SYLLABUS

Compiler Design - 3170701

Credits	Examination Marks				Total Marks
C	Theory Marks		Practical Marks		
	ESE (E)	PA (M)	ESE (V)	PA (I)	
4	70	30	30	20	150

1. **Overview of the Compiler and its Structure** : Language processor, Applications of language processors, Definition-Structure-Working of compiler, the science of building compilers, Basic understanding of interpreter and assembler, Difference between interpreter and compiler, Compilation of source code into target language, Cousins of compiler, Types of compiler. **(Chapter - 1)**
2. **Lexical Analysis** : The Role of the Lexical Analyzer, Specification of Tokens, Recognition of Tokens, Input Buffering, elementary scanner design and its implementation (Lex), Applying concepts of Finite Automata for recognition of tokens. **(Chapter - 2)**
3. **Syntax Analysis** : Understanding Parser and CFG(Context Free Grammars), Left Recursion and Left Factoring of grammar Top Down and Bottom up Parsing Algorithms, Operator-Precedence Parsing, LR Parsers, Using Ambiguous Grammars, Parser Generators, Automatic Generation of Parsers, Syntax-Directed Definitions, Construction of Syntax Trees, Bottom-Up Evaluation of S-Attributed Definitions, L-Attributed Definitions, syntax directed definitions and translation schemes. **(Chapters - 3, 4)**
4. **Error Recovery** : Error Detection & Recovery, Ad-Hoc and Systematic Methods. **(Chapter - 5)**
5. **Intermediate-Code Generation** : Variants of Syntax Trees, Three-Address Code, Types and Declarations, Translation of Expressions, Type Checking, Syntax Directed Translation Mechanisms, Attributed Mechanisms And Attributed Definition. **(Chapter - 6)**
6. **Run-Time Environments** : Source Language Issues, Storage Organization, Stack Allocation of Space, Access to Nonlocal Data on the Stack, Heap Management. **(Chapter - 7)**
7. **Code Generation and Optimization** : Issues in the Design of a Code Generator, The Target Language, Addresses in the Target Code, Basic Blocks and Flow Graphs, Optimization of Basic Blocks, A Simple Code Generator, Machine dependent optimization, Machine independent optimization Error detection of recovery. **(Chapter - 8)**
8. **Instruction-Level Parallelism** : Processor Architectures, Code-Scheduling Constraints, Basic-Block Scheduling, Pass structure of assembler. **(Chapter - 9)**

TABLE OF CONTENTS

Notes

1 Overview of the Compiler and its Structure

Syllabus

Language processor, Applications of language processors, Definition-Structure-Working of compiler, the science of building compilers, Basic understanding of interpreter and assembler. Difference between interpreter and compiler. Compilation of source code into target language, Cousins of compiler, Types of compiler.

Contents

1.1 Language Processor

Defination : A translator is one kind of program that takes one form of program as input and converts it into another form. The input program is called **source language** and the output program is called **target language**.

The source language can be low level language like assembly language or a high level language like C, C++, FORTRAN.

The target language can be a low level language or a machine language.

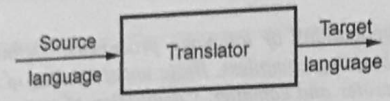


Fig. 1.1.1 Translator

Types of Translator

There are two types of translators **compiler** and **assembler**.

Basic Functions of Translator

1. The translator is used to convert one form of program to another.
2. The translator should convert the source program to a target machine code in such a way that the generated target code should be easy to understand.
3. The translator should preserve the meaning of the source code.
4. The translator should report errors that occur during compilation to its users.
5. The translation must be done efficiently.

1.2 Definition of Compiler

GTU : May-12, Marks 6

In this section we will discuss two things : ***"What is compiler ?*** And ***"Why to write compiler ?"*** Let us start with *"What is compiler?"*

Compiler is a program which takes one language (source program) as input and translates it into an equivalent another language (target program).

During this process of translation if some errors are encountered then compiler displays them as error messages. The basic model of compiler can be represented as follows.

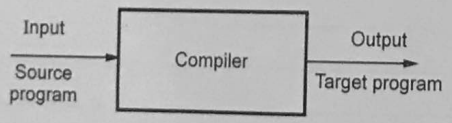


Fig. 1.2.1 Compiler

The compiler takes a source program as higher level languages such as C, PASCAL, FORTRAN and converts it into low level language or a machine level language such as assembly language.

Key Point *Compiler and assemblers are two translators. Translators convert one form of program into another form. Compiler converts high level language to machine level language while assembler converts assembly language program to machine level language.*

Factors that Affect the Design of Compiler are -

1. The choice of source language.
2. The machine architecture on which compiler is executed.
3. The amount of memory available.
4. The type of object code required.

Major Functions Done by Compiler

1. The compilers translate high level source program to machine program.
2. It raises error messages if any, during the process of comilation.
3. The translation of source language to machine language must be done efficiently.
4. While translating, the compiler preserves the meaning of the code.

1.3 Analysis-Synthesis Model

GTU : May-12, Winter-19, Marks 6

The compilation can be done in two parts : Analysis and synthesis. In **analysis part** the source program is read and broken down into constituent pieces. The **syntax** and the **meaning of the source string** is determined and then an **intermediate code** is created from the input source program. In **synthesis part** this intermediate form of the source language is taken and **converted into** an equivalent **target program**. During this process if certain **code** has to be **optimized** for efficient execution then the required code is optimized. The analysis and synthesis model is as shown in Fig. 1.3.1.

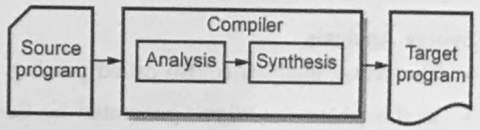


Fig. 1.3.1 Analysis and synthesis model

Review Question

1. *Explain the analysis synthesis model of compilation. List the factors that affect the design of complier. Also list major functions done by compiler.* **GTU : May-12, Winter-19, Marks 6**

1.4 Phases of Compiler

GTU : Winter-12, 13, 17, 20, Summer-14, 16, 17, 19, Marks 7

The process of compilation is carried out in two parts : **Analysis and synthesis.** Again the analysis is carried out in three phases : **Lexical analysis, syntax analysis** and **semantic analysis.** And the synthesis is carried out with the help of **intermediate code generation, code generation** and **code optimization.** Let us discuss these phases one by one.

1. Lexical Analysis

- The lexical analysis is also called **scanning.**
- It is the phase of compilation in which the complete source code is scanned and your source program is broken up into group of strings called **token.**
- A token is a sequence of characters having a collective meaning. For example if some assignment statement in your source code is as follows,

  total = count + rate * 10

Then in lexical analysis phase this statement is broken up into series of tokens as follows.

1. The identifier **total**
2. The **assignment** symbol
3. The identifier **count**
4. The **plus** sign
5. The identifier **rate**
6. The **multiplication** sign
7. The constant number **10**

The blank characters which are used in the programming statement are eliminated during the lexical analysis phase.

2. Syntax Analysis

- The syntax analysis is also called **parsing.**
- In this phase the tokens generated by the lexical analyser are grouped together to form a **hierarchical structure.**
- The syntax analysis determines the structure of the source string by grouping the tokens together. The hierarchical structure generated in this phase is called **parse tree** or **syntax tree.** For the expression **total = count + rate *10** the parse tree can be generated as follows.

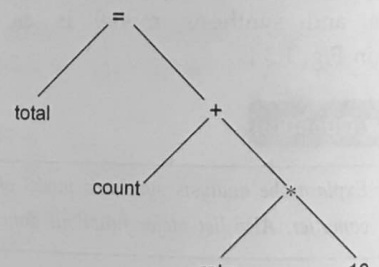


Fig. 1.4.1 Parse tree for total = count + rate *10

In the statement *'total = count + rate *10 '* first of all *rate*10* will be considered because in arithmetic expression the multiplication operation should be performed before the addition. And then the addition operation will be considered. For building such type of syntax tree the production rules are to be designed. The rules are usually expressed by context free grammar. For the above statement the production rules are -

(1) E ← identifier
(2) E ← number
(3) E ← E1 + E2
(4) E ← E1*E2
(5) E ← (E)

where E stands for an expression.

- By rule (1) count and rate are expressions and
- by rule(2) 10 is also an expression.
- By rule (4) we get rate*10 as expression.
- And finally count +rate*10 is an expression.

3. Semantic Analysis

- Once the syntax is checked in the syntax analyser phase the next phase i.e. the semantic analysis determines the meaning of the source string.
- For example meaning of source string means matching of parenthesis in the expression, or matching of if ...else statements or performing arithmetic operations of the expressions that are type compatible, or checking the scope of operation.

For example,

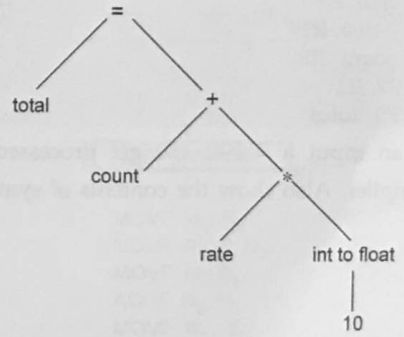


Fig. 1.4.2 Semantic analysis

Thus these three phases are performing the task of analysis. After these phases an intermediate code gets generated.

4. Intermediate Code Generation

- The intermediate code is a kind of code which is easy to generate and this code can be easily converted to target code. This code is in variety of forms such as *three address code, quadruple, triple, posix.*
- Here we will consider an intermediate code in three address code form. This is like an assembly language. The **three address code** consists of instructions each of which has at the **most three operands.** For example,

```
t1 := int_to_float (10)
t2 := rate × t1
t3 := count + t2
total := t3
```

5. Code Optimization

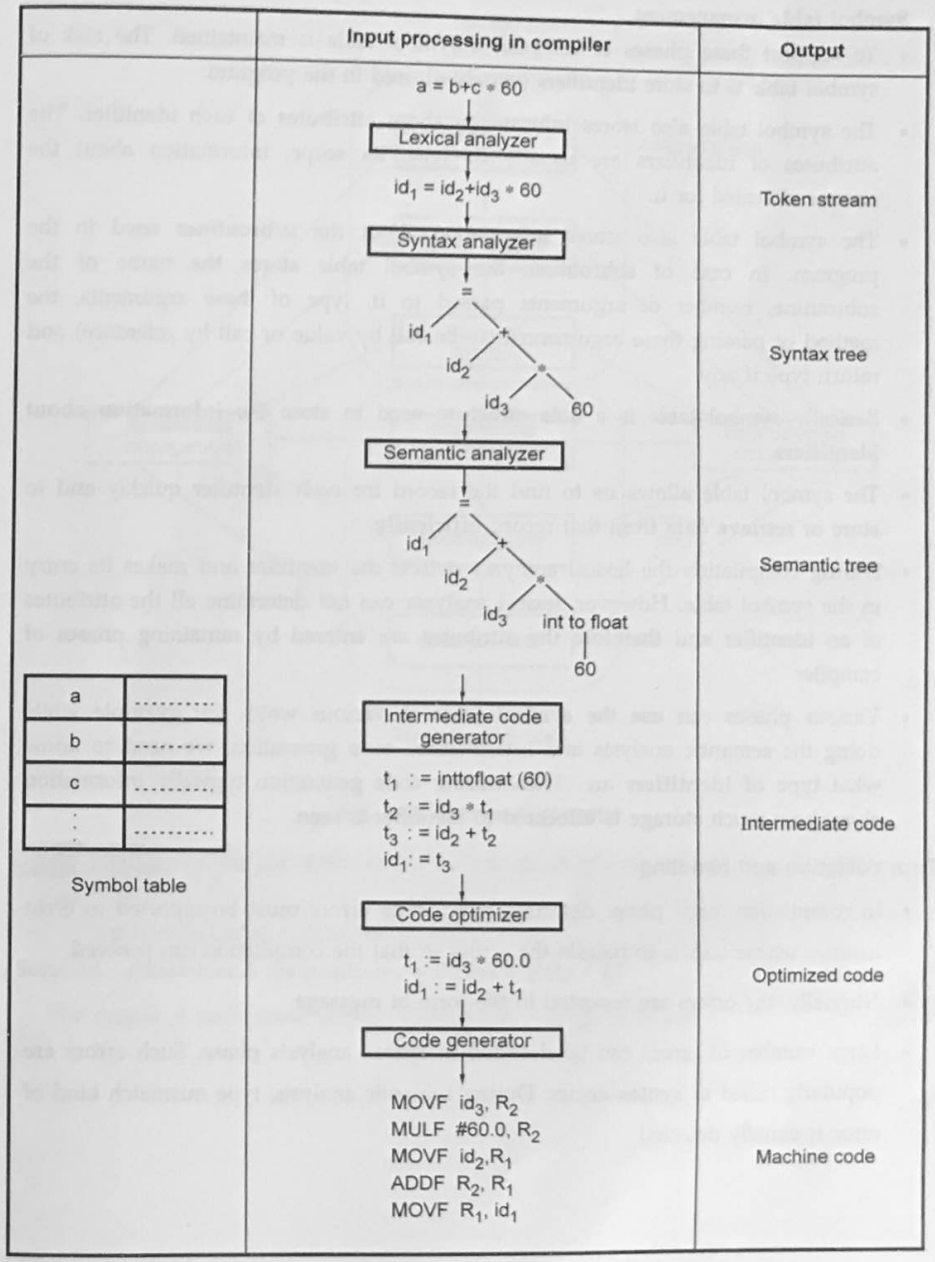
- The code optimization phase attempts to improve the intermediate code.
- This is necessary to have a faster executing code or less consumption of memory.
- Thus by optimizing the code the overall running time of the target program can be improved.

6. Code Generation

- In code generation phase the target code gets generated.
- The intermediate code instructions are translated into sequence of machine instructions.

```
MOV rate, R1
MUL #10.0, R1
MOV count, R2
ADD R2, R1
MOV R1, total
```

Example - Show how an input a = b+c *60 get processed in compiler. Show the output at each stage of compiler. Also show the contents of symbol table.



Symbol table management

- To support these phases of compiler a symbol table is maintained. The task of symbol table is to store identifiers (variables) used in the program.
- The symbol table also stores information about **attributes** of each identifier. The attributes of identifiers are usually its type, its scope, information about the storage allocated for it.
- The symbol table also stores information about the **subroutines** used in the program. In case of subroutine, the symbol table stores the name of the subroutine, number of arguments passed to it, type of these arguments, the method of passing these arguments(may be call by value or call by reference) and return type if any.
- Basically symbol table is a data structure used to store the **information about identifiers.**
- The symbol table allows us to find the record for each identifier quickly and to **store or retrieve data** from that record **efficiently**.
- During compilation the lexical analyzer detects the identifier and makes its entry in the symbol table. However, lexical analyzer can not determine all the attributes of an identifier and therefore the attributes are entered by remaining phases of compiler.
- Various phases can **use** the **symbol table** in various ways. For example while doing the semantic analysis and intermediate code generation, we need to know what **type of identifiers** are. Then during code generation typically information about how much **storage is allocated** to identifier is seen.

Error detection and handling

- In compilation, each phase detects errors. These errors must be reported to error handler whose task is to handle the errors so that the compilation can proceed.
- Normally, the errors are reported in the form of **message**.
- Large number of errors can be detected in syntax analysis phase. Such errors are popularly called as **syntax errors**. During semantic analysis; type mismatch kind of error is usually detected.

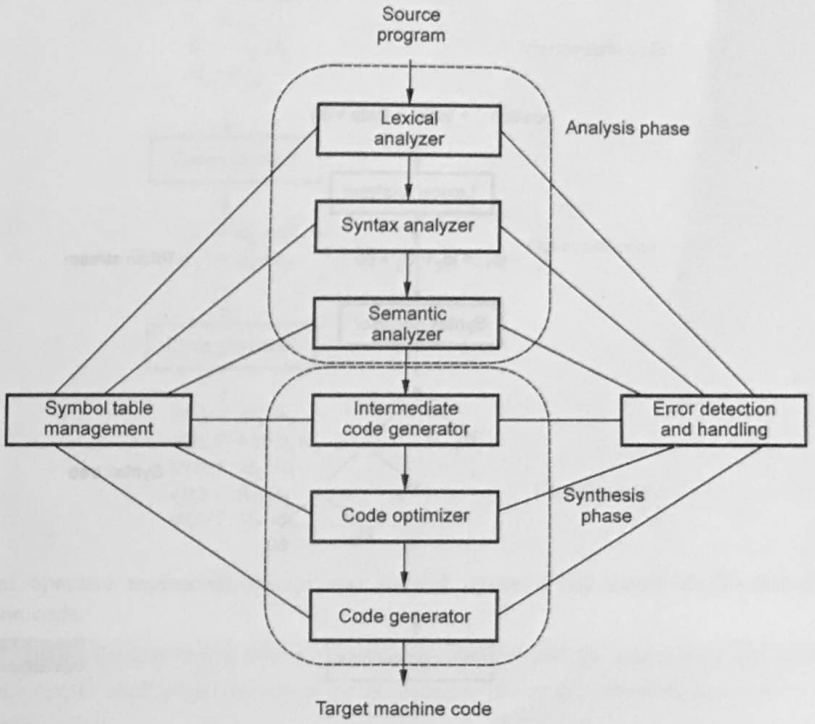


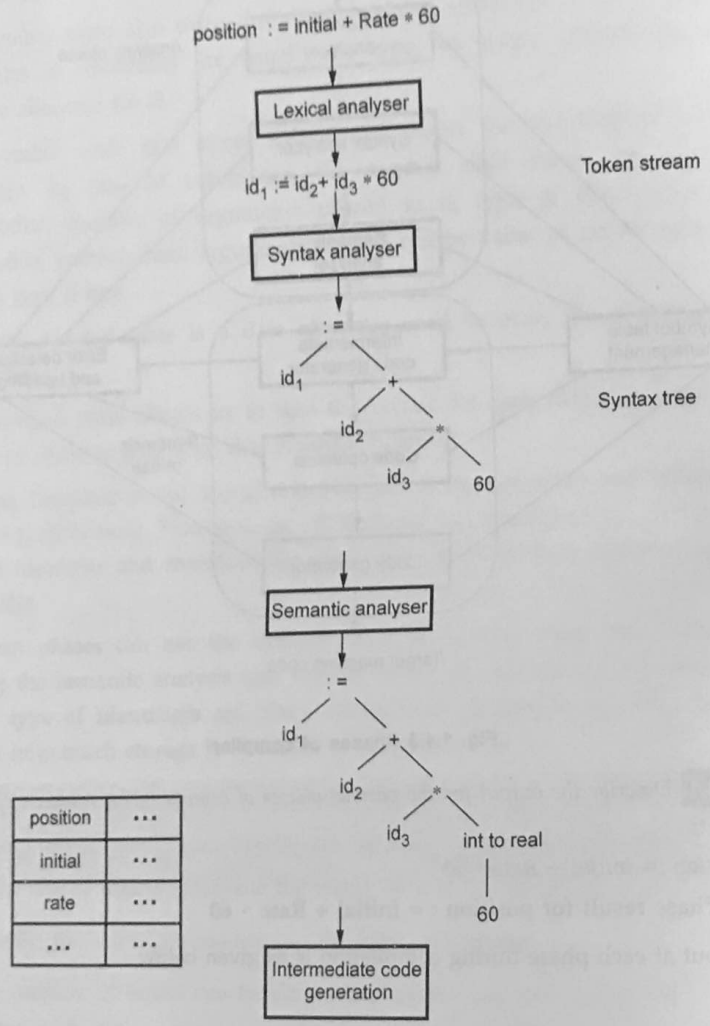
Fig. 1.4.3 Phases of compiler

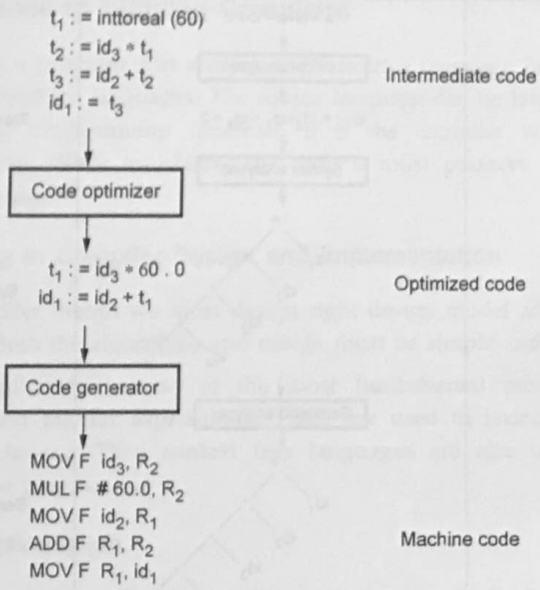
Example 1.4.1 *Describe the output for the various phases of compiler with respect to following statements :*

*position := initial + Rate * 60*

Solution : Phase result for position : = initial + Rate * 60

The output at each phase during compilation is as given below -





First operand represents source and second operand represents destination in the machine code.

Example 1.4.2 *Explain lexical analysis phase of a compiler and for a statement given below, write output of all phases (except of an optimization phase) of a compiler. Assume a, b and c of type float*

$$a = a + b * c * 2;$$

GTU : Winter-12, Marks 7

Solution : See Fig. 1.4.4 on next page.

Review Questions

1. *Draw structure of compiler. Also explain analysis phase in brief.* **GTU :Winter-13, Marks 7**
2. *List out phases of a compiles. Write a brief note on lexical analyzer.* **GTU : Summer-14, Marks 6**
3. *Explain different phases of compiler* **GTU : Summer-14, 16, 17, Winter-20, Marks 7**
4. *Explain analysis phase of source program with example.* **GTU : Winter-17, Marks 4, Summer-19, Marks 7**

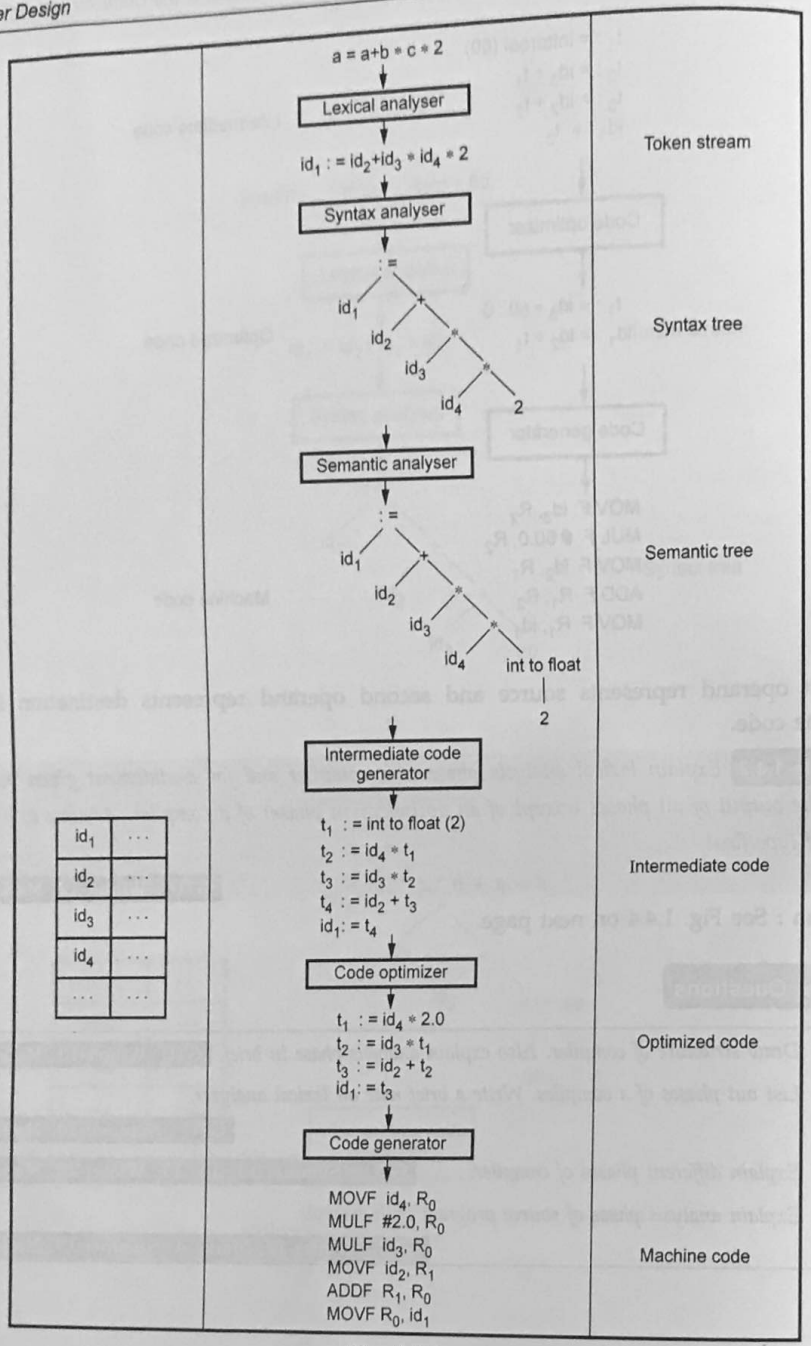


Fig. 1.4.4

1.5 The Science of Building Compilers

- Compiler is a program that accepts all the source language programs and convert them in a machine languages. The source language can be large or small, it might contain any programming construct, it is the compiler who converts it into machine code. While translating the code it must preserve the meaning of the source language.

1.5.1 Modeling in Compiler Design and Implementation

- In the compiler design we must design right design model and must choose right algorithm. Both the algorithms and design must be simple and efficient.
- Many compilers make use of the most fundamental model i.e **Finite State machines** and **regular expressions**. These are used in lexical analysis phase for identifying **tokens**. Then **context free languages** are also used to describe the syntax of the program.

1.5.2 Code Optimization

- The desiring feature of any compiler is to produce **optimized code.** Optimized code means the code that **executes efficiently** on the machine. For the compiler the optimized code becomes **complex** and **important**.
- For optimization following are the compiler design **objectives** that are used -
  1. When optimization is done then the meaning of the source program must be preserved. This is called **correct optimization.**
  2. Due to optimization the performance of the program must get improved.
  3. Compiler time required to execute the optimized code must be reasonable.
  4. The efforts required due to optimization must be manageable.

1.6 Applications of Language Processors

1.6.1 Optimization for High Level Programming Languages

- Compiler takes the higher level languages as input and produces the target code. The high level languages are less efficient as compared to low level languages. Their target code runs slowly. But low level languages are difficult to understand and are not portable. Optimizing compilers generally improve the performance of the generated code by eliminating the abstractions posed by the high level languages.
- As new languages get introduced the newer features are getting added up. But the most commonly addressed programming language features are - aggregated data

types, arrays, structures, control flows, loops and procedure invocation. The **data flow optimization** mechanism is used to optimize the flow of control.

- Java has built in **garbage collection** facility that automatically frees the memory of those variables that are no longer in use. These features increase the run time overhead. Compiler optimization will reduce this overhead.
- Dynamic optimization is one important optimization technique that extracts the required information dynamically and thus reduces the overhead at run time.

1.6.2 Optimization for Computer Architecture

- The high performance systems take advantage of two basic techniques : **parallelism** and **memory hierarchy.**
- All the modern architectures make use of instruction level parallelism. This **parallelism** is hidden from the programmer. The compilers can rearrange the instructions in such a way that parallelism can be more efficient.
- The **memory hierarchy** means arranging several levels of storage with different speeds and sizes. The level closest to the processor is fastest but smallest in size. In compilers the more emphasis is given for making the memory hierarchy efficient.

1.6.3 Designing of New Computer Architecture

- In modern computer several effective architectures are used. To exploit the features of these architectures, the compilers are developed in processor design stage, it is then run on **simulators** and then used to evaluate the architectural features.
- The RISC, CISC architectures are highly influenced by the compilers.
- Various specialized architectures are getting invented. For instance - SIMD (Single Instruction Multiple Data), VLIW (Very Long Instruction Word) machines, symbolic arrays, multiprocessors with distributed shared memory. The development in these architecture leads the development and improvement in compiler technologies.

1.6.4 Program Translation

The compilation technology involves translation of the program from high level to machine level language. Following are some important **applications** of program translation techniques -

Binary Translation

- Compiler are used to translate binary code of one machine to run it on another machine. Thus due to binary translation the one machine code can be run on another machine.

Hardware Synthesis

- There are hardware description languages such as VHDL, Verilog and so on. These languages help to write a specification file in which the hardware is described. The hardware synthesis tools translate this description into gates and physical layout. These tools help in obtaining the optimizing circuit.

Query Interpreter

- The SQL (Structured Query Languages) are used to search database. The SQL interpreters compile or interpret the queries given at the command prompt.

Compiled Simulation

- Simulation is a general technique in which the design is validated. Simulation is very expensive. Instead of writing simulator it is faster to compile the design to produce machine code. Compiled simulation is used to simulate design written by the VHDL or Verilogs.

1.6.5 Software Productivity Tools

- Programs are the most important elements of any software systems. The errors in the program can cause crashing of the entire system. Hence testing is done to locate errors in the program.
- Data flow analysis technique is used to locate errors along the execution path. This technique used in testing is mainly derived in compilers. Other important techniques of locating errors are -

Type Checking

The type checking is an effective tool used to catch the inconsistencies in the data types. When operation with wrong types is carried out then this technique identifies the errors.

Bounds Checking

- Boundary value checking is a technique in which **array index boundary** can be checked using bounds check technique. This technique is used to check the overflow of the buffers in the program.

Memory Management Tools

- **Garbage collector** is an excellent example of memory management tools. There are memory management errors such as memory leaks which are mostly occur in C or C++. Various tools are developed to help the programmer to find out the memory management errors.

1.7 Basic Understanding of Interpreter and Assembler

Definition : An interpreter is a kind of translator which produces the result directly when the source language and data is given to it as input.

- It **does not produce the object code** rather each time the program needs execution.
- The model for interpreter is as shown in Fig. 1.7.1.

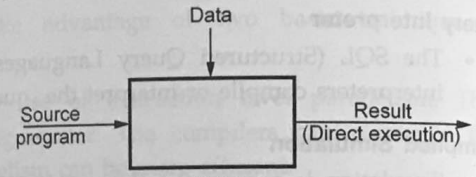


Fig. 1.7.1 Interpreter

- Languages such as BASIC, SNOBOL, LISP can be translated using interpreters. JAVA also uses interpreter.
- The process of interpretation can be carried out in following phases.
  1. Lexical analysis
  2. Syntax analysis
  3. Semantic analysis
  4. Direct execution

Advantages :

- Modification of user program can be easily made and implemented as execution proceeds.
- Type of object that denotes a variable may change dynamically.
- Debugging a program and finding errors is simplified task for a program used for interpretation.
- The interpreter for the language makes it machine independent.

Disadvantages :

- The **execution** of the program is **slower.**
- **Memory** consumption is **more.**

Assembler

Compiler is a kind of translator that converts the high level source program into the machine language and assembler is a kind of translator that converts low level source program into the machine language.

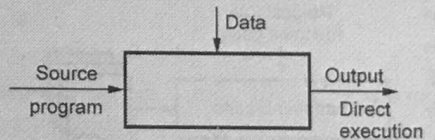
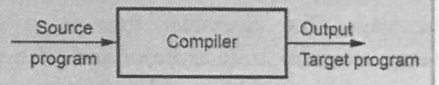
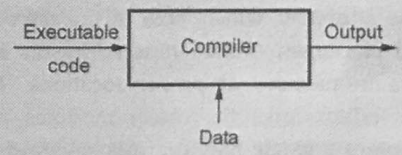
Example of high level language is C,C++, PASCAL and so on.

Example of low level language is assembly language.

1.8 Difference between Interpreter and Compiler

GTU ; Winter-17, 20, Summer-19, Marks 4

The analysis phase of interpreter and compiler is same i.e. in both lexical, syntactic and semantic analysis is performed.

Sr. No.	Interpreter	Compiler
1.	**Demerit :** The source program gets interpreted every time it is to be executed, and every time the source program is analyzed. Hence interpretation is **less efficient** than Compiler.	**Merit :** In the process of compilation the program is analyzed only once and then the code is generated. Hence compiler is efficient than interpreter.
2.	The interpreters do not produce object code.	The compilers produce object code.
3.	**Merit :** The interpreters can be made **portal** because they do not produce object code.	**Demerit :** The compilers has to be present on the host machine when particular program needs to be compiled.
4.	**Merit :** Interpreters are **simpler** and give us **improved debugging environment.**	**Demerit :** The compiler is a **complex** program and it requires large amount of memory.
5.	An interpreter is a kind on translator which produces the results directly when the source language and data is given to it as input. 	An compiler is a kind of translator which takes only source program as input and converts it into object code.  Then loader performs loading and link editing and prepares an executable code. Compiler takes this executable code and data as input and produces output. 
6.	**Examples of interpreter :** **A UPS Debugger** is basically a graphical source level debugger but it contains built in C interpreter which can handle multiple source files.	**Example of Compiler :** **Borland C compiler** or **Turbo C compiler** compiles the programs written in C or C++.

Review Questions

1. *What is the difference between compiler and interpreter?* **GTU : Winter-17, Marks 3, Summer-19, Marks 4**
2. *Compare and contrast compiler and interpreter* **GTU : Winter-20, Marks 3**

1.9 Compilation of Source Code into Target Language

GTU : Winter-13, Marks 8

To create an executable form of your source program only a compiler program is not sufficient. You may require several other programs to create an executable target program. After a synthesis phase a target code gets generated by the compiler. This target program generated by the compiler is processed further before it can be run which is as shown in the Fig. 1.9.1.

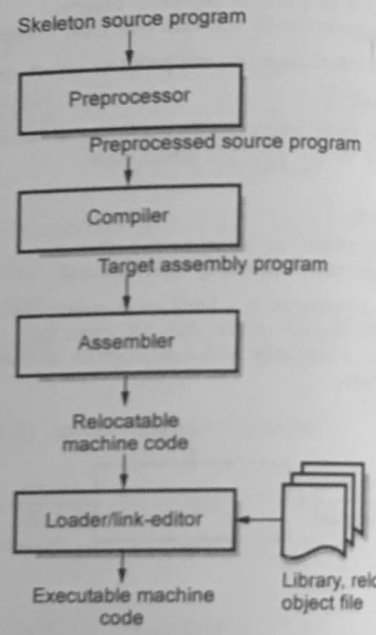


Fig. 1.9.1 (a) Process of execution of program

The compiler takes a source program written in high level language as an input and converts it into a target assembly language. The assembler then takes this target assembly code as input and produces a relocatable machine code as an output. Then a program **loader** is called for performing the task of **loading and link editing**. The task of loader is to perform the relocation of an object code. **Relocation** of an object code means allocation of load time addresses which exist in the memory and placement of load time addresses and data in memory at proper locations. The link editor links the object modules and prepares a single module from several files of relocatable object modules to **resolve** the **mutual references**. These files may be library files and these library files may be referred by any program.

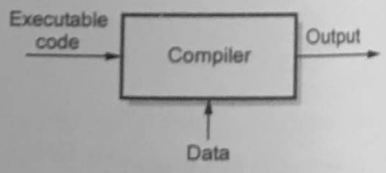


Fig. 1.9.1 (b) Process of execution of program

Properties of Compiler

When a compiler is built it should posses following properties.

1. The compiler itself must be **bug-free**.
2. It must generate **correct machine code**.
3. The generated machine **code must run fast**.
4. The **compiler** itself must run **fast** (compilation time must be proportional to program size).
5. The compiler must be **portable** (i.e. modular, supporting separate compilation).
6. It must give **good diagnostics** and error messages.
7. The generated code must work well with **existing debuggers**.
8. It must have **consistent optimization**.

Review Question

1. *Explain the roles of linker, loader and preprocessor.* **GTU : Winter-13, Marks 8**

1.10 Cousins of Compiler

GTU : May-12, Winter-18, Marks 4

Sometimes the output of preprocessor may be given as input to the compiler.

Cousins of compiler means the context in which the compiler typically operates. Such contexts are basically the programs such as preprocessor, assemblers, loaders and link editors.

Let us discuss them in detail.

1. Preprocessors - The output of preprocessors may be given as the input to compilers. The tasks performed by the preprocessors are given as below

Preprocessors allow user to use macros in the program. Macro means some set of instructions which can be used repeatedly in the program. Thus macro preprocessing task is be done by preprocessors.

Preprocessor also allows user to include the header files which may be required by the program.

For example :

```
#include<stdio.h>
```

By this statement the header file *stdio.h* can be included and user can make use of the functions defined in this header file. This task of preprocessor is called file **inclusion**.

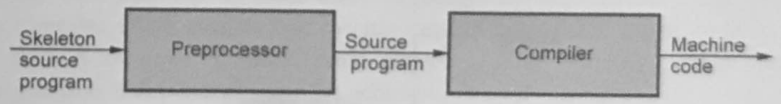


Fig. 1.10.1 Role of preprocessor

Macro preprocessor - Macro is a small set of instructions. Whenever in a program macroname is identified then that name is replaced by macro definition (i.e. set of instruction defining the corresponding macro).

For a macro there are two kinds of statements **macro definition** and **macro use**. The macro definition is given by keyword like *"define"* or *"macro"* followed by the name of the macro.

For example :

```
# define PI 3.14
```

Whenever the "PI" is encountered in a program it is replaced by a value 3.14.

2. **Assemblers** - Some compilers produce the assembly code as output which is given to the assemblers as an input. The assembler is a kind of translator which takes the assembly program as input and produces the machine code as output.

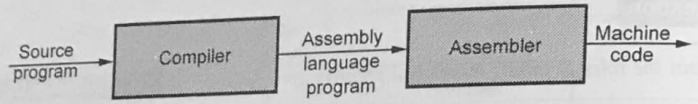


Fig. 1.10.2 Role of assembler

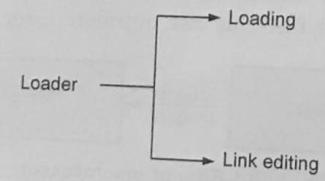
An assembly code is a mnemonic version of machine code. The typical assembly instructions are as given below.

```
MOV a, R1
MUL #5,R1
ADD #7,R1
MOV R1,b
```

The assembler converts these instructions in the **binary language** which can be understood by the machine. Such a binary code is often called as **machine code**. This machine code is a *relocatable* machine code that can be passed directly to the loader/linker for execution purpose.

The assembler converts the assembly program to low level machine language using two passes. A pass means one complete scan of the input program. The end of second pass is the relocatable machine code.

3. Loaders and Link Editors - Loader is a program which performs two functions: Loading and link editing. Loading is a process in which the relocatable machine code is



read and the relocatable addresses are altered. Then that code with altered instructions and data is placed in the memory at proper location. The job of link editor is to make a single program from several files of **relocatable machine code**. If code in one file refers the location in another file then such a reference is called **external reference**. The link editor resolves such external references also.

Review Questions

1. *What does the linker do ? What does the loader do ? What does the preprocessor do ? Explain their role (s) in compilation process.* **GTU : May-12, Marks 4**

(Hint: For working of Linker and Loader refer section 1.9)

2. *List the cousins of compiler and explain the role of any one of them.* **GTU : Winter-18, Marks 3**

1.11 Types of Compiler

In this section we will discuss various types of compilers.

1.11.1 Incremental Compiler

Incremental compiler is a compiler which performs the recompilation of only modified source rather than compiling the whole source program.

The basic **features** of incremental compiler are,

1. It tracks the dependencies between output and the source program.
2. It produces the same result as full recompile.
3. It performs less task than the recompilation.
4. The process of incremental compilation is effective for maintenance.

1.11.2 Cross Compiler

Basically there exists three types of languages

1. **Source language** i.e. the application program.
2. **Target language** in which machine code is written.
3. The **Implementation language** in which a **compiler is written**.

There may be a case that all these three languages are different. In other words there may be a compiler which run on one machine and produces the target code for another machine. Such a compiler is called **cross compiler**. Thus by using cross compilation technique platform independency can be achieved.

To represent cross compiler T diagram is drawn as follows

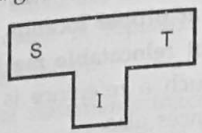


Fig. 1.11.1 (a) T diagram with S as source, T as target and I as implementation language

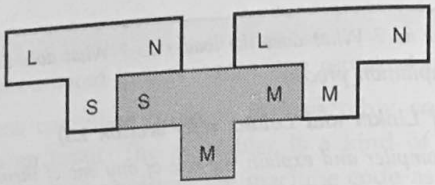


Fig. 1.11.1 (b) Cross compiler

For source language L the target language N gets generated which runs on machine M.

For example :

For the first version of EQN compiler, the compiler is written in C and the command are generated for TROFF, which is as shown in Fig. 1.9.1.

The cross compiler for EQN can be obtained by running it on PDP-11 through C copiler, which produces output for PDP-11 as shown below -

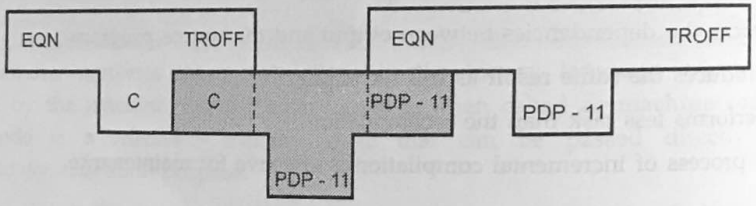


Fig. 1.11.2 Cross compiler for EQN

1.12 Short Questions and Answers

Q.1 What is translator ?

Ans. : Translator is a kind of program which converts one form of the source language into another form of the target language.

Q.2 Enlist the two commonly used translators in operating system. What is their purpose ?

Ans. : The assembler and the compiler are the two commonly used translators in operating system. The assembler converts the source program written in assembly

language into equivalent machine code. The compiler converts the source program written in high level language into the equivalent machine code.

Q.3 What is compiler ?

Ans. : The compiler is a kind of translator which converts the high level language into the machine level language. The examples of high level languages are C, Fortran, C++, Pascal and so on.

Q.4 What are the phases of compiler ?

Ans. : Various phases of compiler are lexical analysis, syntax analysis or parsing, semantic analysis, intermediate code generation, code generation and code optimization.

Q.5 What are the cousins of compiler ?

Ans. : Cousins of compiler means the context in which the compiler typically operates. Such contexts are basically the programs such as preprocessor, assemblers, loaders and link editors.

Q.6 What is an interpreter ?

Ans. : An interpreter is a kind of translator which produces the result directly when the source language and data is given to it as input. It does not produce the object code rather each time the program needs execution.

Q.7 What is the advantage of front end and back end model of compiler ?

Ans. : Following are the advantages of using the front end and back end model of the compiler -

- By keeping the same front end and attaching different back ends one can produce a compiler for same source language on different machines.
- By keeping different front ends and same back end one can compile several different languages on the same machine.

Q.8 What are machine dependant and machine independent phases ?

Ans. : The machine dependent phases are code generation and code optimization phases. The machine independent phases are lexical analyzers, syntax analyzers, semantic analyzers.

Q.9 What are the factors affecting number of passes in compiler ?

Ans. : Various factors affecting the number of passes in compiler are -

1. Forward reference 2. Storage limitations 3. Optimization.

Q.10 Define the term cross compiler.

Ans. : There may be a compiler which run on one machine and produces the target code for another machine. Such a compiler is called cross compiler.

Q.11 What are the phases included in front end of a compiler ? What does the front end produce ?

Ans. : The phases of front end of compiler are lexical analysis, syntax analysis and semantic analysis. The front end produces an intermediate code.

Q.12 Mention few cousins of the compiler.

Ans. : Some of the cousins of the compiler are -

i) Preprocessor ii) Macro preprocessor iii) Assemblers iv) Loaders and Linkers

Q.13 How will you group the phases of compiler ?

Ans. : The phases of compiler are grouped in such a way that some phases will act as a front end while the remaining phases will act as a back end of the compiler. The front end is basically for analysis of the source program while the back end is for synthesis.

Q.14 What is the purpose of programming languages ?

Ans. : The purpose of programming languages is to make effective use of computer to solve the complex problems.

1.13 Multiple Choice Questions

Q.1 What is compiler ?

a Compiler is an editor.

b Compiler is a program that converts high level source program into the machine code.

c Compiler is a program that converts low level source program into the machine code.

d Compiler is a general purpose application program.

Q.2 What are the stages of compilation process ?

a Requirement analysis, design, implementation, testing and maintenance.

b Documentation, coding and testing.

c Testing and quality assurance.

d Lexical analysis, syntax analysis, intermediate code generation, code generation and code optimization.

Q.3 The languages such as C,C++, PASCAL and FORTRAN are referred as _____.

a databases

b high level programming languages.

c low level programming languages.

d middle level programming languages.

Q.4 The definition of interpreter is _____.

a general purpose application program.

b representation of the system which is implemented from the design.

c kind of translator that does the conversion line by line as program runs.

d a program used for editing the source code.

Q.5 Following languages are translated using the interpreters

a C, PASCAL, FORTRAN.

b LISP, SNOBOL, JAVA.

c Assembly language

d Expert system, knowledge based system.

Q.6 Translation of low level language to machine code is done by _____.

a compiler

b interpreter

c assembler

d loader

Q.7 Cross compiler is a compiler _____.

a that runs on one machine but produces object code for another machine.

b which is written in a language that is different from the source program.

c is written in the same language of source program.

d generates the object code for the host machine only.

Q.8 Incremental compiler is _____.

a that runs on one machine but produces object code for another machine.

b which is written in a language that is different from the source program.

c is written in the same language of source program.

d that allows a modified portion of the program to be recompiled.

Q.9 An ideal compiler is _____.

a that takes less time for compilation

b which converts the high level source program to machine level language.

c which produces the object code which is smaller in size and execute faster.

d All of the above.

Q.10 Interpreter is preferred than compiler because _____.

a it takes less time to execute.

b it is helpful in initial phases of program development process.

c debugging is faster and easier.

d it requires less number of resources.

Q.11 An interpreter is a program that _____ .

a converts the high level language into a machine level language by producing the object code.

b produces the results directly when the source language and data is given as input.

c automates the translation of assembly language into machine language.

d places the source program in the memory and prepares for execution.

Q.12 Compiler is preferred than interpreter because _____ .

a it takes less time to execute.

b it is helpful in initial phases of program development process.

c debugging is faster and easier.

d it requires less number of resources.

Q.13 Compiler is preferred than interpreter because _____ .

a converts the high level language into a machine level language by producing the object code.

b produces the results directly when the source language and data is given as input.

c automates the translation of assembly language into machine language.

d places the source program in the memory and prepares for execution.

Q.14 A system program that places an executable program into the memory ready for execution is _____ .

a assembler b compiler

c loader d linker

Q.15 Which of the following system program converts the assembly language into the object code ?

a Assembler b Compiler

c Loader d Linker

Q.16 Syntax directed translation engines are _____ .

a loaders b operating system

c compilers d compiler construction tools

Q.17 Storage mapping is done by _____ .

a loader b linker

c compiler d operating system

Q.18 The front end and back end model of compiler is beneficial because _____ .

a it takes less time to execute the source code.

b same program can be compiled on different machines

c it takes less space for execution.

d programs written in different languages can be compiled by the same compiler.

Q.19 The external references are resolved by _____.

a loader b link editors

c compilers d assemblers

Q.20 Compilers are generally written by _____.

a computer users. b professional programmers

c database administrators d project managers

Answer Keys for Multiple Choice Questions

Q.1	b	Q.6	c	Q.11	b	Q.16	d
Q.2	d	Q.7	a	Q.12	a	Q.17	c
Q.3	b	Q.8	d	Q.13	a	Q.18	b, d
Q.4	c	Q.9	d	Q.14	c	Q.19	b
Q. 5	b	Q.10	b, c	Q.15	a	Q.20	b

❑❑❑

Notes

2 Lexical Analysis

Syllabus

The Role of the Lexical Analyzer, Specification of Tokens, Recognition of Tokens, Input Buffering, elementary scanner design and its implementation (Lex), Applying concepts of Finite Automata for recognition of tokens.

Contents

2.1 The Role of the Lexical Analyzer

GTU : Winter-15, 19, 20, May-12, Summer-19, Marks 7

Lexical analyzer is the **first phase** of compiler. The lexical analyzer reads the input source program from left to right one character at a time and generates the sequence of tokens. Each **token** is a single logical cohesive unit such as **identifier, keywords, operators** and **punctuation marks**. Then the parser to determine the syntax of the source program can use these tokens. The role of lexical analyzer in the process of compilation is as shown below -

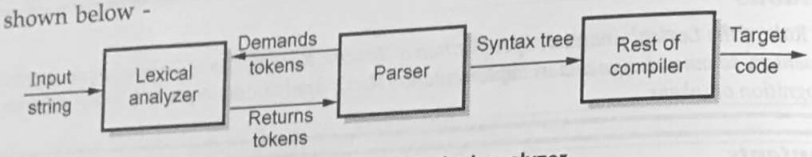


Fig. 2.1.1 Role of lexical analyzer

As the lexical analyzer scans the source program to recognize the tokens it is also called as scanner. Apart from **token identification** lexical, analyzer also performs following functions.

Functions of lexical analyzer

1. It produces stream of tokens.
2. It eliminates blank and comments.
3. It generates symbol table which stores the information about identifiers, constants encountered in the input.
4. It keeps track of line numbers.
5. It reports the error encountered while generating the tokens.

The lexical analyzer works in two phases. In first phase it performs scan and in the second phase it does lexical analysis; means it generates the series of tokens.

2.1.1 Tokens, Patterns, Lexemes

Let us learn some terminologies, which are frequently used when we talk about the activity of lexical analysis.

Tokens : It describes the class or category of input string. For example, identifiers, keywords, constants are called tokens.

Patterns : Set of rules that describe the token.

Lexemes : Sequence of characters in the source program that are matched with the pattern of the token. For example, int, i, num, ans, choice.

Let us take one example of programming statement to clearly understand these terms -

```
if (a<b)
```

Here "if", "(" ,"a" ,"<" ,"b" ,")" are all lexemes. And "if" is a keyword, "(" is opening parenthesis, "a" is identifier, "<" is an operator and so on.

Now to define the identifier pattern could be -

1. Identifier is a collection of letters.
2. Identifier is a collection of alphanumeric characters and identifier's beginning character should be necessarily a letter.

When we want to compile a given source program, we submit this program to compiler. A compiler scans the source program and produces sequence of tokens therefore lexical analysis is also called as **scanner**. For example

Example 2.1.1 *Generate appropriate tokens for given piece of source code.*

```
int MAX (int a, int b)
{
        if (a>b)
        return a;
        else
        return b;
}
```

Solution :

Lexeme	Token
int	keyword
MAX	identifier
(	operator
int	keyword
a	identifier
,	operator
int	keyword
b	identifier
)	operator
{	operator
if	keyword
⋮	⋮

- The blank and new line characters can be ignored. These stream of tokens will be given to syntax analyzer.

Example 2.1.2 *Define lexeme, token and pattern. Identify the lexemes that make up the tokens in the following program segment. Indicate corresponding token and pattern.*

```
void swap(int a,int b)
{
    int k;
    k=a;
    a=b;
    b=k;
}
```

GTU : Winter-15, Marks 7

Solution : Lexeme, pattern and token - Refer section 2.1.1.

Lexeme	Token
void	keyword
swap	identifier
(	operator
int	keyword
a	identifier
,	operator
int	keyword
b	identifier
)	operator
{	operator
int	keyword
k	identifier
k	identifier
=	operator
a	identifier
a	identifier
=	operator
b	identifier
b	identifier

=	operator
k	identifier
}	operator

Patterns

1) **Identifier**
   - i) Identifier is a collection of letters.
   - ii) Identifier is a collection of alphanumeric chracters.
   - iii) The first character of identifier must be a letter.

2) **Operator**
   - i) Operator can be arithmetic, logical, relational operators.
   - ii) The parenthesis are considered as operators.
   - iii) Comma is treated as separation operator.
   - iv) Assignment is denoted by operator.

3) **Keyword**
   - i) Keyword are special words to which some meaning is associated with.
   - ii) int, void are keywords for denoting data types.

Review Questions

1. *How do the parser and scanner communicate ? Explain with the block diagram communication between them.* **GTU : May-12, Marks 4**
2. *Define lexemes, patterns and tokens* **GTU : Summer-19, Winter-19, 20, Marks 3**

2.2 Specification of Tokens

GTU : Nov.-11, May-12, Winter-13, 19, Summer-16, Marks 7

To specify tokens **regular expressions** are used. When a pattern is matched by some regular expression then token can be recognized. Let us understand the fundamental concepts of language.

2.2.1 Strings and Language

String is a collection of finite number of alphabets or letters. The strings are synonymously called as words.

- The length of a string is denoted by | S |.
- The empty string can be denoted by ε.
- The empty set of strings is denoted by Φ.

Following terms are commonly used in strings.

Term	Meaning
Prefix of string	A string obtained by removing zero or more tail symbols. For example, for string Hindustan the prefix could be 'Hindu'.
Suffix of string	A string obtained by removing zero or more leading symbols. For example, for string Hindustan the suffix could be 'stan'.
Substring	A string obtained by removing prefix and suffix of a given string is called substring. For example, for string Hindustan the string 'indu' can be a substring.
Sequence of string	Any string formed by removing zero or more not necessarily the contiguous symbols is called sequence of string. For example, Hisan can be sequence of string.

2.2.2 Operations on Language

As we have seen that the language is a collection of strings. There are various operations which can be performed on the language.

Operation	Description
Union of two languages L1 and L2.	L1∪ L2 = {set of strings in L1 and strings in L2}.
Concatenation of two languages L1 and L2.	L1·L2 = {set of strings in L1 followed by set of strings in L2}.
Kleen closure of L.	$L^* = \bigcup_{i=0}^{\infty} L^i$ L^* denotes zero or more concatenations of L.
Positive closure of L.	$L^+ = \bigcup_{i=1}^{\infty} L^i$ L^+ denotes one or more concatenations of L.

For example, Let L be the set of alphabets such as L= {A, B, C ...Z, a, b, c...z} and D be the set of digits such as D = { 0, 1, 2...9} then by performing various operations as discussed above new languages can be generated as follows -

- L∪ D is a set of letters and digits.
- LD is a set of strings consisting of letters followed by digits.
- L^5 is a set of strings having length of 5 each.
- L^* is a set of strings having all the strings including ε.

- L^+ is a set of strings except ε.

2.2.3 Regular Set

The finite set which denotes a regular language and the set which can be described by regular expression is called **regular set.**

For example : A set of identifier is a regular set because it can be represented using regular expression.

2.2.4 Regular Expressions

Regular expressions are mathematical symbolisms which describe the set of strings of specific language. It provides convenient and useful notation for representing tokens. Here are some rules that describe definition of the regular expressions over the input set denoted by Σ.

1. ε is a regular expression that denotes the set containing empty string.
2. If R1 and R2 are regular expressions then R = R1 + R2 (same can also be represented as R = R1|R2) is also regular expression which represents **union** operation.
3. If R1 and R2 are regular expressions then R = R1.R2 is also a regular expression which represents **concatenation** operation.
4. If R1 is a regular expression then R = R1* is also a regular expression which represents **kleen closure**.

A language denoted by regular expressions is said to be a **regular set or a regular language**. Let us see some examples of regular expressions.

Example 2.2.1 *Write a Regular Expression (R.E.) for a language containing the strings of length two over Σ = {0, 1}.*

Solution : R.E. = (0+1) (0+1)

Example 2.2.2 *Write a regular expression for a language containing strings which end with "abb" over Σ = {a, b}.*

Solution : R.E. = (a+b)*abb

Example 2.2.3 *Write a regular expression for a recognizing identifier.*

Solution : For denoting identifier we will consider a set of letters and digits because identifier is a combination of letters or letter and digits but having first character as letter always. Hence R.E. can be denoted as,

R.E. = letter (letter+digit)*

where Letter = (A, B,...Z, a, b,...z) and digit = (0, 1, 2, ...9).

Example 2.2.4 *Design the regular expression (r.e.) for the language accepting all combinations of a's except the null string over Σ = {a}.*

Solution : The regular expression has to built for the language

L = {a, aa, aaa, ...}

This set indicates that there is no null string. So we can write,

$R = a^+$

The + is called positive closure.

Example 2.2.5 *Design regular expression for the language containing all the strings with any number of a's and b's.*

Solution : The R.E. will be

$R.E. = (a + b)^*$

The set for this R.E will be -

L = {ε, a, aa, ab, b, ba, bab, abab, ...any combination of a and b}

The $(a + b)^*$ means any combination of a and b even a null string.

Example 2.2.6 *Construct a regular expression for the language containing all strings having any number of a's and b's except the null string.*

Solution : R.E. = $(a + b)^+$

This regular expression will give the set of strings of any combination of a's and b's except a null string.

Example 2.2.7 *Write a regular expression for a recognizing identifier.*

Construct the R.E. for the language accepting all the strings which are ending with 00 over the set Σ = {0, 1}.

Solution : The R.E. has to be formed in which at the end there should be 00. That means

R.E. = (Any combination of 0's and 1's) 00

$R.E. = (0 + 1)^* 00$

Thus the valid strings are 100, 0100, 1000... we have all strings ending with 00.

Example 2.2.8 *Write R.E. for the language accepting the strings which are starting with 1 and ending with 0, over the set Σ = {0,1}.*

Solution : The first symbol in R.E. should be 1 and the last symbol should be 0.

So, $R.E. = 1(0 + 1)^* 0$

Note that the condition is strictly followed by keeping starting and ending symbols correctly. In between them there can be any combination of 0 and 1 including null string.

Example 2.2.9 *Write a regular expression to denote the language L over Σ^*, where Σ = {a,b,c} in which every string will be such that any number of a's followed by any number of b's followed by any number of c's.*

Solution : Any number of a's means a^*. Similarly any number of b's and any number of c's means b^* and c^*. So the regular expression is = a^* b^* c^*.

Example 2.2.10 *Write R.E. to denote a language L over Σ^*, where Σ = {a, b} such that the 3rd character from right end of the string is always a.*

Solution :

r =	any number of a's and b's	a	either a or b	either a or b
		3rd	2nd	1st

∴ $R.E. = (a + b)^* a (a + b) (a + b)$.

Thus the valid strings are babb, baaa or abb, baba or aaab and so on.

Example 2.2.11 *Construct R.E. for the language which consists of exactly two b's over the set Σ = {a, b}.*

Solution : There should be exactly two b's.

any number of a's	b	any number of a's	b	any number of a's

Hence $R.E. = a^* ba^*b a^*$

a^* indicates either the string contains any number of a's or a null string. Thus we can derive any string having exactly two b's and any number of a's.

Example 2.2.12 *Write a regular definition for the language of all strings of 0's and 1's with an even number of 0's and odd number of 1's.* **GTU : Nov.-11, Marks 4**

Solution : The regular expression for even number of 0's is (00)*. The regular expression for odd number of 1's is 1(11)*.

The regular expression that contains even number of 0 and odd number of 1's is,

1(11)* + (00)*.

Example 2.2.13 *Find the regular expression corresponding to given statement, subset of* $\{0,1\}^*$

The language of all strings containing at least one 0 and at least one 1.

2. *The language of all strings containing 0's and 1's both are even.*
3. *The language of all strings containing atmost one pair of consecutive 1's.*
4. *The language of all strings that do not end with 01.* **GTU : May-12, Marks 4**

Solution : 1. R.E. = $(0+1)^+$

2. R.E. = (00 + 11)*
3. R.E. = 0* 110*
4. R.E. = 1* + 0* + (10)*

Example 2.2.14 *Write down the regular expression for the binary strings with even length.*

GTU : Winter-13, Marks 3

Solution : (aa+ab+ba+bb)*

Example 2.2.15 *Write a regular definition for :*

1. *The language of all strings that do not end with 01.*
2. *All strings of digit that contain no leading 0's.* **GTU : Winter-19, Marks 3**

Solution : (1) r.e. = (0+1)*(00+11+10)

(2) r.e. = $([1-9]\,[0-9]^*)^*$

Review Question

1. *What is regular expression, give all the algebraic properties of regular expression*

GTU : Summer-16, Marks 7

2.3 Recognition of Tokens

For a programming language there are various types of tokens such as identifier, keywords, constants and operators and so on. The token is usually represented by a pair token type and token value.

Token type	Token value

Fig. 2.3.1 Token representation

The token type tells us the category of token and token value gives us the information regarding token. The token value is also called **token attribute**. During lexical analysis process the **symbol table** is maintained. The token value can be a pointer to symbol table in case of identifier and constants. The lexical analyzer reads the input program and generates a symbol table for tokens.

For example :

We will consider some encoding of tokens as follows.

Token	Code	Value
if	1	–
else	2	–
while	3	–
for	4	–
identifier	5	Ptr to symbol table
constant	6	Ptr to symbol table
<	7	1
<=	7	2
>	7	3
>=	7	4
!=	7	5
(	8	1
)	8	2
+	9	1
–	9	2
=	10	–

Consider, a program code as

```
if(a<10)
        i=i+2;
else
        i=i–2;
```

Our lexical analyzer will generate following token stream.

1, (8,1), (5,100), (7,1), (6,105), (8,2), (5,107), 10, (5,107), (9,1), (6,110), 2,(5,107), 10, (5,107), (9,2), (6,110).

The corresponding symbol table for identifiers and constants will be,

Location counter	Type	Value
100	identifier	a
⋮	⋮	
105	constant	10
⋮	⋮	
107	identifier	i
⋮	⋮	
110	constant	2

In above example, scanner scans the input string and recognizes "if" as a keyword and returns token type as 1 since in given encoding code 1 indicates keyword "if" and hence 1 is at the beginning of token stream. Next is a pair (8,1) where 8 indicates parenthesis and 1 indicates opening parenthesis '('. Then we scan the input 'a' it recognizes it as identifier and searches the symbol table to check whether the same entry is present. If not it inserts the information about this identifier in symbol table and returns 100. If the same identifier or variable is already present in symbol table then lexical analyzer does not insert it into the table instead it returns the location where it is present.

2.4 Input Buffering

GTU : Nov.-11, May-12, Winter-13, 14, 16, 18, 19, 20, Summer-14, 18, Marks 7

The lexical analyzer scans the input string from left to right one character at a time. It uses two pointers begin_ptr **(bp)** and forward_ptr **(fp)** to keep track of the portion of the input scanned. Initially both the pointers point to the first character of the input string as shown below -

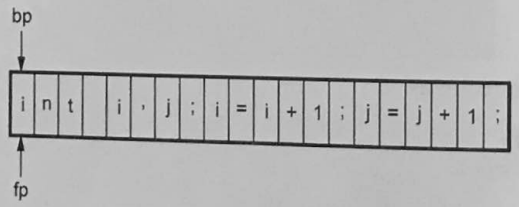


Fig. 2.4.1 Initial configuration

The forward_ptr moves ahead to search for end of lexeme. As soon as the blank space is encountered, it indicates end of lexeme. In above example as soon as forward_ptr (fp) encounters a blank space the lexeme "int" is identified.

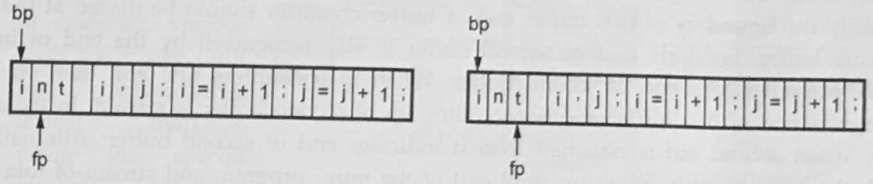


Fig. 2.4.2 Input buffering

The fp will be moved ahead at white space. When fp encounters white space, it ignore and moves ahead. Then both the begin_ptr (bp) and forward_ptr (fp) are set at next token i.

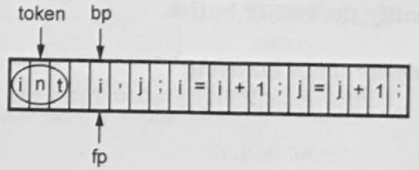


Fig. 2.4.3 Input buffering

The input character is thus read from secondary storage. But reading in this way from secondary storage is costly. Hence buffering technique is used. A block of data is first read into a buffer, and then scanned by lexical analyzer. There are two methods used in this context : one buffer scheme and two buffer scheme.

1. One buffer scheme

In this one buffer scheme, only one buffer is used to store the input string. But the problem with this scheme is that if lexeme is very long then it crosses the buffer boundary, to scan rest of the lexeme the buffer has to be refilled, that makes overwriting the first part of lexeme.

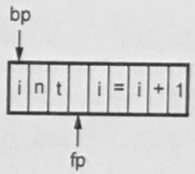


Fig. 2.4.4 One bufffer scheme storing input string

2. Two buffer scheme

To overcome the problem of one buffer scheme, in this method two buffers are used to store the input string. The first buffer and second buffer are scanned alternately. When end of current buffer is reached the other buffer is filled. The only problem with this method is that if length of the lexeme is longer than length of the buffer then scanning input cannot be scanned completely.

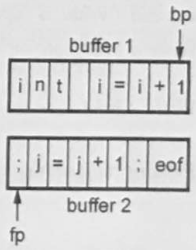


Fig. 2.4.5 Two buffer scheme storing input string

Initially both the bp and fp are pointing to the first character of first buffer. Then the fp moves towards right in search of end of lexeme. As soon as blank character is recognized, the string between bp and fp is identified as corresponding token. To identify the **boundary** of first buffer **end of buffer** character should be placed at the end of first buffer. Similarly end of second buffer is also recognized by the end of buffer mark present at the end of second buffer. When fp encounters first **eof**, then one can recognize end of first buffer and hence filling up of second buffer is started. In the same way when second **eof** is obtained then it indicates end of second buffer. Alternatively both the buffers can be filled up until end of the input program and stream of tokens is identified. This **eof** character introduced at the end is called **sentinel which is used to identify the end of buffer.**

Code for input buffering

```
if(fp==eof(buff1)) /*encounters end of first buffer*/
{
 /*Refill buffer2*/
 fp++;
}
else if (fp==eof(buff2) /*encounters end of second buffer*/
{
 /*Refill buffer1*/
  fp++;
}

else if(fp==eof(input))
return; /*terminate scanning*/
else
        fp++;
              /* still remaining input has to scanned */
```

Review Questions

1. *Write the two methods used in lexical analyzer for buffering the input. Which technique is used for speeding up the lexical analyzer.* **GTU : Nov.-11, Marks 7**
2. *Explain : What is input buffering ?* **GTU : May-12, Marks 3**
3. *Write a brief note on input buffering techniques to lexical analyzer.* **GTU : Winter-13, 16, Marks 7**
4. *Write a brief note on input buffering techniques.* **GTU : Summer-14, 18, Winter-18, 19, 20, Marks 7**
5. *Explain buffer pairs and sentinels.* **GTU : Winter-14, Marks 7**

2.5 Elementary Scanner Design and its Implementation (Lex)

GTU : Winter-20, Marks 4

- For efficient design of compiler, various tools have been built for constructing lexical analyzers using the special purpose notations called **regular expressions.**
- Basically LEX is a unix utility which generates the lexical analyzer.

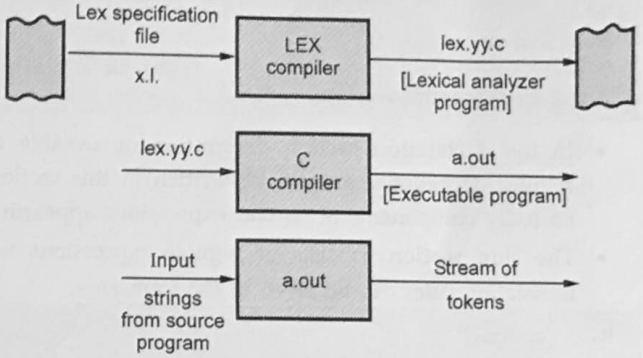


Fig. 2.5.1 Generation of lexical analyzer using LEX

- A LEX lexer is very much faster in finding the tokens as compared to the handwritten LEX program in C.
- LEX scans the source program in order to get the stream of tokens and these tokens are related together so that various programming constructs such as expressions, block statements, procedures, control structures can be realised.
- The LEX specification file can be created using the extension .l(often pronounced as **dot L**). For example, the specification file can be **x.l** .
- This **x.l** file is then given to LEX compiler to produce **lex.yy.c**.
- This **lex.yy.c.** is a C program which is actually a **lexical analyzer program.** The LEX specification file stores the regular expressions for the tokens and the **lex.yy.c.** file consists of the **tabular representation** of the transition diagrams constructed for the regular expression. The lexemes can be recognized with the help of this tabular representation of transition diagram.
- Finally the compiler compiles this generated **lex.yy.c** and produces an object program **a.out.** When some input stream is given to a.out then sequence of tokens get generated. The above described scenario can be modelled below.

2.5.1 Structure of LEX

Now the question arises how do we write the specification file? Well, the LEX program consists of three parts -

1. Declaration section 2. Rule section and 3. Procedure section.

```
%{
Declaration section
%}
%%
Rule section
%%
Auxiliary procedure section
```

- In the **declaration section** declaration of variable constants can be done. Some regular definitions can also be written in this section. The regular definitions are basically components of regular expressions appearing in the rule section.
- The **rule section** consists of regular expressions with associated actions. These translation rules can be given in the form as -

```
R1   {action1}
R2   {action2}
.
.
.
Rn   {actionn}
```

Where each R_i is a regular expression and each **action**$_i$ is a program fragment describing what action is to be taken for corresponding regular expression. These actions can be specified by piece of C code.

- And third section is a **auxiliary procedure** section in which all the required procedures are defined. Sometimes these procedures are required by the actions in the rule section.
- The lexical analyzer or scanner works in co-ordination with parser. When activated by the parser, the lexical analyzer begins reading its remaining input, character by character at a time. When the string is matched with one of the regular expressions R_i then corresponding action$_i$ will get executed and this action$_i$ returns the control to the parser.
- The repeated search for the lexeme can be made in order to return all the tokens in the source string. The lexical analyzer ignores white spaces and comments in this process. Let us learn some LEX programming with the help of some examples -


```
%{
%}
%%
      "Rama" |
      "Seeta" |
      "Geeta" |
      "Neeta" |                  printf("\n Noun");
      "Sings" |
      "dances" |
      "eats"              printf ("\n Verb");
%%
      main ()
      {
            yylex();
      }
      int yywrap()
      {
            return 1;
      }
```

This is a simple program that recognizes noun and verb from the string clearly. There are 3 sections in above program.

- The section starting and ending with %, { and %} respectively is a **definition section**.
- The section starting with %% is called **rule section**. This section is closed by %%. Within %% consists of regular expressions and actions. Rule 1 gives the definition of noun and second rule gives the definition of verb.
- The third section consists of two functions the **main function** and the **yywrap function**. In main function call to **yylex** routine is given. This function is defined in **lex.yy.c** program. First we will compile our above program (x.*l*) using lex compiler and then the lex compiler will generate a ready C program named **lex.yy.c**. This **lex.yy.c** makes use of regular expression and corresponding actions defined in **x.*l***. Hence our above program **x.*l*** is called lex specification file.
- When we compile **lex.yy.c** file using command cc, here cc means compile C. We get an output file named **a.out**. (This is a default output file on LINUX platform). On execution of **a.out** we can give the input string.
- Following commands are used to run the lex program x.*l*.

```
$    lex x.l        ------>  This command generates lex.yy.c
$    cc lex.yy.c    ------>  This command compiles lex.yy.c (sometimes
                             gcc can be used)
$    .\a.out        ------>  This command runs an executable file
```

After entering these commands a blank space for entering input gets available. There we can give some valid input.

```
Rama eats
Noun
verb
Seeta sings
Noun
verb
```

Then press either **control + c** or **control + d** to come out of the output.

Notations used in Regular Expressions of LEX

Regular expression	Meaning
*	Matches with zero or more occurrences of preceeding expression. For example, 1* occurrence of 1 for any number of times.
.	Matches any single character other than new line character.
[ ]	A character class which matches any character within the bracket. For example, [a-z] matches with any alphabet in lower case.
()	Group of regular expressions together put into a new regular expression.
" "	The string written in quotes matches literally. For example, "Hanumaan" matches with the string Hanumaan.
$	Matches with the end of line as last character.
+	Matches with one or more occurrences of preceeding expression. For example, [0-9]+ any number but not empty string.
?	Matches zero or one occurrence of preceding regular expression. For example, [+-]?[0-9]+ a number with unary operator.
^	Matches the beginning of a line as first character.
[^]	Used as for negation. For example, [^verb] means except verb match with anything else.
\	Used as escape metacharacter. For example, \n is a newline character. \# prints the # literally
|	To represent the or i.e. another alternative. For example, a | b means match with either a or b.

Built-in Variables

yyin	Of the type FILE*. This point to the current file being parsed by the lexer. It is standard input file that stores input source program.
yyout	Of the type FILE*. This point to the location where the output of the lexer will be written. By default, both yyin and yyout point to standard input and output.
yytext	The text of the matched pattern is stored in this variable (char*) i.e. when lexer matches or recognizes the token from input token the lexeme stored in null terminated string called yytext. Thus current token is returned by this variable.

yyleng	Gives the length of the matched pattern. The value in yyleng is same as strlen() functions.
yylineno	Provides current line number information.
yylval	This is a global variable used to store the value of any token.

Built-in Functions

yylex()	This is a starting point of lex from which scanning of source program starts.
yywrap()	This function is called when end of file is encountered. If yywrap returns 0 the scanner countinues scanning if it returns 1 the scanner does not return tokens.
yylesss(int n)	This function can be used to push back all first n characters of the token being read.
yymore()	This function tells lexer to attach next tokens to current token.
yyerror()	For displaying error messages, this function is used.

2.5.2 LEX Programs

LEX is a scanner program which scans the input. We can further analyse the input by counting the number of words, number of characters and total number of lines appearing within it. Here is a simple program using LEX which counts the total number of words, number of lines and number of characters appearing in the given input.

LEX Program

```
%{
int Char_Cnt=0,Word_Cnt=0,Line_Cnt=0;
%}
word [^ \t\n]+
%%
{word} {Word_Cnt++;Char_Cnt+=yyleng;}
\n {Char_Cnt++;Line_Cnt++;}
. {Char_Cnt++;}
%%
main()
{
yylex();
printf("\nThe character count= %d",Char_Cnt);
printf("\nThe word count= %d",Word_Cnt);
printf("\nThe line count=%d",Line_Cnt);
printf("\n");
}
int yywrap()
{
return 1;
}
```

Anything other than space, tab or newline is considered as word

Output

```
[root@aap root]# lex linecount.l
[root@aap root]# gcc lex.yy.c –ll
[root@aap root]# ./a.out
hello Bhakti!!
You are
a newcomer
here
The character count= 41
The word count= 7
The line count=4
[root@aap root]#
```

LEX Program Using Command Line Argument

The command line parameters are the parameters that are appearing on the shell prompt.

The command line interface is the interface which allows the user to interact with the computer by typing the commands.

In C we can pass these parameters to the main function in the form of character array.

For example :

```
$ cp abc.txt pqr.txt
```

Here

```
argv[0]=cp
argv[1]=abc.txt
argv[2]=pqr.txt
```

As there are three such parameters that are present at the command line interface **argc** value will be 3.

Let us now discuss how to handle the command line parameters with using LEX specification file.

Example 2.5.1 *Write a lex program to check well formedness of the parenthesis. Make use of commandline arguments.*

Solution :

LEX Program :

```
%{
/*Well formedness of brackets*/
/*input is b.c*/
#include<stdio.h>
#include<string.h>
char temp[10];
int i=0,openbracket=0,closebracket=0;
extern FILE *yyin;
%}
```



```
%%
"("[()]*")"";" {
strcpy(temp,yytext);
printf("%s\n",temp);
i=0;
while(temp[i]!=';')
{
if(temp[i]=='(')
openbracket++;
if(temp[i]==')')
closebracket++;
else ;
i++;
}

if(openbracket==closebracket)
printf("Well formed input!\n");
else
printf("not well formed!\n");
}

%%

main(int argc,char *argv[])
{
FILE *fp=fopen(argv[1],"r");
yyin=fp;
yylex();
fclose(fp);
}
```

Output (run 1)

```
[root@aap Syssoftprograms]# vi well.l
[root@aap Syssoftprograms]# lex well.l
[root@aap Syssoftprograms]# gcc lex.yy.c –ll
[root@aap Syssoftprograms]# ./a.out b.c
(()());
Well formed input!
```

Output (run 2)

```
[root@aap Syssoftprograms]# vi b.c
[root@aap Syssoftprograms]# lex well.l
[root@aap Syssoftprograms]# gcc lex.yy.c –ll
[root@aap Syssoftprograms]# ./a.out b.c

()());
not well formed!
```

Program explanation : In above program, we have used a file b.c which is our input file. While running the LEX program we have given the input to the program via the input file "b.c" using commandline parameter. Note that argv[0] ="./a.out" and in argv[1]="b.c". In the **main()** function we have opened the file "b.c." in read mode by

```
fp=fopen(argv[1],"r");
```

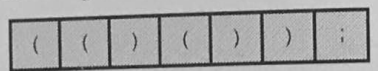
The file pointer fp of opened file "b.c" will then be assinged to the standard input file pointer yyin. Just do not forget to declare yyin as,

```
extern FILE *yyin;
```

(we did that in the declaration section!!!)

Then the yylex() routine is invoked to call LEX.

The logic of the above program is very simple. We have maintained one temp array in which the input expression is copied. For example -



temp [ ]

Then we have maintained two counters 'openbracket' and 'closebracket'. As soon as we come across the openbracket the 'openbracket' counter will be incremented and as soon as we come across the closing bracket, the 'closebracket' counter will be incremented. The regular expression specifies that opening bracket should occur prior to closing bracket. This whole logich should we work out untill we come across semicolon. If both the openbracket and closebracket counters are same then declare "well formed input!" otherwise declare "not well formed!".

Example 2.5.2 *Write a LEX program to recognize valid operators of C program*

Solution :

```
%{
      /* Program Name: valid_operators.l*/
%}
identifier [a-zA-Z][a-zA-Z0-9]*
delim                [ \t\n]
white_sp   {delim}+
letter     [A-Za-z]
digit      [0-9]
identifier {letter}({letter}|{digit})*
%%
{white_sp}
if                                {/* do nothing, simply ignore white space*}
then                              {printf("\n\t%s is a KEYWORD",yytext);}
else                              {printf("\n\t%s is a KEYWORD",yytext);}
{identifier}                      {printf("\n\t%s is a KEYWORD",yytext);}
"<"                               {printf("\n\t%s is an IDENTIFIER",yytext);}
OPERATOR",yytext);}               {printf("\n\t%s is a LESS THAN
```



```
">"                               {printf("\n\t%s is a GREATER THAN
OPERATOR",yytext);}
"<="                              {printf("\n\t%s is a LESS THAN EQUAL TO
                                  OPERATOR",yytext);}
">="                              {printf("\n\t%s is a GREATER THAN EQUAL
TO
                                  OPERATOR",yytext);}
"=="                              {printf("\n\t%s is a  EQUAL TO
OPERATOR",yytext);}
"!="                              {printf("\n\t%s is a  NOT EQUAL TO
                                        OPERATOR",yytext);}
\".*\"                            {printf("\n\t%s is a STRING",yytext);}

%%

int main(int argc, char **argv)
{
      if(argc > 1)
      {
            FILE *file;
            file = fopen(argv[1],"r");
            if(!file)
            {
                  printf("Could not open %s\n",argv[1]);
                  exit(0);
            }
            yyin = file;
      }
      yylex();
      printf("\n\n");
      return 0;
}
```

```
/*************************************************************************
Following is a C program which contains fragment of code. This file is taken as an input
by the above LEX program and then the output is produced
Program Name: test.c
*************************************************************************/
if(a<2) then
  printf("The number is 2");
else
  printf("The number is something else");
```

Output

```
[root@localhost ]#lex valid_operators.l
[root@localhost ]#cc lex.yy.c
[root@localhost ]#./a.out test.c
```

```
if is a KEYWORD
( is a STRING
a is an IDENTIFIER
> is a GREATER THAN OPERATOR
) is a STRING
then is a KEYWORD
printf("The number is 2")  is a STRING
else is a KEYWORD
printf("The number is something else") is a STRING
```

Example 2.5.3 *Write a lex program to identify comments in the program.*

Solution :

```
/*************************************************************
Program to identify and count the numbers of comment lines.
*************************************************************/
%{
int c=0,state=1;
%}

%%
"//".* {c++;printf("\n This is a single line comment statement");}
"/*" { state=0;printf("\n Comment statement begins");}
"*/" { c++;
      if (!state)
         state=1;
       printf("\n Comment statement ends");
      }
%%
main(int argc,char ** argv)
{
  yylex();
  printf("\nNumber of comment lines : %d",c);
}

yywrap()
{
 if(state==0)
 {
   printf("\nUnterminated comment");
   return 1;
 }
}
```

Review Question

1. *Write a short note on LEX tool.* **GTU : Winter-20, Marks 4**

2.6 Applying Concepts of Finite Automata for Recognition of Tokens

There is a close relationship between a finite automata and the regular expression. We can show this relation int Fig. 2.6.1.

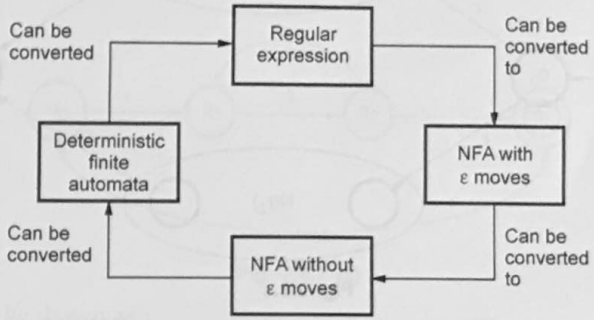


Fig. 2.6.1 Relationship between FA and regular expression

The Fig. 2.6.1 shows that it is convenient to convert the regular expression to NFA with ε moves. Le us see the theorem based on this conversion.

2.6.1 Construction of a NFA from Regular Expression (Thompson's Construction)

To construct NFA from regular expression r the Thompson's construction is used. The input string of r is parsed and following constructs are used to build an equivalent NFA.

1. For r = ε

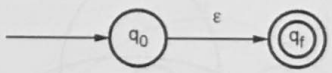


Fig. 2.6.2

2. When r = a for $\sum = \{a\}$ the NFA is

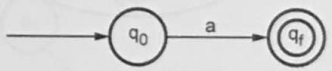


Fig. 2.6.3

3. When $r = r_1 + r_2$ then NFA can be drawn as -

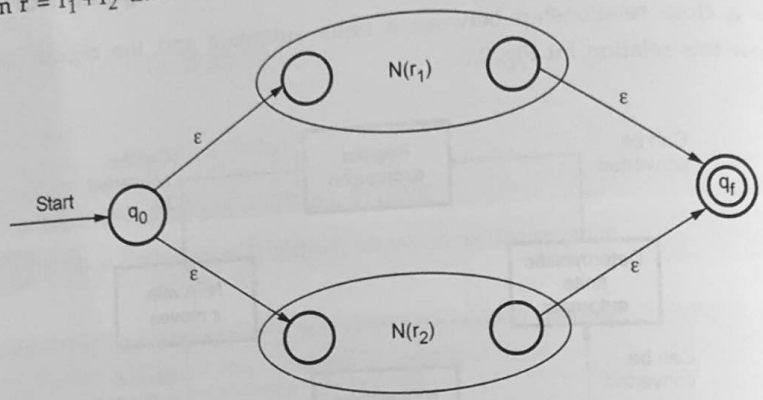


Fig. 2.6.4

Here $N(r_1)$ represents NFA for regular expression r_1 and $N(r_2)$ represents NFA for regular expression r_2.

4. When $r = r_1 \cdot r_2$ then NFA can be drawn as -

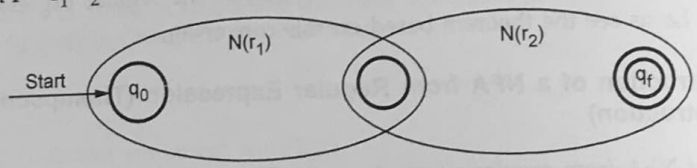


Fig. 2.6.5

5. When $r = (r_1)^*$ then the NFA can be as drawn below -

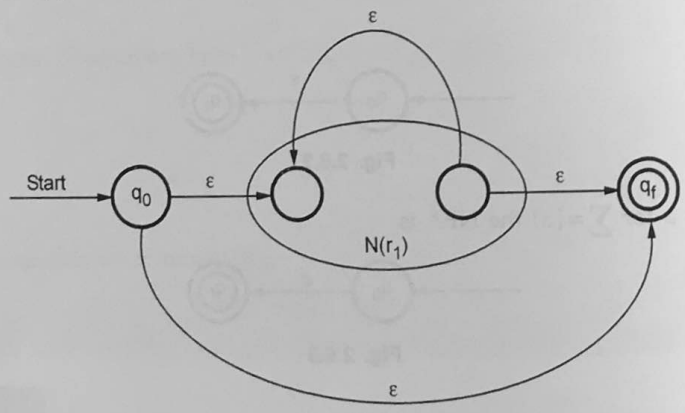


Fig. 2.6.6

Thus these constructs are useful for creating NFA equivalent to regular expression.

Example 2.6.1 *Construct NFA equivalent to r = a* b.*

Solution : We will parse the regular expression r and construct NFA.

Let $r = r_1, r_2 = a^* b$. We can say $r_1 = a^*$ and $r_2 = b$. Let us draw NFA for $r_1 = a^*$ as -

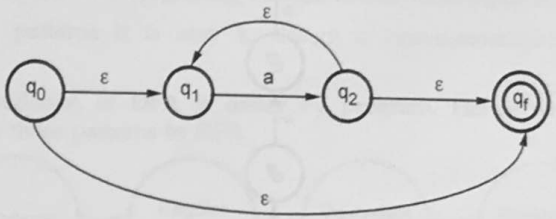


Fig. 2.6.7 (a)

The r = a* b can be drawn as :

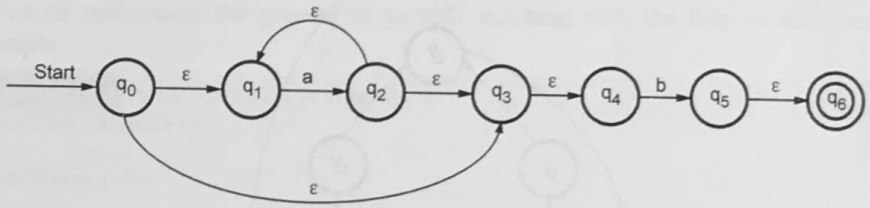


Fig. 2.6.7 (b)

Example 2.6.2 *Build NFA for the regular expression r = (a + b)* ab.*

Solution : As r = (a + b)* ab, we will build NFA for (a + b)*.

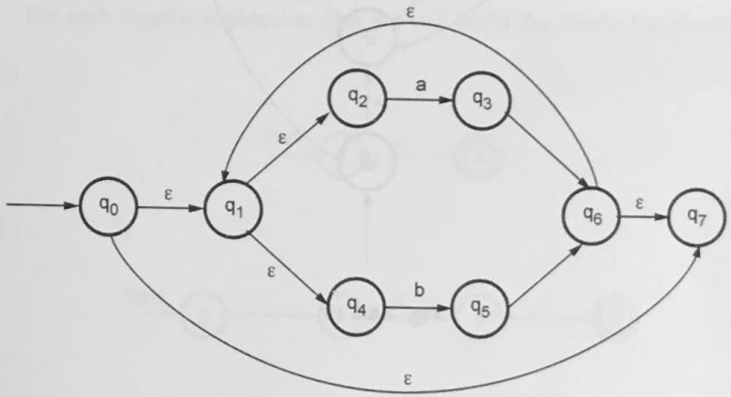


Fig. 2.6.8 (a)

- Then we will build r = (a + b)*ab. (Refer Fig. 2.6.8 (b))

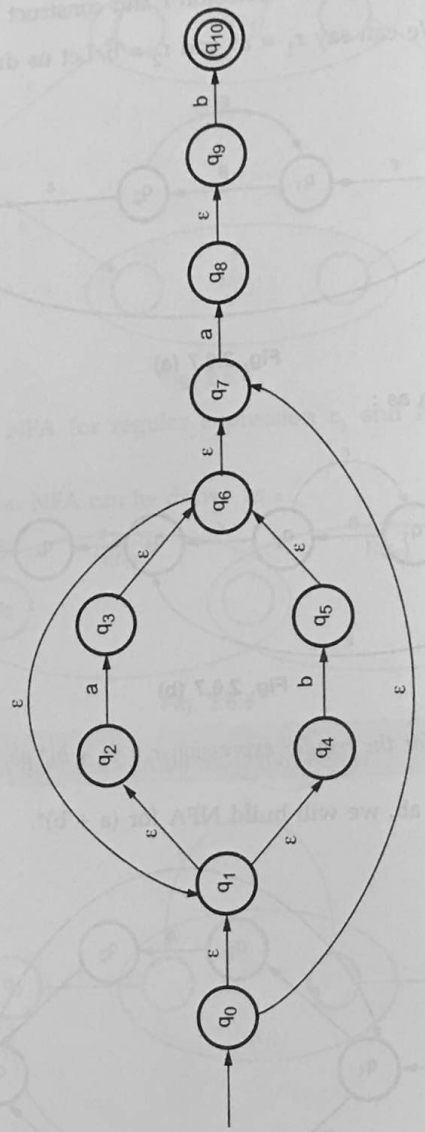


Fig. 2.6.8 (b)

2.7 Design of Lexical Analyzer Generator

GTU : Summer-16, 18, 19, Marks 7

- To design lexical analyzer generator, the **pattern** of **regular expressions** are designed first.
- These patterns are for recognizing various tokens form input string.
- From these patterns it is easy to design a Non-deterministic Finite Automata (NFA).
- But the simulation of DFA is easier by program. Hence we convert the NFA drawn from these patterns to DFA.

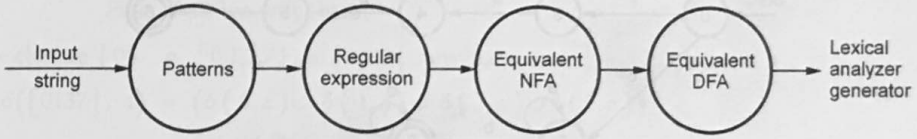


Fig. 2.7.1 Building of lexical analyzer generator

Let us understand the process of pattern matching with the help of some suitable example.

Example 2.7.1 *A LEX program is given below -*

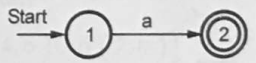
Auxiliary Definitions
(none)
Translation rules
a
abb
a^*b^+

Implement the LEX Program as DFA.

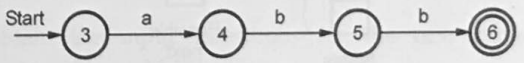
Solution : We will implement the given LEX program as DFA in following steps -

Step 1 : For each regular expression first we will build the Finite Automata (FA).

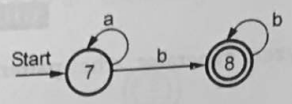
Pattern 1 :



Pattern 2 :



Pattern 3 :



Step 2 : Now we will combine all these patterns by designing NFA with ε .

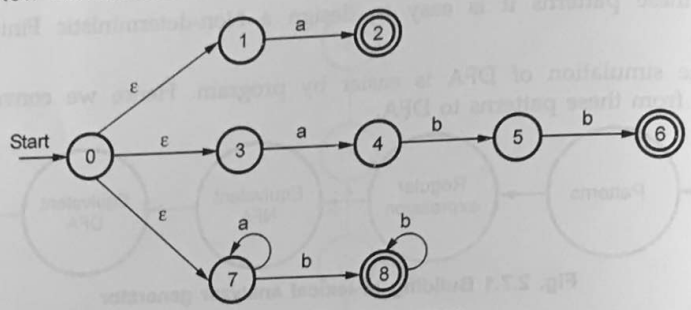
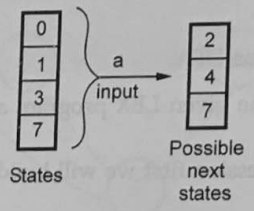


Fig. 2.7.2 Combined NFA

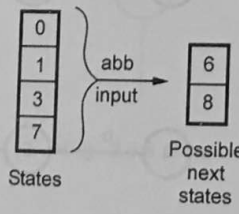
Step 3 : The above drawn combined NFA can recognize the pattern in the given LEX Program.

For instance

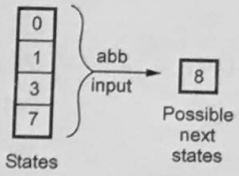
String **a** matches with **pattern 1 .**



String **abb** matches with **pattern 2 .**



String **aab** matches with **pattern 3 .**



Step 4 : Now we will convert the combined NFA to DFA.

First of all we will obtain

ε- closure $\{0\} = \{0,1,3,7\} = [0137]$ **new state.**

$\delta([0137], a) = (\delta(0,a) \cup \delta(1,a) \cup \delta(3,a) \cup \delta(7,a))$

$= (\phi \cup 2 \cup 4 \cap 7)$

$= (2,4,7)$

$= (2,4,7) \rightarrow$ **new state.**

$\delta([0137], b) = (\delta(0,b) \cup \delta(1,b) \cup \delta(3,b) \cup \delta(7,b))$

$= (\phi \cup \phi \cup \phi \cup 8)$

$= [8] \rightarrow$ **new state**

The pattern announced for [0137] will be **none** because 0, 1, 3 or 7 all these states are non-final states.

As in above computation two new states i.e.[2, 4 , 7] and [8] are formed, we will find input transitions for these states.

$\delta([247], a) = (\delta(2,a) \cup \delta(4,a) \cup \delta(7,a))$

$= \phi \cup \phi \cup 7$

$= [7] \rightarrow$ **new state**

$\delta([247], b) = (\delta(2,b) \cup \delta(4,b) \cup \delta(7,b))$

$= (\phi \cup 5 \cup 8)$

$= [58] \rightarrow$ **new state.**

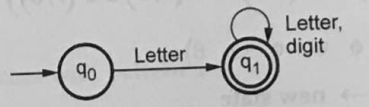
Thus we will obtain the input transitions for the new states that are getting generated. Finally the transition table can be -

State	Input		Pattern recognised
	a	b	
[0137]	[247]	[8]	None
[247]	[7]	[58]	a
[8]	-	[8]	a*b+
[7]	[7]	[8]	None
[58]	-	[68]	a*b+
[68]	-	[8]	abb

2.7.1 Transition Diagrams for Programming Constructs

Example 2.7.2 *Write a regular expression for identifier and keyword and design a transition diagram for it.*

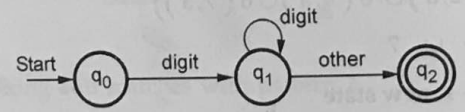
Solution : r.e = letter (letter + digit)*



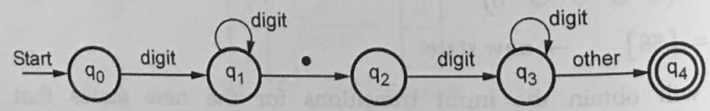
Example 2.7.3 *Write a regular expression for defining constants (unsigned numbers). Design the transition graphs for them.*

Solution : The constant can be defined by three different ways -

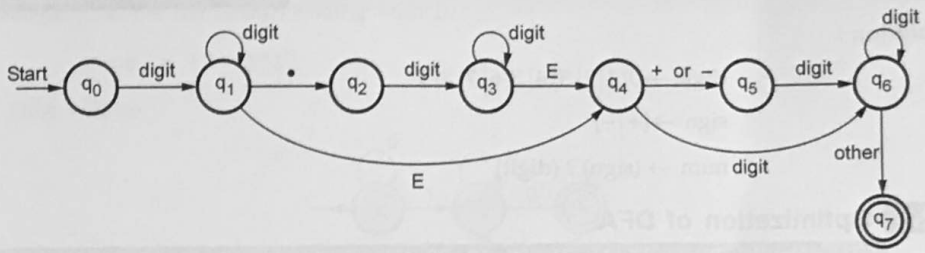
i) r.e = digit$^+$



ii) r.e = $(\text{digit})^+ \bullet (\text{digit})^+$

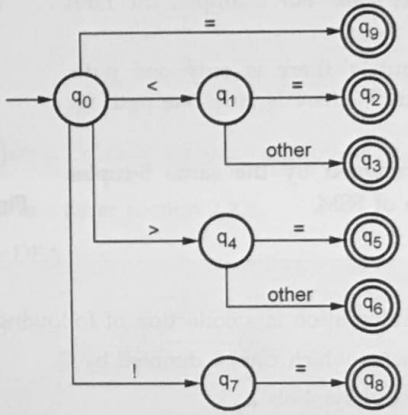


iii) r.e = digit$^+$ (•digit$^+$)?(E(+ | −)?digit$^+$)?



Example 2.7.4 *Write a regular expression for relation operators. Design a transition diagram for them.* **GTU : Summer-18, Marks 4**

Solution : r.e = (< | > |< =|> =|=| ! =)



During lexical analysis, the LEX program uses the regular expression and the generated lexical analyser writes the procedure or a separate function for each state of DFA. The symbols along the edges represent the input parameters for these functions. At the final state of corresponding DFA the corresponding token can be identified.

Example 2.7.5 *Unsigned numbers are strings such as 5280, 39.37, 6.336E4 or 1.894E-4, give the regular definitions for the above mentioned strings.* **GTU : Summer-16, Marks 7**

Solution : Refer example 2.7.3.

Example 2.7.6 *Draw the state transition diagram for the unsigned numbers* **GTU : Summer-16, Marks 7**

Solution : Refer example 2.7.3.

Example 2.7.7 *Give regular definition for signed and unsigned numbers.*

GTU : Summer-19, Marks 3

Solution :

digit $\rightarrow 0|1|2|3|4|5|6|7|8|9$

sign $\rightarrow [+|-]$

num $\rightarrow$ (sign) ? (digit)*

2.8 Optimization of DFA

GTU : May-12, Winter-13, 15, 16, 18, 19, 20, Summer-15, 16, 17, 18, 19, 20, Marks 7

2.8.1 Deterministic Finite Automata (DFA)

The finite automata is called deterministic finite automata if there is only one path for a specific input from current state to next state. For example, the DFA can be shown as below.

From state S_0 for input '*a*' there is only one path, going to S_2. Similarly from S_0 there is only one path for input b going to S_1.

The DFA can be represented by the same 5-tuples described in the definition of FSM.

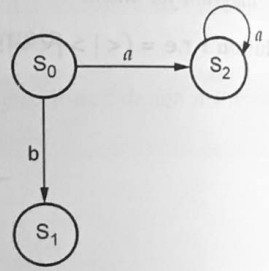


Fig. 2.8.1 Deterministic finite automata

Definition of DFA

A deterministic finite automation is a collection of following things -

1) The **finite set of states** which can be denoted by Q.
2) The **finite set of input** symbols Σ.
3) The **start state** q_0 such that $q_0 \in Q$.
4) A set of **final states** F such that $F \in Q$.
5) The **mapping function** or transition function denoted by δ. Two parameters are passed to this transition function : One is current state and other is input symbol. The transition function returns a state which can be called as next state.

For example, $q_1 = \delta(q_0, a)$ means from current state q_0, with input *a* the next state transition is q_1.

In short, the DFA is a five tuple notation denoted as :

$$A = (Q, \Sigma, \delta, q_0, F)$$

The name of DFA is A which is a collection of above described five elements.

Example 2.8.1 *Draw deterministic finite automata for the binary strings endings with 10.*

GTU : Winter-13, Marks 4

Solution : r.e. for the strings ending with 10.

$$\text{r.e.} = (0+1)^*10.$$

DFA will be

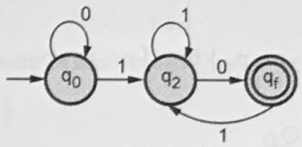


Fig. 2.8.2

Example 2.8.2 *What are regular expressions. Find the regular expression described by DFA = {{A, B}, {0, 1}, δ, A, {B}}, where δ is detailed in following table*

GTU : Summer-15, Marks 7

	0	1
A	A	B
B	ϕ	A

Please note B is accepting state. Describe the language defined by regular expression.

Solution : Regular expression : Refer section 2.2.2.

The transition graph for DFA is -

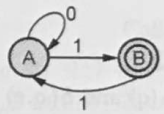


Fig. 2.8.3

$$\text{r.e.} = [0^*1(11)^*]^*$$

L = {The words that contain any number of zero's but always end with odd number of one's}

2.8.2 NFA to DFA Conversion

There are three methods to convert NFA to DFA.

1. Using subset construction method
2. Direct method
3. Using DFA tree method

Let us understand how to Convert NFA to DFA using subset construction method.

1. Method for converting NFA with ε to DFA

Step 1 : Consider $M = (Q, \Sigma, \delta, q_0, F)$ is a NFA with ε. We have to convert this NFA with ε to equivalent DFA denoted by

$$M_D = (Q_D, \Sigma, \delta_D, q_0, F_D)$$

Then obtain,

ε - closure(q_0) = $\{p_1, p_2, p_3, \cdots p_n\}$ then $[p_1, p_2, p_3, \cdots p_n]$ becomes a start state of DFA.

Now $[p_1, p_2, p_3, \cdots p_n] \in Q_D$

Step 2 : We will obtain δ transitions on $[p_1, p_2, p_3, \cdots p_n]$ for each input.

$$\delta_D([p_1, p_2, \cdots p_n], a) = \varepsilon \text{ - closure}(\delta(p_1, a) \cup \delta(p_2, a) \cup \ldots \delta(p_n, a))$$

$$= \bigcup_{i=1}^{n} \varepsilon \text{ - closure}\, \delta(p_i, a)$$

where a is input $\in \Sigma$.

Step 3 : The states obtained $[p_1, p_2, p_3, \cdots p_n] \in Q_D$. The states containing final state in p_i is a final state in DFA.

Definition of ε closure

The ε - closure (p) is a set of all states which are reachable from state p on ε -transitions such that :

i) ε - closure (p) = p where $p \in Q$.

ii) If there exists ε - closure (p) = {q} and $\delta(q, \varepsilon) = r$ then ε - closure (p) = {q, r}.

Example 2.8.3 *Find ε - closure for the following NFA with ε .*

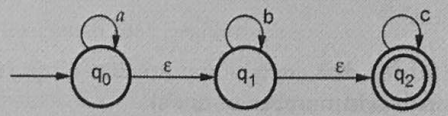


Fig. 2.8.4

Solution :

ε - closure $(q_0) = \{q_0, q_1, q_2\}$ means **self state + ε - reachable states.**

ε - closure $(q_1) = \{q_1, q_2\}$ means q_1 is a self state and q_2 is a state obtained from q_1 with ε input.

ε - closure $(q_2) = \{q_2\}$

Example 2.8.4 *Convert the given NFA into its equivalent DFA -*

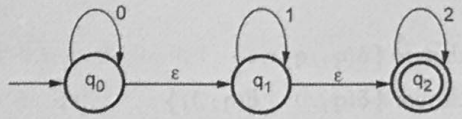


Fig. 2.8.5

Solution : Let us obtain ε - closure of each state.

ε - closure $(q_0) = \{q_0, q_1, q_2\}$

ε - closure $(q_1) = \{q_1, q_2\}$

ε - closure $(q_2) = \{q_2\}$

Now we will obtain δ′ transition. Let ε - closure $(q_0) = \{q_0, q_1, q_2\}$ call it as **state A.**

$\delta'(A, 0)$ = ε - closure $\{\delta((q_0, q_1, q_2), 0)\}$

= ε - closure $\{\delta(q_0, 0) \cup \delta(q_1, 0) \cup \delta(q_2, 0)\}$

= ε - closure $\{q_0\}$

= $\{q_0, q_1, q_2\}$ i.e. **state A**

$\delta'(A, 1)$ = ε - closure $\{\delta((q_0, q_1, q_2), 1)\}$

= ε - closure $\{\delta(q_0, 1) \cup \delta(q_1, 1) \cup \delta(q_2, 1)\}$

= ε - closure $\{q_1\}$

= $\{q_1, q_2\}$ Call it as **state B.**

$\delta'(A, 2)$ = ε - closure $\{\delta((q_0, q_1, q_2), 2)\}$

= ε - closure $\{\delta(q_0, 2) \cup \delta(q_1, 2) \cup \delta(q_2, 2)\}$

= ε - closure $\{q_2\}$

= $\{q_2\}$ Call it as **state C.**

Thus we have obtained

$\delta'(A, 0)$ = A

$\delta'(A, 1)$ = B

$\delta'(A, 2)$ = C

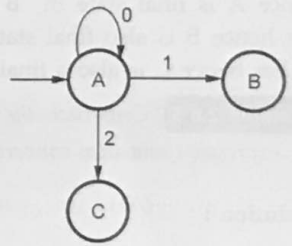


Fig. 2.8.6

Now we will find transitions on states B and C for each input.

Hence $\delta'(B, 0)$ = ε - closure $\{\delta(q_1, q_2), 0\}$

= ε - closure $\{\delta(q_1, 0) \cup \delta(q_2, 0)\}$

$= \varepsilon$ - closure $\{\phi\}$

$= \phi$

$\delta'(B,1) = \varepsilon$ - closure $\{\delta(q_1,q_2),1\}$

$= \varepsilon$ - closure $\{\delta(q_1,1) \cup \delta(q_2,1)\}$

$= \varepsilon$ - closure $\{q_1\}$

$= \{q_1,q_2\}$ i.e. **state B itself.**

$\delta'(B,2) = \varepsilon$ - closure $\{\delta(q_1,q_2),2\}$

$= \varepsilon$ - closure $\{\delta(q_1,2) \cup \delta(q_2,2)\}$

$= \varepsilon$ - closure $\{q_2\}$

$= \{q_2\}$ i.e. **state C.**

Hence $\delta'(B,0) = \phi$

$\delta'(B,1) = B$

$\delta'(B,2) = C$

The partial transition diagram will be in Fig. 2.9.7.

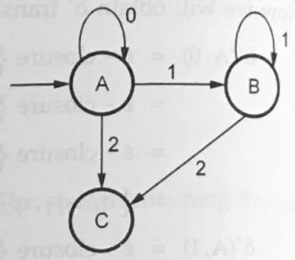


Fig. 2.8.7

Now we will obtain transitions for C :

$\delta'(C,0) = \varepsilon$ - closure $\{\delta(q_2,0)\}$

$= \varepsilon$ - closure $\{\phi\} = \phi$

$\delta'(C,1) = \varepsilon$ - closure $\{\delta(q_2,1)\}$

$= \varepsilon$ - closure $\{\phi\} = \phi$

$\delta'(C,2) = \varepsilon$ - closure $\{\delta(q_2,2)\} = q_2$

Hence the DFA is as shown in Fig. 2.9.8.

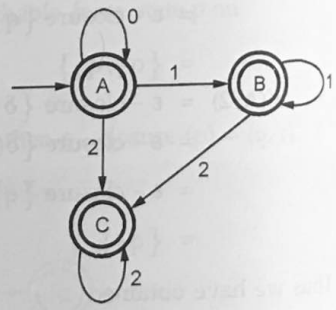


Fig. 2.8.8

As $A = \{q_0,q_1,q_2\}$ in which final state q_2 lies hence A is final state in $B = \{q_1,q_2\}$ the state q_2 lies hence B is also final state in $C = \{q_2\}$, the state q_2 lies hence C is also a final state.

Example 2.8.5 *Construct the NFA using thompson's notation for the following regular expression and then convert it to DFA* $a^+(c|d)b^*f\#$

GTU : Summer-15, Marks 7

Solution :

ε - closure $(q_0) = \{q_0, q_1\}$

ε - closure $(q_1) = \{q_1\}$

ε - closure $(q_2) = \{q_2,q_3,q_5,q_6,q_7\}$

ε - closure $(q_3) = \{q_3\}$

ε - closure $(q_4) = \{q_4,q_3,q_5,q_6,q_6,q_7\}$

ε - closure $(q_5) = \{q_5,q_6,q_7\}$

ε - closure $(q_6) = \{q_6\}$

ε - closure $(q_7) = \{q_7\}$

ε - closure $(q_8) = \{q_8,q_{10},q_{11},q_{13}\}$

ε - closure $(q_9) = \{q_9,q_{10},q_{11},q_{13}\}$

ε - closure$(q_{10}) = \{q_{10},q_{11},q_{13}\}$

ε - closure $(q_{11}) = \{q_{11}\}$

ε - closure $(q_{12}) = \{q_{11},q_{12},q_{13}\}$

ε -closure $(q_{13}) = \{q_{13}\}$

ε -closure $(q_{14}) = \{q_{14}\}$

ε -closure $(q_{15}) = \{q_{15}\}$

Let, $\varepsilon - \text{closure}(q_{10}) = \{q_{10}, q_1\}$ be **state A**

$\therefore \quad \delta'(A,a) = \varepsilon$ - closure $\{\delta(q_0,q_1),a\} = \varepsilon$ - closure $\{\delta(q_0,a),\cup\ \delta(q_1,a)\}$

$= \varepsilon$ - closure $\{q_2\}$

$= \{q_2,q_3,q_5,q_6,q_7\}$ **call it as B**

$\therefore \quad \delta'(A,b) = \varepsilon$ - closure $\{\delta((q_0,q_1)b)\}$

$= \varepsilon$ - closure $\{\delta(q_0,b),\cup\ \delta(q_1,b)\}$

$= \varepsilon$ - closure $\{\phi\}$

$= \phi$

$\delta(A,c) = \phi$

$\delta(A,d) = \phi$

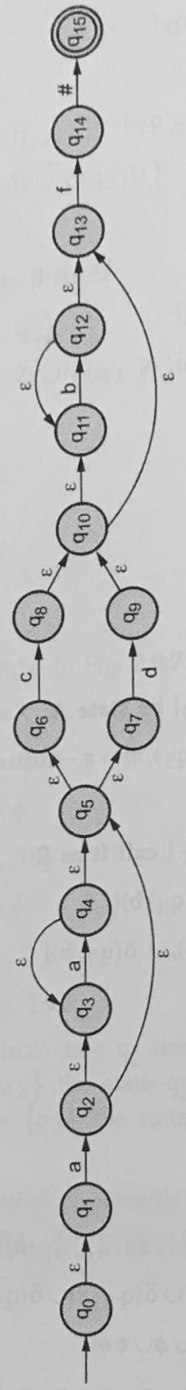
$\delta(A,f) = \phi$

$\delta(B,a) = \varepsilon$ - closure $(\delta((q_2,q_3,q_5,q_6,q_7)a))$

$= \varepsilon$ - closure $(\delta(q_2,a) \cup \delta(q_3,a) \cup \delta(q_5,a) \cup \delta(q_6,a),\ \cup \delta(q_7,a))$

$= \varepsilon$ - closure $(\phi \cup q_4 \cup \phi \cup \phi\phi)$

$= \varepsilon$ - closure $(q_4) = (q_3,q_4,q_5,q_6,q_7)$ **Call as state C**



δ **(B, a) = C**

δ(B, b) = ε - closure($\delta((q_2, q_3, q_5, q_6, q_7), b)$)

= ε - closure($\delta(q_2, b) \cup \delta(q_3, b) \cup \delta(q_5, b) \cup \delta(q_6, b) \cup \delta(q_7, b)$)

= ε - closure($\phi \cup \phi \cup \phi \cup \phi \cup \phi$)

δ **(B, b) =** ϕ

δ(B, c) = ε - closure(q_8)

= $\{q_8, q_{10}, q_{11}, q_{13}\}$ **Call it as state D**

δ **(B, c) = D**

δ(B, d) = ε - closure(q_9)

= $\{q_9, q_{10}, q_{11}, q_{13}\}$ Call it as state E

δ(B, d) = E

δ(B, f) = ϕ

δ(C, a) = ε - closure($\delta(q_3, q_4, q_5, q_6, q_7)$ a)

= ε - closure($\delta(q_3, a) \cup \delta(q_4, a) \cup \delta(q_5, a) \cup \delta(q_6, a) \cup \delta(q_7, a)$)

= ε - closure(q_4) = C

δ(C, b) = ε - closure($\delta((q_3, q_4, q_5, q_6, q_7)$ b))

= ε - closure($\delta(q_3, b) \cup \delta(q_4, b) \cup \delta(q_5, b) \cup \delta(q_6, b) \cup \delta(q_7, b)$)

= ε - closure (ϕ)

δ(C, b) = ϕ

δ(C, c) = ε - closure($\delta(q_3, q_4, q_5, q_6, q_7)$ c))

= ε - closure{ q_8}

δ(C, c) = D

δ(B, f) = ϕ

δ(C, d) = ε - closure($\delta((q_3, q_4, q_5, q_6, q_7)$ d))

= ε - closure{ q_9}

δ(C, d) = E

δ(C, f) = ε - closure($\delta((q_3, q_4, q_5, q_6, q_7)$ f))

δ(C, f) = ϕ

δ(D, a) = ε - closure($\delta(q_8, q_{10}, q_{11}, q_{13})$, a)

$\delta(D, a) = \phi$

$\delta(D, b) = \varepsilon\text{-closure}(\delta((q_{12})$

$\delta(D, b) = (q_{11}, q_{12}, q_{13})$ Call it as State F

$\therefore \quad \delta(D, b) = F$

$\delta(D, c) = \delta(D, d) = \phi$

$\delta(D, f) = \varepsilon\text{-closure}((q_{14}) = q_{14}$ i.e. State G

$\delta(D, f) = G$

$\delta(E, a) = \varepsilon\text{-closure}(\delta((q_9, q_{10}, q_{11}, q_{13}), a))$

$\delta(E, b) = F$

$\delta(E, c) = \delta(E, d) = \phi$

$\delta(E, f) = G$

$\delta(F, a) = \varepsilon\text{-closure}(\delta((q_{11}, q_{12}, q_{13},), a))$

$\delta(F, a) = \varepsilon\text{-closure}(\phi) = \phi$

$\delta(F, b) = \varepsilon\text{-closure}((q_{12}) = F$

$\delta(F, c) = \delta(F, d) = \phi$

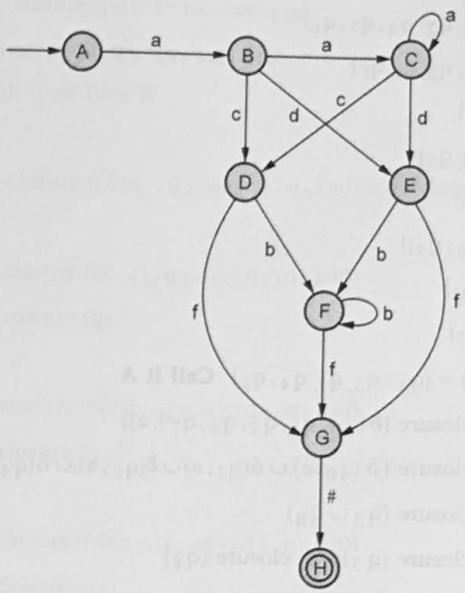
$\delta(F, f) = G$

$\delta(G, a) = \delta(G, b) = \delta(G, c) = \delta(G, d) = \delta(G, f) = \phi$

$\delta(G, \#) = \varepsilon - \text{closure}(q_{15})$ i.e. State H

$\therefore \quad \delta(G, \#) = H$

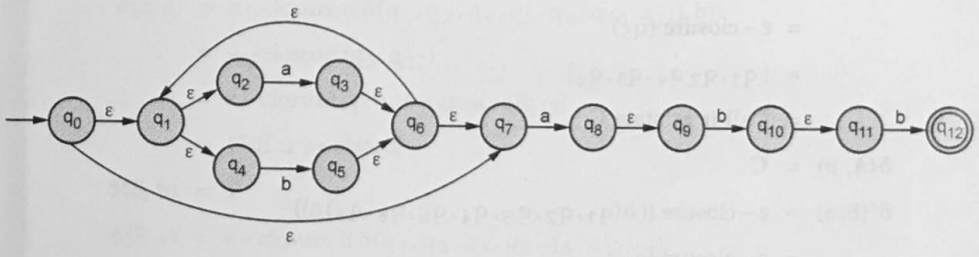
	a	b	c	d	e	#
A	B	ϕ	ϕ	ϕ	ϕ	ϕ
B	C	ϕ	D	E	ϕ	ϕ
C	C	ϕ	D	E	ϕ	ϕ
D	ϕ	F	ϕ	ϕ	G	ϕ
E	ϕ	F	ϕ	ϕ	G	ϕ
F	ϕ	F	ϕ	ϕ	G	ϕ
G	ϕ	ϕ	ϕ	ϕ	ϕ	H
H	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ



Example 2.8.6 *Construct the NFA for following Regular Expression using Thompson's construction. Apply subset construction method to convert into DFA. (a + b)*abb#*

GTU : Winter-15, 18, 19, 20, Marks 7

Solution : The NFA using Thompson's construction is as follows :



ε - closure $(q_0) = \{q_0, q_1, q_2, q_4, q_7\}$

ε - closure $(q_1) = \{q_1, q_2\}$

ε - closure $(q_2) = \{q_2\}$

ε - closure $(q_3) = \{q_1, q_2, q_3, q_4, q_6\}$

ε - closure $(q_4) = \{q_4\}$

ε - closure (q_5) = $\{q_1, q_2, q_4, q_5, q_6\}$

ε - closure (q_6) = $\{q_1, q_2, q_6, q_7\}$

ε - closure (q_7) = $\{q_7\}$

ε - closure (q_8) = $\{q_8, q_9\}$

ε - closure (q_9) = $\{q_9\}$

ε - closure(q_{10}) = $\{q_{10}, q_{11}\}$

ε - closure (q_{11}) = $\{q_{11}\}$

ε - closure (q_{12}) = $\{q_{12}\}$

Let, $\varepsilon - \text{closure}(q_0) = \{q_1, q_1, q_2, q_4, q_7\}$ **Call it A**

$\therefore$ $\delta'(A, a) = \varepsilon$ - closure $(\delta(q_0, q_1, q_2, q_4, q_7), a))$

$= \varepsilon$ - closure $(\delta(q_0, a), \cup \delta(q_1, a) \cup \delta(q_2, a), \cup \delta(q_4, a) \cup \delta(q_7, a))$

$= \varepsilon$ - closure $(q_3 \cup q_8)$

$= \varepsilon$ - closure $(q_3) \cup \varepsilon$ - closure (q_8)

$= \{q_1, q_2, q_3, q_4, q_6\} \cup \{q_8, q_9\}$

$= \{q_1, q_2, q_3, q_4, q_6, q_8, q_9\}$

= Call it as state B

$\therefore$ $\delta'(A, a) = B$

$\delta'(A, b) = \varepsilon - \text{closure}(\delta(q_0, q_1, q_2, q_4, q_7), b))$

$= \varepsilon - \text{closure}(q_5)$

$= \{q_1, q_2, q_4, q_5, q_6\}$

= Call it as state C

$\delta(A, b) = C$

$\delta'(B, a) = \varepsilon - \text{closure}((\delta(q_1, q_2, q_3, q_4, q_6, q_8, q_9)\, a))$

$= \varepsilon - \text{closure}(q_3)$

$= \{q_1, q_2, q_3, q_4, q_6\}$

= Call it as state D

$\delta'(B, a) = D$

$\delta'(B, b) = \varepsilon - \text{closure}((\delta(q_1, q_2, q_3, q_4, q_6, q_8, q_9), b))$

$= \varepsilon - \text{closure}(q_5 \cup q_{10})$

$= \varepsilon - \text{closure}\{q_5\} \cup \varepsilon - \text{closure}\{q_{10}\}$

$= \{q_1, q_2, q_4, q_5, q_6, q_{10}, q_{11}\}$

= Call it as state E

$\delta(B, b) = E$

$\delta'(C, a) = \varepsilon - \text{closure}((\delta(q_1, q_2, q_4, q_5, q_6), a)) = \varepsilon - \text{closure}(q_3)$

$\delta'(C, a) = D$

$\delta(C, b) = \varepsilon - \text{closure}((\delta(q_1, q_2, q_4, q_5, q_6), b))$

$= \varepsilon - \text{closure}(q_5)$

$\delta(C, a) = C$

$\delta(D, a) = \varepsilon - \text{closure}((\delta(q_1, q_2, q_3, q_4, q_6), a))$

$= \varepsilon - \text{closure}(q_3)$

$\delta(D, a) = D$

$\delta(D, b) = \varepsilon - \text{closure}((\delta(q_1, q_2, q_3, q_4, q_6), b))$

$= \varepsilon - \text{closure}(q_5)$

$\delta(D, b) = C$

$\delta(E, a) = \varepsilon - \text{closure}((\delta(q_1, q_2, q_4, q_5, q_6, q_{10}, q_{11}), a))$

$= \varepsilon - \text{closure}(q_3)$

$\delta(E, a) = D$

$\delta(E, a) = \varepsilon - \text{closure}((\delta(q_1, q_2, q_4, q_5, q_6, q_{10}, q_{11}), b))$

$= \varepsilon - \text{closure}(q_5, q_{12})$

$= \varepsilon - \text{closure}\{q_5\} \cup \varepsilon - \text{closure}\{q_{12}\}$

= Call it as state F

$\therefore$ $\delta(E, b) = F$

$\delta(F, a) = \varepsilon - \text{closure}((\delta(q_1, q_2, q_4, q_5, q_6, q_{12}), a))$

$= \varepsilon - \text{closure}(q_3)$

$\delta(F, a) = D$

$\delta(F, b) = \varepsilon - \text{closure}((\delta(q_1, q_2, q_4, q_5, q_6, q_{12}), b))$

$= \varepsilon - \text{closure}(q_5)$

$\delta(F, b) = C$

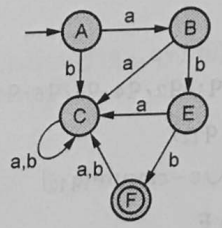
The transiition table will be

State \ i/p	a	bs
A	B	C
B	D	E
C	D	C
D	D	C
E	D	F
F	D	C

State C and D are both non- final states and their input transitions are same. Hence C = D.

The optimized DFA will be

State \ i/p	a	b
A	B	C
B	C	E
C	C	C
E	C	F
F	C	C



Example 2.8.7 *Convert the following NFA-∧ into equivalent NFA. Here ε is a ∧-transition.*

GTU : May-12, Marks 7

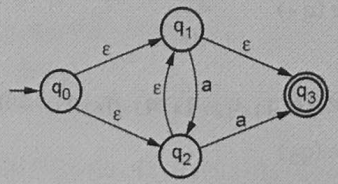


Fig. 2.8.9

Solution :

ε - closure $(q_0) = \{q_0, q_1, q_2, q_3\}$

ε - closure $(q_1) = \{q_1, q_3\}$

ε - closure $(q_2) = \{q_2, q_1, q_3\} = \{q_1, q_2, q_3\}$

ε - closure $(q_3) = \{q_3\}$

We will obtain δ transitions for each state for input a.

$\delta'(q_0, a) = \varepsilon$ - closure $(\delta(\delta^{\wedge}(q_0, \varepsilon), a))$

$= \varepsilon$ - closure $(\delta(\varepsilon$ - closure $(q_0), a))$

$= \varepsilon$ - closure $(\delta((q_0, q_1, q_2, q_3), a)) = \varepsilon$ - closure $(q_2 \cup q_3)$

$= \varepsilon$ - closure $(q_2) \cup \varepsilon$ - closure (q_3)

$= \{q_1, q_2, q_3\} \cup \{q_3\}$

$\delta'(q_0, a) = \{q_1, q_2, q_3\}$

$\delta'(q_1, a) = \varepsilon$ - closure $(\delta((\varepsilon$ - closure $(q_1), a))$

$= \varepsilon$ - closure $(\delta((q_1, q_3), a)) = \varepsilon$ - closure (q_2)

$\delta'(q_1, a) = \{q_1, q_2, q_3\}$

$\delta'(q_2, a) = \varepsilon$ - closure $(\delta(\varepsilon$ - closure $(q_2), a))$

$= \varepsilon$ - closure $(\delta((q_1, q_2, q_3), a)) = \varepsilon$ - closure (q_2, q_3)

$= \varepsilon$ - closure $(q_2) \cup \varepsilon$ - closure (q_3)

$\delta'(q_2, a) = \{q_1, q_2, q_3\}$

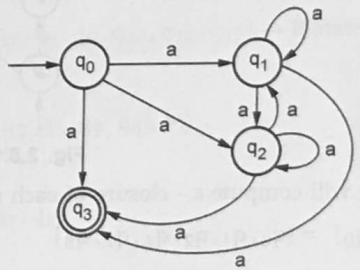
$\delta'(q_3, a) = \varepsilon$ - closure $(\delta(\varepsilon$ - closure $(q_3), a)) = \varepsilon$ - closure $(\delta(q_3, a))$

$= \varepsilon$ - closure (ϕ)

$\delta'(q_3, a) = \phi$

The transition table will be

State \ i/p	a
→ q_0	$\{q_1, q_2, q_3\}$
q_1	$\{q_1, q_2, q_3\}$
q_2	$\{q_1, q_2, q_3\}$
* q_3	ϕ



Example 2.8.8 *Draw NFA from regular expression using Thomson's construction and convert it into DFA.*

$(a|b)^* ab^* a$

GTU : Summer-19, Marks 7

Solution : Step 1 : We will construct NFA using Thomson's construction.

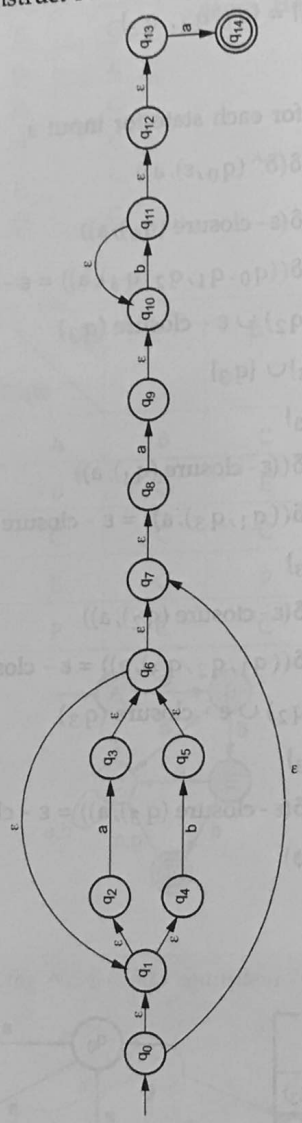


Fig. 2.8.10

Step 2 : We will compute ε - closure of each state

ε - closure(q_0) = {$q_0, q_1, q_2, q_4, q_7, q_8$}

ε - closure(q_1) = {q_1, q_2, q_4}

ε - closure(q_2) = {q_2}

ε - closure(q_3) = {$q_3, q_6, q_1, q_2, q_4, q_7, q_8$}

= {$q_1, q_2, q_3, q_4, q_6, q_7, q_8$}

ε - closure(q_4) = {q_4}

ε - closure(q_5) = {$q_5, q_6, q_7, q_8, q_1, q_2, q_4$}

= {$q_1, q_2, q_4, q_5, q_6, q_7, q_8$}

ε - closure(q_6) = {$q_6, q_7, q_8, q_1, q_2, q_4$}

= {$q_1, q_2, q_4, q_6, q_7, q_8$}

ε - closure(q_7) = {q_7, q_8}

ε - closure(q_8) = {q_8}

ε - closure(q_9) = {$q_9, q_{10}, q_{12}, q_{13}$}

ε - closure(q_{10}) = {q_{10}}

ε - closure(q_{11}) = {q_{11}, q_{12}, q_{13}}

ε - closure(q_{12}) = {q_{12}, q_{13}}

ε - closure(q_{13}) = {q_{13}}

ε - closure(q_{14}) = {q_{14}}

Step 3 : Let,

ε - closure(q_0) = { $q_0, q_1, q_2, q_4, q_7, q_8$ } **Call it A**

$\therefore$ δ' (A, a) = ε - closure ($\delta(q_0, q_1, q_2, q_4, q_7, q_8)$, a))

= ε - closure ($q_3 \cup q_9$)

= ε - closure (q_3) ∪ ε - closure(q_9)

= {$q_1, q_2, q_3, q_4, q_6, q_7, q_8$}∪{$q_9, q_{10}, q_{12}, q_{13}$}

= {$q_1, q_2, q_3, q_4, q_6, q_7, q_8, q_9, q_{10}, q_{12}, q_{13}$} → **B state**

$\therefore$ **δ'(A, a)** = **B**

δ' (A, b) = ε - closure ($\delta(q_0, q_1, q_2, q_4, q_7, q_8)$, b))

= ε - closure (q_5)

= {$q_1, q_2, q_4, q_5, q_6, q_7, q_8$}

= Call it as state **C**

$\therefore$ $\delta'(A, b) = C$

$\delta'(B, a) = \varepsilon\text{-closure}(\delta(q_1, q_2, q_3, q_4, q_6, q_7, q_8, q_9, q_{10}, q_{12}, q_{13}), a))$

$= \varepsilon\text{-closure}(q_3 \cup q_9 \cup q_{14})$

$= \{q_1, q_2, q_3, q_4, q_6, q_7, q_8, q_9, q_{10}, q_{12}, q_{13}, q_{14}\}$

$=$ Call it as state D.

$\therefore$ $\delta'(B, a) = D$

$\delta'(B, b) = \varepsilon\text{-closure}(\delta(q_1, q_2, q_3, q_4, q_6, q_7, q_8, q_9, q_{10}, q_{12}, q_{13}), b))$

$= \varepsilon\text{-closure}(q_5 \cup q_{11})$

$= \{q_1, q_2, q_4, q_5, q_6, q_7, q_8, q_{11}, q_{12}, q_{13}\} \rightarrow E$

$\delta'(B, b) = E$

$\delta'(C, a) = \varepsilon\text{-closure}(\delta(q_1, q_2, q_4, q_5, q_6, q_7, q_8), a))$

$= \varepsilon\text{-closure}(q_3 \cup q_9)$

$\delta'(C, a) = B$

$\delta'(C, b) = \varepsilon\text{-closure}(\delta(q_1, q_2, q_4, q_5, q_6, q_7, q_8), b))$

$= \varepsilon\text{-closure}(q_5)$

$\therefore$ $\delta'(C, b) = C$

$\delta'(D, a) = \varepsilon\text{-closure}(\delta(q_1, q_2, q_3, q_4, q_6, q_7, q_8, q_9, q_{10}, q_{12}, q_{13}, q_{14}), a))$

$= \varepsilon\text{-closure}(q_3 \cup q_9 \cup q_{14})$

$\delta'(D, a) = D$

$\delta'(D, b) = \varepsilon\text{-closure}(\delta(q_1, q_2, q_3, q_4, q_6, q_7, q_8, q_9, q_{10}, q_{12}, q_{13}, q_{14}), a))$

$= \varepsilon\text{-closure}(q_5 \cup q_{11})$

$\delta'(D, a) = E$

$\delta'(E, a) = \varepsilon\text{-closure}(\delta(q_1, q_2, q_4, q_5, q_6, q_7, q_8, q_{11}, q_{12}, q_{13}), a))$

$= \varepsilon\text{-closure}(q_3 \cup q_9 \cup q_{14})$

$\delta'(E, a) = D$

$\delta'(E, b) = \varepsilon\text{-closure}(\delta(q_1, q_2, q_4, q_5, q_6, q_7, q_8, q_{11}, q_{12}, q_{13}), b))$

$= \varepsilon\text{-closure}(q_5) = C$

$\therefore$ $\delta'(E, b) = C$

The transition table is

State \ i/p	a	b
→ A	B	C
B	D	E
C	B	C
(D)	D	E
E	D	C

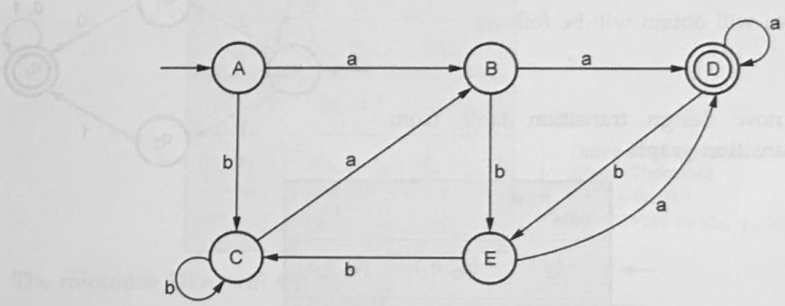


Fig. 2.8.11

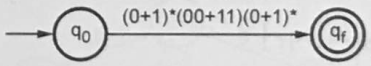
2. Direct Method

In this method the NFA with ε is built using Thompson's Construction Method. The ε is eliminated and then the NFA without ε is converted to DFA. Let us understand this method with the help of example

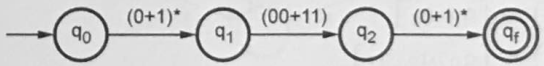
Example 2.8.9 *Construct deterministic finite automaton to accept the regular expression* $(0+1)^*(00+11)(0+1)^*$

Solution : The FA for r.e.= $(0+1)^*(00+11)(0+1)^*$ is as follows -

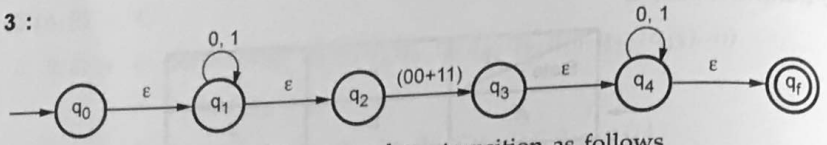
Step 1 :



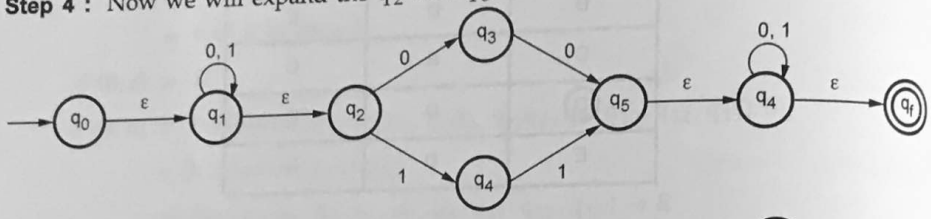
Step 2 :



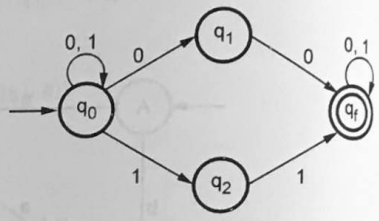
Step 3 :



Step 4 : Now we will expand the q_2 and q_3 transition as follows



Step 5 : Now we will eliminate ε transitions and the FA which we will obtain will be follows



Step 6 : We now design transition table from above drawn transition graph-

	State \ Input	0	1
→	q_0	$\{q_0, q_1\}$	$\{q_0, q_2\}$
	q_1	q_f	–
	q_2	–	q_f
*	q_f	q_f	q_f

The $\{q_0, q_1\}$ is assumed to be a new state $[q_0, q_1]$ and $\{q_0, q_2\}$ is assumed to be a new state $[q_0, q_2]$. Hence we will find input transitions on these states. First we will find and transition on q_0 and q_1

$\delta([q_0, q_1], 0) = \{q_0, q_1, q_f\} = [q_0, q_1, q_f] \rightarrow$ **new state**

$\delta([q_0, q_1], 1) = \{q_0, q_2\} = [q_0, q_2] \rightarrow$ **new state.**

Now we will apply input 0 and 1 transitions on the newly formed state

$\delta([q_0, q_1, q_f], 0) = [q_0, q_1, q_f]$

$\delta([q_0, q_1, q_f], 1) = [q_0, q_1, q_f] \rightarrow$ **new state**

$\delta([q_0, q_2], 0) = [q_0, q_1]$

$\delta([q_0, q_2], 1) = [q_0, q_2, q_f]$

$\delta([q_0, q_2, q_f], 0) = [q_0, q_1, q_f]$

$\delta([q_0, q_2, q_f], 1) = [q_0, q_2, q_f]$

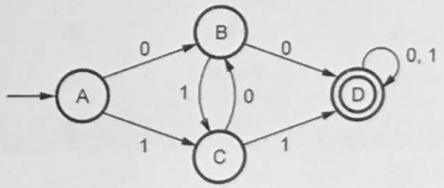
Now no new state getting generated. Hence we will write transition table as -

	State \ Input	0	1	
→	$[q_0]$	$[q_0, q_1]$	$[q_0, q_2]$	
	$[q_1]$	q_f	–	These are non-reachable states from the start state q_0 ∴ Eliminate them
	$[q_2]$	–	q_f	
*	$[q_f]$	q_f	q_f	
	$[q_0, q_1]$		$[q_0, q_2]$	
	$[q_0, q_2]$	$[q_0, q_1]$	$[q_0, q_2, q_f]$	
*	$[q_0, q_1, q_f]$	$[q_0, q_1, q_f]$	$[q_0, q_2, q_f]$	These are equivalent states. Therefore $[q_0, q_2, q_f]$ can be replaced by $[q_0, q_1, q_f]$
*	$[q_0, q_2, q_f]$	$[q_0, q_1, q_f]$	$[q_0, q_2, q_f]$	

The minimize DFA will then be -

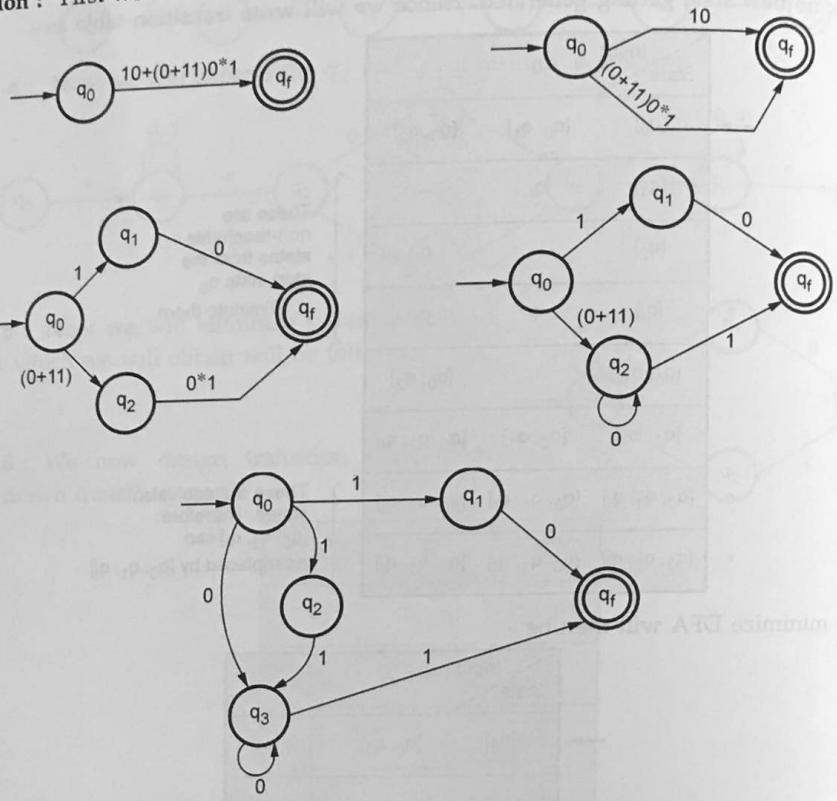
	State \ Input	0	1
→	$[q_0]$	$[q_0, q_1]$	$[q_0, q_2]$
	$[q_0, q_1]$	$[q_0, q_1, q_f]$	$[q_0, q_2]$
	$[q_0, q_2]$	$[q_0, q_1]$	$[q_0, q_2, q_f]$
*	$[q_0, q_1, q_f]$	$[q_0, q_1, q_f]$	$[q_0, q_1, q_f]$

If we assume $A = [q_0]$, $B = [q_0, q_1]$, $C = [q_0, q_2]$ $D = [q_0, q_1, q_f]$ then the DFA will be -



Example 2.8.10 *Design a FA from given regular expression* $10 + (0 + 11)\,0^*\,1$.

Solution : First we will construct the transition diagram for given regular expression.



Now we have got NFA without ε. Now we will convert it to required DFA for that, we will first write a transition table for this NFA.

State \ Input	0	1
q_0	q_3	$\{q_1, q_2\}$
q_1	q_f	ϕ
q_2	ϕ	q_3
q_3	q_3	q_f
q_f	ϕ	ϕ

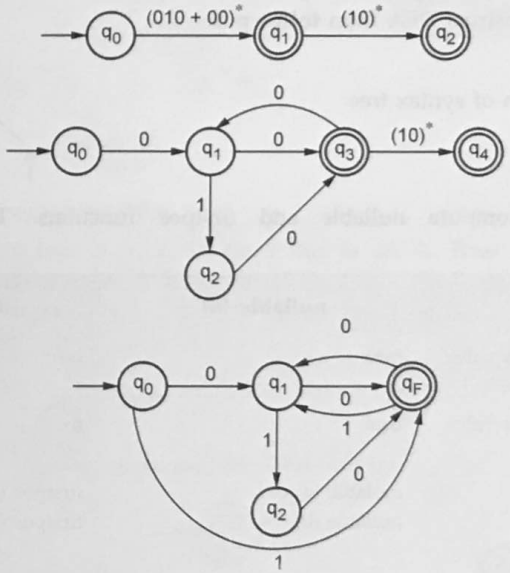
The equivalent DFA will be

State \ Input	0	1
$[q_0]$	$[q_3]$	$[q_1, q_2]$
$[q_1]$	$[q_f]$	ϕ
$[q_2]$	ϕ	$[q_3]$
$[q_3]$	$[q_3]$	$[q_f]$
$[q_1, q_2]$	$[q_f]$	$[q_3]$
$[q_f]$	ϕ	ϕ

Example 2.8.11 *Construct a DFA for a given regular expression* $(010+00)^*(10)^*$.
GTU : May-12, Summer-20, Marks 7

Solution : We will construct DFA for r.e. (010 + 00)* (10)* using direct method.

Step 1 :



Requird DFA

Fig. 2.8.12

Example 2.8.12 *Construct DFA accepting the strings of binary digits which are even numbers.*
GTU : Summer-20, Marks 7

Solution :

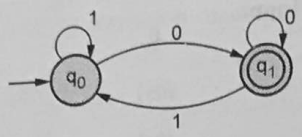


Fig. 2.8.13

The strings that end with 0 are the even numbers.

3. DFA Tree Method

The Regular expression can be directly converted to DFA by constructing binary tree. Let us understand this method with the help of examples.

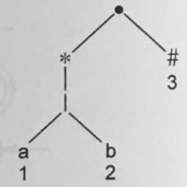
Example 2.8.13 *Draw the DFA for the augmented regular expression (a|b)*# directly using syntax tree.*

Solution : The grammar is said to be an augmented grammar if it is of the form (r)#.

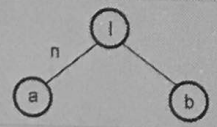
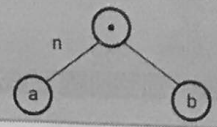
We will construct a syntax tree T for (r)# i.e. (a|b)*# and then compute four functions - **1) nullable 2) firstpos 3) lastpos 4) followpos**, by traversing T.

Finally we will construct DFA from **followpos**.

Step 1 : Construction of syntax tree



Step 2 : We will compute nullable and firstpos functions. The rules for this computations are -

Node (n)	nullable (n)	firstpos (n)
n is a leaf node and its value is labeled with position x	false	{x}
n is a leaf node and its value is ∈ or null	true	ϕ
	nullable (a) or nullable (b)	firstpos (a) ∪ firstpos (b)
	nullable (a) and nullable (b)	if nullable (a) then firstpos (a) ∪ firstpos (b) else firstpos (a)

	true	firstpos (a)

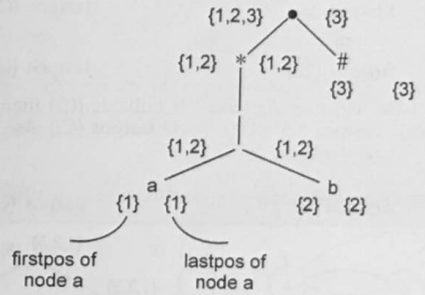
The rules for lastpos are same as the rules of firstpos except that a and b are reversed.

The computation of followpos function will be using these rules -

Rule 1 : If node n represents concatenation with left child a and right child b and x is the position in lastpos (a) then all positions in firstpos (b) are in followpos (x).

Rule 2 : If n is a Kleen closure node and x is a position in lastpos (n) then all positions in firstpos (n) are in followpos (x).

The firstpos and lastpos and followpos for each node are as follows -



Node	followpos
1	{1,2,3}
2	{1,2,3}
3	-

Step 3 : Now we will construct DFA.

Step a : The firstpos of root is {1, 2, 3}. Mark this as set A. Thus A will be the start state in DFA. Now consider input a from the set {1, 2, 3} - the 1 and 3 represents input a. So followpos (1) ∪ followpos (3) = {1, 2, 3} i.e. A. From set {1, 2, 3} - the 2 represents input b. So followpos (2) = {1, 2, 3} i.e. A -

∴ Dtran [A, a] = A

Dtran [A, b] = A

As no new states are getting generated, the DFA will be

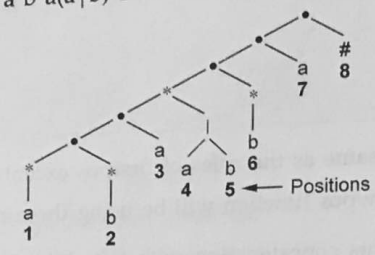
Fig. 2.8.14 DFA for (a|b)*#

Example 2.8.14 *Construct a DFA without constructing NFA for following regular expression. Find minimized DFA.*

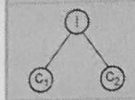
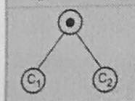
*a*b*a(a|b)*b*a#*

GTU : Winter-19, Marks 7

Solution : Consider r.e. = a*b*a(a|b)*b*a# we will construct a binary tree for given r.e



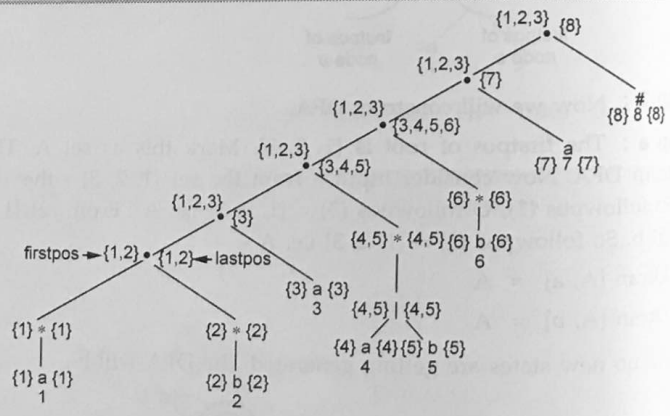
Now we will obtain **nullable(n)**, **firstpos(n)** and **lastpos(n)** using following table

Node n	nullable (n)	firstpos (n)	lastpos (n)
Leaf ε	true	ϕ	ϕ
Leaf i	false	{ i }	{ i }
	nullable (C_1) or nullable (C_2)	firstpos (C_1) or firstpos (C_2)	lastpos (C_1) or lastpos (C_2)
	nullable (C_1) and nullable (C_2)	if nullable (C_1) then firstpos (C_1) U firstpos (C_2) else firstpos (C_1)	if nullable (C_2) then lastpos (C_1) U lastpos (C_2) else lastpos (C_2)
*	true	firstpos (C_1)	lastpos (C_1)

Let us write firstpos and lastpos in the tree at every node.

To the left we write firstpos and to the right we write lastops.

Now we will compute **followpos** using following algorithm.




```
for each node n in the tree do
    if n is a cat-node with left child C1 and right child C2 then    //Cat-node means
                                                                     //Concatenation
        for each i in lastpos(C1) do
        followpos(i) := followpos(i) ∪ firstpos(C2)
    end do
    else if n is a star-node
```

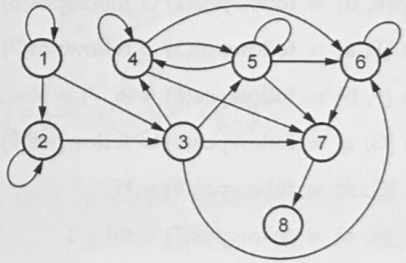


```
        for each i in lastpos(n) do
            followpos(i) := followpos(i) ∪ firstpos(n)
        end do
    end if
end do
```

For above tree, for each node followpos is as follows -

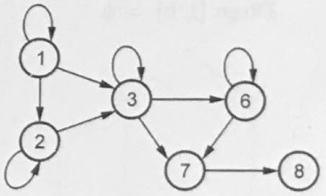
Node	followpos
1	{ 1, 2, 3 }
2	{ 2, 3 }
3	{ 4, 5, 6, 7 }
4	{ 4, 5, 6, 7 }
5	{ 4, 5, 6, 7 }
6	{ 6, 7 }
7	{ 8 }
8	-

The transition graph for above table is as follows



If we assume state 3 = state 4 = state 5. Then minimized transition table and graph will be

1	{ 1, 2, 3 }
2	{ 2, 3 }
3	{ 3, 6, 7 }
6	{ 6, 7 }
7	{ 8 }
8	----



Assume A = firstpos(root) = {1, 2, 3} Here root is the root of tree shown on page 2-49

Dtran [A, a] = followpos(1)∪ followpos(3)
= {1, 2, 3, 6, 7} = B

Dtran [A, b] = followpos(2) = {2, 3} = C

Dtran [B, a] = followpos(1) ∪ followpos(3) ∪ followpos(7)
= {1, 2, 3, 6, 7, 8} = D.

Dtran [C, b] = followpos(2) ∪ followpos(6)
= {2, 3, 6, 7} = E

Dtran [C, a] = followpos(3) = {3, 6, 7} = F

Dtran [C, b] = followpos(2) = {2, 3} = C.

Dtran [D, a] = followpos(1) ∪ followpos(3) ∪ followpos(7) = D.

Dtran [D, b] = followpos(2) ∪ followpos(6) = E

Dtran [E, a] = followpos(3) ∪ followpos(7)
= {3, 6, 7, 8} = G.

Dtran[E, b] = followpos(2) ∪ followpos(6) = E

Dtran [F, a] = followpos(3) ∪ followpos(7) = G

Dtran [F, b] = followpos(6) = {6, 7} = H.

Dtran [G, a] = followpos(3) ∪ followpos(7) = G

Dtran [G, b] = followpos(6) = H

Dtran [H, a] = followpos(7) = {8} = I

Dtran [H, b] = followpos(6) = H

Dtran [I, a] = ϕ

Dtran [I, b] = ϕ.

A = {1, 2, 3}. For Dtran [A,a] we refer 'a' position in A = {1,2,3} and those are 1 and 3. Hence take their followpos.

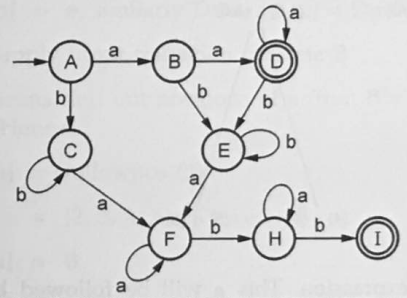
Transition Table

States	I/P a	b
A	B	C
B	D	E
C	F	C
D	D	E
E	G	E
F	G	H
G	G	H
H	H	I
I	ϕ	ϕ

Minimized DFA

As F = G

States	I/P a	b
→ A	B	C
B	D	E
C	F	C
* D	D	E
E	F	E
F	F	H
H	H	I
* I	ϕ	ϕ



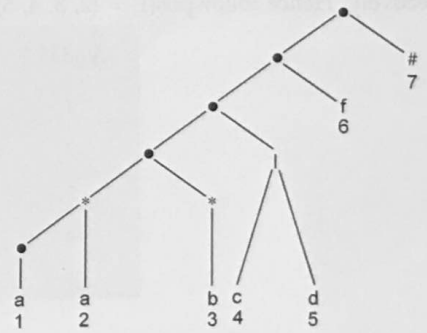
This is required minimzed DFA

Example 2.8.15 *Construct DFA by syntax tree construction method $a^+b^*(c|d)f\#$ optimize the resultant DFA.* **GTU : Summer-15, Marks 7**

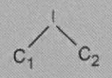
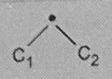
Solution : Consider

r.e. = $a^+b^*(c|d)f\#$ we will rewrite it as

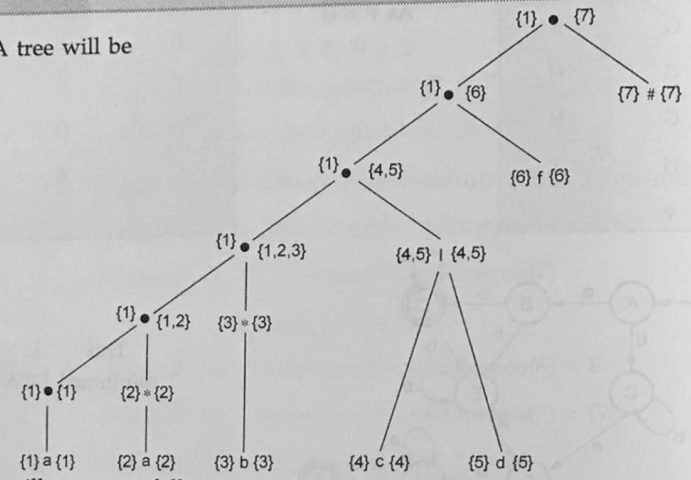
r.e. = $aa^*b^*(c|d)f\#$ we will binary tree for given r.e.



Now we will obtain nullable (n), firstpos(n) and lastpos(n) using following table

Node n	nullable (n)	firstpos (n)	lastpos (n)
Leaf ε	true	ϕ	ϕ
Leaf i	false	{ i }	{ i }
	nullable (C_1) or nullable (C_2)	firstpos (C_1) or firstpos (C_2)	lastpos (C_1) or lastpos (C_2)
	nullable (C_1) and nullable (C_2)	if nullable (C_1) then firstpos (C_1) ∪ firstpos (C_2) else firstpos (C_1)	if nullable (C_2) then lastpos (C_1) ∪ lastpos (C_2) else lastpos (C_2)

The DFA tree will be



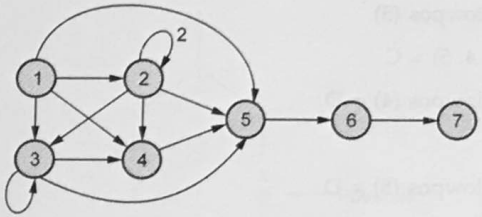
Now we will construct followpos.

Node 1 means **a** from aa* (i.e. a^+) expression. This **a** will be followed by i) any **a** from **a*** i.e. state 2. ii) It a* gives out ∈ then **a** will be followed by any **b** from b^* i.e. state 3 iii) If b^* gives out ε then **a** will be followed by **c** or **d** i.e. state 4 or 5 respectively. Hence followpos(1) = {2, 3, 4, 5}

Node	followpos
1	{1, 2, 3, 4, 5}
2	{2, 3, 4, 5}
3	{3, 4, 5}
4	{6}
5	{6}

6	{7}
7	-

The transition graph will be



Assume first pos (root) = {1} call it as **state A** we will obtain **or** transition on state A.

Dtran [A, a] = followpos (1)

= {2, 3, 4, 5} = **State B**

∴ **Dtran [A, a] = B**

Dtran [A, b] = ϕ, similarly Dtran [A, c] = Dtran [A, d] = **Dtran [A, f]** = ϕ.

Now we will apply input transition on state B

Dtran [B, a] means find out positions of **a** from B = {2, 3, 4, 5}. The node 2 denotes the position of **a**. Hence

Dtran [B, a] = followpos (2)

= {2, 3, 4, 5} i.e. B

∴ **Dtran [B, a] = B**

Dtran [B, b] = position of **b** in {2, 3, 4, 5} = 3

∴ Dtran [B, b] = followpos (3)

= {3, 4, 5} call it as state C

∴ **Dtran [B, b] = C**

Dtran [B, c] = followpos (4)

= {6} call it as state **D**.

∴ **Dtran [B, c] = D**

Now Dtran [B, d] = followpos (5)

= {6}

∴ **Dtran [B, d] = D**

Dtran [B, f] = ϕ

Dtran [C, a] = followpos (position of a in {3, 4, 5})

= followpos (ϕ) = ϕ

$\therefore$ **Dtran [C, b]** = followpos (3)

= {3, 4, 5} = C

Dtran [C, c] = followpos (4) = D

$\therefore$ **Dtran [C, c] = D**

$\therefore$ Dtran [C, d] = followpos (5) = D

$\therefore$ **Dtran [C, d] = D**

$\therefore$ **Dtran [C, f] = ϕ**

Dtran [D, a] = Dtran [D, b] = Dtran [D, c] = Dtran [D, d] = ϕ

Dtran [D, f] = followpos(6) = {7} call it as **state E**

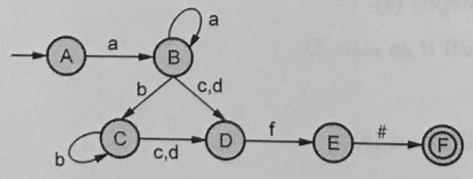
$\therefore$ Dtran [E, a] = Dtran [E, b] = Dtran [E, c] = Dtran [E, d] = Dtran [E, f] = ϕ

Dtran [E, #] = F i.e. accept state

The transition table will be as follows

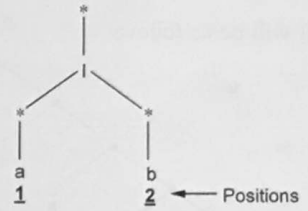
	a	b	c	d	f	#
A	B	ϕ	ϕ	ϕ	ϕ	ϕ
B	B	C	D	D	ϕ	ϕ
C	ϕ	C	D	D	ϕ	ϕ
D	ϕ	ϕ	ϕ	ϕ	E	ϕ
E	ϕ	ϕ	ϕ	ϕ	ϕ	F
F	-	-	-	-	-	-

The DFA is

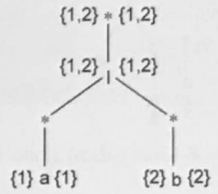


Example 2.8.16 *Construct DFA for following regular expression. Use firstpos, lastpos, and followpos functions $(a^*|b^*)^*$ to construct DFA.* **GTU : Winter-15, Marks 7**

Solution : We will construct binary tree for given regular expression



Now we will obtain firstpos and lastpos for node n.



Now we will obtain followpos for each node position

Node	follow pos
1	{1, 2}
2	{1, 2}

The transition graph for above table will be

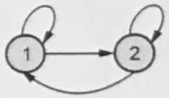
Assume firstpos (root) = {1, 2} **call it as state A**

Dtran [A, a] = followpos(1)

= {1, 2} = A

Dtran [A, b] = followpos(2)

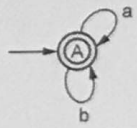
= {1, 2} = A



The transition table for DFA

State \ Input	a	b
→ A	A	A

The **Transition graph** for DFA

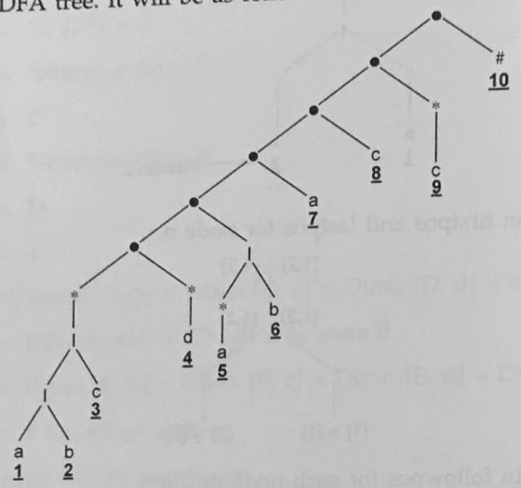


Example 2.8.17 *Convert the $(a|b|c)^* d^* (a^*|b|ac^+)\#$ regular expression to DFA directly and draw its DFA.* **GTU : Summer-16, Marks 7**

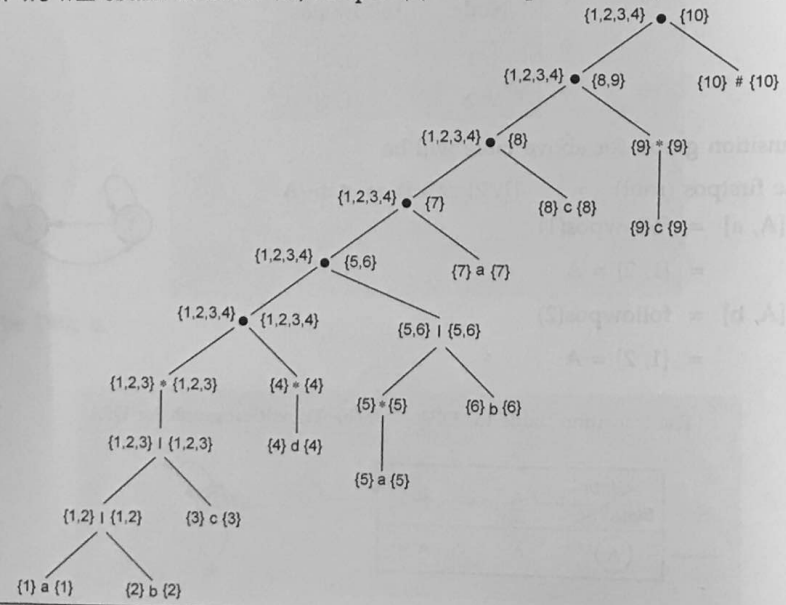
Solution : The given r.e. is re-written as

$(a|b|c)^* d^* (a^*|b)acc^*\#$

Let us draw the DFA tree. It will be as follows



Now we will obtain **nullable (n),** firstpos (n) and **lastpos(n)** using following table

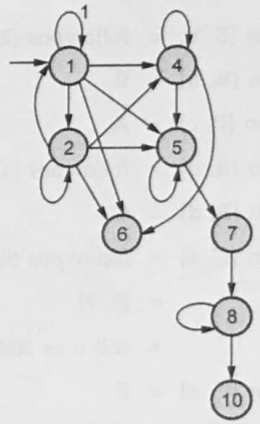


Now we will compute followpos of each node

Node	followpos
1	{1, 2, 3, 4, 5, 6}
2	{1, 2, 3, 4, 5, 6}
3	{1, 2, 3, 4, 5, 6}
4	{4, 5, 6}
5	{5, 7}
6	{7}
7	{8}
8	{9, 10}
9	{9, 10}
10	-

By observing followpos values of each node, state 8 = 9 then replace state 9 by 8. The minimized table will be -

Node	followpos
1	{1, 2, 3, 4, 5, 6}
2	{1, 2, 3, 4, 5, 6}
3	{1, 2, 3, 4, 5, 6}
4	{4, 5, 6}
5	{5, 7}
6	{7}
7	{8}
8	{8, 10}
10	-



Now, we will obtain Dtran for each state and each input

Assume A = firstpos (root) = {1, 2, 3, 4, 5, 6}

Dtran [A, a] = followpos (1) ∪ followpos (5)

= {1, 2, 3, 4, 5, 6, 7}

= call it as B

∴ **Dtran [A, a] = B**

Dtran [A, b] = followpos (2) ∪ followpos (6)

= {1, 2, 3, 4, 5, 6, 7}

Dtran [A, b] = B

Dtran [A, c] = followpos (3) = A

Dtran [A, d] = followpos (4)

= {4, 5, 6}

= call it as state C

∴ **Dtran [A, d] = C**

Dtran [B, b] = followpos (1) ∪ followpos (5) ∪ followpos (7)

= {1, 2, 3, 4, 5, 6, 7, 8}

= call it as state D

∴ **Dtran [B, a] = D**

Dtran [B, b] = followpos (2) ∪ followpos (6)

∴ **Dtran [B, b] = B**

Dtran [B, c] = A

Dtran [B, d] = followpos (4)

∴ **Dtran [B, d] = C**

Dtran [C, a] = followpos (5)

= {5, 7}

= call it as state E

∴ **Dtran [C, a] = E**

Dtran [C, b] = followpos (6)

= {7}

= call it as state F

∴ **Dtran [C, b] = F**

Dtran [C, c] = ϕ

Dtran [C, d] = followpos (4)

∴ **Dtran [C, d] = C**

Dtran [D, a] = followpos (1) ∪ followpos (5) ∪ followpos (7)

∴ **Dtran [D, a] = D**

Dtran [D, b] = followpos (2) ∪ followpos (6)

∴ **Dtran [D, b] = B**

Dtran [D, c] = followpos (8)

= {8, 10}

= call it as G

∴ **Dtran [D, c] = G**

Dtran [D, d] = followpos (4)

∴ **Dtran [D, d] = C**

Dtran [E, a] = followpos (5) ∪ followpos (7)

= {5, 7, 8}

= Call it as state H

∴ **Dtran [E, a] = H**

Dtran [E, b] = Dtran [E, c] = Dtran [E, d] = ϕ

Dtran [F, a] = Followpos (7)

= {8} → Call it as state I

∴ **Dtran [F, a] = I**

Dtran [F, b] = Dtran [F, c] = Dtran [F, d] = ϕ

Dtran [G, a] = Dtran [G, b] = ϕ

Dtran [G, c] = followpos (8) i.e. G

∴ **Dtran [G, c] = G**

Dtran [G, d] = ϕ

Dtran [H, a] = followpos (5) ∪ followpos (7)

Dtran [H, a] = H

Dtran [H, b] = ϕ

Dtran [H, c] = Followpos (8) = G

∴ **Dtran [H, d]** = ϕ

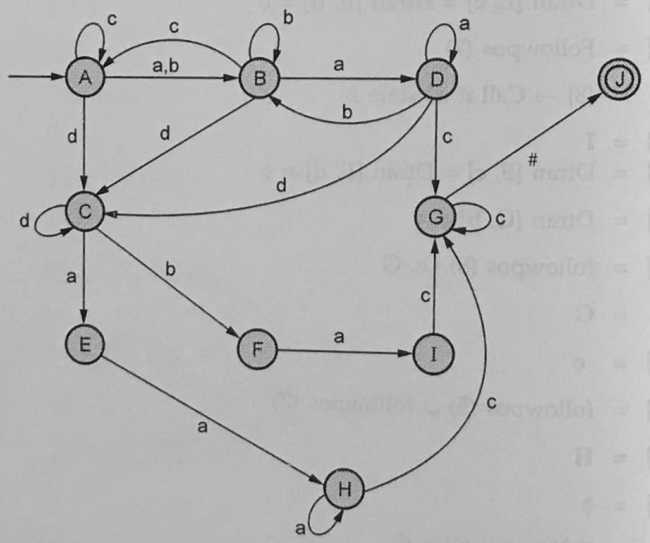
Dtran [I, a] = Dtran [I, b] = Dtran [I, d] = ϕ

Dtran [I, c] = followpos (8)

∴ Dtran [I, c] = G

The transition table for DFA is

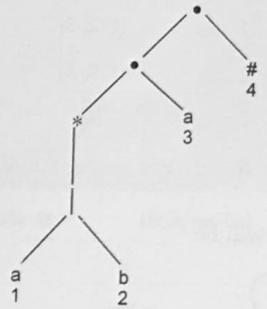
	a	b	c	d
A	B	B	ϕ	C
B	D	B	ϕ	C
C	E	F	ϕ	C
D	D	B	G	C
E	H	ϕ	ϕ	ϕ
F	I	ϕ	ϕ	ϕ
G	ϕ	ϕ	G	ϕ
H	H	ϕ	G	ϕ
I	ϕ	ϕ	G	ϕ



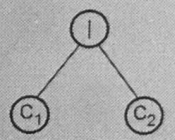
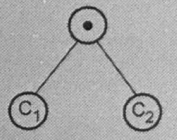
Example 2.8.18 *Construct DFA for the following regular expression using syntax tree with firstpos, lastpos and followpos function.* $(a|b)^*a$ **GTU : Winter-16, Marks 7**

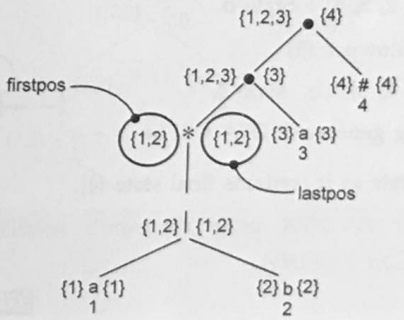
Solution :

Step 1 : We will construct syntax tree



Step 2 : We will compute nullable, firstpos and lastpos using following table.

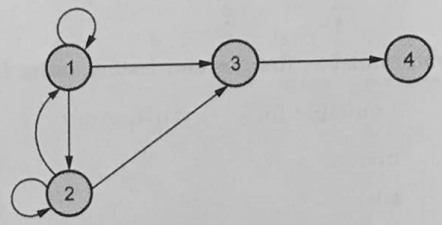
Node n	nullable (n)	firstpos (n)	lastpos (n)
Leaf ε	true	ϕ	ϕ
Leaf i	false	{i}	{i}
	nullable (C_1) or nullable (C_2)	firstpos (C_1) or firstpos (C_2)	lastpos (C_1) or lastpos (C_2)
	nullable (C_1) and nullable (C_2)	If nullable (C_1) then firstpos (C_1) firstpos (C_2) else firstpos (C_1) ∪	If nullable (C_2) then lastpos (C_1) lastpos (C_2) else lastpos (C_2) ∪
*	true	firstpos (c_1)	lastpos (c_1)



To the left we write firstpos and to the right we write lastpos. Now we will compute **followpos.**

Node	Followpos
1	{1, 2, 3}
2	{1, 2, 3}
3	{4}
4	-

Step 3 : The transition graph will be



Step 4 : Assume firstpos (root) = {1, 2, 3} as

State A

Dtran [A, a] = Followpos (1) ∪ Followpos (3)

= {1, 2, 3, 4} → **State B**

Dtran [A, b]= Followpos (2)

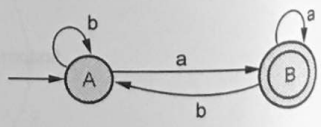
= {1, 2, 3} i.e. state A

Dtran [B, a] = Followpos (1) ∪ Followpos (3)

= {1, 2, 3, 4} = State B

Dtran [B, b] = Followpos (2)

= {1, 2, 3} i.e. state A



As no new state getting generated. DFA will be

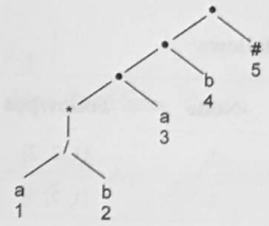
The state B is a final state as it contains final state {4}.

Example 2.8.19 *Construct the NFA using thompson's notation for following regular expression and then convert it to DFA.*

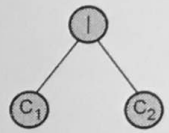
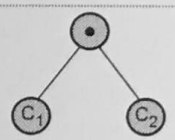
*(a/b)*ab#*

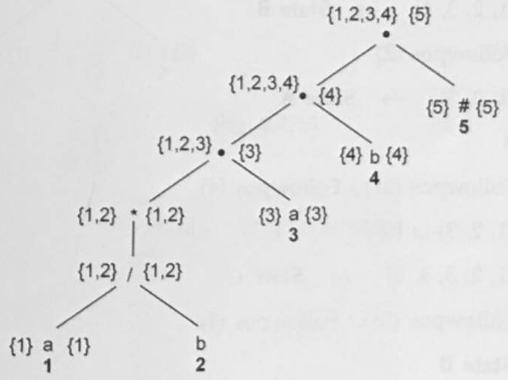
GTU : Summer-17, Marks 7

Solution : Step 1 : We will construct syntax tree.



Step 2 : We will compute nullable, firstpos and lastpos using following table

Node n	nullable (n)	firstpos (n)	lastpos (n)
Leaf ε	true	ϕ	ϕ
Leaf i	false	{i}	{i}
	nullable (C_1) or nullable (C_2)	firstpos (C_1) or firstpos (C_2)	lastpos (C_1) or lastpos (C_2)
	nullable (C_1) and nullable (C_2)	If nullable (C_1) then firstpos (C_1) firstpos (C_2) else firstpos (C_1) ∪	If nullable (C_2) then lastpos (C_1) lastpos (C_2) else lastpos (C_2) ∪
*	true	firstpos (c_1)	lastpos (c_1)

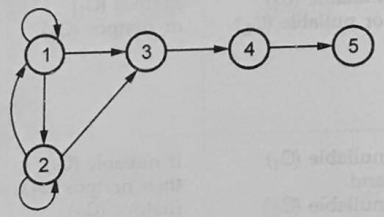


The left value written in curly bracket is firstpos and that written to the right is **lastpos.**

The firstpos is computed as follows.

Node	Followpos
1	{1, 2, 3}
2	{1, 2, 3}
3	{4}
4	{5}
5	-

Step 3 : The transition graph will be



Step 4 : Assume firstpos(root) = {1, 2, 3} = **A**

Dtram [A, a] = Followpos (1) ∪ Followpos (3)

= {1, 2, 3} ∪ {4}

= {1, 2, 3, 4} → **State B**

Dtran [A, b] = Followpos (2)

= {1, 2, 3} → **State A**

Dtran [B, a] = B

Dtran [B, b] = Followpos (2) ∪ Followpos (4)

= {1, 2, 3} ∪ {5}

= {1, 2, 3, 4, 5} → **State C**

Dtran [C, a] = Followpos (3) ∪ Followpos (1)

= **State B**

Dtran [C, a] = B

Dtran [C, b] = Followpos (2)

Dtran [C, b] = A

The DFA will be shown in Fig. 1

The state C contains final state {5}. Hence C is a final state in DFA.

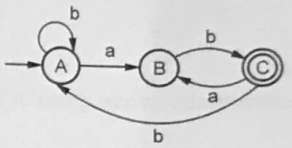


Fig. 2.8.15

Example 2.8.20 *Draw DFA from regular expression without constructing NFA.*
(a |b |c) a (b|c)* #*

GTU : Summer-19, Marks 7

Solution : We will construct a tree as follows :

r.e. = (a|b|c|)* a (b|c)* #

Step 1 :

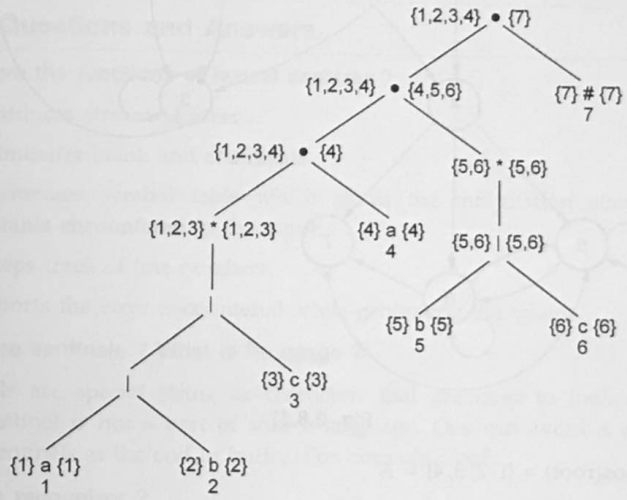


Fig. 2.8.16

Step 2 :

Node	FOLLOWPOS
1	{1, 2, 3, 4}
2	{1, 2, 3, 4}
3	{1, 2, 3, 4}
4	{5, 6, 7}

5	{5, 6, 7}
6	{5, 6, 7}
7	–

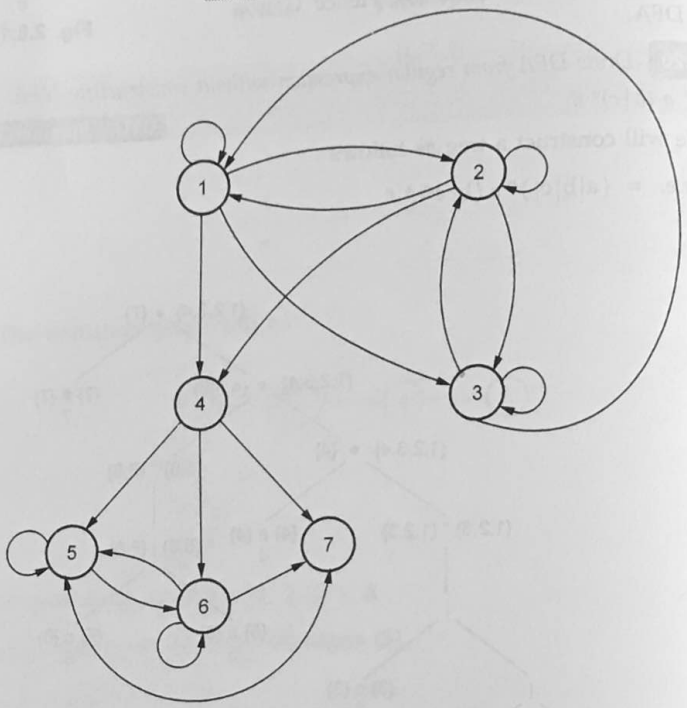


Fig. 2.8.17

Consider, first pos(root) = {1, 2, 3, 4} = A

Dtran [A, a] = FOLLOWPOS(1) ∪ FOLLOPOS (4)

= {1, 2, 3, 4}∪ {5, 6, 7} = {1, 2, 3, 4, 5, 6, 7} = B

Dtran [A, b] = FOLLOWPOS(2) = {1, 2, 3, 4} = A

Dtran [A, c] = FOLLOWPOS(3) = {1, 2, 3, 4} = A

Dtran [B, a] = FOLLOWPOS(1) ∪ FOLLOWPOS(4) = B

Dtran [B, b] = FOLLOWPOS(2) ∪ FOLLOPOS(5)

= {1, 2, 3, 4}∪ {5, 6, 7} = {1, 2, 3, 4, 5, 6, 7}

Dtran [B, b] = B

Dtran [B, c] = FOLLOWPOS(3) ∪ FOLLOPOS(6)

= {1, 2, 3, 4}∪ {5, 6, 7} = {1, 2, 3, 4, 5, 6, 7}

Dtran [B, c] = B

As no new state getting generated, we can construct DFA using above computations as follows.

As state B = {1, 2, 3, 4, 5, 6, 7}, i.e. It contains 7 a final state, we will consider B as final state.

Review Questions

1. *Explain subset construction method for constructing DFA from an NFA with an example* **GTU : Winter-16, Marks 7**
2. *Explain subset construction method with example.* **GTU : Summer-18, Marks 7**

2.9 Short Questions and Answers

Q.1 What are the functions of lexical analyzer ?

Ans. : 1. It produces stream of tokens.
2. It eliminates blank and comments.
3. It generates symbol table which stores the information about identifiers, constants encountered in the input.
4. It keeps track of line numbers.
5. It reports the error encountered while generating the tokens.

Q.2 What are sentinels ? What is its usage ?

Ans. : Sentinels are special string or characters that are used to indicate an end of buffer. The sentinel is not a part of source language. One can avoid a test on end of buffer using sentinels at the end of buffer. For example : 'eof'.

Q.3 What is recognizer ?

Ans. : Recognizers are machines. These are the machines which accepts the strings belonging to certain language. If the valid strings of such languages are accepted by the machine then it is said that the corresponding language is accepted by that machine, otherwise it is rejected. For example -

1. The finite state machines are recognizers for regular expressions or regular languages.
2. The push down automata are recognizers for context free languages.

Q.4 What is the role of lexical analyzer ?

Ans. : The lexical analyzer scans the source program and separates out the tokens from it.

Q.5 Give the transition diagram for an identifier.

Ans. : The regular expression for denoting identifier will be -

r.e. = letter (letter + digit)*

The transition diagram will be

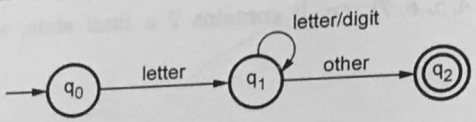


Fig. 2.9.1 Transition diagram for identifier

Q.6 What are the possible error recovery actions in lexical analyzer ?

Ans. : Various possible error recovery actions in lexical analyzer are -

i) Deleting extra characters that might appear in the source statement.

ii) Inserting missing character

iii) Interchanging two adjacent characters.

iv) Replacing an incorrect character by the correct character.

Q.7 What is the input to lexical analyzer generator ? What is its output ?

Ans. : The lexical analyzer generator takes the source program as input and generates the stream of tokens as output.

Q.8 Write the Regular expressions for identifier and number.

Ans. : 1. Regular expression for identifier is

R.E.=letter(letter+digit)*

2. Regular expression for integer number is

R.E.=digit.digit*

Q.9 Define tokens, patterns and lexemes.

Ans. : Token : It describes the class or category of input string. For example, identifiers, keywords, constants are called tokens.

Pattern : The set of rules that describe the token.

Lexemes : It represents the sequence of characters in the source program that are matched with pattern of token.

Q.10 Write the regular expression for the identifier and whitespace.

Ans. : Refer answer of Q.8 for regular expression of identifier

Regular expression for white space is

r.e. = (|\t\n)*

Q.11 Why is buffering used in lexical analysis ? What are the commonly used buffering methods ?

Ans. : The lexical analyzer reads the input source program and identifies the tokens from it. The input is read from the secondary storage. But reading the input in this way is costly. Hence the input source program are sentences are stored in the buffer. That is why buffering is required in lexical analysis.

The one buffer and two buffer schemes are the commonly used buffering methods.

Q.12 Write regular expression to describe a language consists of strings made of even number of a and b.

Ans. : r.e.=(aa+bb+(ab+ba)(aa+bb)*(ab+ba))*

2.10 Multiple Choice Questions

Q.1 Grouping of characters into tokens is done by _____.

a lexical analyser b syntax analysers

c semantic analysers d code generators

Q.2 Whether the token is formed from the given pattern or not is dependant upon _____.

a compiler b target language

c source language d operating system

Q.3 Keeping track of the line numbers is a task of _____.

a syntax analyser b lexical analyser

c semantic analyser d none of the above

Q.4 Following is a regular expression for identifier -

a letter b letter.Digit

c Digit d letter(letter+Digit)*

Q.5 Lexical analyser stores the input in _____.

a input buffer b symbol table

c linked list d none of the above

Q.6 A pattern can be represented by _____.

a context free grammar b regular grammar

c regular expression d algorithm

Q.7 Setinel is _____ .

a character b string

c indication for end of buffer d none of the above

Q.8 Input buffer is used by _____.

a lexical analyser b syntax analyser

c application Program d code generator

Q.9 The main difference between NFA and DFA is that _____ .

a the ε can be present in DFA

b the ε can be present in NFA

c the NFA leads to more than two different states for the same input symbol.

d none of the above

Q.10 Finite automata is meaningless if _____.

a there are more than two states for the same input symbol.

b there is no start state.

c if there are more than one final states.

d if there is only one state present in it

Q.11 What is true about the regular expression _____.

a it is used to represent the pattern

b it consists of or, concatenation and kleen closure.

c it is always represented by a finite automata.

d all of the above

Q.12 The two regular expressions are said to be equivalent if _____.

a they contain same number of symbols.

b if they are of the same length

c if they represent the same set of operations

d if they represent the same set of tokens

Q.13 The (0+1)*(00+11)(0+1)* means _____ .

a the strings contain even number of zeros and even number of ones

b the strings is of even length

c the strings contains at least one pair of zeros or one pair of ones.

d the strings contains exactly one pair of zeros or one pair of ones.

Q.14 Choose the correct statement

a DFA does not simulate NFA

b DFA simulates NFA

c DFA sometimes simulates NFA

d Simulation of DFA to NFA depends upon the NFA

Q.15 Which of the following regular expressions denote the language comprising of all the strings of even length over the length (a,b) ?

a (a|b)* b (a|b) (a|b)*

c (aa|ab|bb|ba)* d (a|b)(a|b)(a|b)*

Q.16 LEX produces _____.

a lexical analyser b syntax analyser

c semantic analyser d none of the above

Q.17 Regular expressions and associated actions are present in _____.

a declaration section b rule section

c procedure section d all of the above

Q.18 In LEX specification file the C statements are allowed in _____.

a Declaration section b Rule section for specifying actions

c Procedure section d All of the above

Q.19 Redeclaration of some variable is _____.

a syntactical error b lexical Error

c semantic Error d logical Error

Q.20 Lexical Error can be detected at _____.

a compile time

b run time

c both at compile time and run time

d none of the above

Q.21 In lexical analyser generator _____.

a regular expression are converted to equivalent NFA

b regular expressions are converted to equivalent regular grammar

c FA is converted to regular grammar.

d no conversion takes place

Q.22 What is true about LEX _____.

a LEX is a lexical analyser generator

b LEX is a compiler

c LEX is a database.

d LEX is a specification file used for generating token

Answer Keys for Multiple Choice Questions

Q.1	a	Q.2	c	Q.3	b	Q.4	d
Q.5	a	Q.6	c	Q.7	c	Q.8	a
Q.9	c	Q.10	b	Q.11	d	Q.12	d
Q.13	c	Q.14	b	Q.15	c	Q.16	a
Q.17	b	Q.18	d	Q.19	b	Q.20	a
Q.21	a	Q.22	a, d				

❑❑❑

3 Syntax Analysis

Syllabus

Understanding Parser and CFG(Context Free Grammars), Left Recursion and Left Factoring of grammar Top Down and Bottom up Parsing Algorithms, Operator - Precedence Parsing, LR Parsers, Using Ambiguous Grammars, Parser Generators, Automatic Generation of Parsers. Syntax - Directed Definitions, Construction of Syntax Trees, Bottom - Up Evaluation of S - Attributed Definitions, L - Attributed Definitions, syntax directed definitions and translation schemes.

Contents

3.1 Understanding Parser and CFG(Context Free Grammars)

GTU : Summer-16, Winter-20, Marks 7

The syntax analysis is the second phase in compilation. The syntax analyzer (Parser) basically checks for the syntax of the language. A syntax analyzer takes the tokens from the lexical analyzer and groups them in such a way that some programming structure (syntax) can be recognized. After grouping the tokens if at all, any syntax cannot be recognized then syntactic error will be generated. This overall process is called syntax checking of the language.

Definition of parser : A parsing or syntax analysis is a process which takes the input string w and produces either a parse tree (syntactic structure) or generates the syntactic errors.

For example :

a = b+10;

The above programming statement is first given to lexical analyzer. The lexical analyzer will divide it into group of tokens. The syntax analyzer takes the tokens as input, and generates a tree like structure called **parse tree**.

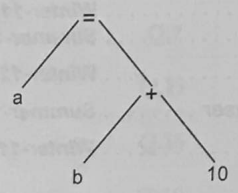


Fig. 3.1.1 Parse tree for a = b + 10

The parse tree drawn above is for some programming statement. It shows how the statement gets parsed according to their syntactic specification.

3.1.1 Role of Parser

In the process of compilation the parser and lexical analyzer work together. That means, when parser requires string of tokens it invokes lexical analyzer. In turn, the lexical analyzer supplies tokens to syntax analyzer (parser)

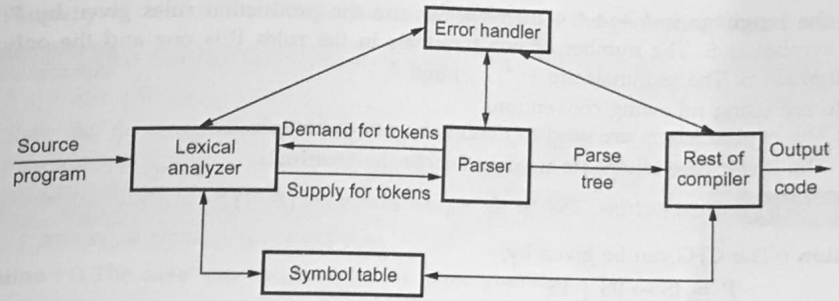


Fig. 3.1.2 Role of parser

The parser collects sufficient number of tokens and builds a parse tree. Thus by building the parse tree, parser smartly finds the syntactical errors if any. It is also necessary that the parser should recover from commonly occurring errors so that remaining task of process the input can be continued.

3.1.2 Why Lexical and Syntax Analyzer are Separated Out ?

The lexical analyzer scans the input program and collects the tokens from it. On the other hand parser builds a parser tree using these tokens. These are two important activities and these activities are independently carried out by these two phases. Separating out these two phases has two advantages - Firstly it **accelerates** the process of **compilation** and secondly the **errors** in the source input can be **identified precisely**.

3.1.3 Concept of Context Free Grammar

Definition :

The context free grammar can be formally defined as a set denoted by G = (V, T, P, S) where V and T are set of non-terminals and terminals respectively. P is set of production rules, where each production rule is in the form of

non-terminal → non-terminals

or non-terminal → terminals

S is a start symbol.

For example,

$$P = \{ S \rightarrow S + S$$
$$S \rightarrow S * S$$
$$S \rightarrow (S)$$
$$S \rightarrow 4 \}$$

If the language is 4 + 4 * 4 then we can use the production rules given by P. The start symbol is S. The number of non-terminals in the rules P is one and the only non terminal i.e. S. The terminals are +, *, (,) and 4.

We are using following conventions.

1. The capital letters are used to denote the **non-terminals.**
2. The lower case letters are used to denote the **terminals.**

Example 3.1.1 *Construct the CFG for the regular expression* $(0+1)^*$

Solution : The CFG can be given by,

$$P = \{S \rightarrow 0S \mid 1S$$
$$S \rightarrow \varepsilon \quad \}$$

The rules are in combination of 0's and 1's with the start symbol. Since $(0+1)^*$ indicates {ε, 0, 1, 01, 10, 00, 11, ...} in this set ε is a string. So in the rules we can set the rule $S \rightarrow \varepsilon$.

Example 3.1.2 *Construct a grammar for the language containing strings of atleast two a's.*

Solution : Let G = (V, T, P, S)

where V = {S, A}

T = {*a*, b}

P = {S → A*a* A*a* A

A → *a*A | bA | ε }

The rule S → A*a*A*a*A is something in which the two *a*'s are maintained since at least two *a*'s should be there in the strings. And A → *a*A |bA|ε gives any combination of *a*'s and b's i.e. this rules gives the strings of (*a* + b)*.

Thus the logic for this example will be

(any thing) *a* (any thing) *a* (any thing)

So before *a* or after *a* there could be any combination of *a*'s and b's.

Example 3.1.3 *Construct a grammar generating* $L = w\,c\,w^T$ *where* $w \in \{a, b\}^*$

Solution : The strings which can be generated for given L is {*aa*c*aa*, bcb, *a*bcb*a*, b*aca*b ...}

The grammar could be

S → *a* S *a*

S → b S b

S → c

Since the language L = w c w ^{T} where w ∈ (*a* + b)*

Hence S → *a* S *a* or S → b S b. The string abcba can be generated from given production rules as

S

a S *a*

a b S b

a b c b *a*

Thus any of this kind of string could be derived from the given production rules.

Example 3.1.4 *i) Consider the grammar*

S – > SS+ |SS|a*

Show that the string aa+a can be generated by the grammar.*

Construct the parse tree for it. Is the grammar ambiguous ? Justify.

ii) Write production rules for producing following language. Strings of 0's and 1's with equal numbers of 0's and 1's.

Solution : i) The aa+a* can be generated by given grammar

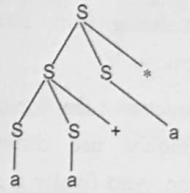


Fig. 3.1.3 Parse tree

The given grammar is not ambiguous. As unique tree is getting generated for any given string.

ii) The production rules for producing strings having equal number of 0's and equal number of 1's is -

S → 0B | 1A

A → 0 | 0S | 1AA

B → 1 | 1S | 0BB

Example 3.1.5 *Construct leftmost and rightmost derivation for the sentence abab*

S → aSbS | bSaS |ε **GTU : Winter-20, Marks 3**

Solution :

Leftmost derivation		Rightmost derivation	
S		S	
a<u>S</u>bS	∵ S → aSbS	aSb<u>S</u>	∵ S → aSbS
ab<u>S</u>aSbS	∵ S → bSaS	aSb	∵ S → ε
aba<u>S</u>bS	∵ S → ε	ab<u>S</u>a<u>S</u>b	∵ S → bSaS
abab<u>S</u>	∵ S → ε	ab<u>S</u>ab	∵ S → ε
abab	∵ S → ε	abab	∵ S → ε

Review Question

1. *Write a short note on context free grammar (CFG) explain it with suitable example.* **GTU : Summer-16, Marks 7**

3.2 Top Down and Bottom Up Parsing Algorithms

GTU : Summer-19, Marks 3

As we know, there are two parsing techniques, these parsing techniques work on the following principle.

1. The parser scans the input string from left to right and identifies that the derivation is leftmost or rightmost.
2. The parser makes use of production rules for choosing the appropriate derivation. The different parsing techniques use different approaches in selecting the appropriate rules for derivation. And finally a parse tree is constructed.

When the parse tree can be constructed from root and expanded to leaves then such type of parser is called Top-down parser. The name itself tells us that the parse tree can be built from top to bottom.

When the parse tree can be constructed from leaves to root, then such type of parser is called as bottom-up parser. Thus the parse tree is built in bottom up manner.

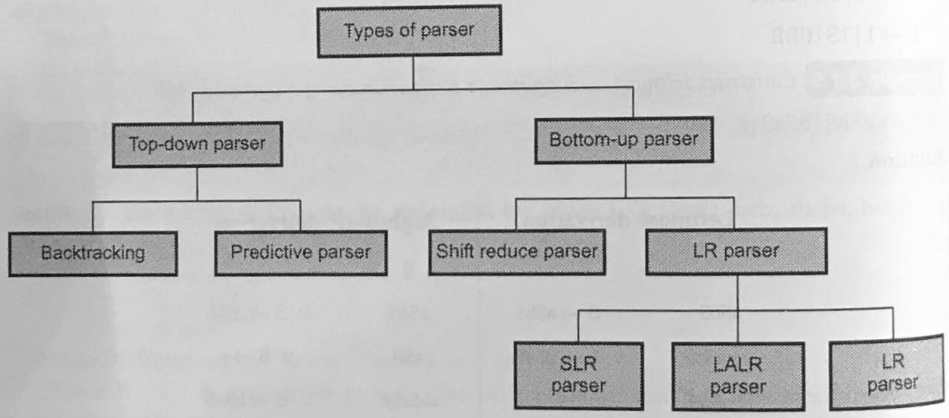


Fig. 3.2.1 Parsing techniques

Difference between Top down and Bottom up Parser

Sr. No.	Top down parser	Bottom up parser
1.	Parse tree can be built from root to leaves.	Parse tree is built from leaves to toot
2.	This is simple to implement.	This complex to to implement.
3.	This is less efficient parsing techniques. Various problems that occur during top down technique are ambiguity left recursion	When the bottom up parser handles ambiguous grammar conflicts occur in parse table.
4.	It is applicable to small class of languages.	It is applicable to a broad class of languages.
5.	Various parsing techniques are 1) Recursive descent parser 2) Predictive parser	Various parsing techniques are 1) Shift reduce 2) Operator precedence 3) LR parser.

Review Question

1. *Differentiate top down parsing and bottom up parsing* **GTU : Summer-19, Marks 3**

3.3 Top - Down Parsing

GTU : Winter-11,12,13,14,15,16,17,18,19,20, Summer-12,14,15,17,18,19, Marks 8

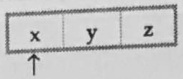
In top-down parsing the parse tree is generated from top to bottom (i.e. from root to leaves). The derivation terminates when the required input string terminates. The leftmost derivation matches this requirement. The main task in top-down parsing is to find the appropriate production rule in order to produce the correct input string. We will understand the process of top-down parsing with the help of some example.

Consider a grammar.

$$S \rightarrow xPz$$

$$P \rightarrow yw \mid y$$

Consider the input string xyz is as shown below.



Input buffer

Now we will construct the parse tree for above grammar deriving the given input string. And for this derivation we will make use of top down approach.

Step 1 :

The first leftmost leaf of the parse tree matches with the first input symbol. Hence we will advance the input pointer. The next leaf node is P. We have to expand the node P. After expansion we get the node y which matches with the input symbol y.

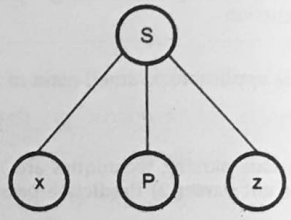


Fig. 3.3.1 (a)

Step 2 :

Now the next node is w which is not matching with the input symbol. Hence we go back to see whether there is another alternative of P. The another alternative for P is y which matches with current input symbol. And thus we could produce a successful parse tree for given input string.

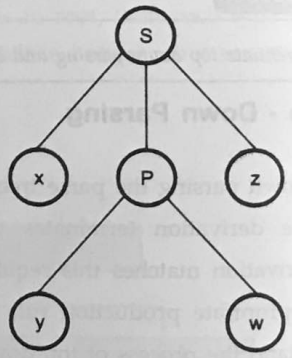


Fig. 3.3.1 (b)

Step 3 : We halt and declare that the parsing is completed successfully.

In top-down parsing selection of proper rule is very important task. And this selection is based on trial and error technique. That means we have to select a particular rule and if it is not producing the correct input string then we need to backtrack and then we have to try another production. This process has to be repeated until we get the correct input string. After trying all the productions if we found every production unsuitable for the string match then in that case the parse tree cannot be built.

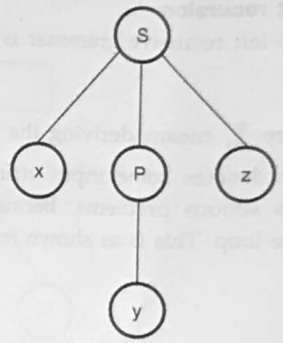


Fig. 3.3.1 (c)

3.3.1 Problems with Top - Down Parsing

There are certain problems in top-down parsing. In order to implement the parsing we need to eliminate these problems. Let us discuss these problems and how to remove them. These problem are (1) Backtracking (2) Left Recursion (3) Left factoring (4) Ambiguity.

1) Backtracking

Backtracking is a technique in which for expansion of non-terminal symbol we choose one alternative and if some mismatch occurs then we try another alternative if any.

For example :

$S \rightarrow xPz$

$P \rightarrow yw \mid y$

Then

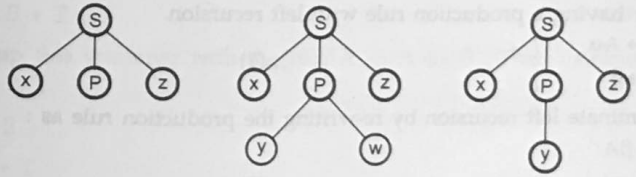


Fig. 3.3.2

If for a non-terminal there are multiple production rules beginning with the same input symbol then to get the correct derivation we need to try all these alternatives. Secondly, in backtracking we need to move some levels upward in order to check the possibilities. This increases lot of overhead in implementation of parsing. And hence it becomes necessary to eliminate the backtracking by modifying the grammar.

2) Left recursion

The left recursive grammar is a grammar which is as given below.

$$A \overset{+}{\Rightarrow} A\alpha$$

Here $\overset{+}{\Rightarrow}$ means deriving the input in one or more steps. The A can be non-terminal and α denotes some input string. If left recursion is present in the grammar then it creates serious problems. Because of left recursion the top-down parser can enter in infinite loop. This is as shown in the Fig. 3.3.3.

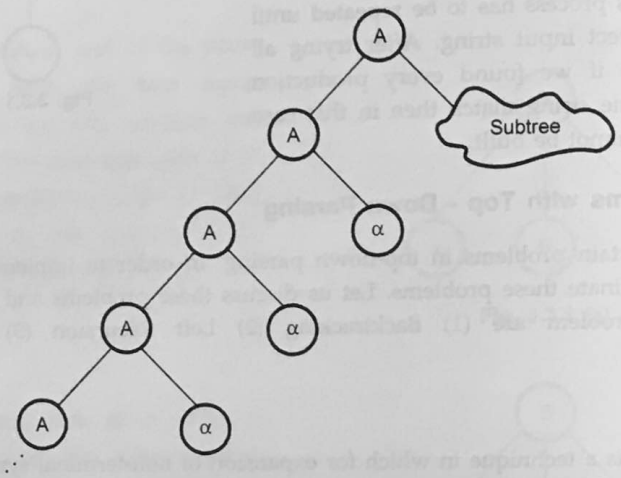


Fig. 3.3.3 Left recursion

Thus expansion of A causes further expansion of A only and due to generation of A, Aα, Aαα, Aααα, ..., the input pointer will not be advanced. This causes major problem in top-down parsing and therefore elimination of left recursion is a must.

To eliminate left recursion we need to modify the grammar. Let, G be a context free grammar having a production rule with left recursion.

$$\left.\begin{array}{l} A \rightarrow A\alpha \\ A \rightarrow \beta \end{array}\right\} \quad \ldots(1)$$

Then we eliminate left recursion by re-writing the production rule as :

$$\left.\begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' \\ A' \rightarrow \varepsilon \end{array}\right\} \quad \ldots(2)$$

Thus a new symbol A' is introduced. We can also verify, whether the modified grammar is equivalent to original or not.

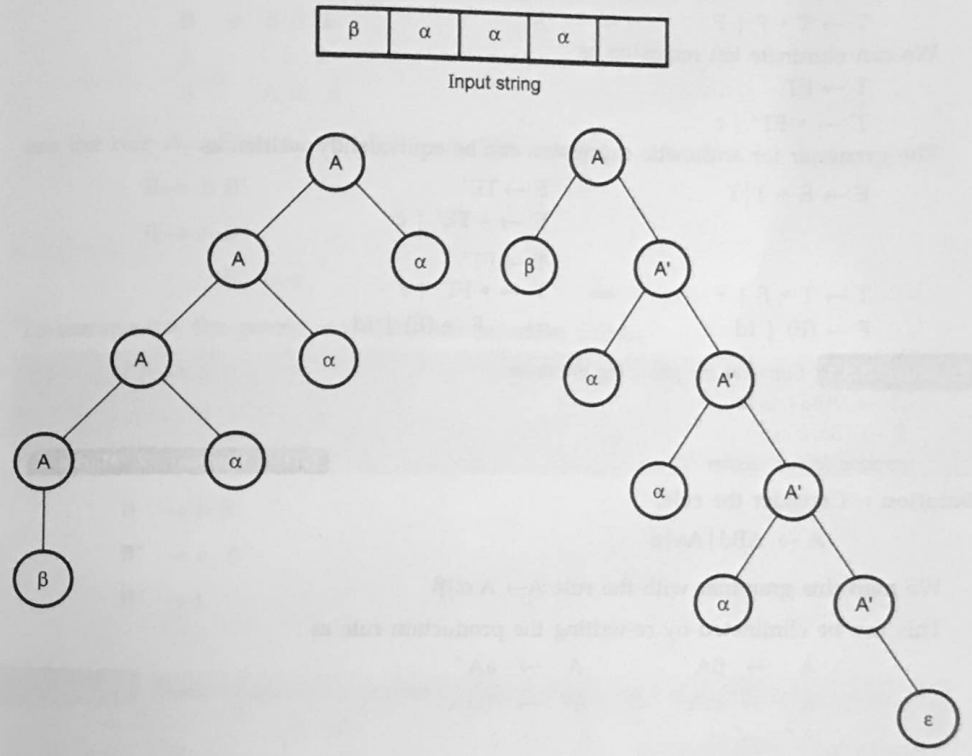


Fig. 3.3.4

For example : Consider the grammar

$$E \rightarrow E + T \mid T$$

We can map this grammar with the rule $A \rightarrow A\alpha \mid \beta$. Then using equation (2) we can say,

$A = E$

$\alpha = + T$

$\beta = T$

Then the rule becomes,

$$E \rightarrow TE'$$

$$E' \rightarrow + TE' \mid \varepsilon$$

Similarly for the rule,

$T \rightarrow T * F \mid F$

We can eliminate left recursion as

$T \rightarrow FT'$

$T' \rightarrow * FT' \mid \varepsilon$

The grammar for arithmetic expression can be equivalently written as -

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

$\Rightarrow$

$E \rightarrow TE'$

$E' \rightarrow + TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow * FT' \mid \varepsilon$

$\Rightarrow \quad F \rightarrow (E) \mid id$

Example 3.3.1 *Consider the following grammar*

$A \rightarrow ABd \mid Aa \mid a$

$B \rightarrow Be \mid b$

remove left recursion

GTU : Summer-12, Marks 3

Solution : Consider the rule,

$A \rightarrow ABd \mid Aa \mid a$

We map this grammar with the rule $A \rightarrow A\alpha \mid \beta$

This can be eliminated by re-writing the production rule as :

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \Rightarrow$

$A' \rightarrow \varepsilon$

$A \rightarrow aA'$

$A' \rightarrow BdA'$

$A' \rightarrow \varepsilon$

For $A \rightarrow Aa \mid a$

$\downarrow \quad \downarrow\downarrow\downarrow$

$A \quad A\alpha\beta$

This left recursion can be eliminated as -

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \Rightarrow$

$A' \rightarrow \varepsilon$

$A \rightarrow aA'$

$A' \rightarrow aA'$

$A' \rightarrow \varepsilon$

To summarize

$A \rightarrow ABd \mid Aa \mid a \Rightarrow A \rightarrow aA'$

$A' \rightarrow BdA' \mid aA'$

$A' \rightarrow \varepsilon$

$B \rightarrow Be \mid b$

We map this grammar with the rule $A \rightarrow A\alpha \mid \beta$.

$B \rightarrow B\,e \mid b$

$\downarrow \quad \downarrow\downarrow \downarrow$

$A \quad A\,\alpha\,\beta$

use the rule $A \rightarrow \beta A'$, $A \rightarrow \alpha A'$, $A' \rightarrow \varepsilon$. Then we get

$B \rightarrow b\,B'$

$B \rightarrow e\,B'$

$B' \rightarrow \varepsilon$

To summarize, the grammar without left recursion will be

$A \rightarrow a\,A'$

$A' \rightarrow B\,d\,A' \mid a\,A'$

$A' \rightarrow \varepsilon$

$B \rightarrow b\,B'$

$B' \rightarrow e\,B'$

$B' \rightarrow \varepsilon$

Example 3.3.2 *Eliminate left recursion from the following grammer and rewrite the grammar.*

S–> Aa|b

A–> Ac|Sd|ε

GTU : Summer-12, Marks 3

Solution :

Let,

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid \varepsilon$

be the given grammar. If we replace S by Aa|b in the rule $A \rightarrow Sd$ then the grammar becomes.

$A \rightarrow Ac \mid Aad \mid bd \mid \varepsilon$

This rule is left recursive. We will map $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \beta_1 \mid \beta_2$ to this rule.

This gives $A \rightarrow \beta_1 A' \mid \beta_2 A'$ and $A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \varepsilon$

$A \rightarrow bdA' \mid A'$

$A \rightarrow cA' \mid adA' \mid \varepsilon$

Hence after eliminating the left recursion the grammar becomes

$S \rightarrow Aa \mid b$

$A \rightarrow bdA' \mid A'$

$A' \rightarrow cA' \mid adA' \mid \varepsilon$

Example 3.3.3 *Define : Left Recursive state the rule to remove left recursive from the grammar. Eliminate left recursive from following grammar.*

$S \rightarrow A\,a \mid b$

$A \rightarrow Ac \mid Sd \mid f$

GTU : Winter-15, Marks 7

Solution : The left recursive grammar is a grammar which is as given below :

$$A \overset{+}{\Rightarrow} A\alpha$$

To eliminate left recursion we apply following rule.

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid\mid \beta_1 \mid \beta_2$ is a rule then

$A \rightarrow \beta_1 A' \mid \beta_2 A'$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \varepsilon$

If we replace S by Aa|b then there exists only one rule

$A \rightarrow Ac \mid Aad \mid bd \mid f$

After eliminating left recursion the grammar becomes

$A \rightarrow bdA' \mid fA'$

$A \rightarrow cA' \mid adA' \mid \varepsilon$

To summerize

$S \rightarrow Aa \mid b$

$A \rightarrow bdA' \mid fA'$

$A' \rightarrow cA' \mid adA' \mid \varepsilon$

Example 3.3.4 *Write rule(s) to check grammar is left recursive or not. Remove left recursive from the following grammar*

$S \rightarrow aBDh$

$B \rightarrow Bb \mid c$

$D \rightarrow EF$

$E \rightarrow g \mid \varepsilon$

$F \rightarrow f \mid \varepsilon$

GTU : Winter-20, Marks 4

Solution : The rule for removal of left recursion is as follows -

If $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \beta_1 \mid \beta_2$ is a left recursive rule then, to eliminate left recursion,

$A \rightarrow \beta_1 A' \mid \beta_2 A'$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \varepsilon$

From given grammar, only one non-terminal i.e. B gives left recursive grammar =

$B \rightarrow Bb \mid c$

We will map the above production rule with left recursive rule as

B	→	B	b	|	c
↓		↓	↓	↓	
A		A	α_1	β_1	

For eliminating left recursion

$B \rightarrow cB'$

$B' \rightarrow bB'$

To summerize, the grammar on eliminating recursion,

$S \rightarrow aBDh$

$B \rightarrow cB'$

$B' \rightarrow bB'$

$D \rightarrow EF$

$E \rightarrow g \mid \varepsilon$

$F \rightarrow f \mid \varepsilon$

3) Left factoring

If the grammar is left factored then it becomes suitable for the use. Basically left factoring is used when it is not clear that which of the two alternatives is used to expand the non-terminal. By left factoring we may be able to re-write the production in which the decision can be deferred until enough of the input is seen to make the right choice.

In general if

$$A \rightarrow \alpha\beta_1 | \alpha\beta_2$$

is a production then it is not possible for us to take a decision whether to choose first rule or second. In such a situation the above grammar can be left factored as

$$A \rightarrow \alpha A'$$
$$A \rightarrow \beta_1 | \beta_2$$

For example : Consider the following grammar.

$$S \rightarrow iEtS \mid iEtSeS \mid a$$
$$E \rightarrow b$$

The left factored grammar becomes,

$$S \rightarrow iEtSS' \mid a$$
$$S' \rightarrow eS \mid \varepsilon$$
$$E \rightarrow b$$

Example 3.3.5 *Do left factoring in the following grammar -*
$A \rightarrow aAB \mid aA \mid a$
$B \rightarrow bB \mid b$

Solution : If the rule is $A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \ldots$ is a production then the grammar needs to be left factored. Consider

$$A \rightarrow a\,A\,B \mid a\,A \mid a$$
$$\downarrow\ \downarrow \quad \downarrow\downarrow \ \downarrow\downarrow$$
$$\alpha\ \beta_1 \quad \alpha\beta_2 \ \alpha\beta_3$$

We have to convert it to

$$A \rightarrow \alpha A \qquad A \rightarrow \alpha A'$$
$$\Rightarrow$$
$$A' \rightarrow \beta_1 | \beta_2 \qquad A' \rightarrow AB \mid A \mid \varepsilon$$

$$B \rightarrow b\ B \mid b$$
$$\downarrow\ \downarrow\ \downarrow\ \downarrow \qquad \Rightarrow \quad B \rightarrow bB'$$
$$A\ \ \alpha\ \ \beta_1\ \alpha\ \ \beta_2 \qquad\qquad B' \rightarrow B \mid \varepsilon$$

Similarly

To summarize, the grammar with left factor operation will be -

$$A \rightarrow a\,A'$$
$$A' \rightarrow AB \mid A \mid \varepsilon$$
$$B \rightarrow b\,B'$$
$$B' \rightarrow B \mid \varepsilon$$

Example 3.3.6 *Perform the left factoring of following grammar*
$A \rightarrow ad \mid a \mid ab \mid abc \mid b$

Solution : For left factoring the rule is if $A \rightarrow \alpha\beta_1 | \alpha\beta_2$ is a production then by left factoring we get,

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 | \beta_2$$

Consider

Then

$$A \rightarrow aA'$$
$$A' \rightarrow d \mid b \mid bc \mid \varepsilon$$

$$A \rightarrow a\ d \mid a \mid ab \mid a\ bc \mid b$$
$$\alpha \quad \beta_1\ \beta_2\ \beta_3\ \beta_4$$

Hence we get

$$A \rightarrow aA \mid 'b$$
$$A \rightarrow d \mid b \mid bc \mid \varepsilon$$

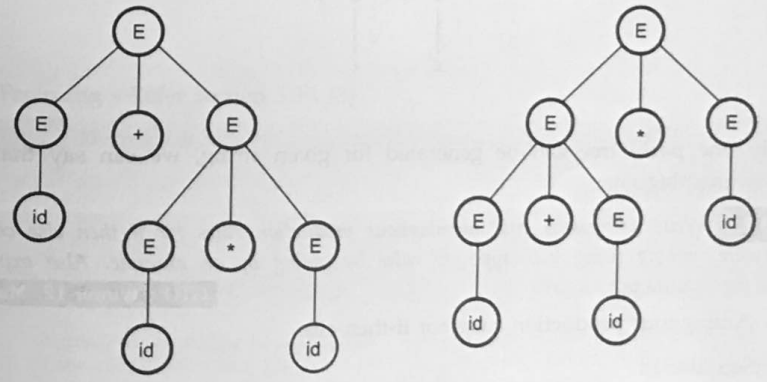
4) Ambiguity

The ambiguous grammar is not desirable in top-down parsing. Hence we need to remove the ambiguity from the grammar if it is present.

For example :

$$E \rightarrow E + E \mid E * E \mid id$$

is an ambiguous grammar. We will design the parse tree for id + id * id as follows.



(a) Parse tree 1 (b) Parse tree 2

Fig. 3.3.5 Ambiguous grammar

For **removing the ambiguity** we will apply one rule : If the grammar has left associative operator (such as +, −, *, /) then induce the left recursion and if the grammar has right associative operator (exponential operator) then induce the right recursion.

The **unambiguous grammar** is

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow id$

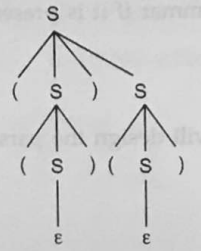
Note one thing that the grammar is unambiguous but it is left recursive and elimination of such left recursion is again a must.

Example 3.3.7 *Check whether following grammar is ambiguous or not*

$S \rightarrow (S)S$

$S \rightarrow \varepsilon$

Solution : Consider the input string (( ))( ) for which we will draw the parse tree.



As only one parse tree can be generated for given string, we can say that given grammar is unambiguous.

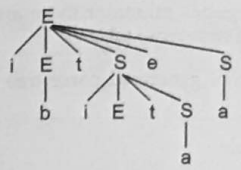
Example 3.3.8 *Write ambiguous and unambiguous production rules for if then else construct. Illustrate parsing using both types of rules by giving by an example. Also explain left factoring and its use.* **GTU : Winter-12, Marks 7**

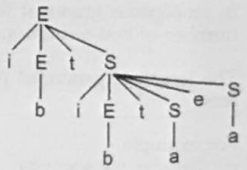
Solution : Ambiguous production rules for if-then-else

$S \rightarrow iEtSeS \mid iEtS \mid a$

$E \rightarrow b$

Consider the string **ibtibtaea** can be parsed. Following parse trees show the parsing of this string.





Unambiguous production rules are as given below. The ambiguity arise due to use only one non terminal i.e. S. We will use non terminal A that denotes match statement and B denotes unmatched statement. The unambiguous grammar will be -

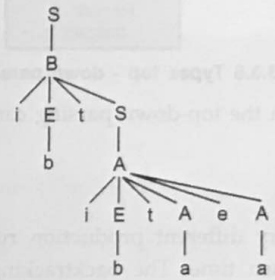
$S \rightarrow A \mid B$

$A \rightarrow iEtAeA \mid a$

$B \rightarrow iEtS \mid iEtAeB$

$E \rightarrow b$

The parsing of string ibtibtaea



Left Factoring - Refer section 3.3.1 (3)

Example 3.3.9 *Distinguish between ambiguous and unambiguous grammar ?* **GTU : Winter-19, Marks 4**

Solution :

Sr. No.	Ambiguous Grammar	Unambiguous Grammar
1)	A grammar is said to be ambiguous if for atleast one string generated by it, it produces more than one parse tree.	A grammar is said to be unambiguous if for all the strings generated by it, it produces exactly one parse tree.

2)	In ambiguous grammar there are less number of non-terminals.	In unambiguous grammar there are more number of non-terminals.
3)	The length of generated parse tree is less.	The length of generated parse tree is large.
4)	For example - $E \rightarrow E + E \mid E * E \mid id$.	For example - $E \rightarrow E+T$ $E \rightarrow T$ $T \rightarrow T*F$ $T \rightarrow F$ $F \rightarrow id$

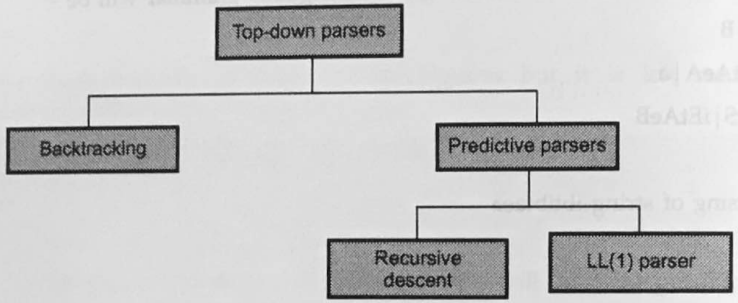


Fig. 3.3.6 Types top - down parsers

There are two types by which the top-down parsing can be performed.

1. Backtracking
2. Predictive parsing

A backtracking parser will try different production rules to find the match for the input string by backtracking each time. The backtracking is powerful than predictive parsing. But this technique is slower and it requires exponential time in general. Hence backtracking is not preferred for practical compilers.

As the name suggests the predictive parser tries to predict the next construction using one or more lookahead symbols from input string. There are two types of predictive parsers :

1. Recursive descent
2. LL(1) parser.

Let us discuss these types along with some examples.

3.3.2 Recursive Descent Parser

A parser that uses collection of recursive procedures for parsing the given input string is called **Recursive Descent (RD) Parser**. In this type of parser the CFG is used to

build the recursive routines. The R.H.S. of the production rule is directly converted to a program. For each non-terminal a separate procedure is written and body of the procedure (code) is R.H.S. of the corresponding non-terminal.

Basic steps for construction of RD parser

The R.H.S. of the rule is directly converted into program code symbol by symbol.

1. If the input symbol is non-terminal then a call to the procedure corresponding to the non-terminal is made.
2. If the input symbol is terminal then it is matched with the lookahead from input. The lookahead pointer has to be advanced on matching of the input symbol.
3. If the production rule has many alternates then all these alternates has to be combined into a single body of procedure.
4. The parser should be activated by a procedure corresponding to the start symbol.

Let us take one example to understand the construction of RD parser. Consider the grammar having start symbol E.

$E \rightarrow num\ T$

$T \rightarrow *\ num\ T \mid \varepsilon$

```
procedure E
{
   if lookahead = num then
   {
      match(num) ;
      T;                        /* call to procedure T */
   }
   else
      error;
   if lookahead = $
   {
      declare success;   /* Return on success */
   }
   else
      error;
}/*end of procedure E*/
procedure T
{
   if lookahead = '*'
   {
      match('*');
      if lookahead ='num'
      {
         match(num);
```

```
        T;
      }                    /* inner if closed*/
      else
          error
   }                       /* outer if closed*/
   else NULL               /*indicates ε Here the other alternate is
                                combined into same procedure T */
                           /* end of procedure T*/
}
procedure match( token t )
{
   if lookahead=t
   lookahead = next_token;      /*lookahead pointer is advanced*/
   else
   error
}                                /*end of procedure match*/
procedure error
{
   print("Error!!!");
}                                /*end of procedure error*/
```

Fig. 3.3.7 Pseudo code for recursive descent parser

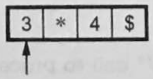
The parser constructed above is recursive descent parser for the given grammar. We can see from above code that the procedures for the non-terminals are written and the procedure body is simply mapping of R.H.S. of the corresponding rule.

Consider that the input string is 3*4. We will parse this string using above given procedures.

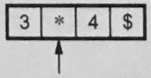
The parser will be activated by calling procedure E. Since the first input character 3 is matching with num the procedure match (num) will be invoked and then the lookahead will point to next token. And a call to procedure T is given.

A match with '*' is found hence lookahead = next_token.

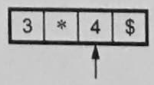
Now '4' is matching with num hence again the procedure for match (num) is fulfilled. Then procedure for T is invoked. And T is replaced by ε.



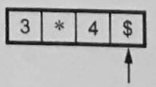
E→ num T



T→ * num T



T→ *num T



Declare Success !

As lookahead pointer points to $ we quit by reporting success. Thus the input string can be parsed using recursive descent parser.

Construction of recursive descent parser is easy. However the programming language that we choose for RD parser should support recursion. The internal details are not accessible in this type of parser. For instance; in this parser we cannot access the current leftmost sentential form. Secondly at any instant, we cannot access the stack containing recursive calls.

Writing procedures for left recursive grammar $(A \rightarrow \alpha\beta_1 | \alpha\beta_2)$ is crucial. And to make such procedure simpler first we have to left factor the grammar. We cannot write recursive descent parsers for all types context free grammar.

Example 3.3.10 *Consider the following grammar*

$E \rightarrow T+E \mid T$

$T \rightarrow V*T \mid V$

$V \rightarrow id$

Write down procedures for nonterminals of the grammar to make a recursive descent parser.

Solution :

```
Procedure  E ( )
   {
      T ( );
      if ( lookahead = '+')
      {
          match('+');
          E ( );
      }
      else
          error ( );
      if (lookahead = '$')  { declare SUCCESS ;}
Procedure T ( )
   {
      V ( );
      if (lookahead = '*')
      {
          match ('*');
          T ( );
      }
      else
          error ( );
   }
Procedure V ( )
   {
      if (lookahead = 'id')
          match ('id');
      else
```

```
            error ( );
    }
Procedure match (token t)
{
    if (lookahead = t)
    lookahead = next_token;
    else
        error;
}
Procedure error ( )
{
        Print (" Error ! ");
}
```

Example 3.3.11 *Write down C program for recursive descent parser for :*

$S \rightarrow ABC$ $B \rightarrow 1B|\wedge$

$A \rightarrow 0A1|\wedge$ $C \rightarrow 1C0|\wedge$

GTU : Winter-13, Marks 7

Solution :

```
procedure S ()
{
    A ();
    B () ;
    C () ;
}
procedure A ()
{
    if (lookahead = '0')
    {
        match ('0') ;
        A ();
        if (lookahead = '1')
        match ('1') ;
    }
    else
        return ;
}
procedure B ()
{
    if (lookahead = '1')
    {
        match ('1') ;
        B () ;
    }
    else
        return ;
}
```



```
procedure C ()
{
    if (lookahead = '1')
    {
        match ('1') ;
        C () ;
        if (lookahead = '0')
            match ('0') ;
    }
    else
        return ;
}
procedure  match (token t )
{
    if (lookahead = t)
      lookahead = next_ token ;
    else
        error ;
```

Example 3.3.12 *Implement the following grammar using recursive descent parser.*

S-> Aa | bAc| bBa, A-> d,B -> d **GTU : Summer-14, Marks 8, Winter-18, Marks 7**

Solution :

```
Procedure S( )
        {
            if (lookahead = 'b')
            {
                B( ) ;          // or A( ) because both gives 'd'
                if (lookahead  = 'c')
                    match ('c')
                else if (lookahead = 'a')
                        match ('a')
                else if (loohahead = '$')
                {
                    declare SUCCESS ;
                }
            else
            {
                A( ) ;
                    if (lookahead = 'a')
                        match ('a') ;
            }
        }
        Procedure A( )
        {
            if (lookahead = 'd')
                match ('d')
```

```
        else
            error ( );
    }
    Procedure B( )
    {
        if (lookahead = 'd')
            match ('d')
        else
            error( );
    }
```

Example 3.3.13 *Construct recursive descent parser for following grammar.*

$E \rightarrow T\ A$

$A \rightarrow +\ T\ A$

$A \rightarrow \varepsilon$

$T \rightarrow F\ B$

$B \rightarrow *\ F\ B$

$B \rightarrow \varepsilon$

$F \rightarrow (E)$

$F \rightarrow id$

GTU : Summer-19, Marks 7

Solution : The routines for recursive descent parser is

```
E()
{
        T();
        A();
}
A()
{
        iF (lookahead =='+')
        {
            match (' + ')
            T();
            A();
        }
    else
return ;
}
T()
{
        F();
```



```
        B();
}
B()
{
        if (lookahead == '*')
        {
                macth ('*') ;
                    F();
                    B();
        }
            else
                return;
}
F()
{
        if (lookahead = = '(' )
        {
                match (' (' );
                E () ;
            if (lookahead = = ')' )
            match ( ' ) ' ) ;
        }
        else
            match ('id');
}
    match (char   t )
{
        if (lookahead = = t)
                lookhead = getchar();
            else
                    printf("Error") ;
}
main ()
{
        E() ;
            if (lookahead == $)
            Printf (" Parsing successful");
```

Advantages of recursive descent parser

1. Recursive descent parsers are simple to build.
2. Recursive descent parsers can be constructed with the help of parse tree.

Limitations of recursive descent parser

1. Recursive descent parsers are not very efficient as compared to other parsing techniques.
2. There are chances that the program for RD parser may enter in an infinite loop for some input.
3. Recursive descent parser can not provide good error messaging
4. It is difficult to parse the string if lookahead symbol is arbitrarily long.

3.3.3 Predictive LL(1) Parser

- This top-down parsing algorithm is of non-recursive type.
- In this type of parsing a **table is built**.
- For LL(1) - the **first L** means the input is scanned from left to right. The **second** L means it uses **leftmost derivation** for input string. And the **number 1** in the input symbol means it uses only one **input symbol (lookahead)** to predict the parsing process.
- The simple block diagram for LL(1) parser is as shown in Fig. 3.3.8.

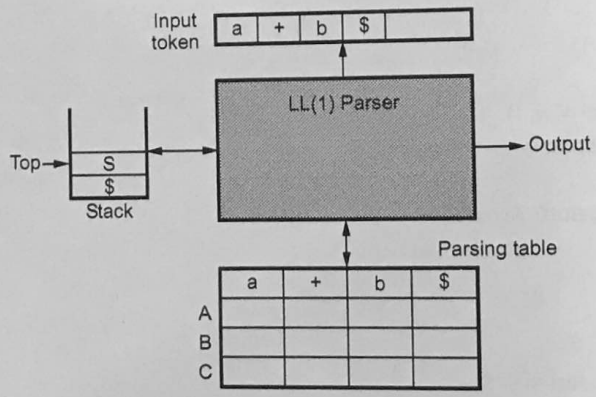


Fig. 3.3.8 Model for LL(1) parser

- The data structures used by LL(1) are **i) input buffer ii) stack iii) parsing table.**
- The LL(1) parser uses **input buffer** to store the input tokens. The **stack** is used to hold the left sentential form.The symbols in R.H.S. of rule are pushed into the stack **in reverse order** i.e. from right to left. Thus use of stack makes this algorithm non-recursive. The **table** is basically a two dimensional array. The table has row for non-terminal and column for terminals. The table can be represented as M[A,a] where A is a non-terminal and a is current input symbol.

Working

- The parsing program reads top of the stack and a current input symbol. With the help of these two symbols the parsing action is determined. The parsing actions can be

Top of stack	Input token	Parsing action
$	$	Parsing successful, halt.
a	A	Pop a and advance lookahead to next token.
a	B	Error.
A	a	Refer table M[A,a] if entry at M[A,a] is error report Error.
A	a	Refer table M[A,a] if entry at M[A,a] is A → PQR then pop A then push R, then push Q, then push P.

- The parser consults the table M[A,a] each time while taking the parsing actions hence this type of parsing method is called **table driven parsing algorithm.**
- The configuration of LL(1) parser can be defined by **top of the stack** and a **lookahead token.**
- One by one configuration is performed and the input is successfully parsed if the parser reaches the halting configuration.
- When the stack is empty and next token is $ then it corresponds to successful parse.

3.3.3.1 Construction of Predictive LL(1) Parser

The construction of predictive LL(1) parser is based on two very important functions and those are **FIRST** and **FOLLOW.**

For **construction** of Predictive LL(1) parser we have to follow the following steps -

Step 1 : Computation of FIRST and FOLLOW function.

Step 2 : Construct the Predictive Parsing Table using FIRST and FOLLOW functions.

Step 3 : Parse the input string with the help of Predictive Parsing Table.

FIRST function

FIRST(α) is a set of terminal symbols that are first symbols appearing at R.H.S. in derivation of α. If $\alpha \Rightarrow \varepsilon$ then ε is also in FIRST (α).

Following are the rules used to compute the FIRST functions.

1. If the terminal symbol **a** the FIRST(a) = {a}.
2. If there is a rule $X \rightarrow \varepsilon$ then FIRST(X) = {ε}.
3. For the rule $A \rightarrow X_1 X_2 X_3 \ldots X_k$ FIRST(A) = (FIRST(X_1) ∪ FIRST(X_2) ∪ FIRST(X_3)... ∪ FIRST(X_k).
   Where k $X_j \leq n$ such that $1 \leq j \leq k-1$.

FOLLOW function

FOLLOW (A) is defined as the set of terminal symbols that appear immediately to the right of A. In other words

FOLLOW(A) = { a | $S \overset{*}{\Rightarrow} \alpha$ Aa β where α and β are some grammar symbols may be terminal or non-terminal.

The rules for computing FOLLOW function are as given below -

1. For the start symbol **S** place $ in FOLLOW(S).
2. If there is a production $A \rightarrow \alpha B \beta$ then everything in FIRST(β) without ε is to be placed in FOLLOW(B).
3. If there is a production $A \rightarrow \alpha B \beta$ or $A \rightarrow \alpha B$ and FIRST(β)= {ε} then FOLLOW(A) = FOLLOW(B) or FOLLOW(B)=FOLLOW(A). That means everything in FOLLOW(A) is in FOLLOW(B).

Let us take some examples to compute FIRST and FOLLOW functions using above rules.

Example 3.3.14 *Consider grammar*

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid id$

Find the FIRST and FOLLOW functions for the above grammar.

Solution :

$E \rightarrow TE'$ is a rule in which the first symbol at R.H.S. is T. Now $T \rightarrow FT'$ in which the first symbol at R.H.S. is F and there is a rule for F as $F \rightarrow (E) \mid id$.

$\therefore$ FIRST(E) = FIRST(T) = FIRST(F)

As $F \rightarrow (E)$

$F \rightarrow id$

Hence FIRST(E) = FIRST(T) = FIRST(F) = { (,id }

$T \rightarrow FT' \longrightarrow$ replace F by R.H.S. rule
$T \rightarrow (E)T' \mid idT'$

This can be added in FIRST (T)

FIRST(E′) = (+, ε)

As $E' \rightarrow +TE'$

$E' \rightarrow \varepsilon$ by referring computation rule 2

The first symbol appearing at R.H.S. of production rule for E′ is added in the FIRST function.

FIRST(T′)=(*, ε)

As $T' \rightarrow * TE'$

$E' \rightarrow \varepsilon$

The first terminal symbol appearing at R.H.S. of production rule for T′ is added in the FIRST function. Now we will compute FOLLOW function.

FOLLOW(E) -

i) As there is a rule $F \rightarrow (E)$ the symbol ')' appears immediately to the right of E.
Hence ')' will be in FOLLOW(E).

ii) The computation rule is $A \rightarrow \alpha B \beta$ we can map this rule with $F \rightarrow (E)$ then A = F, α = (, B = E, β =).

FOLLOW(B) = FIRST(β) = FIRST( )) = {) }
FOLLOW(E) = {)}

Since E is a start symbol, add $ to follow of E.

Hence FOLLOW(E) = {),$}

FOLLOW(E′) -

i) $E \rightarrow TE'$ the computational rule is $A \rightarrow \alpha B \beta$.

$\therefore$ A = E′, α = T, B = E′, β = ε then by computational rule 3 everything in FOLLOW(A) is in FOLLOW(B) i.e. everything in FOLLOW(E) is in FOLLOW(E′).

$\therefore$ FOLLOW(E′) = {), $ }

ii) E′→ +TE′ the computational rule is A →α Bβ.

∴ A = E′, α = +T, B = E′, β=ε then by computational rule 3 everything in FOLLOW(A) is in FOLLOW(B) i.e. everything in FOLLOW(E′) is in FOLLOW(E′).

FOLLOW(E′) = {),$ }

We can observe in the given grammar that) is really following E.

FOLLOW(T) -

We have to observe two rules

E → TE′

E′ → +TE′

i) Consider

E→ TE′ we will map it with A →α B β

A = E, α = ε, B = T, = β = E' by computational rule 2 FOLLOW(B) = {FIRST (β) – ε}.

That is FOLLOW(T) = {FIRST(E′) – ε}

= {{+,ε}–ε}

= {+}

ii) Consider E′→ +TE′ we will map it with A →α Bβ

A = E′, α = +, B = T, β = E′ by computational rule 3 FOLLOW(A) = FOLLOW(B) i.e. FOLLOW(E′) = FOLLOW(T)

FOLLOW(T) = {),$}

Finally FOLLOW(T) = {+} ∪ {),$}

FOLLOW(T) = {+ ,), $}

We can observe in the given grammar that + and) are really following T.

FOLLOW(T′) -

T → FT′

We will map this rule with A →α Bβ then A=T, α=F, B=T′, β=ε then FOLLOW(T′) = FOLLOW(T) = {+,),$}

T → *FT′

We will map this rule with A →α Bβ then A=T, α=*F, B=T′, β= ε′ then FOLLOW(T′) = FOLLOW(T) = {+,),$}

Hence FOLLOW(T′) = {+,),$}

FOLLOW(F) -

Consider T → FT′ or T′ → *FT′ then by computational rule 2,

T → FT′
A →α Bβ
A = T, α = ε, B = F, β = T′
FOLLOW(B) = {FIRST (β) – ε}
FOLLOW(F) = {FIRST(T′) – ε}
FOLLOW(F) = {*}

T′ → *FT′
A →α Bβ
A = T′, α = *, B = F, β = T′
FOLLOW(B) = {FIRST(β) – ε}
FOLLOW(F) = {FIRST(T′) – ε}
FOLLOW(F) = {*}

Consider T′→ *FT′ by computational rule 3

T' → *FT′
A →α Bβ
A = T', α =*, B = F, β = T′
FOLLOW(A) = FOLLOW (B)
FOLLOW(T') = FOLLOW(F)
Hence **FOLLOW(F) = {+,),$}**

Finally FOLLOW(F) = {*} ∪ {+,),$}

FOLLOW(F) = {+,*,),$)

To summarize above computation

Symbols	FIRST	FOLLOW
E	{(,id}	{),$}
E'	{+,ε}	{),$}
T	{(,id}	{+,),$}
T'	{*,ε}	{+,),$}
F	{(,id}	{+,*,),$}

Algorithm for predictive parsing table -

The construction of predictive parsing table is an important activity in predictive parsing method. This algorithm requires FIRST and FOLLOW functions.

Input : The Context Free Grammar G.

Output : Predictive Parsing table M.

Algorithm :

For the rule $A \rightarrow \alpha$ of grammar G

1. For each a in FIRST(α) create entry M[A,a] = $A \rightarrow \alpha$ where a is terminal symbol.
2. For ε in FIRST(α) create entry M[A,b] = $A \rightarrow \alpha$
   Where b is the symbols from FOLLOW(A).
3. If ε is in FIRST(α) and $ is in FOLLOW(A) then create entry in the table M[A, $] = $A \rightarrow \alpha$.
4. All the remaining entries in the table M are marked as SYNTAX ERROR.

Example 3.3.15 *Construct predictive parsing table for following grammar.*

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid id$

Solution : We have computed FIRST and FOLLOW of all the non-terminal symbols.

Symbols	FIRST	FOLLOW
E	{(,id}	{),$}
E'	{+,ε}	{),$}
T	{(,id}	{+,),$}
T'	{*,ε}	{+,),$}
F	{(,id}	{+,*,),$}

Create the table as follows.

	id	+	*	(	)	$
E						
E'						
T						

T'						
F						

Now we will fill up the entries in the table using the above given algorithm. For that consider each rule one by one.

Step 1 : $E \rightarrow TE'$

$A \rightarrow \alpha$

$A = E, \alpha = TE'$

FIRST(TE') if $E' = \varepsilon$ then FIRST(T) = {(,id}

M[E, (] = $E \rightarrow TE'$

M[E, id] = $E \rightarrow TE'$

Step 2 : $E' \rightarrow +TE'$

$A \rightarrow \alpha$

$A = E', \alpha = +TE'$

FIRST(+TE') = {+ }

Hence M[E,+] = $E' \rightarrow +TE'$

Step 3 : $E' \rightarrow \varepsilon$

$A \rightarrow \alpha$

$A = E', \alpha = \varepsilon$ then

FOLLOW(E') = {),$}

M[E',)] = $E' \rightarrow \varepsilon$

M[E, $] = $E' \rightarrow \varepsilon$

Step 4 : $T \rightarrow FT'$

$A \rightarrow \alpha$

$A = E', \alpha = FT'$

FIRST(FT') = FIRST(F) = {(,id}

Hence M[F,(] = $T \rightarrow FT'$

And M[F,id] = $T \rightarrow FT'$

Step 5 : $T \rightarrow *FT'$

$A \rightarrow \alpha$

$A = T, \alpha = *FT'$

$FIRST(*FT') = \{*\}$

Hence $M[T,*] = T \rightarrow *FT'$

Step 6 : $T' \rightarrow \varepsilon$

$A \rightarrow \alpha$

$A = T', \alpha = \varepsilon$

$FOLLOW(T') = \{+,),\$\}$

Hence $M[T',+] = T' \rightarrow \varepsilon$

$M[T',)] = T' \rightarrow \varepsilon$

$M[T',\$] = T' \rightarrow \varepsilon$

Step 7 : $F \rightarrow (E)$

$A \rightarrow \alpha$

$A = F, \alpha = (E)$

$FIRST((E)) = \{ (\}$

Hence $M[F,(] = F \rightarrow (E)$

Step 8 : $F \rightarrow id$

$A \rightarrow \alpha$

$A = F, \alpha = id$

$FIRST(id) = \{ id \}$

Hence $M[F,id] = F \rightarrow id$

The complete table can be as shown below.

	id	+	*	(	)	$
E	$E \rightarrow TE'$	Error	Error	$E \rightarrow TE'$	Error	Error
E'	Error	$E' \rightarrow +TE'$	Error	Error	$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$E \rightarrow FT'$	Error	Error	$T' \rightarrow FT'$	Error	Error
T'	Error	$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$	Error	$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$	Error	Error	$F \rightarrow (E)$	Error	Error

Now the input string **id + id * id $** can be parsed using above table. At the initial configuration the stack will contain start symbol E, in the input buffer input string is placed.

Stack	Input	Action
$(E)	(id) + id * id $	

Now symbol E is at top of the stack and input pointer is at first **id**, hence **M[E,id]** is referred. This entry tells us $E \rightarrow TE'$, so we will push E' first then T.

Stack	Input	Action
$E' T	id + id * id $	$E \rightarrow TE'$
$E' T' F	id + id * id $	$E \rightarrow FT'$
$E' T' id	id + id * id $	$F \rightarrow id$
$E' T'	+ id * id $	
$E'	+ id * id $	$T' \rightarrow \varepsilon$
$E' T+	+ id * id $	$E' \rightarrow +TE'$
$E' T	id * id $	
$E' T' F	id * id $	$T \rightarrow FT'$
$E' T' id	id * id $	$F \rightarrow id$
$E' T'	* id $	
$E' T' F*	* id $	$T \rightarrow *FT'$
$E' T' F	id $	
$E' T' id	id $	$F \rightarrow id$
$E' T'	$	
$E'	$	$T \rightarrow \varepsilon$
$	$	$E' \rightarrow \varepsilon$

Thus it is observed that the input is scanned from left to right and we always follow left most derivation while parsing the input string. Also at a time only one input symbol is referred to taking the parsing action. Hence the name of this parser is LL(1). The LL(1) Parser is a table driven predictive parser. The left recursion and ambiguous grammar is not allowed for LL(1) parser.

Example 3.3.16 *Show that following grammar :*

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

is LL (1).

Solution : Consider the grammar :

$S \rightarrow AaAb$

$S \rightarrow BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

Now we will compute FIRST and FOLLOW functions.

FIRST(S) = {a, b} if we put

$S \rightarrow AaAb$

$S \rightarrow aAb$ When $A \rightarrow \varepsilon$

Also $S \rightarrow BbBa$

$S \rightarrow bBa$ When $B \rightarrow \varepsilon$

FIRST(A) = FIRST(B) = {ε}

FOLLOW(S) = {$}

FOLLOW(A) = FOLLOW(B) = {a, b}

The LL(1) parsing table is

	a	b	$
S	$S \rightarrow AaAb$	$S \rightarrow BbBa$	
A	$A \rightarrow \varepsilon$	$A \rightarrow \varepsilon$	
B	$B \rightarrow \varepsilon$	$B \rightarrow \varepsilon$	

Now consider the string "ba". For parsing -

Stack	Input	Action
$S	ba$	$S \rightarrow BbBa$
$aBbB	ba$	$B \rightarrow \varepsilon$
$aBb	ba$	
$aB	a$	$B \rightarrow \varepsilon$
$a	a$	
$	$	Accept

This shows that the given grammar is LL(1).

Example 3.3.17 *For the following grammar find FIRST and FOLLOW sets for each of non terminal.* $S \rightarrow aAB \mid bA \mid \varepsilon$

$A \rightarrow aAb \mid \varepsilon$

$B \rightarrow bB \mid \varepsilon$

Solution : FIRST(S) = The first terminal symbol appearing on R.H.S.

$\therefore$ FIRST(S) = $\{a, b, \varepsilon\}$

FIRST(A) = The first terminal symbol appearing on R.H.S.

FIRST(A) = $\{a, \varepsilon\}$

FIRST(B) = The first terminal symbol appearing on R.H.S. of production rule for B

FIRST(B) = $\{b, \varepsilon\}$

Now, we will compute FOLLOW function as follows -

FOLLOW(S) = $\{\$\}$ $\because$ S is a start symbol.

FOLLOW(A) = $\{b\}$ because

Consider the rule,

$A \rightarrow \alpha B \beta$ which can be mapped with

$A \rightarrow a\ A\ B$ Then the FIRST(β) = FIRST(B) = $\{b, \varepsilon\}$

$\downarrow\ \downarrow\ \downarrow$

$\alpha\ B\ \beta$

Then without ε remains b. Hence b ∈ FOLLOW(A). Now consider the rule $A \rightarrow \alpha B$ then "everything in

FOLLOW(A) = FOLLOW(B). This rule can be mapped with

$S \rightarrow b\ A$

$\downarrow\ \ \downarrow\ \downarrow$ $\therefore$ everything in FOLLOW(S) = FOLLOW(A)

$A\ \ \alpha\ B$

$\therefore$ FOLLOW(A) = $\{\$\}$

To summarize FOLLOW(A) = $\{b, \$\}$

Now consider the rule,

$S \rightarrow a\ A\ B$

$\downarrow\ \ \underbrace{\ \ \downarrow\ \ }\ \downarrow$

if we map. $A \rightarrow \alpha\ B$ Then according to this rule,

$\therefore$ everything in FOLLOW(S) = FOLLOW(B)

$\therefore$ FOLLOW(B) = $\{\$\}$.

FIRST(A) = $\{a, \varepsilon\}$	FOLLOW(A) = $\{b, \$\}$
FIRST(B) = $\{b, \varepsilon\}$	FOLLOW(B) = $\{\$\}$
FIRST(S) = $\{a, b, \varepsilon\}$	FOLLOW(S) = $\{\$\}$

Example 3.3.18 *Consider the grammar*

$S \rightarrow iCtSA \mid a$

$A \rightarrow eS \mid \varepsilon$

$C \rightarrow b$

Whether it is LL(1) grammar. Give the explanation whether (i, t, e, b, a) are terminal symbols.

Solution : Let the given grammar will be,

$S \rightarrow iCtSA \mid a$

$A \rightarrow eS \mid \varepsilon$

$C \rightarrow b$

Now we will compute FIRST and FOLLOW for given nonterminals.

FIRST(S) = {i, a} FOLLOW(S) = {e, $}

FIRST(A) = {e, ε} FOLLOW(A) = {e, $}

FIRST(C) = {b} FOLLOW(C) = {t}

The predictive parsing table will be

	a	b	e	t	i	$
S	S → a				S → iCtSA	
A			A → eS A → ε			A → ε
C		C → b				

As we have got multiple entries in M[A, e] given grammar is **not LL(1) grammar.**

The (i, t, e, a, b) are the terminal symbols because they do not derive any production rule.

Example 3.3.19 *Construct the predictive parser for the following grammar.*

$S \rightarrow (L) \mid a$

$L \rightarrow L, S \mid S$

GTU : Winter-15, Marks 7, Summer-18, Marks 4

Solution : As the given grammar is left recursive because of

$L \rightarrow L, S \mid S.$

We will first eliminate left recursion . As $A \rightarrow A\,\alpha \mid \beta$ can be converted as

$A \rightarrow \beta\, A'$

$A' \rightarrow \alpha\, A' \mid \varepsilon$

We can write $L \rightarrow L, S \mid S$ as

$L \rightarrow SL'$

$L' \rightarrow ,SL' \mid \varepsilon$

Now the grammar taken for predictive parsing is -

$S \rightarrow (L) \mid a$

$L \rightarrow SL'$

$L' \rightarrow , SL' \mid \varepsilon$

Now we will compute FIRST and FOLLOW of non - terminals

$\therefore$ FIRST(S) = {(, a}

FIRST(L) = {(, a}

FIRST(L') = {', ε}

FOLLOW(S) = {',), $}

FOLLOW(L) = FOLLOW(L') = {)}

The predictive parsing table can be constructed as

	a	(	)	'	$
S	S → a	S → (L)			
L	L → SL'	L → SL'			
L'			L' → ε	L' → , SL'	

Example 3.3.20 *Construct the behaviour of the parser on the sentence (a, a) using the grammar specified above. $S \rightarrow (L) \mid a$*

Solution : As we have constructed a predictive parsing table in Example 3.3.19 we will parse the string (a, a) using that table as shown below.

State	Input	Actions
$ S	(a, a) $	
$) L (	(a, a) $	S → (L)
$) L	a, a) $	L → SL'
$) L' S	a, a) $	S → a
$) L' a	a, a) $	
$) L'	, a) $	L' → , SL'

$) L' S,	, a) $	
$) L' S	a) $	S → a
$) L' a	a) $	
$) L'	) $	L' → ε
$)	) $	
$	$	Accept

Example 3.3.21 *Compute FIRST and FOLLOW sets for all nonterminals in the following grammar* $S \rightarrow Aa|bAc|Bc|bBa$
$A \rightarrow d$
$B \rightarrow d$

Solution : FIRST(S) = first terminal symbol appearing on RHS.

$= \{b, d\}$

FIRST(A) = first terminal symbol appearing on RHS.

$= \{d\}$

FIRST(B) = first terminal symbol appearing on RHS.

$= \{d\}$

FOLLOW(S) $= \{\$\}$ ∵ Start Symbol.

Consider the rule

S → Aa which can be mapped with $A \rightarrow \alpha B \beta$.

$S \rightarrow A\ a$ with $\alpha = \varepsilon$
$A \quad B\ \beta$

Hence FOLLOW(A) = FIRST(β) = FIRST(a) = {a}

Similarly S → bAc. Which can be mapped with $A \rightarrow \alpha B \beta$.

$\therefore \quad S \rightarrow b\ A\ c$
$A \quad \alpha\ B\ \beta$

Then FOLLOW(A) = FIRST (β) = FIRST(c) = {c}

∴ **FOLLOW(A)** = {a, c}

Consider the rule,

$S \rightarrow b\ B\ a$
$A \quad \alpha\ B\ \beta$

FOLLOW(B) = FIRST(β) = FIRST(a) = {a}

$S \rightarrow B\ c$ with $\alpha = \varepsilon$
$A \quad B\ \beta$

FOLLOW(B) = FIRST(β) = FIRST(c) = {c}

∴ **FOLLOW(B)** = {a, c}

To summarize,

	FIRST	FOLLOW
S	{b,d}	{$}
A	{d}	{a,c}
B	{d}	{a,c}

Example 3.3.22 *Is the following grammar suitable for LL(1) parsing ? If not make it suitable for LL(1) parsing. Compute FIRST and FOLLOW sets. Generate the parsing table.*
$S \rightarrow AB$
$A \rightarrow Cd|\varepsilon$
$B \rightarrow BaAC|c$
$C \rightarrow b|\varepsilon$

GTU : Winter-11, Marks 7

Solution : The given grammar is not suitable for LL(1) parsing because the production rule.

B → BaAC

is left recursive. We will first eliminate the left recursion.

Consider the rule B → BaAC | c we will map it with $A \rightarrow A\alpha \mid \beta$ as follows -

$B \rightarrow BaAC \mid c$
$A \quad A\ \alpha \quad \beta$

This can be eliminated by re-writing the production rule as :

$A \rightarrow \beta A'$
$A' \rightarrow \alpha A'$ ⇒
$A' \rightarrow \varepsilon$

$B \rightarrow c\ B'$
$B' \rightarrow aACB'$
$B' \rightarrow \varepsilon$

'Now the production rules are

$S \rightarrow AB$
$A \rightarrow Ca \mid \varepsilon$
$B \rightarrow cB'$
$B' \rightarrow aACB'$
$B' \rightarrow \varepsilon$
$C \rightarrow b \mid \varepsilon$

	First	Follow
S	{b, c, a}	{ $ }
A	{b, a, ε}	{a, b, c}
B	{ c }	{ $ }
B'	{a, ε}	{ $ }
C	{b, ε}	{a, $}

The predictive parsing table can be as shown below -

	a	b	c	$
S	$S \rightarrow AB$	$S \rightarrow AB$	$S \rightarrow AB$	
A	$A \rightarrow Ca$ $A \rightarrow \varepsilon$	$A \rightarrow Ca$ $A \rightarrow \varepsilon$	$A \rightarrow \varepsilon$	
B			$B \rightarrow cB'$	
B'	$B' \rightarrow aACB'$			$B' \rightarrow \varepsilon$
C	$C \rightarrow \varepsilon$	$C \rightarrow b$		$C \rightarrow \varepsilon$

Example 3.3.23 *Construct predictive parsing table for following.*

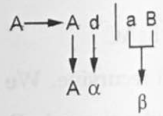
S–> A
A–> aB | Ad
B–> bBC | f
C–> g

GTU : Summer-12, Marks 7

Solution : The rule. $A \rightarrow Ad \mid aB$ is left recursive. We will eliminate the left recursion

Then we get, $A \rightarrow aBA'$

$A' \rightarrow dA' \mid \varepsilon$



Hence now rules becomes -

$S \rightarrow A$

$A \rightarrow aBA'$

$A' \rightarrow dA' \mid \varepsilon$

$B \rightarrow bBC \mid f$

$C \rightarrow g$

The FIRST and FOLLOW are

	First	Follow
S	{ a }	{ $ }
A	{ a }	{ $ }
A'	{d, ε}	{ $ }
B	{ b, f }	{d, g}
C	{ g }	{d, g}

The parsing table will be

	a	b	d	f	g	$
S	$S \rightarrow A$					
A	$A \rightarrow aBA'$					
A'			$A' \rightarrow dA'$			$A' \rightarrow \varepsilon$
B		$B \rightarrow dBC$		$B \rightarrow f$		
C					$C \rightarrow g$	

Consider the string **abfgd**

$ S	abfgd$	$S \rightarrow A$
$ A	abfgd$	$A \rightarrow aBA'$
$ A'Ba	abfgd$	
$ A'B	bfgd$	$B \rightarrow bBC$
$ A'CBb	bfgd$	
$ A'CB	fgd$	$B \rightarrow f$
$ A'Cf	fgd$	
$ A'C	gd$	$C \rightarrow g$
$ A'g	gd$	
$ A'	d$	$A' \rightarrow dA'$
$ A'd	d$	
$ A'	$	$A' \rightarrow \varepsilon$
$	$	ACCEPT

Example 3.3.24 *Find out FIRST and FOLLOW set for all the nonterminals*

$S \rightarrow AcB \mid cbB \mid Ba$

$A \rightarrow da \mid BC$

$B \rightarrow g \mid \varepsilon$

$C \rightarrow h \mid \varepsilon$

GTU : Summer-12, Marks 6

Solution : Let the grammar be

$S \rightarrow AcB \mid cbB \mid Ba$

$A \rightarrow da \mid BC$

$B \rightarrow g \mid \varepsilon$

$C \rightarrow h \mid \varepsilon$

The first and follow for each nonterminal are

	First	Follow
S	{a, c, d, g, h}	{ $ }
A	{d, g, h, ε}	{ c }
B	{g, ε}	{ a, h c, $}
C	{h, ε}	{ c }

Example 3.3.25 *Test whether the following grammer is LL (1) or not. Construct predictive parsing table for it.*

$S \rightarrow 1\,AB \mid \epsilon$

$A \rightarrow 1AC \mid 0C$

$B \rightarrow 0S$

$C \rightarrow 1$

GTU : Summer-12, Marks 7

Solution : Let, the grammar be - $S \rightarrow 1AB \mid \varepsilon$

$A \rightarrow 1AC \mid 0C$

$B \rightarrow 0S$

$C \rightarrow 1$

We will find out the FIRST and FOLLOW for this grammar -

	First	Follow
S	{1, ε}	{ $ }
A	{1, 0}	{0, 1}
B	{ 0 }	{ $ }
C	{ 1 }	{0, $}

The predictive parsing table will be

	0	1	$
S		S → 1 AB	S → ε
A	A → 0C	A → 1 AC	
B	B → 0S		
C		C → 1	

Consider string 110110

Stack	Input	Action
$ S	110110 $	S → 1AB
$ BA1	110110 $	
$ BA	10110 $	A → 1AC
$ BCA1	10110 $	
$ BCA	0110 $	A → 0C
$ BCC0	0110 $	
$ BCC	110 $	C → 1
$ BC1	10 $	
$ BC	10 $	C → 1
$ B1	10 $	
$ B	0 $	B → 0S
$ S0	0 $	
$ S	$	S → ε
$	$	Accept

The given grammar is LL(1).

Example 3.3.26 *Draw parsing table for table driven parser for the given grammar. Is the grammar LL (1) ?*

$A \rightarrow AaB \mid x \qquad B \rightarrow BCb \mid Cy \qquad C \rightarrow Cc \mid \wedge.$

GTU : Winter-13, Marks 7

Solution : Let,

$A \rightarrow A\,a\,B \mid x$

$B \rightarrow B\,C\,b \mid C\,y$

$C \rightarrow C\,c \mid \wedge$

We will compute FIRST and FOLLOW sets for given grammar.

FIRST (A) = {x} FOLLOW (A) = {a, $}

FIRST (B) = {c, y, ε} FOLLOW (B) = {b, c}

FIRST (C) = {c, ^} FOLLOW (C) = {y, b}

The parsing table is

	a	b	c	x	y	$
A				A → x		
B			B → BCb		B → Cy B → BCb	
C			C → Cc C → ^			

As there are multiple entries in c and y column, given grammar is not LL (1)

Example 3.3.27 *Implement the following grammar using table driven parser and check whether it is LL(1) or not. S-> aBDh, B-> cC, C-> bC /^, D -> EF, E-> g /^, F-> f/^*

GTU : Summer-14, Marks 8

Solution : We will first compute FIRST and FOLLOW

FIRST (S) = FIRST (aBDh) = {a}

FIRST (B) = FIRST (cC) = {c}

FIRST (C) = FIRST (bC) ∪ FIRST {ε} = {b, ε}

FIRST (D) = FIRST (EF) = {g, f, ε}

FIRST (E) = FIRST (g) ∪ FIRST (ε) = {g,ε}

FIRST (F) = FIRST (f) ∪ FIRST (ε) = {f,ε}

FOLLOW (S) = {$}

FOLLOW (B) = FIRST (dh) = FIRST (D) ∪ FIRST (h)

FOLLOW (B) = {g, f, h}

FOLLOW (C) = FOLLOW (B) ∵ B → cC

= {g, f, h}

FOLLOW (D) = {h}

FOLLOW (E) = {f, h}

FOLLOW (F) = FOLLOW (D) ∵ D → EF

= {h}

The LL(1) parsing table will be -

	a	b	c	g	f	h	$
S	S → aBDh						
B			B → cC				
C		C → bC		C → ε	C → ε	C → ε	
D				D → EF	D → EF	D → EF	
E				E → g	E → ε	E → ε	
F					F → f	F → ε	

Example 3.3.28 *For the following grammar.*

D → TL;

L → L, id | id

T → int | float

1) Remove left recursion (if required)

2) Find first and follow for each non-terminal for resultant grammar

3) Construct LL (1) Parsing table

*4) Parse the following String (Show Stack actions Clearly) and drawe parse Parse tree for the input : **int id, id;***

GTU : Summer-15, Marks 7

Solution : 1) Formula for removing the left recursion is as follows :

if A → A α |β is the rule. Then it can be converted to

A → β A′

A′ → α A′|ε

D → TL ;	No action as the rule is not left recursive.
L → L, id | id	L → id L′ L′→, id L′ L′→ ε
T → int | Float	No action as the rule rule has no left recursion.

2) The first and follow functions :

FIRST (D) = {int, float} FOLLOW (D) = {$}

FIRST (L) = {id} FOLLOW (L) = {;}

FIRST (L′) = {, , ε} FOLLOW (L′) = {;}

FIRST (T) = {int, float} FOLLOW (T) = {id}

3) Construction of Parsing table :

	id	,	int	float	;	$
D			D → TL;	D → TL;		
L	L → idL′					
L′		L′ → , idL′			L′ → ε	
T			T → int	T → float		

4) Parsing of string int id, id;

Stack	Input	Action
$D	int id,id;$	D → TL
$;LT	int id,id;$	T→ int
$;Lint	int id,id;$	
$;L	id,id;$	L→ id L′
$;L′id	id,id;$	
$;L′	, id;$	L′ → , id L′
$;L′id,	, id;$	
$;L′id	id;$	
$;L′	; $	L′→ ε
$	; $	
$	$	ACCEPT

Example 3.3.29 *Develop predictive parser for the following grammar,*

$S' \rightarrow S$

$S \rightarrow aA \mid b \mid cB \mid d$

$A \rightarrow aA \mid b$

$B \rightarrow cB \mid d$

GTU : Summer-15, Marks 7

Solution : We will compute FIRST and FOLLOW functions :

FIRST (S′) = FIRST (S) = {a, b, c, d}

FIRST (A) = {a, b}

FIRST (B) = {c, d}

FOLLOW (S′) = FOLLOW (S) = {$)

FOLLOW (A) = { }

FOLLOW (B) = { }

The predictive parsing table will be -

	a	b	c	d	$
S′	S′→ S	S′→ S	S′→ S	S′→ S	
S	S→ aA	S→b	S→cB	S→d	
A	A→ aA	A→b			
B			B→cB	B→d	

The string **aab** can be derived as

Stack	Input Buffer	Action
$S′	a a b $	S′→S
$S	a a b $	S → a A
$Aa	a a b $	
$A	a b $	A → a A
$Aa	ab $	
$A	b$	A → b
$b	b $	
$	$	Accept

Example 3.3.30 *Construct LL (1) parsing table for the following grammer. Also show moves made by input string : abba*

$S \rightarrow aBa$

$B \rightarrow bB \mid \varepsilon$

GTU : Winter-16, Marks 7

Solution : We will compute FIRST and FOLLOW functions :

FIRST (S) = {a}	FOLLOW (S) = {$}
FIRST (B) = {b, ε}	FOLLOW (B) = {a}

The LL(1) parsing table will be

	a	b	$
S	S → aBa		
B	B → ε	B → bB	

Simulation abba

$S	a b b a $	S → aBa
$aBa	a b b a $	
$aB	b b a $	B → bB
$aBb	b b a $	
$ B	b a $	B → bB
$ aBb	b a $	
$aB	a $	B → ε
$a	a $	
$	$	ACCEPT

Example 3.3.31 *Check following grammer is LL(1) or not ?*

$S \rightarrow aB \mid \varepsilon$

$B \rightarrow bC \mid \varepsilon$

$C \rightarrow cS \mid \varepsilon$

GTU : Summer-17, Marks 7

Solution :

Let,

$S \rightarrow aB \mid \varepsilon$

$B \rightarrow bC \mid \varepsilon$

$C \rightarrow cS \mid \varepsilon$

The FIRST and FOLLOW are computed as

Symbol	FIRST	FOLLOW
S	{a, ε}	{$}
B	{b, ε}	{$}
C	{c, ε}	{$}

The parsing table will be

	a	b	c	$
S	S → aB			S → ε
B		B → bC		B → ε
C			C → cS	C → ε

Simulation :

Consider derivation of **abc**.

State	Input	Action
$ S	abc $	S → aB
$ Ba	abc $	B → bC
$ Cb	bc $	C → cS
$ Sc	c $	S → ε
$	$	ACCEPT

Example 3.3.32 *Construct LL(1) parsing table for the following grammar.*

GTU : Winter-18, Marks 7

Solution : Let, the given grammar is -

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid a$

The given grammar has some rules that are left recursive. We will remove left recursion, using the rule.

If $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \beta_1 \mid \beta_2$ is a rule then

$A \rightarrow \beta_1 A' \mid \beta_2 A'$, $A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \varepsilon$

Hence on eliminating left recursion we get -

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

For constructing LL(1) parsing table for above grammar, Refer example 3.3.11.

Example 3.3.33 *Construct LL(1) parsing table for following grammar. Check whether the grammar is LL(1) or not.*

$A \rightarrow A\ a\ B$

$A \rightarrow x$

$B \rightarrow B\ C\ b$

$B \rightarrow C\ y$

$C \rightarrow C\ c$

GTU : Summer-19, Marks 7

Solution : We will find FIRST and FOLLOW for grammar -

	FIRST	FOLLOW
A	{x}	{a, $}
B	{c, y}	{b, y, c}
C	{c, ε}	{b, y, c}

The predictive parsing table will be ,

	a	b	x	y	c	$
A			A → x			
B				B → BCb B → Cy		
C					C → Cc	

As there are multiple entries in M[B, y] the given grammar is not LL(1).

Review Questions

1. *Write an algorithm for eliminating left recursion.* **[Refer Section 3.3.1(2)]** **GTU : Winter-11, Marks 3**
2. *Explain non-recursive predictive parsers. Draw the block diagram of it.* **GTU : Summer-12, Marks 4**
3. *Write down the algorithm for left factoring.* **[Refer Section 3.3.1(3)]** **GTU : Winter-13,19, Marks 3**
4. *Explain recursive-descent and predictive parsing.* **[Refer Section 3.3.2]** **GTU : Winter-14, Marks 7**
5. *What is left factoring and left recursion? Explain it with suitable example.* **GTU : Summer-17, Marks 7**
6. *What is ambiguous grammar? Describe with example.* **GTU : Winter-18, Marks 3**

3.4 Bottom Up Parsing

GTU : Winter-12,13,14, Summer-14,15, Marks 12

- In bottom-up parsing method, the input string is taken first and we try to reduce this string with the help of grammar and try to obtain the start symbol.
- The process of parsing halts successfully as soon as we reach to start symbol.

- The parse tree is constructed from bottom to up that is from leaves to root.
- In this process, the input symbols are placed at the leaf nodes after successful parsing.
- The bottom-up parse tree is created starting from leaves, the leaf nodes together are reduced further to internal nodes, these internal nodes are further reduced and eventually a root node is obtained.
- The internal nodes are created from the list of terminal and non-terminal symbols. This involves -

  Non-terminal for internal node = Non-terminal ∪ terminal
- In this process, basically parser tries to identify R.H.S. of production rule and replace it by corresponding L.H.S. This activity is called **reduction.**
- Thus the **crucial but prime task** in bottom-up parsing is to find the productions that can be used for reduction.
- The bottom-up parse tree construction process indicates that the tracing of derivations are to be done in **reverse order.**

For example

Consider the grammar for declarative statement,

S → TL;

T → int | float

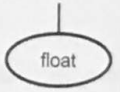
L → L,id | id

The input string is **float id,id,id;**

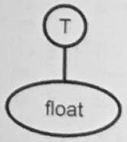
Parse Tree

Step 1 : We will start from leaf node

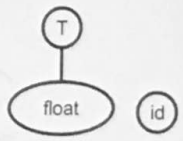
Step 2 :



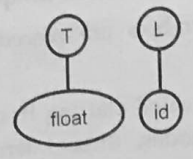
Step 3 : Reducing float to T. T → float.



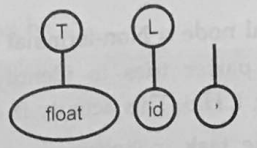
Step 4 : Read next string from input.



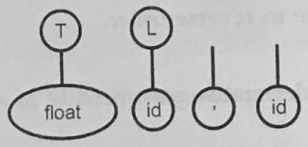
Step 5 : Reducing id to L. L → id.



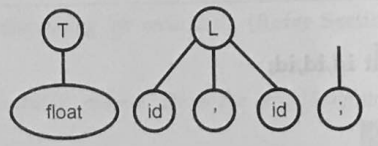
Step 6 : Read next string from input.



Step 7 : Read next string from input.



Step 8 : $\underbrace{\text{id, id}}_{\text{L}}$ gets reduced.



Step 9 : $\underbrace{\text{TL}}_{\text{S}}$; reduced to.

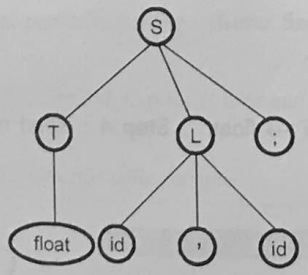


Fig. 3.4.1 Bottom-up parsing

Step 10 : The sentential form produced while constructing this parse tree is

float id,id;

T id,id;

T L,id;

T L;

S

Step 11 : Thus looking at sentential form we can say that the rightmost derivation in reverse order is performed.

Thus basic steps in bottom-up parsing are

1) Reduction of input string to start symbol.
2) The sentential forms that are produced in the reduction process should trace out rightmost derivation in reverse.

Handle Pruning

As said earlier, the crucial task in bottom-up parsing is to find the substring that could be reduced by appropriate non-terminal. Such a substring is called **handle**.

In other words **handle** is a string of substring that matches the right side of production and we can reduce such string by a non-terminal on left hand side production. Such reduction represents one step along the reverse of rightmost derivation. Formally we can define handle as follow.

Defination of Handle : "Handle of right sentential form γ is a production A → β and a position of γ where the string β may be found and replaced by A to produce the previous right sentential form in rightmost derivation of γ".

For example

Consider the grammar

$$E \rightarrow E+E$$
$$E \rightarrow id$$

Now consider the string **id + id + id** and the rightmost derivation is

$$E \underset{rm}{\Rightarrow} \mathbf{E + E}$$
$$E \underset{rm}{\Rightarrow} E + \mathbf{E + E}$$
$$E \underset{rm}{\Rightarrow} E + \mathbf{E + id}$$
$$E \underset{rm}{\Rightarrow} E + \mathbf{id} + id$$
$$E \underset{rm}{\Rightarrow} \mathbf{id} + id + id$$

The bold strings are called handles.

Right sentential form	Handle	Production
id + id + id	id	E → id
E + id + id	id	E → id
E + E + id	id	E → id
E + E + E	E + E	E → E + E
E + E	E + E	E → E + E
E		

Thus bottom parser is essentially a process of detecting handles and using them in reduction. This process is called **handle pruning**.

3.4.1 Shift Reduce Parser

Shift reduce parser attempts to construct parse tree from leaves to root. Thus it works on the same principle of **bottom up parser**. A shift reduce parser requires following data structures.

1. The **input buffer** storing the input string.
2. A **stack** for storing and accessing the L.H.S. and R.H.S. of rules.

The initial configuration of Shift reduce parser is as shown in Fig. 3.4.2.

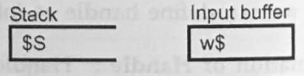


Fig. 3.4.2 Initial configuration

The parser performs following basic **operations**.

1. Shift : Moving of the symbols from input buffer onto the stack, this action is called shift.

2. Reduce : If the handle appears on the top of the stack then reduction of it by appropriate rule is done. That means R.H.S. of rule is popped of and L.H.S. is pushed in. This action is called Reduce action.

3. Accept : If the stack contains start symbol only and input buffer is empty at the same time then that action is called accept. When accept state is obtained in the process of parsing then it means a successful parsing is done.

4. Error : A situation in which parser cannot either shift or reduce the symbols, it cannot even perform the accept action is called as error.

Let us take some examples to learn the shift-reduce parser.

Example 3.4.1 *Consider the grammar*

$E \rightarrow E - E$
$E \rightarrow E * E$
$E \rightarrow id$

*Perform Shift-Reduce parsing of the input string "id1–id2*id3".*

Solution :

Stack	Input buffer	Parsing action
$	id1–id2*id3$	Shift
$id1	–id2*id3$	Reduce by E→ id
$E	–id2*id3$	Shift
$E–	id2*id3$	Shift
$E–id2	*id3$	Reduce by E→ id
$E–E	*id3$	Shift
$E–E*	id3$	Shift
$E–E*id3	$	Reduce by E→ id
$E–E*E	$	Reduce by E→ E*E
$E–E	$	Reduce by E→ E–E
$E	$	Accept

Here we have followed two rules.

1. If the **incoming operator** has **more priority** than in stack operator then perform **shift.**
2. If **in stack operator** has **same or less priority** than the priority of incoming operator then perform **reduce.**

Example 3.4.2 *Consider the following grammar*

$S \rightarrow TL;$
$T \rightarrow int \mid float$
$L \rightarrow L,id \mid id$

Parse the input string int id,id; using shift-reduce parser.

Solution :

Stack	Input buffer	Parsing action
$	int id,id;$	Shift

$int	id,id;$	Reduce by T→ int
$T	id,id;$	Shift
$Tid	id;$	Reduce by L→ id
$TL	id;$	Shift
$TL,	id;$	Shift
$TL,id	;$	Reduce by L → L, id
$TL	;$	Shift
$TL;	$	Reduce by S→ TL;
$S	$	Accept

Example 3.4.3 *Write unambiguous production rules for producing arithmetic expression consisting of symbols id, *, –, ( ) and ^, where ^ represents exponent. Parse following string using shift – reduce parser :*
*id – id * id ^ id * (id ^ id) ^ id*
Explain various conflicts of a shift – reduce parser. **GTU : Winter-12, Marks 12**

Solution : The unambiguous production rules for arithmetic expression will be

$E \rightarrow E+T$
$E \rightarrow T$
$T \rightarrow T*F$
$T \rightarrow F$
$F \rightarrow F-P$
$F \rightarrow P$
$P \rightarrow P \wedge Q$
$P \rightarrow Q$
$Q \rightarrow (E)$
$Q \rightarrow id$

Consider the string **id-id *id ∧ id * (id ∧ id) ∧ id** for parsing using shift - reduce.

Stack	Input buffer	Parsing Action
$	id – id * id ∧ id * (id ∧ id) ∧ id$	Shift
$ id	– id * id ∧ id * (id ∧ id) ∧ id$	Reduce by Q → id
$ Q	– id * id ∧ id * (id ∧ id) ∧ id$	Reduce by P → Q

$ P	– id * id ∧ id * (id ∧ id) ∧ id$	Reduce by F → P
$ F	– id * id ∧ id * (id ∧ id) ∧ id$	Shift
$ F –	id * id ∧ id * (id ∧ id) ∧ id$	Shift
$ F – id	* id ∧ id * (id ∧ id) ∧ id$	Reduce by Q → id
$ F – Q	* id ∧ id * (id ∧ id) ∧ id$	Reduce by P → Q
$ F	* id ∧ id * (id ∧ id) ∧ id$	Reduce by T → F
$ T	* id ∧ id * (id ∧ id) ∧ id$	Shift
$ T *	* id ∧ id * (id ∧ id) ∧ id$	Shift
$ T * id	∧ id * (id ∧ id) ∧ id$	Reduce by Q → id
$ T * Q	∧ id * (id ∧ id) ∧ id$	Reduce by P → Q
$ T * P	∧ id * (id ∧ id) ∧ id$	Shift
$ T * P∧	id * (id ∧ id) ∧ id$	Shift
$ T * P∧ id	* (id ∧ id) ∧ id$	Reduce by Q → id
$ T * P∧Q	* (id ∧ id) ∧ id$	Reduce by P → P ∧ Q
$T * P	* (id ∧ id) ∧ id$	Reduce by F → P
$T * F	* (id ∧ id) ∧ id$	Reduce by T → T*F
$T	* (id ∧ id) ∧ id$	Shift
$T *	(id ∧ id) ∧ id$	Shift
$T * (	id ∧ id) ∧ id$	Shift
$T * (id	∧ id) ∧ id$	Reduce by Q → id
$T * (Q	∧ id) ∧ id$	Reduce by P → Q
$T * (P	∧ id) ∧ id$	Shift
$T * (P∧	id) ∧ id$	Shift
$T * (P∧ id	) ∧ id$	Reduce by Q → id
$T * (P∧ Q	) ∧ id$	Shift
$T * (P ∧ Q)	∧ id$	Reduce by P → P ∧ Q
$T * (P)	∧ id$	Reduce by F → P
$T * (F)	∧ id$	Reduce by T → F
$T * (T)	∧ id$	Reduce by E → T
$T * (E)	∧ id$	Reduce by Q → (E)
$T * Q	∧ id$	Reduce by P → Q
$T * P	∧ id$	Shift
$T * P ∧	id$	Shift

$T * P ∧ id	$	Reduce by Q → id
$T * P∧Q	$	Reduce by P → P∧Q
$T * P	$	Reduce by F → P
$T * F	$	Reduce by T → T*F
$ T	$	Reduce by E → T
$ E	$	Accept

Example 3.4.4 *How top down and bottom up parser will parse the string 'bbd' using grammar A → bA/d. Show all steps clearly.* **GTU : Summer-15, Marks 7**

Solution : I) For the **top down** parsing we will use LL(1) parsing method.

Let, A → bA | d be the grammar

FIRST (A) = {b, d}

FOLLOW (A) = {$}

The Predictive **Parsing table** will be :

	b	d	$
A	A → bA	A → d	

The Parsing of input **bbd**

Stack	Input	Action
$A	b bd $	A → bA
$ Ab	b bd $	
$A	b d $	A → bA
$ Ab	b d $	
$A	d$	A → d
$d	d$	
$	$	Accept

II] For the bottom up parsing we will use **shift reduce** Parsing method. Consider the input **bbd**.

Stack	Input buffer	Parsing action
$	bbd$	Shift
$b	bd$	Shift
$bb	d$	Shift
$bbd	$	Reduce by A → d
$bbA	$	Reduce by A → bA
$bA	$	Reduce by A → bA
$A	$	Accept

Review Questions

1. *Explain shift - reduce parsing with suitable example.* **GTU : Winter-13, Marks 7**
2. *What is bottom - up parsing ? Discuss shift reduce parsing technique in brief. What is a handle ?* **GTU : Summer-14, Marks 8**
3. *What do you understand by a handle ? Explain the stack implementation of shift reduce parser with the help of example.* **GTU : Winter-14, Marks 7**

3.5 Operator - Precedence Parser

GTU : Summer-14,15,16,17, Winter-11,15,16,17,20, Marks 8

A grammar G is said to be **operator precedence** if it posses following properties -

1. No production on the right side is ε .
2. There should not be any production rule possessing two adjacent non-terminals at the right hand side.

Consider the grammar for arithmetic expressions.

E → EAE | (E) | – E | id

A→ + | – | * | / | ^

This grammar is not an operator precedent grammar as in the production rule.

E → EAE

It contains two consecutive non-terminals. Hence first we will convert it into equivalent operator precedence grammar by removing A.

E→ E + E | E – E | E * E | E / E | E ^ E

E → (E)| – E| id

In operator precedence parsing we will first define precedence relations $<\cdot$, $\doteq$ and $\cdot>$ between pair of terminals. The meaning of these relations is

$p <\cdot q$	p gives more precedence than q.
$p \doteq q$	p has same precedence as q.
$p \cdot> q$	p takes precedence over q.

These meanings appear similar to the less than, equal to and greater than operators. Now by considering the precedence relation between the arithmetic operators we will construct the operator precedence table. The operators precedences we have considered are id,+,*,$.

	id	+	*	$
id		$\cdot>$	$\cdot>$	$\cdot>$
+	$<\cdot$	$\cdot>$	$<\cdot$	$\cdot>$
*	$<\cdot$	$\cdot>$	$\cdot>$	$\cdot>$
$	$<\cdot$	$<\cdot$	$<\cdot$	

Fig. 3.5.1 Precedence relation table

Now consider the string.

id + id * id

We will insert $ symbols at the start and end of the input string. We will also insert precedence operator by referring the precedence relation table.

$ <· id ·> + <· id ·> * <· id ·> $

We will follow following steps to parse the given string -

i) Scan the input from left to right until first $\cdot>$ is encountered.

ii) Scan backwards over = until $<\cdot$ is encountered.

iii) The handle is a string between $<\cdot$ and $\cdot>$.

The parsing can be done as follows.

$ <· id ·> + <· id ·> * <· id ·> $	Handle id is obtained between <· ·> Reduce this by E → id
E + <· id ·> * <· id ·> $	Handle id is obtained between <· ·> Reduce this by E → id

E + E * <· id ·> $	Handle id is obtained between <· ·> Reduce this by E → id
E + E * E	Remove all the non-terminals.
+ *	Insert $ at the beginning at the end. Also insert the precedence operators.
$ <· + <· * ·> $	The * operator is surrounded by <· ·>. This indicates that * becomes handle. That means we have to reduce E*E operation first.
$ <· + ·> $	Now + becomes handle. Hence we evaluate E + E.
$ $	Parsing is done.

Advantage of Operator Precedence Parsing

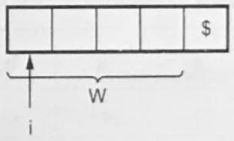
1. This type of parsing is simple to implement.

Disadvantages of Operator Precedence Parsing

1. The operator like minus has two different precedence (unary and binary). Hence it is hard to handle tokens like minus sign.
2. This kind of parsing is applicable to only small class of grammars.

3.5.1 Operator Precedence Parsing Algorithm

1. Set i pointer to first symbol of string w. The string will be represented as follows.



2. If $ is on the top of the stack and if a is the symbol pointed by i **then return**.
3. If a is on the top of the stack and if the symbol b is read by pointer i then

   a) **if** $a <\cdot b$ or $a \doteq b$ **then**

      push b on to the stack

      advance the pointer i to next input symbol.

   b) **else if** $a \cdot> b$ **then**

      while (top symbol of the stack $\cdot>$ recently popped terminal symbol)

      {

      pop the stack. Here popping the symbol means reducing the terminal

symbol by equivalent non terminal.

}

c) **else** error ( ).

Example 3.5.1 *Construct an operator precedence parser for the grammar.*

$S \rightarrow iEtS \mid iEtSeS \mid a$

$E \rightarrow b \mid c \mid d$

Where a, b, c, d, e, i, t are terminals. **GTU : Summer-15, Marks 7**

Solution : Let,

$S \rightarrow iEtS \mid iEtSeS \mid a$

$E \rightarrow b \mid c \mid d$

be the given grammar.

Here, i stands for **if**

E stands for **Expression**

t stands for **then**

S stands for **statement**

The symbols a, b, c, d are the terminal symbols.

- The precedence if > then > else.

  $\therefore \quad i \cdot > t \cdot > e$
- Similarly $ symbol will have least precedence.
- The terminal symbols a, b, c, and d have highest precedence over i, t and e.
- The nonterminal symbols have less precedence over a, b, c, d, i, e, t. Hence the precedence table can be designed as follows -

	i	t	e	a	b	c	d	$
i		·>	·>	<·	<·	<·	<·	·>
t	<·		·>	<·	<·	<·	<·	·>
e	<·	<·		<·	<·	<·	<·	·>
a	·>	·>	·>					·>
b	·>	·>	·>					·>
c	·>	·>	·>					·>
d	·>	·>	·>					·>
$	<·	<·	<·	<·	<·	<·	<·	

Simulation consider a valid string **ibtaea** for **simulation.**

Stack	Input Scanned	Relation	Action
$	↑ibtaea $	<·	Push i
$i	↑btaea $	<·	Push b
$ib	↑taea $	·>	Reduce by E → b
$iE	↑taea $	<·	Push t
$iEt	↑aea $	<·	Push a
$iEta	↑ea $	·>	Reduce by S → a
$iEtS	↑ea $	<·	Push e
$iEtSe	↑a $	<·	Push a
$iEtSea	↑$	·>	Reduce by S → a
$iEtSeS	$	·>	Reduce by S → iEtSeS
$S	$		ACCEPT

3.5.2 Precedence Functions

During operator precedence parsing, the table of precedence relation is not stored. Instead of it, the operator precedence parser use precedence functions to map terminal symbols to integers so that the precedence relations between the symbols are implemented by numerical comparison.

For example :

Consider two functions f and g. For these functions the precedence relation can be shown with some integers.

Precedence table -

	+	*	(	)	id	$
f	2	4	0	6	6	0
g	1	3	5	0	5	0

From this precedence table we can design a precedence graph, and precedence function. Here is an algorithm for constructing precedence function-

Algorithm

Input : An operator precedence matrix

Output : Precedence function.

Method

1. Create functions f_a and g_a for each grammar terminal a and for $.
2. Partition the symbols in groups so that f_a and g_a are in the same group if $a \doteq b$
3. Create a directed graph by adding edges, in following manner
   i) If $a \lessdot b$ then add an edge from and g_b and f_a.
   ii) If $a \gtrdot b$ then add an edge from f_a and g_b.
4. If the constructed graph contains no cycle then there exists precedence function.

Example 3.5.2 *Construct precedence graph using following precedence table*

	+	*	(	)	id	$
f	2	3	0	4	4	0
g	1	3	5	0	5	0

Solution :

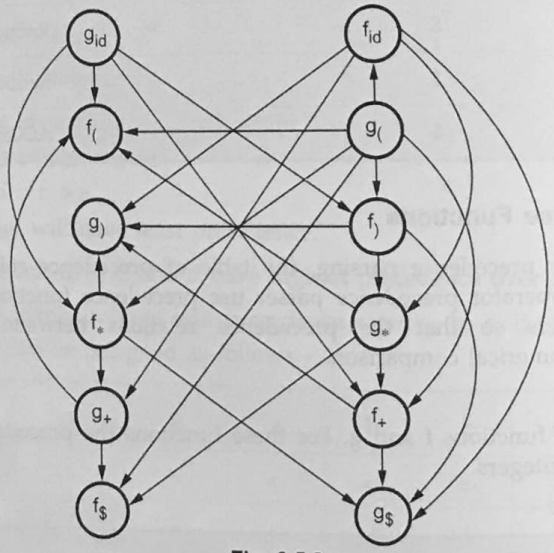


Fig. 3.5.2

Review Questions

1. *Compute the operator precedence matrix and precedence function for the following grammar if it exists. +,*,–,/,id,num,(and) are terminal symbols.*
   $G \rightarrow E$
   $E \rightarrow E+T|E-T|T$
   $T \rightarrow T*F|T / F / F$
   $F \rightarrow num|id|(E)$ **[Refer Section 3.5]** **GTU : Winter-11, Summer-17, Marks 7**

2. *Define an operator precedence grammar. Also write down the rules to find relationship between each pair of terminal symbols.* **GTU : Summer-14, Marks 8**
3. *Explain operator precedence Parsing method with example.* **GTU : Winter-15, Marks 7**
4. *Where do we use operator precedence parsing technique? Give the general precedence table for operating precedence parsing. Consider all generalized rules* **GTU : Summer-16, Marks 7**
5. *Write short note on operator precedence parsing with an example* **GTU : Winter-16, Marks 7**
6. *What is operator grammar? Generate precedence function table for following grammar*
   E->EAE | id
   *A->+|**
   GTU : Winter-17, Marks 7
7. *Write rules to check grammar is operator grammar or not.* **GTU : Winter-20, Marks 3**

3.6 LR Parsers

This is the most efficient method of bottom-up parsing which can be used to parse the large class of context free grammars. This method is also called **LR(k) parsing**. Here

- **L** stands for left to right scanning.
- **R** stands for rightmost derivation in reverse.
- **k** is number of input symbols. When k is omitted k is assumed to be 1.

Properties of LR parser -

LR parser is **widely used for following reasons.**

1. LR parsers can be constructed to **recognize most of the programming languages for** which context free grammar can be written.
2. The class of grammar that can be parsed by **LR parser** is a **superset** of class of grammars that can be parsed using **predictive parsers**.
3. LR parser works using **non backtracking shift reduce technique** yet it is efficient one.
4. LR Parsers **detect syntactical errors** very efficiently.

Structure of LR parsers

The structure of LR parser is as given in following Fig. 3.6.1.

It consists of **input buffer** for storing the input string, a **stack** for storing the grammar symbols, **output** and a **parsing table comprised** of two parts, namely **action** and **goto**. There is one parsing program which is actually a driving program and reads the input symbol one at a time from the input buffer.

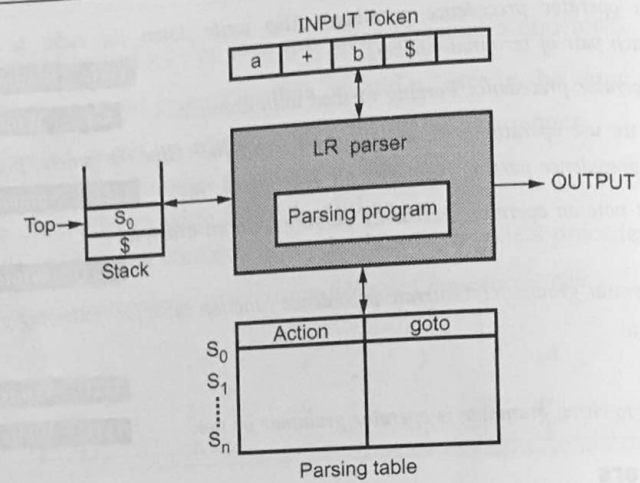


Fig. 3.6.1 Structure of LR parser

The **driving program** works on following line.

1. It initializes the stack with start symbol and invokes scanner (lexical analyzer) to get next token.
2. It determines s_j the state currently on the top of the stack and a_i the current input symbol.
3. It consults the parsing table for the action $[s_j, a_i]$ which can have one of the four values.
   i) s_i means shift state i.
   ii) r_j means reduce by rule j.
   iii) Accept means successful parsing is done.
   iv) Error indicates syntactical error.

Types of LR parser

Following diagram represents the types of LR parser.

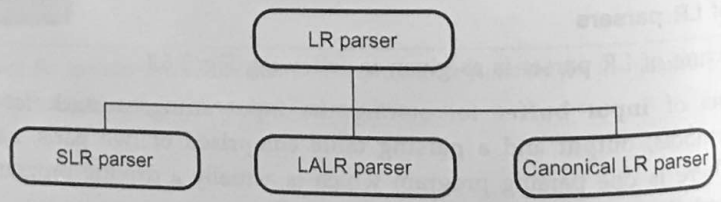


Fig. 3.6.2 Techniques of LR parsers

The SLR means simple LR parser, LALR means Lookahead LR parser and canonical LR or simply "LR" parser - these are the three members of LR family. The overall structure of all these LR parsers is same. All are table driven parsers. The relative powers of these parsers is SLR(1) ≤ LALR(1) ≤ LR(1). That means canonical LR parses larger class than LALR and LALR parses larger class than SLR parser.

3.7 Simple LR Parsing (SLR)

GTU : Summer-12,14,15,16,17,18,19, Winter-12,13,14,15,16,19 Marks 8

We will start by the simplest form of LR parsing called SLR parser. It is the weakest of the three methods but it is easiest to implement. The parsing can be done as follows.

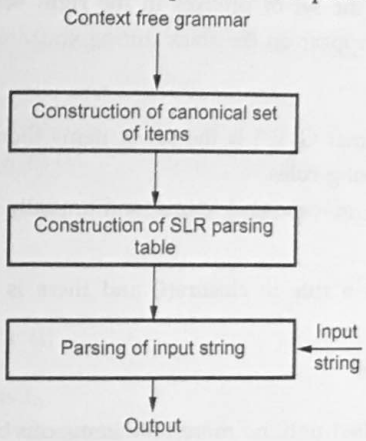


Fig. 3.7.1 Working of SLR(1)

A grammar for which SLR parser can be constructed is called SLR grammar.

Definition of LR(0) items and related terms -

1) The LR(0) item for grammar G is production rule in which symbol • is inserted at some position in R.H.S. of the rule. For example

   $S \rightarrow \bullet ABC$

   $S \rightarrow A \bullet BC$

   $S \rightarrow AB \bullet C$

   $S \rightarrow ABC \bullet$

   The production $S \rightarrow \varepsilon$ generates only one item $S \rightarrow \bullet$.

2) **Augmented grammar** : If a grammar G is having start symbol S then augmented grammar is a new grammar G′ in which S′ is a new start symbol such that $S' \rightarrow S$.

The purpose of this grammar is to indicate the acceptance of input. That is when parser is about to reduce $S' \rightarrow S$ it reaches to acceptance state.

3) **Kernel items** : It is collection of items $S' \rightarrow \bullet S$ and all the items whose dots are not at the leftmost end of R.H.S. of the rule.

Non-Kernel items : The collection of all the items in which $\bullet$ are at the left end of R.H.S. of the rule.

4) **Functions closure and goto :** These are two important functions required to create collection of canonical set of items.

5) **Viable prefix :** It is the set of prefixes in the right sentential form of production $A \rightarrow \alpha$. This set can appear on the stack during shift/reduce action.

Closure operation -

For a context free grammar G, if I is the set of items then the function closure(I) can be constructed using following rules.

1. Consider I is a set of canonical items and initially every item I is added to closure(I).
2. If rule $A \rightarrow \alpha \bullet B\beta$ is a rule in closure(I) and there is another rule for B such as $B \rightarrow \gamma$ then

   closure(I) : $A \rightarrow \alpha \bullet B\beta$

   $B \rightarrow \bullet \gamma$

This rule has to be applied until no more new items can be added to closure(I).

The meaning of rule $A \rightarrow \alpha \bullet B\beta$ is that during derivation of the input string at some point we may require strings derivable from $B\beta$ as input. A non-terminal immediately to the right of $\bullet$ indicates that it has to be expanded shortly.

Goto operation -

The function goto can be defines as follows.

If there is a production $A \rightarrow \alpha \bullet B\beta$ then goto($A \rightarrow \alpha \bullet B\beta$,B) = $A \rightarrow \alpha B \bullet \beta$. That means simply shifting of $\bullet$ one position ahead over the grammar symbol (may be terminal or non-terminal). The rule $A \rightarrow \alpha \bullet B\beta$ is in I then the same goto function can be written as goto(I,B).

Example 3.7.1 *Consider the grammar*

$X \rightarrow Xb \mid a$

Compute closure(I) and goto(I).

Solution : Let,

$I : X \rightarrow \bullet Xb$

Closure(I) $= X \rightarrow \bullet Xb$

$= X \rightarrow \bullet a$

The goto function can be computed as

goto(I,X) = $X \rightarrow X \bullet b$

$X \rightarrow \bullet X b$ gives $X \rightarrow X \bullet b$

$X \rightarrow \bullet a$ gives $X \rightarrow a \bullet$

Similarly goto (I, a) gives $X \rightarrow a \bullet$

Example 3.7.2 *Consider the grammar*

$S \rightarrow AS \mid b$

$A \rightarrow SA \mid a$

Compute closure (I) and goto (I).

Solution : We will first write the grammar using dot operator.

$S \rightarrow \bullet AS$

$S \rightarrow \bullet b$

$A \rightarrow \bullet SA$

$A \rightarrow \bullet a$ } Closure (I)

Let us call this as state as I_0.

Now we apply goto on each symbol in I_0.

I_0 : $S \rightarrow \bullet AS$
$S \rightarrow \bullet b$
$A \rightarrow \bullet SA$
$A \rightarrow \bullet a$

I_1 : goto (I_0, A)
$S \rightarrow A \bullet S$
$S \rightarrow \bullet b$
$A \rightarrow \bullet SA$
$A \rightarrow \bullet a$

I_2 : goto (I_0, b)
$S \rightarrow b \bullet$

I_3 : goto (I_0, S)
$A \rightarrow S \bullet A$
$A \rightarrow \bullet SA$
$A \rightarrow \bullet a$
$S \rightarrow \bullet AS$
$S \rightarrow \bullet b$

I_4 : goto (I_0, a)
$A \rightarrow a \bullet$

Example 3.7.3 *Consider the following grammar :*

$S \rightarrow Aa \mid bAc \mid Bc \mid bBa$

$A \rightarrow d$

$B \rightarrow d$

Compute closure (I) and goto (I).

Solution : First we will write the grammar using dot operator.

I_0 : $S \rightarrow \bullet Aa$

$S \rightarrow \bullet bAc$

$S \rightarrow \bullet Bc$

$S \rightarrow \bullet bBa$

$A \rightarrow \bullet d$

$B \rightarrow \bullet d$

Now we will apply goto on each symbol from state I_0. Each goto (I) will generate new subsequent states.

I_1 : goto (I_0, A)

$S \rightarrow A \bullet a$

I_2 : goto (I_0, b)

$S \rightarrow b \bullet Ac$

$S \rightarrow b \bullet Ba$

$A \rightarrow \bullet d$

$B \rightarrow \bullet d$

I_3 : goto (I_0, B)

$S \rightarrow B \bullet c$

I_4 : goto (I_0, d)

$A \rightarrow d \bullet$

$B \rightarrow d \bullet$

Construction of canonical collection of set of item -

1. For the grammar G initially add $S' \rightarrow \bullet S$ in the set of item C.
2. For each set of items I_i in C and for each grammar symbol X (may be terminal non-terminal) add closure (I_i, X). This process should be repeated by apply goto(I_i, X) for each X in I_i such that goto(I_i, X) is not empty and not in C. The set items has to constructed until no more set of items can be added to C.

Now we will consider one grammar and construct the set of items by applying closure and goto functions.

Example :

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

In this grammar we will add the augmented grammar $E' \rightarrow \bullet E$ in the I then we have to apply closure(I).

I_0 :

$E' \rightarrow \bullet E$

$E \rightarrow \bullet E + T$

$E \rightarrow \bullet T$

$T \rightarrow \bullet T * F$

$T \rightarrow \bullet F$

$F \rightarrow \bullet (E)$

$F \rightarrow \bullet id$

The item I_0 is constructed starting from the grammar $E' \rightarrow \bullet E$. Now immediately right to • is E. Hence we have applied closure(I_0) and thereby we add E-productions with • at the left end of the rule. That means we have added $E \rightarrow \bullet E + T$ and $E \rightarrow \bullet T$ in I_0. But again as we can see that the rule $E \rightarrow \bullet T$ which we have added, contains non-terminal T immediately right to •. So we have to add T-productions in I_0.
$T \rightarrow \bullet T * F$ and $T \rightarrow \bullet F$.

In T-productions after • comes T and F respectively. But since we have already added T productions so we will not add those. But we will add all the F-productions having dots. The $F \rightarrow \bullet (E)$ and $F \rightarrow \bullet id$ will be added. Now we can see that after dot and id are coming in these two productions. The (and id are terminal symbols and are not deriving any rule. Hence our closure function terminates over here. Since there is no rule further we will stop creating I_0.

Now apply goto(I_0, E)

$E' \rightarrow \bullet E$, $E \rightarrow \bullet E + T$ —Shift dot to right→ $E' \rightarrow E \bullet$, $E \rightarrow E \bullet + T$

Thus I_1 becomes,

goto(I_0, E)
I_1 : E′→ E •
E → E • +T

Since in I_1 there is no non-terminal after dot we cannot apply closure(I_1).

By applying goto on T of I_0,

goto(I_0, T)
I_2 : E → T •
T → T • * F

Since in I_2 there is no non-terminal after dot we cannot apply closure(I_2).

By applying goto on F of I_0,

goto(I_0, F)
I_3 : T→ F •

Since after dot in I_3 there is nothing, hence we cannot apply closure(I_3).

By applying goto on (of I_0. But after dot E comes hence we will apply closure on E, then on T, then on F.

goto(I_0, ( )
I_4 : T → (• E)
E →• E + T
E →• T
T →• T * F
T →• F
F →• (E)
F →• id

By applying goto on id of I_0,

goto(I_0, id)
I_5 : F → id •

Since in I_5 there is no non-terminal to the right of dot we cannot apply closure function here. Thus we have completed applying gotos on I_0. We will consider I_1 for applying goto. In I_2 there are two productions E′→ E• and E → E• + T. There is no point applying goto on E′→ E•, hence we will consider E′→ E• +T for application of goto.

goto(I_1, +)
I_6 : E → E + • T
T →• T * F
T →• F
FT →• (E)
F →• id

There is no other rule in I_1 for applying goto. So we will move to I_2. In I_2 there are two productions E→ T• and T→ T•* F. We will apply goto on *.

goto(I_2, *)
I_7 : T → T * • F
F →• (E)
F →• id

The goto cannot be applied on I_3. Apply goto on E in I_4. In I_4 there are two productions having E after dot (F→• E and E→• E+ T). Hence we will apply goto on both of these productions. The I_8 becomes,

goto(I_4, E)
I_8 : F → (E•)
E → E • + T

If we will apply goto on (I_4, T) but we get E→ T• and T→ T•* F which is I_2 only. Hence we will not apply goto on T. Similarly we will not apply goto on F, (and id as we get the states I_3, I_4, I_5 again. Hence these gotos cannot be applied to avoid repetition.

There is no point applying goto on I_5. Hence now we will move ahead by applying goto on I_6 for T.

goto(I_6, T)
I_9 : E → E + T•
T → T • * F

Then,

goto(I_7, F)

I_{10} : T → T * F •

Then,

goto(I_8,))

I_{11} : F → (E)•

Applying goto on I_9, I_{10}, I_{11} is of no use. Thus now there is no item that can be added in the set of items. The collection of set of items is from I_0 to I_{11}.

Construction SLR parsing table -

As we have seen in the structure of SLR parser that are two parts of SLR parsing table and those are **action** and **goto**. By considering basic parsing actions such as shift, reduce, accept and error we will fill up the action table. The goto table can be filled up using goto function. Let us see the algorithm -

Input : An Augmented grammar G'

Output : SLR parsing table.

Algorithm :

1. Initially construct set of items C = {I_0, I_1, I_2...I_n} where C is a collection of set of LR(0) items for the input grammar G'.
2. The parsing actions are based on each item I_i. The actions are as given below.
   a. If A → α • aβ is in I_i and goto(I_i, a)= I_j then set action[i, a] as "shift j". Note that a must be a terminal symbol.
   b. If there is a rule A → α • is in I_i then set action[i, a] to "reduce A → α" for all symbols a, where a ∈ FOLLOW(A). Note that A must not be an augmented grammar S'.
   c. If S' → S is in I_i then the entry in the action table action[i, $] ="accept".
3. The goto part of the SLR table can be filled as : The goto transitions for state i is considered for non-terminals only. If goto(I_i, A) = I_j then goto[I_i, A] = j.
4. All the entries not defined by rule 2 and 3 are considered to be "error".

Example 3.7.4 *Construct the SLR parsing table for*

*1) E → E+T 2) E → T 3) T → T * F 4) T → F 5) F → (E) 6) F → id*

GTU : Summer-14, Marks 7

Solution : We will first construct a collection of canonical set of items for the above grammar. The set of items generated by this method are also called **SLR(0) items**. As there is no lookahead symbol in this set of items, zero is put in the bracket.

I_0 :
E'→ • E
E→ • E + T
E→ • T
T→ • T*F
T→ • F
F→ • (E)
F→ • id

goto (I_0, E)
I_1 : E'→ E •
E→ E • + T

goto (I_0, T)
I_2 : E→ T •
T→ T • * F

goto (I_0, F)
I_3 : T→ F •

goto (I_0, ( )
I_4 : T→ (• E)
E→ • E + T
E→ • T
T→ • T*F
T→ • F
F→ • (E)
F→ • id

goto (I_0, id)
I_5 : F→ id •

goto (I_1, +)
I_6 : E→ E + • T
T→ • T*F
T→ • F
F→ • (E)
F→ • id

goto (I_2, *)
I_7 : T→ T* • F
F→ • (E)
F→ • id

goto (I_4, E)
I_8 : F→ (E •)
E→ E • +T

goto (I_6, T)
I_9 : E→ E + T •
T→ T • *F

goto (I_7, F)
I_{10} : T→ T*F •

goto (I_8,))
I_{11} : F→ (E) •

We can design a DFA for above set of items as follows.

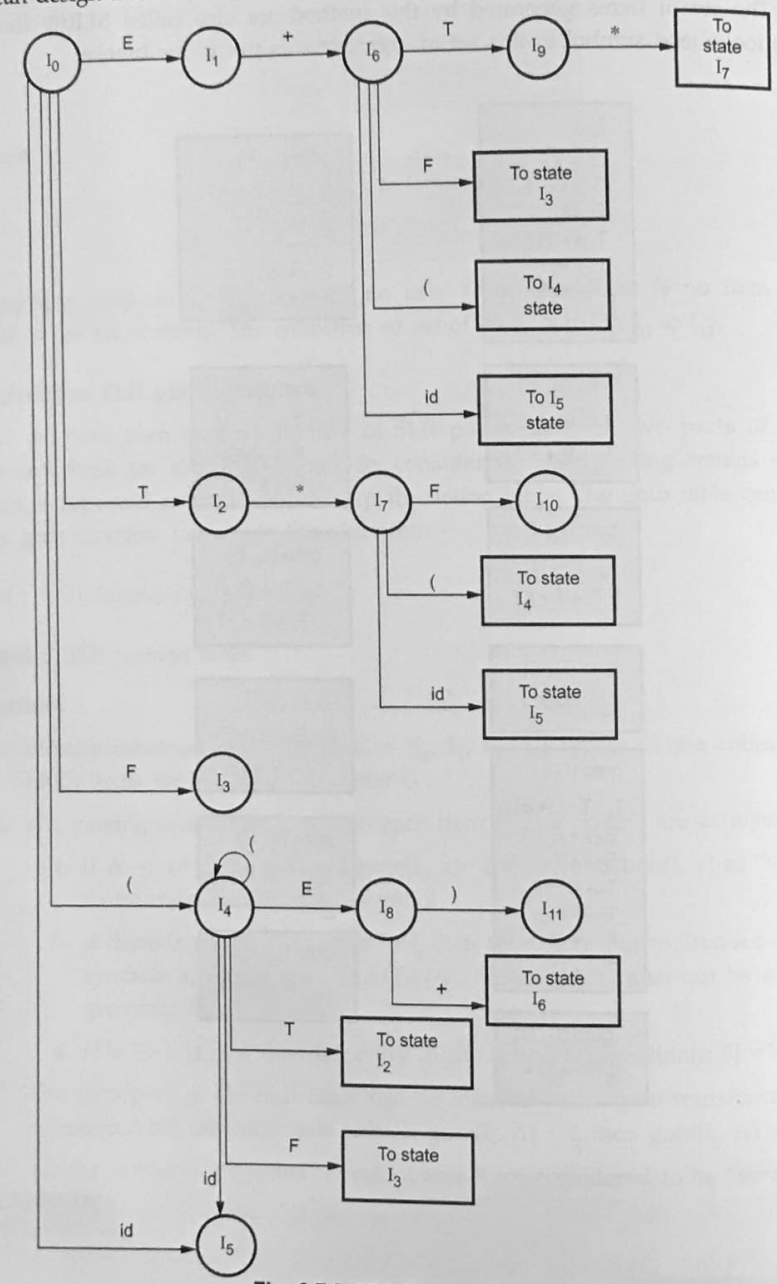
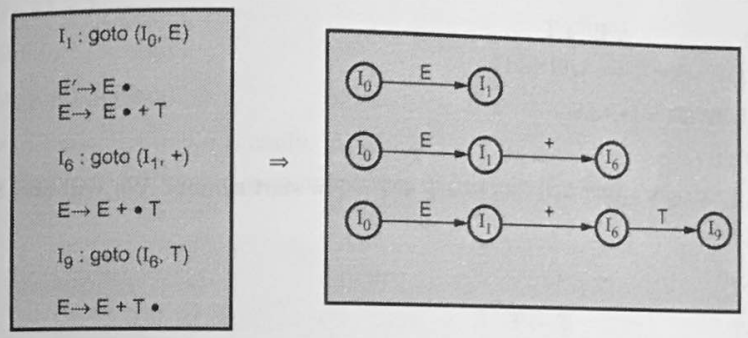


Fig. 3.7.2 DFA for set of items

In the given DFA every state is final state. The state I_0 is initial state. Note that the DFA recognizes viable prefixes.

For example - For the item,



The viable prefixes E, E+ and E+T are recognized here continuing in this fashion. The DFA can be constructed for set of items. Thus DFA helps in recognizing valid viable prefixes that can appear on the top of the stack.

Now we will also prepare a FOLLOW set for all the non-terminals because we require FOLLOW set according to rule 2.b of parsing table algorithm.

[**Note for readers :** Below is a manual method of obtaining FOLLOW. We have already discussed the method of obtaining FOLLOW by using rules. This manual method can be used for getting FOLLOW quickly]

FOLLOW(E') = As E' is a start symbol $ will be placed

= {$}

FOLLOW(E) :

$E' \rightarrow E$ that means E' = E = start symbol. ∴ we will add $.

$E \rightarrow E+T$ the + is following E. ∴ we will add +.

$F \rightarrow (E)$ the) is following E. ∴ we will add).

∴ FOLLOW(E) = {+,), $}

FOLLOW(T) :

As $E' \rightarrow E$, $E \rightarrow T$ the E' = E = T = start symbol. ∴ we will add $.

$E \rightarrow E+T$

$E \rightarrow T+ T$ $\because E \rightarrow T$

The + is following T hence we will add +.

$T \rightarrow T * F$

As * is following T we will add *.

$F \rightarrow (E)$

$F \rightarrow (T)$ $\quad \because E \rightarrow T$

As) is following T we will add)

$\therefore$ FOLLOW(T) = {+,*,),$}

FOLLOW(F) :

As $E' \rightarrow E$, $E \rightarrow T$ and $T \rightarrow F$ the E' = E = T = F = start symbol. We will add $.

$E \rightarrow E+T$

$E \rightarrow T+T$ $\quad \because E \rightarrow T$

$E \rightarrow F+T$ $\quad \because T \rightarrow F$

The + is following F hence we will add +.

$T \rightarrow T * F$

$T \rightarrow F * T$ $\quad T \rightarrow F$

As * is following F we will add *.

$F \rightarrow (E)$

$F \rightarrow (T)$ $\quad E \rightarrow T$

$F \rightarrow (F)$ $\quad T \rightarrow F$

As) is following F we will add).

FOLLOW(F) = {+, *,), $}

Building of parsing table

Now from canonical collection of set of items, consider I_0.

I_0 :

$E' \rightarrow \bullet E$

$E \rightarrow \bullet E+T$

$E \rightarrow \bullet T$

$T \rightarrow \bullet T * F$

$T \rightarrow \bullet F$

$F \rightarrow \bullet (E)$

$F \rightarrow \bullet id$

Consider $F \rightarrow \bullet (E)$

$A \rightarrow \alpha \bullet a\beta$

A = F, $\alpha = \varepsilon$, a = (, β = E)

$goto(I_0, ( ) = I_4$

$\therefore$ action[0, ( ] = shift 4

Similarly for $F \rightarrow \bullet$ id

Entry in the action table action[0, id] = shift 5 $\because goto(I_0, id) = I_5$

Other item in I_0 does not give any action. Hence we will find the actions from I_0 to I_{11}.

State	Action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4					
1		s6							
2			s7						
3									
4	s5			s4					
5									
6	s5			s4					
7	s5			s4					
8		s6			s11				
9			s7						
10									
11									

Thus SLR parsing table is filled up with the shift actions. Now we will fill it up with reduce and accept action.

According to the rule 2.c from parsing table algorithm, there is a production $E' \rightarrow E \bullet$ in I_1. Hence we will add the action "accept" in action[1, $].

Now in state I_2

$E \rightarrow T \bullet$

$A \rightarrow \alpha \bullet$ rule 2.b

A = E, α = T

FOLLOW(E) = {+,), $}

Hence add rule E → T in the row of state 2 and in the column of +,) and $. In the given example of grammar E → T is rule number 2. ∴ action[2,+] = r2, action[2,)]=r2, action[2,$]=r2.

Similarly, Now in state I_3

T → F •

A → α • rule 2.b

A=T, α= F

FOLLOW(F)={+,*,),$}

Hence add rule T→ F in the row of state 3 and in the column of +,*,) and $. In the given example of grammar T → F is rule number 4. Therefore action[3,+]=r4, action[3,)]=r4, action[3,$]=r4. Thus we can find the match for the rule A→ α • from remaining states I_4 to I_{11} and fill up the action table by respective "reduce" rule. The table with all the action[ ] entries will be

State	Action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4					
1		s6				Accept			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4					
5		r6	r6		r6	r6			
6	s5			s4					
7	s5			s4					
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Now we will fill up the goto table for all the non-terminals. In state I_0, goto(I_0, E) = I_1. Hence goto[0, E] =1, similarly goto(I_0, T) = I_2. Hence goto[0, T] = 2. Continuing in this fashion we can fill up the goto entries of SLR parsing table.

Finally the SLR(1) parsing table will look as -

State	Action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				Accept			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Remaining blank entries in the table are considered as syntactical errors.

Parsing the input using parsing table -

Now it's the time to parse the actual string using above parsing table. Consider the parsing algorithm.

Input : The input string w that is to be parsed and parsing table.

Output : Parse w if w ∈ L(G) using bottom-up. If w ∉ L(G) then report syntactical error.

Algorithm :

1. Initially push 0 as initial state onto the stack and place the input string with $ as end marker on the input tape.
2. If S is the state on the top of the stack and a is the symbol from input buffer pointed by a lookahead pointer then.
   a) If action[S, a] = shift j then push a, then push j onto the stack. Advance the input lookahead pointer.
   b) If action[S, a] = reduce A→ β then pop 2*|β| symbols. If i is on the top of the stack then push A, then push goto[i, A] on the top of the stack.

c) If action[S, a] = accept then halt the parsing process. It indicates the successful parsing.

Let us take one valid string for the grammar.

1) E → E+T
2) E → T
3) T → T * F
4) T → F
5) F → (E)
6) F → id

Input string : id * id+id

We will consider two data structures while taking the parsing actions and those are - Stack and input buffer.

Stack	Input buffer	Action table	goto table	Parsing action
$0	id*id+id$	[0,id]=s5		Shift
$0id5	*id+id$	[5,*]=r6	[0,F]=3	Reduce by F → id
$0F3	*id+id$	[3,*]=r4	[0,T]=2	Reduce by T → F
$0T2	*id+id$	[2,*]=s7		Shift
$0T2*7	id+id$	[7,id]=s5		Shift
$0T2*7id5	+id$	[5,+]=r6	[7,F]=10	Reduce by F → id
$0 T2*7F10	+id$	[10,+]=r3	[0,T]=2	Reduce by T → T * F
$0T2	+id$	[2,+]=r2	[0,E]=1	Reduce by E → T
$0E1	+id$	[1,+]=s6		Shift
$0E1+6	id$	[6,id]=s5		Shift
$0E1+6id5	$	[5,$]=r6	[6,F]=3	Reduce by F → id
$0E1+6F3	$	[3,$]=r4	[6,T]=9	Reduce by T → F
$0E1+6T9	$	[9,$]=r1	[0,E]=1	E → E +T
$0E1	$	Accept		Accept

In the above table at first row we get action[0, id] = s5, that means shift id from input buffer onto the stack and then push the state number 5. On the second row we get action[5, *] as r6 that means reduce by rule 6, F→ id, hence in the stack id is replaced by F. By referring goto[0, F] we get state number 3 hence 3 is pushed onto the stack. Note that for every reduce action goto is performed. This process of parsing is continued in this fashion and finally when we get action[1, $] = Accept we halt by successfully parsing the input string.

Example 3.7.5 *Consider the following grammar*

$E \rightarrow E + T \mid T$

$T \rightarrow TF \mid F$

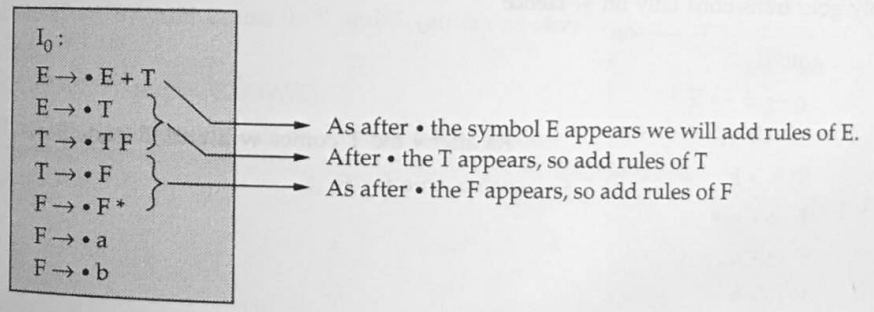
$F \rightarrow F * \mid a \mid b.$

*Construct the SLR parsing table for this grammar. Also parse the input a * b + a.*

Solution : Let us number the production rules in the grammar.

1) E → E + T
2) E → T
3) T → TF
4) T → F
5) F → F *
6) F → a
7) F → b

Now we will build the canonical set of SLR (0) items. We will first introduce an augmented grammar. We will first introduce an augmented grammar E' → • E and then the initial set of items I_0 will be generated as follows.



Now we will use goto function. From state I_0 goto on E, T, F and a, b will be applied step by step. Each goto transition will generate new state I_i.

I_1 : goto (I_0, E)

E' → E•

E → E• + T

I_2 : goto (I_0, T)

$E \rightarrow T \bullet$

$T \rightarrow T \bullet F$

$F \rightarrow \bullet F *$

$F \rightarrow \bullet a$

$F \rightarrow \bullet b$

I_3 : goto (I_0, F)

$T \rightarrow F \bullet *$

$F F \rightarrow F \bullet *$

I_4 : goto (I_0, a)

$F \rightarrow a \bullet$

I_5 : goto (I_0, b)

$F \rightarrow b \bullet$

Now we will start applying goto transitions on state I_1. From I_1 state it is possible to apply goto transitions only on +. Hence

I_6 : goto $(I_1, +)$

$E \rightarrow E + \bullet T$

$T \rightarrow \bullet TF$

$T \rightarrow \bullet F$

$F \rightarrow \bullet F *$

$F \rightarrow \bullet a$

$F \rightarrow \bullet b$

As after • the T comes will add T transitions same is true for F.

The goto transitions will be applied on I_2 state now. We will choose F to apply goto transition because there is no point in applying goto on T.

I_7 : goto (I_2, F)

$T \rightarrow TF \bullet$

$T \rightarrow F \bullet *$

If we apply goto on a or b from state I_2 then we will get $F \rightarrow a \bullet$ or $F \rightarrow b \bullet$ which are states I_4 and I_5 respectively.

Hence we will not consider I_4 and I_5 again. Now move to state I_3; from I_3 the goto transitions on * . Hence

I_8 : goto $(I_3, *)$

$F \rightarrow F \bullet *$

As there is no point in applying goto on state I_4 and I_5. We will choose state I_6 for goto transition.

I_9 : goto (I_6, T)

$E \rightarrow E + T \bullet$

$T \rightarrow T \bullet F$

$F \rightarrow \bullet F *$

$F \rightarrow \bullet a$

$F \rightarrow \bullet b$

Now we will first obtain FOLLOW of E, T and F. As the FOLLOW computation is required when the SLR parsing table is building.

FOLLOW(E)= {+, $}

FOLLOW(T)= {+, a, b, $}

FOLLOW(F)= {+, *, a, b, $}

As by rule 2 and 4, $E \rightarrow T$ and $T \rightarrow F$ we can state E = T = F. But E is a start symbol. Then by rule 2 and 4, T and F can act as start symbol. ∴ we have added $ in FOLLOW(E).

FOLLOW(T) and FOLLOW(F).

The SLR parsing table can be constructed as follows.

State	Action					goto		
	+	*	a	b	$	E	T'	F
0			s4	s5		1	2	3
1	s6				Accept			
2	r2		s4	s5	r2			7
3	r4	s8	r4	r4	r4			
4	r6	r6	r6	r6	r6			
5	r6	r6	r6	r6	r6			
6			s4	s5			9	3
7	r3	s8	r3	r3	r3			
8	r5	r5	r5	r5	r5			
9	r1		s4	s5	r1			7

Now we will parse the input a * b + a using above parse table.

Stack	Input buffer	Action
$0	a*b+a$	Shift
$0a4	*b+a$	reduce by F → a
$0F3	*b+a$	Shift
$0F3*8	b+a$	reduce by F → F *
$0F3	b+a$	reduce by T → F
$0T2	b+a$	Shift
$0T2b5	+a$	reduce by F → b
$0T2F7	+a$	reduce by T → TF
$0T2	+a$	reduce by E → T
$0E1	+a$	Shift
$0E1+6	a$	Shift
$0E1+6a4	$	reduce by F → a
$0E1+6F3	$	reduce by T → F
E1+6T9	$	reduce by E → E + T
$0E1	$	Accept

Thus input string is parsed.

Example 3.7.6 *Construct the collection of LR(0) item sets and draw the goto graph for the following grammar* $S \rightarrow S\ S\ |a|\varepsilon$. *Indicate the conflicts (if any) in the various states of the SLR parser.*

Solution : We will number the production rules in the grammar.

1) $S \rightarrow SS$

2) $S \rightarrow a$

3) $S \rightarrow \varepsilon$

Now let us construct canonical set of items -

I_0 : $S' \rightarrow \bullet S$ This is augmented grammar.
$S \rightarrow \bullet SS$
$S \rightarrow \bullet a$
$S \rightarrow \bullet$

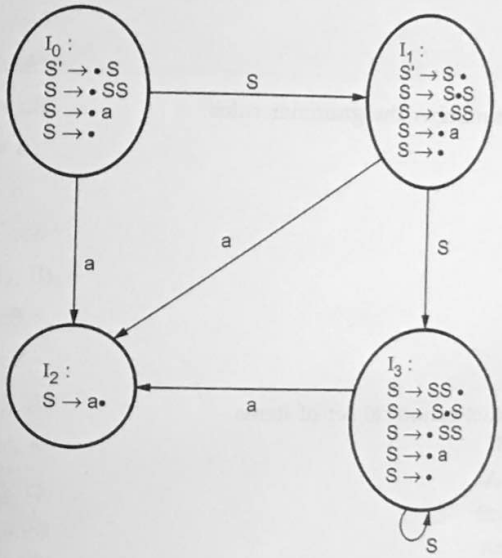
I_1 : goto (I_0, S)
$S' \rightarrow \bullet S$

$S \rightarrow S \bullet S$
$S \rightarrow \bullet SS$
$S \rightarrow \bullet a$
$S \rightarrow \bullet$

I_2 : goto (I_0, a)
$S \rightarrow a \bullet$

I_3 : goto (I_1, S)
$S \rightarrow SS \bullet$
$S \rightarrow S \bullet S$
$S \rightarrow \bullet SS$
$S \rightarrow \bullet a$
$S \rightarrow \bullet$

The goto graph will be



FOLLOW(S) = {a, $}

The construction of SLR(1) table will be.

In I_3 state there is a production rule $S \rightarrow \bullet\ a$. If we apply goto (I_3, a) then we will get I_2 state. Hence M[3, a] = S2. In I_3 there is a production $S \rightarrow SS \bullet$ which matches with $A \rightarrow \alpha\bullet$ Rule $S \rightarrow SS$ is rule number 1. And FOLLOW(S) = {a, $}

$\therefore$ M[3, a] = M[3, $] = r_1

	Action		Goto
	a	$	S
0	S2		1
1	S2	Accept	3
2	r2	r2	
3	S2 r1	r1	

Conflict occurs. [shift-reduce conflict]

Hence conflict occurs in state I_3.

Example 3.7.7 *Construct LR(0) parsing table for the following grammar*

$S \rightarrow cA \mid ccB$

$A \rightarrow cA \mid a$

$B \rightarrow ccB \mid b$

Solution : We will first number the grammar rules

1) S → cA
2) S → ccB
3) A → cA
4) A → a
5) B → ccB
6) B → b

Now we will construct canonical set of items

I_0 : S' → • S
S → • cA
S → • ccB
A → • cA
A → • a
B → • ccB
B → • b

I_1 : goto (I_0, S)
S' → S •

I_2 : goto (I_0, c)
S → c • A
S → c • cB

A → c • A
B → c • cB
A → • cA
A → • a

I_3 : goto (I_0, a)
A → a •

I_4 : goto (I_0, b)
B → b •

I_5 : goto (I_2, A)
S → cA •
A → cA •

I_6 : goto (I_2,c)
S → cc• B
B → cc• B
A → c•A
A → • cA
A → • a
B → • b
B → • ccB

I_7 : goto (I_6, B)
S → ccB •
B → ccB •

I_8 : goto (I_6, A)
A → cA •

I_9 : goto (I_6, c)
B → c • cB
A → c • A
A → • cA
A → • a

I_{10} : goto (I_9, c)
B → cc • B
A → c • A
A → • cA
A → • a

$B \rightarrow \bullet\ ccB$

$B \rightarrow \bullet\ b$

I_{11} : goto (I_{10}, B)

$B \rightarrow\ ccB \bullet$

FOLLOW(S) = { $ }

FOLLOW(A) = { $ }

FOLLOW(B) = { $ }

The SLR parsing table can be constructed as follows -

	Action				goto		
	a	b	c	$	s	A	B
0	S3	S4	S2		1		
1				Accept			
2	S3		S6			5	
3				r4			
4				r6			
5				r1/r3			
6	S3	S4	S9			8	7
7				r2/r5			
8				r3			
9	S3			S10		8	
10	S3	S4	S9			8	11
11				r5			

As there occurs reduce-reduce conflict given grammar is not SLR(0).

Example 3.7.8 *Check whether the following grammar is SLR(1) or not. Explain your answer with reasons.*

$S \rightarrow L = R$

$S \rightarrow R$

$L \rightarrow * R$

$L \rightarrow id$

$R \rightarrow L$

Solution : We will first build a canonical set of items. Hence a collection of LR(0) items is as given below.

I_0 : $S' \rightarrow \bullet S$
$S \rightarrow \bullet L = R$
$S \rightarrow \bullet R$
$L \rightarrow \bullet * R$
$L \rightarrow \bullet id$
$R \rightarrow \bullet L$

I_1 : goto (I_0, S)
$S' \rightarrow S\bullet$

I_2 : goto (I_0, L)
$S \rightarrow L \bullet = R$
$R \rightarrow L \bullet$

I_3 : goto(I_0, R)
$S \rightarrow R \bullet$

I_4 : goto(I_0, *)
$L \rightarrow * \bullet R$
$R \rightarrow \bullet L$
$L \rightarrow \bullet * R$
$L \rightarrow \bullet id$

I_5 : goto(I_0, id)
$L \rightarrow id \bullet$

I_6 : goto(I_2, =)
$S \rightarrow L = \bullet R$
$R \rightarrow \bullet L$
$L \rightarrow \bullet * R$
$L \rightarrow \bullet id$

I_7 : goto (I_4, R)
$L \rightarrow * R \bullet$

I_8 : goto (I_4, L)
$R \rightarrow L \bullet$

I_9 : goto(I_6, R)
$S \rightarrow L = R \bullet$

Now we will build the parsing table for the above set of items.

State	Action				goto		
	=	*	id	$	S	L	R
0							
1							
2	s6/r5						
3							
4		s4	s5				
5							
6		s4	s5				
7							
8							
9							

As we are getting multiple entries in Action [2, =] = s6 and r5. That means shift and reduce both a shift/reduce conflict occurs on input symbol =. Hence given grammar is not SLR(1) grammar.

Example 3.7.9 *Generate the SLR parsing table for the following grammar*

$S \rightarrow Aa \mid bAc \mid bBa$

$A \rightarrow d$

$B \rightarrow d$

GTU : Summer-12, Marks 8, Winter-19, Marks 7

Solution : We will number out the production rules

1) S → Aa
2) S → bAc
3) S → bBa
4) A → d
5) B → d

Now we will construct canonical set of items.

I_0 : S' → • S
S → • Aa
S → • bAc
S → • bBa
A → • d

I_1 : goto (I_0, S)
S' → S •

I_2 : goto (I_0, A)
S → A • a

I_3 : goto (I_0, b)
S → b • Ac
S → b • Ba
A → • d
B → • d

I_4 : goto (I_0, d)
A → d•

I_5 : goto (I_2, a)
S → Aa •

I_6 : goto (I_3, A)
S → bA • c

I_7 : goto (I_3, B)
S → bB • a

I_8 : goto (I_3, d)
A → d •
B → d •

I_9 : goto (I_6, c)
S → bAc •

I_{10} : goto (I_7, a)
S → bBa •

We obtain follow set for each nonterminal

FOLLOW (S) = { $ }

FOLLOW (A) = {a, c}

FOLLOW (B) = { a }

Now the SLR parsing table will be

	Action					goto		
	a	b	c	d	$	S	A	B
0		S3		S4		1	2	
1					Accept			
2	S5							
3				S8			6	7
4	r4		r4					
5					r1			
6			S9					
7	S10							
8	r4 r5		r4					
9					r2			
10					r3			

Example 3.7.10 *Write SLR parsing table for : S → T T → CC C → cC C → d*

GTU : Winter-13, Marks 8

Solution : Let, the given CFG is

S → T, T → CC C → cC C → d.

Let us number the production rules -

1. S → T
2. T → CC
3. C → cC
4. C → d

Now we will build the canonical set of SLR (0) items.

I_0 : S → •T
T → •CC
C → • c C
C → • d

I_1 : goto (I_0, T)
S → T·

I_4 : goto (I_0 , d)
C → d•

I_5 : goto (I_2, C)
T → CC•

I_6 : goto (I_3, C)
C → cC•

I_2 : goto (I_0, C) $T \rightarrow C \bullet C$ $C \rightarrow \bullet\ cC$ $C \rightarrow \bullet d$	We will compute FOLLOW FOLLOW (S) = {$} FOLLOW (T) = {$} FOLLOW (C) = {c,d}
I_3 : goto (I_0, c) $C \rightarrow c \bullet C$ $C \rightarrow \bullet\ cC$ $C \rightarrow \bullet d$	

Parsing table

State	Action			goto		
	c	d	$	S	T	C
0	S3	S4			1	2
1			ACCEPT			
2	S3	S4				5
3	S3	S4				6
4	r4	r4	r4			
5			r2			
6	r3	r3	r3			

Parsing of string cdcd will be

Stack	Input	Action
$ 0	cdcd $	shift 3
$ 0c3	dcd $	shift 4
$ 0c3d4	cd $	reduce by C → d
$ 0c3C6	cd $	reduce by C → cC
$ 0C2	cd $	Shift 3
$ 0C2c3	d $	Shift 4
$ 0C2c3d4	$	reduce by C → d

$ 0C2c3C6	$	reduce by C → cC
$ 0C2C5	$	reduce by T → CC
$ 0T1	$	ACCEPT

Example 3.7.11 *Show that the following grammer $S \rightarrow AaAb \mid BbBa$ $A \rightarrow \varepsilon$ $B \rightarrow \varepsilon$ is LL(1) but not SLR(1).* **GTU : Winter-14, Summer-17,19 Marks 7**

Solution : For LL(1) grammar refer example 3.3.16. To show that this grammar is not SLR(1) we will construct a canonical collection of set of item -

I_0 : $S \rightarrow \bullet\ AaAb$

$S \rightarrow \bullet\ BbBa$

$A \rightarrow \bullet$

$B \rightarrow \bullet$

I_1 : goto (I_0, A)

$S \rightarrow A \bullet aAb$

I_2 : goto (I_0, B)

$S \rightarrow B \bullet bBa$

I_3 : goto (I_1, a)

$S \rightarrow Aa \bullet Ab$

$A \rightarrow \bullet$

I_4 : goto (I_2, b)

$S \rightarrow Bb \bullet Ba$

$B \rightarrow \bullet$

I_5 : goto (I_3, A)

$S \rightarrow AaA \bullet b$

I_6 : goto (I_4, B)

$S \rightarrow BbB \bullet a$

I_7 : goto (I_5, b)

$S \rightarrow AaAb \bullet$

I_8 : goto (I_6, a)

$S \rightarrow BbBa \bullet$

The parsing table can be constructed as follows -

State	Action			goto		
	a	b	$	S	A	B
0	r3/r4	r3/r4			1	2
1	S3					
2		S4				
3	r3	r3			5	
4	r4	r4				6
5		S7				
6	S8					
7						
8						

As conflicting entries appear in above table. This grammar is not SLR(1).

Example 3.7.12 *Construct SLR parsing table for the following grammer.*

1) $E \rightarrow E - T \mid T$ 2) $T \rightarrow F \uparrow T \mid F$ 3) $F \rightarrow (E) \mid id$ **GTU : Summer-15, Marks 7**

Solution : We will construct a canonical set of items for given production rules.

I_0 :
$E' \rightarrow \bullet E$
$E \rightarrow \bullet E - T$
$E \rightarrow \bullet T$
$T \rightarrow \bullet F \uparrow T$
$T \rightarrow \bullet F$
$F \rightarrow \bullet (E)$
$F \rightarrow \bullet id$

I_1 : goto (I_0, E)
$E' \rightarrow E \bullet$
$E \rightarrow E \bullet - T$

I_2 : goto (I_0, T)
$E \rightarrow T \bullet$

I_3 : goto (I_0, F)
$T \rightarrow F \bullet \uparrow T$
$T \rightarrow F \bullet$

I_4 : goto $(I_1, ()$
$F \rightarrow (\bullet E)$
$E \rightarrow \bullet E - T$
$E \rightarrow \bullet T$
$T \rightarrow \bullet F \uparrow T$
$T \rightarrow \bullet F$
$F \rightarrow \bullet (E)$
$F \rightarrow \bullet id$

I_5 : goto (I_0, id)
$F \rightarrow id \bullet$

I_6 : goto $(I_1, -)$
$E \rightarrow E - \bullet T$
$T \rightarrow \bullet F \uparrow T$
$T \rightarrow \bullet F$
$F \rightarrow \bullet (E)$
$F \rightarrow \bullet id$

I_7 : goto $(I_3, \uparrow)$
$T \rightarrow F \uparrow \bullet T$
$T \rightarrow \bullet F \uparrow T$
$T \rightarrow \bullet F$
$F \rightarrow \bullet (E)$
$F \rightarrow \bullet id$

I_8 : goto (I_4, E)

$F \rightarrow (E \bullet)$

$E \rightarrow E \bullet - T$

I_9 : goto (I_4, T)

$E \rightarrow T\bullet$

I_{10} : goto (I_4, F)

$T \rightarrow F \bullet \uparrow T$

$E \rightarrow F\bullet$

I_{11} : goto (I_6, T)

$E \rightarrow E - T\bullet$

I_{12} : goto (I_7, T)

$T \rightarrow F \uparrow T \bullet$

I_{13} : goto (I_8,))

$F \rightarrow (E)\bullet$

We will number the production rules :

1) $E \rightarrow E - T$

2) $E \rightarrow T$

3) $T \rightarrow F \uparrow T$

4) $T \rightarrow F$

5) $F \rightarrow (E)$

6) $F \rightarrow id$

FOLLOW (E) = { –,), $}

FOLLOW (T) = { –,), $}

FOLLOW (F) = { –, ↑,), $}

The SLR parsing table will be,

	Action						goto		
	id	–	↑	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				Accept			

2		r1			r1	r1			
3		r4	s7		r4	r4			
4	s5			s4			8	9	10
5		r6	r6		r6	r6			
6	s5			s4				11	10
7	s5			s4				12	10
8		s6			s13				
9		r2			r2	r2			
10		r4	s7		r4	r4			
11		r1			r1	r1			
12		r3			r3	r3			
13		r5	r5		r5	r5			

Example 3.7.13 *Construct the collection of sets of LR(0) items for the following grammar.*

$S \rightarrow Aa|bAc|dc|bda$

$A \rightarrow d$

GTU : Summer-15, Marks 7

Solution : Let us number the production rules as below -

1) $S \rightarrow Aa$

2) $S \rightarrow bAc$

3) $S \rightarrow dc$

4) $S \rightarrow bda$

5) $A \rightarrow d$

We will construct collection of LR (0) items :

I_0 : $S' \rightarrow \bullet S$

$S \rightarrow \bullet Aa$

$S \rightarrow \bullet bAc$

$S \rightarrow \bullet dc$

$S \rightarrow \bullet bda$

$A \rightarrow \bullet d$

I_1 : goto (I_0, S)

$S' \rightarrow S \bullet$

I_2 : goto (I_0, A)

$S \rightarrow A \bullet a$

I_3 : goto (I_0, b)

$S \rightarrow b \bullet Ac$

$S \rightarrow b \bullet da$
$A \rightarrow \bullet d$

I_4 : goto (I_0, d)
$S \rightarrow d \bullet c$
$A \rightarrow d \bullet$

I_5 : goto (I_2, a)
$S \rightarrow Aa \bullet$

I_6 : goto (I_3, A)
$S \rightarrow bA \bullet c$

I_7 : goto (I_3, d)
$S \rightarrow bd \bullet a$
$A \rightarrow d \bullet$

I_8 : goto (I_4, c)
$S \rightarrow dc \bullet$

I_9 : goto (I_6, c)
$S \rightarrow bAc \bullet$

I_{10} : goto (I_7, a)
$S \rightarrow bda \bullet$

Example 3.7.14 *Construct SLR parsing table for the following grammar* $S \rightarrow 0S0|1S1|10$.

GTU : Winter-15, Marks 7

Solution : Let us number the production rules in grammar :

1) $S \rightarrow 0S0$ 2) $S \rightarrow 1S1$ 3) $S \rightarrow 10$

We will build canonical set of SLR (0) items :

I_0 : $S' \rightarrow \bullet S$
$S \rightarrow \bullet 0S0$
$S \rightarrow \bullet 1S1$
$S \rightarrow \bullet 10$

I_1 : goto (I_0, S)
$S' \rightarrow S \bullet$

I_2 : goto (I_0, 0)
$S \rightarrow 0 \bullet S0$
$S \rightarrow \bullet 0S0$
$S \rightarrow \bullet 1S1$
$S \rightarrow \bullet 10$

I_3 : goto (I_0, 1)
$S \rightarrow 1 \bullet S1$

$S \rightarrow 1 \bullet 0$
$S \rightarrow \bullet 0S0$
$S \rightarrow \bullet 1S1$
$S \rightarrow \bullet 10$

I_4 : goto (I_2, S)
$S \rightarrow 0S \bullet 0$

I_5 : goto (I_3, S)
$S \rightarrow 1S \bullet 1$

I_6 : goto (I_3, 0)
$S \rightarrow 10 \bullet$
$S \rightarrow 0 \bullet S0$
$S \rightarrow \bullet 0S0$
$S \rightarrow \bullet 1S1$
$S \rightarrow \bullet 10$

I_7 : goto (I_4, 0)
$S \rightarrow 0S0 \bullet$

I_8 : goto (I_5, 1)
$S \rightarrow 1S1 \bullet$

We will compute FOLLOW functions FOLLOW (S) = {0, 1}.

The SLR parsing table will be

	Action			S
	0	1	$	
0	S2	S3		1
1			Accept	
2	S2	S3		4
3	S6	S3		5
4	S7			
5		S8		
6	S2	S3		
7				
8				

Review Questions

1. *Explain SLR parser in detail with the help of an example.* **GTU : Winter-12, Summer-16, Marks 7**
2. *Explain SLR parsing method with example.* **GTU : Winter-16, Summer-18, Marks 7**

3.8 LR(k) Parser

GTU : Summer-12,17,18, Marks 7

The canonical set of items is the parsing technique in which a lookahead symbol is generated while constructing set of items. Hence the collection of set of items is referred as LR(1). The value 1 in the bracket indicates that there is one lookahead symbol in the set of items.

We follow the same steps as discussed in SLR parsing technique and those are

1. Construction of canonical set of items along with the lookahead.
2. Building canonical LR parsing table.
3. Parsing the input string using canonical LR parsing table.

Construction of canonical set of items along with the lookahead

1. For the grammar G initially add S'→• S in the set of item C.
2. For each set of items I_i in C and for each grammar symbol X (may be terminal or non-terminal) add closure (I_i, X). This process should be repeated by applying goto((I_i, X) for each X in I_i such that goto((I_i, X) is not empty and not in C. The set of items has to constructed until no more set of items can be added to C.
3. The closure function can be computed as follows.

   For each item $A \rightarrow \alpha \bullet X \beta$, a and rule $X \rightarrow \gamma$ and $b \in FIRST(\beta\ a)$

   such that $X \rightarrow \bullet\ \gamma$ and b is not in I then add $X \rightarrow \bullet\ \gamma$, b to I.
4. Similarly the goto function can be computed as : For each item $[A \rightarrow \alpha \bullet X\beta, a]$ is in I and rule$[A \rightarrow \alpha X \bullet \beta, a]$ is not in goto items then add $[A \rightarrow \alpha X \bullet \beta, a]$ to goto items.

This process is repeated until no more set of items can be added to the collection C.

Example 3.8.1 $S' \rightarrow S$

$S \rightarrow CC$

$C \rightarrow aC \mid d$

Construct LR(1) set of items for the above grammar.

Solution : We will initially add S'→• S, $ as the first rule in I_0. Now match S'→• S, $ with

$[A \rightarrow \alpha \bullet X\beta, a]$

Hence S'→ • S,$

$A \rightarrow \alpha \bullet X\beta, a$

$A = S', \alpha = \varepsilon, X = S, \beta = \varepsilon, a = \$$

If there is a production $X \rightarrow \gamma$, b then add $X \rightarrow \bullet\ \gamma$, b

$\therefore S \rightarrow \bullet\ CC$, $b \in FIRST\ (\beta\ a)$

$b \in FIRST(\varepsilon\ \$)$ as $\varepsilon\ \$ = \$$

$b \in FIRST(\$)$

$b = \{\$\}$

$\therefore S \rightarrow \bullet\ CC, \$$ will be added in I_0.

Now $S \rightarrow \bullet\ CC, \$$ is in I_0 we will match it with $A \rightarrow \alpha \bullet X\beta, a$

$A = S', \alpha = \varepsilon, X = C, \beta = C, a = \$$

If there is a production $X \rightarrow \gamma$, b then add $X \rightarrow \bullet\ \gamma$,b

$\therefore C \rightarrow \bullet\ aC$, $b \in FIRST(\beta\ a)$

$C \rightarrow \bullet\ d$ $b \in FIRST(C\$)$

$b \in FIRST(C)$ as $FIRST(C) = \{a, d\}$

$b = \{a,d\}$

$C \rightarrow \bullet\ aC$, a or d will be added in I_0.

Similarly $C \rightarrow \bullet\ d$, a/d will be added in I_0.

Hence

I_0 :
$S' \rightarrow \bullet\ S, \$$
$S \rightarrow \bullet\ CC, \$$
$C \rightarrow \bullet\ aC, a/d$
$C \rightarrow \bullet\ d, a/d$

Here a/d is used to denote a or d. That means for the production $C \rightarrow \bullet\ aC$, a and $C \rightarrow \bullet\ aC$, d.

Now apply goto on I_0.

$S' \rightarrow \bullet\ S, \$$
$S \rightarrow \bullet\ CC, \$$
$C \rightarrow \bullet\ aC, a/d$
$C \rightarrow \bullet\ d, a/d$

Hence

goto(I_0, S)
$I_1 : S' \rightarrow S \bullet, \$$

Now apply goto on C in I_0.

$S \rightarrow C \bullet\ C, \$$ add in I_2. Now as after dot C comes we will add the rules of C.

X = C, β = ε, a = $

X → • γ ,b where b ∈ FIRST(β a)

C → • aC b ∈ FIRST(ε $)

C → • d b ∈ FIRST($)

Hence

C→ • aC, $ and C→ • d, $ will be added to I_2

I_2 : goto(I_0, C)
S→ C• C, $
C→ • aC, $
C → • d, $

Now we will apply goto on a of I_0 for rule C → • aC, a/d that becomes C→ • aC, a/d will be added in I_3.

C→ a • C, a/d

As A = C, α = a, X = C, β = ε, a = a/d

Hence X → • γ , b

C → • aC b ∈ FIRST(β a)

C→ • d b ∈ FIRST(ε a) or FIRST(ε d)

b = a/d

Hence

I_3 : goto(I_0, a)
C→ a • C, a/d
C→ • aC, a/d
C→ • d, a/d

Now if we apply goto on d of I_0 in the rule C → • d, a/d then we get C→ d •, a/d hence I_4 becomes

I_4 : goto(I_0, d)
C → d •, a/d

As applying gotos on I_0 is over. We will move to I_1 but we get no new production from I_1 we will apply goto on C in I_2. And there is no closure possible in this state.

I_5 : goto(I_2, C)
S→ CC•, $

We will apply goto on a from I_2 on the rule C→ • aC, $ and we get the state I_6.

C→ a • C, $

A→ α • X β, a

A = C, α = a, X = C, β = ε, a = $

X → • γ

C→ • aC and C → • d

b ∈ FIRST(β a)

b ∈ FIRST(ε $)

b = $

I_6 : goto(I_2, a)
C → a • C, $
C → • aC, $
C→ • d, $

You can note one thing that I_3 and I_6 are different because the second component in I_3 and I_6 is different.

Apply goto on d of I_2 for the rule C→ • d,$.

I_7 : goto(I_2, d)
C → d •, $

Now if we apply goto on a and d of I_3 we will get I_3 and I_4 respectively and there is no point in repeating the states. So we will apply goto on C of I_3.

I_8 : goto(I_3, C)
C → aC •, a/d

For I_4 and I_5 there is no point in applying gotos. Applying gotos on a and d of I_6 gives I_6 and I_7 respectively. Hence we will apply goto on I_6 for C for the rule.

C → a • C, $

I_9 : goto(I_6, C)
C→ aC •, $

For the remaining states I_7, I_8 and I_9 we cannot apply goto. Hence the process of construction of set of LR(1) items is completed. Thus the set of LR(1) items consists of I_0 to I_9 states.

I_0:
S′→ • S, $
S→ • CC, $
C→ • aC, a/d
C→ • d, a/d

I_1: goto (I_0, S)
S′→ S •, $

I_2: goto (I_0, C)
S→ C • C, $
C→ • aC, $
C→ • d, $

I_3: goto (I_0, a)
C→ a • C, a/d
C→ • aC, a/d
C→ • d, a/d

I_4: goto (I_0, d)
C→ d • , a/d

I_5: goto (I_2, C)
S→ CC •, $

I_6: goto (I_2, a)
C′→ a• C, $
C→ • aC, $
C→ • d, $

I_7: goto (I_2, d)
C→ d•, $

I_8: goto (I_3, C)
C→ aC•, a/d

I_9: goto (I_6, C)
C→ aC•, $

Construction of canonical LR parsing table

To construct the canonical LR parsing table first of all we will see the actual algorithm and then we will learn how to apply that algorithm on some example. The parsing table is similar to SLR parsing table comprised of action and goto parts.

Input : An augmented grammar G′.

Output : The canonical LR parsing table.

Algorithm :

1. Initially construct set of items C = {I_0, I_1, I_2...I_n} where C is a collection of set of LR(1) items for the input grammar G′.
2. The parsing actions are based on each item I_i. The actions are as given below.
   a) If [A →α • a β, b] is in I_i and goto(I_i, a) = I_j then create an entry in the action table action[I_i, a] = shift j.

   b) If there is a production [A→ α • , a] in I_i then in the action table action[I_i, a] = reduce by A → α. Here A should not be S′.
   c) If there is a production S′ → S•, $ in I_i then action[i, $] = accept.
3. The goto part of the LR table can be filled as : The goto transitions for state i is considered for non-terminals only. If goto(I_i, A) = I_j then goto[I_i, A] = j.
4. All the entries not defined by rule 2 and 3 are considered to be "error".

Example 3.8.2 *Construct the LR(1) parsing table for the following grammar.*
1) $S \rightarrow CC$
2) $C \rightarrow aC$
3) $C \rightarrow d$

GTU : Summer-18, Marks 7

Solution : First we will construct the set of LR(1) items.

I_0:
S′→ • S, $
S→ • CC, $
C→ • aC, a/d
C→ • d, a/d

I_1: goto (I_0, S)
S′→ S •, $

I_2: goto (I_0, C)
S→ C • C, $
C→ • aC, $
C→ • d,$

I_3: goto (I_0, a)
C→ a • C, a/d
C→ • aC, a/d
C→ • d, a/d

I_4: goto (I_0, d)
C→ d • , a/d

I_5: goto (I_2, C)
S→ CC •, $

I_6: goto (I_2, a)
C′→ a• C, $
C→ • aC, $
C→ • d, $

I_7: goto (I_2, d)
C→ d•, $

I_8: goto (I_3, C)
C→ aC•, a/d

I_9: goto (I_6, C)
C→ aC•, $

The DFA for the set of items can be drawn as shown in Fig. 3.8.1 on next page.

Now consider I_0 in which there is a rule matching with $[A \rightarrow \alpha \bullet a\beta, b]$ as C→ • aC, a/d and if the goto is applied on a then we get the state I_3. Hence we will create entry action[0, a] = shift 3. Similarly,

In I_0

$C \rightarrow \bullet\ d$ a/d

$A \rightarrow \alpha \bullet a\ \beta$, b

A = C, $\alpha = \varepsilon$, a = d, $\beta = \varepsilon$, b = a/d

goto(I_0, d) = I_4

hence action[0, d] = shift 4

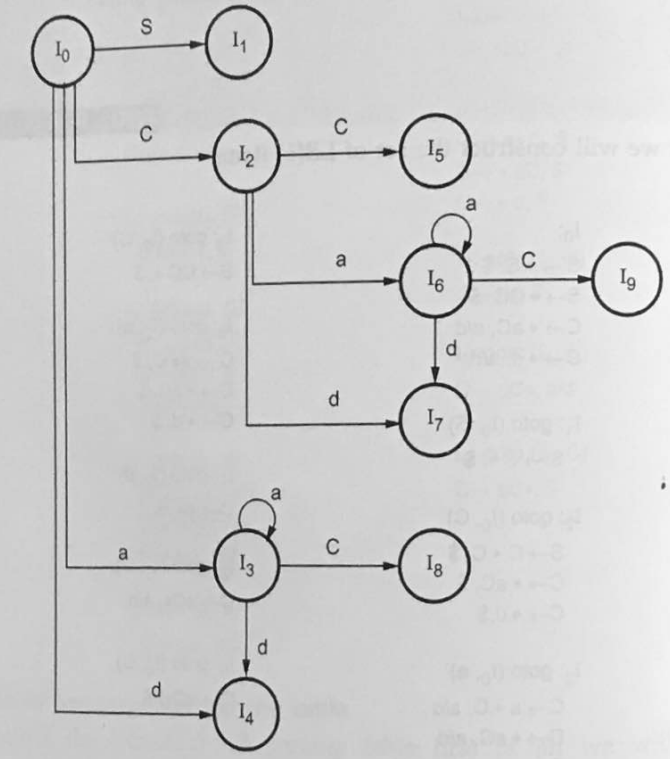


Fig. 3.8.1 DFA [goto graph]

For state I_4

$C \rightarrow d \bullet$, a/d

$A \rightarrow \alpha \bullet$, a

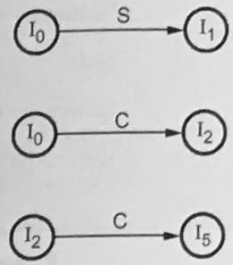
A = C, α = d, a = a/d

action[4, a] = reduce by C→ d i.e. rule 3

$S' \rightarrow S \bullet$, $ in I_1

So we will create action[1, $] = accept.

The goto table can be filled by using the goto functions.



goto (I_0, S) = I_1

goto (I_0, C) = I_2

goto (I_2, C) = I_5

For instance goto(I_0, S) = I_1. Hence goto[0, S] = 1. Continuing in this fashion we can fill up the LR(1) parsing table as follows.

	Action			goto	
	a	d	$	S	C
0	s3	s4		1	2
1			Accept		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

The remaining blank entries in the table are considered as syntactical error.

Parsing the input using LR(1) parsing table

Using above parsing table we can parse the input string"aadd" as

Stack	Input buffer	Action table	goto table	Parsing action
$0	aadd$	action[0, a]=s3		
$0a3	add$	action[3,a]=s3		Shift
$0a3a3	dd$	action[3,d]=s4		Shift
$0a3a3d4	d$	action[4,d]=r3	[3,C]=8	Reduce by C → d

$0a3a3C8	d$	action[8,d]=r2	[3,C]=8	Reduce by C → aC
$0a3C8	d$	action[8,d]=r2	[0,C]=2	Reduce by C → aC
$0C2	d$	action[2,d]=s7		Shift
$0C2d7	$	action[7,$]=r3	[2,C]=5	Reduce by C → d
$0C2C5	$	action[5,$]=r1	[0,S]=1	Reduce by S → CC
$0S1	$	accept		

Thus the given input string is successfully parsed using LR parser or canonical LR parser.

Example 3.8.3 *Construct a DFA whose states are the canonical collection of LR(1) items for the following augmented grammar,*

$S \rightarrow A$

$A \rightarrow BA|\varepsilon$

$B \rightarrow aB|b$

Solution : Initially we will start with the rule

$S \rightarrow \bullet A, \$$

We will add the rules $A \rightarrow \bullet BA$ and $A \rightarrow \bullet$ to I_0.

Now let us map the rule $[A \rightarrow \alpha \bullet X\beta, a]$ with $S \rightarrow \bullet A, \$$.

Such that $A = S,\ \alpha = \varepsilon,\ X = A,\ \beta = \varepsilon,\ a = \$$.

Then second component of $A \rightarrow \bullet BA$ and $A \rightarrow \bullet$

Will be $= \text{FIRST}(\beta a)$

$= \text{FIRST}(\varepsilon, \$)$

$= \text{FIRST}(\$)$

$= (\$)$

$\therefore$ $S \rightarrow \bullet A, \$$

$A \rightarrow \bullet BA, \$$

$A \rightarrow \bullet, \$$

As there is a rule $A \rightarrow \bullet BA$ we must add the rules $B \rightarrow \bullet a\ B$ and $B \rightarrow \bullet b$ to I_0. Now to compute second component of B' S rule we will use

$A \rightarrow \bullet BA, \$$ for mapping with $[A \rightarrow \alpha \bullet X\beta, a]$

$A = A,\ \alpha = \varepsilon,\ X = B,\ \beta = A,\ a = \$$.

Then second component of $B \rightarrow \bullet a\ B$ and $B \rightarrow \bullet b$

will be $= \text{FIRST}(\beta a)$

$= \text{FIRST}(A, \$)$

$= \text{FIRST}(A)$

$= \{a, b, \varepsilon\}$

$= \{a, b\}$

$\therefore$ $B \rightarrow \bullet a\ B, a/b$

$B \rightarrow \bullet b, a/b.$

Hence I_0 will be

$I_0 : S \rightarrow \bullet A, \$$

$A \rightarrow \bullet BA, \$$

$A \rightarrow \bullet, \$$

$B \rightarrow \bullet a\ B, a/b$

$B \rightarrow \bullet b, a/b$

I_1 : goto (I_0, A)

$S \rightarrow \bullet A, \$$

I_2 : goto (I_0, B)

$A \rightarrow B \bullet A, \$$

$A \rightarrow \bullet BA, \$$

$A \rightarrow \bullet, \$$

$B \rightarrow \bullet a\ B, a/b$

$B \rightarrow \bullet b, a/b$

I_3 : goto (I_0, a)

$B \rightarrow a \bullet B, a/b$

$B \rightarrow \bullet a\ B, a/b$

$B \rightarrow \bullet b, \quad a/b$

I_4 : goto (I_0, b)

$B \rightarrow b \bullet, \quad a/b$

I_5 : goto (I_2, A)

$A \rightarrow BA \bullet, \$$

I_6 : goto (I_2, B)

$B \rightarrow a\ B \bullet, a/b$

The DFA can be constructed as follows -

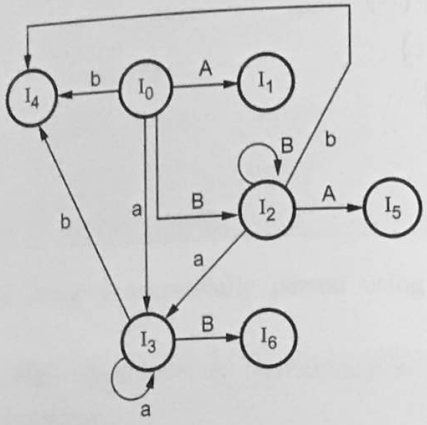


Fig. 3.8.2

Example 3.8.4 *Construct the canonical parsing table for the following grammar*

S'->S
S->CC
C->cC | d

GTU : Summer-17, Marks 7

Solution : Refer example 3.8.2 by assuming a = c

Example 3.8.5 *Construct CLR parsing table for following grammer.*

$S \rightarrow aSA \mid \varepsilon$
$A \rightarrow bS \mid c$

GTU : Summer-17, Marks 7

Solution :

1) $S \rightarrow aSA$
2) $S \rightarrow \varepsilon$
3) $A \rightarrow bS$
4) $A \rightarrow c$

We will construct canonical set of items using augmented grammar.

I_0 :

$S' \rightarrow \bullet S, \$$
$S \rightarrow \bullet aSA, \$$
$S \rightarrow \bullet \quad , \$$

I_1 : goto (I_0, S)

$S' \rightarrow S \bullet, \$$

I_2 : goto (I_0, a)

$S \rightarrow a \bullet SA, \$$
$S \rightarrow \bullet aSA, b/c$
$S \rightarrow \bullet \quad , b/c$

I_3 : goto (I_2, S)

$S \rightarrow aS \bullet A, \$$
$A \rightarrow \bullet bS, \$$
$A \rightarrow \bullet c, \ \$$

I_4 : goto (I_3, A)

$S \rightarrow aSA \bullet, \$$

I_5 : goto (I_3, b)

$A \rightarrow b \bullet S, \$$
$S \rightarrow \bullet aSA, \$$
$S \rightarrow \bullet , \$$

I_6 : goto (I_3, c)

$A \rightarrow c \bullet, \$$

I_7 : goto (I_5, S)

$A \rightarrow bS \bullet, \$$

The parsing table can be

	Action				Goto	
	a	b	c	$	S	A
0	S2			r2	1	
1				ACCEPT	3	
2	S2	r2	r2			4
3		S5	S6			
4				r1		
5	S2			r2	7	
6				r4		
7				r3		

3.9 LALR Parser

GTU : Winter-14,15,16,20, Marks 7

In this type of parser the **lookahead symbol** is generated for each set of item. The table obtained by this method are smaller in size than LR(k) parser. In fact the states of SLR and LALR parsing are always same. Most of the programming languages use LALR parsers.

We follow the same steps as discussed in SLR and canonical LR parsing techniques and those are

1. Construction of canonical set of items along with the lookahead.
2. Building LALR parsing table.
3. Parsing the input string using canonical LR parsing table.

Construction set of LR(1) items along with the lookahead

The construction LR(1) items is same as discussed in section 3.8. But the only **difference** is that : in construction of LR(1) items for LR parser, we have differed the two states if the second component is different but in this case we will merge the two states by merging of first and second components from both the states.

For example in section 3.8 (Refer Example 3.8.2) we have got I_3 and I_6 because of different second components, but for LALR parser we will consider these two states as same by merging these states. i.e.

$I_3 + I_6 = I_{36}$

Hence

I_{36} : goto(I_0, a)

C → a • C, a/d/$

C → • aC, a/d/$

C → • d , a/d/$

Let us take one example to understand the construction of LR(1) items for LALR parser.

Example 3.9.1 $S \rightarrow CC$

$C \rightarrow aC$

$C \rightarrow d$

Construct set of LR(1) items for LALR parser.

Solution : First we will construct set of LR(1) items.

I_0:
S′→ • S, $
S→ • CC, $
C→ • aC, a/d
C→ • d, a/d

I_1: goto (I_0, S)
S′→ S •, $

I_2: goto (I_0, C)
S→ C • C, $
C→ • aC, $
C→ • d,$

I_3: goto (I_0, a)
C→ a • C, a/d
C→ • aC, a/d
C→ • d, a/d

I_4: goto (I_0, d)
C → d •, a/d

I_5: goto (I_2, C)
S→ CC •, $

I_6: goto (I_2, a)
C′→ a• C, $
C→ • aC, $
C→ • d,$

I_7: goto (I_2, d)
C→ d•, $

I_8: goto (I_3, C)
C→ aC•, a/d

I_9: goto (I_6, C)
C→ aC•, $

Now we will merge states **3, 6** then **4, 7** and **8, 9**.

I_0:
S'→ • S, $
S→ • CC, $
C→ • aC, a/d
C→ • d, a/d

I_1: goto (I_0, S)
S'→ S •, $

I_2: goto (I_0, C)
S→ C • C, $
C→ • aC, $
C→ • d, $

I_5: goto (I_2, C)
S→ CC •, $

I_{89}: goto (I_3, C)
C→ aC • a/d/$

I_{36}: goto (I_0, a)
C→ a • C, a/d/$
C→ • aC, a/d/$
C→ • d, a/d/$

I_{47}: goto (I_0, d)
C→ d • , a/d/$

We have merged two states I_3 and I_6 and made the second component as a or d or $. The production rule will remain as it is. Similarly in I_4 and I_7. The set of items consist of states { I_0, I_1, I_2, I_{36}, I_{47}, I_5, I_{89}}

Construction of LALR parsing table -

The algorithm for construction of LALR parsing table is as given below.

Step 1 : Construct the LR(1) set of items.

Step 2 : Merge the two states I_i and I_j if the first component (i.e. the production rules with dots) are matching and create a new state replacing one of the older state such as $I_{ij} = I_i \cup I_j$

Step 3 : The parsing actions are based on each item I_i. The actions are as given below.
a) If [A → α • a β, b] is in I_i and goto(I_i, a) = I_j then create an entry in the action table action[I, a] = shift j.
b) If there is a production [A→ α • , a] in I_i then in the action table action [I_i, a]= reduce by A→ α. Here A should not be S'.
c) If there is a production S'→ S •, $ in I_i then action[i, $] = accept.

Step 4 : The goto part of the LR table can be filled as : The goto transitions for state i is considered for non-terminals only. If goto(I_i, A) = I_j then goto[I_i, A] = j.

Step 5 : If the parsing action conflict then the algorithm fails to produce LALR parser and grammar is not LALR(1). All the entries not defined by rule 3 and 4 are considered to be "error".

Example 3.9.2 $S \rightarrow CC$
$C \rightarrow aC$
$C \rightarrow d$

Construct the parsing table for LALR(1) parser. **GTU : Winter-20, Marks 7**

Solution : First the set LR(1) items can be constructed as follows with merged states.

I_0:
S'→ • S, $
S→ • CC, $
C→ • aC, a/d
C→ • d, a/d

I_1: goto (I_0,S)
S'→ S •, $

I_2: goto (I_0, C)
S→ C • C, $
C→ • aC, $
C→ • d, $

I_{36}: goto (I_0, a)
C→ a • C, a/d/$
C→ • aC, a/d/$
C→ • d, a/d/$

I_{47}: goto (I_0, d)
C→ d • , a/d/$

I_5: goto (I_2, C)
S→ CC •, $

I_{89}: goto (I_3, C)
C→ aC • a/d/$

Now consider state I_0 there is a match with the rule [A→α•aβ, b] and goto (I_i, a) = I_j.

C→• aC, a/d/$ and if the goto is applied on a' then we get the state I_{36}. Hence we will create entry action[0, a] = shift 36. Similarly,

In I_0

C →• d a/d

A →α• a β, b

A = C, α = ε, a = d, β = ε, b = a/d

goto(I_0, d) = I_{47}

Hence action[0, d] = shift 47

For state I_{47}

C→ d• , a/d/$

A→ α•,a

A = C, α = d, a = a/d/$

action[47, a] = reduce by C → d i.e. rule 3

action[47, d] = reduce by C → d i.e. rule 3

action[47, $] = reduce by C → d i.e. rule 3

S'→ S •, $ in I_1

So we will create action[1, $] = accept.

The goto table can be filled by using the goto functions.

For instance goto(I_0, S) = I_1. Hence goto[0, S] = 1. Continuing in this fashion we can fill up the LR(1) parsing table as follows.

State	Action			goto	
	a	d	$	S	C
0	s36	s47		1	2
1			Accept		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

The string belonging to given grammar can be parsed using LALR parser. The blank entries are supposed to be syntactical errors.

Parsing the input string using LALR parser

The string having regular expression = a*da*d ∈ grammar G. We will consider input string as "aadd" for parsing by using LALR parsing table.

Stack	Input buffer	Action table	Goto table	Parsing action
$0	aadd$	action[0,a]=s36		
$0a36	add$	action[36,a]=s36		Shift
$0a36a36	dd$	action[36,d]=s47		Shift
$0a36a36d47	d$	action[47,d]=r36	[36,C]=89	Reduce by C → d
$0a36a36C89	d$	action[89,d]=r2	[36,C]=89	Reduce by C → aC
$0a36C89	d$	action[89,d]=r2	[0,C]=2	Reduce by C → aC
$0C2	d$	action[2,d]=s47		Shift
$0C2d47	$	action[47,$]=r36	[2,C]=5	Reduce by C → d
$0C2C5	$	action[5,$]=r1	[0,S]=1	Reduce by S → CC
$0S1	$	accept		

Thus the LALR and LR parser will mimic one another on the same input.

Example 3.9.3 *Construct LALR parsing table for the following grammar :*

$S \rightarrow Aa$

$S \rightarrow bAc$

$S \rightarrow dc$

$S \rightarrow bda$

$A \rightarrow d$

Parse the input string bdc using table generated by you. **GTU : Winter-14, Marks 7**

Solution : Let us first number the production rules as below.

1) S → Aa
2) S → bAc
3) S → dc
4) S → bda
5) A → d

Now we will construct canonical set of LR(1) items for the above grammar.

I_0 :

S' → • S, $

S → • Aa, $

S → • bAc, $

S → • dc, $

S → • bda, $

A → • d, a

In above set of items we will start from S → • S, $. The second component is $ initially. After • the S comes, hence will add the rules deriving S. Now we have got the rule.

S → • Aa, $

It is resembling with A → α • x β, a

Now we can map x to A then the second component of X → is FIRST (β a)

In our rule

A → • d and second component is FIRST ($a) = a

Hence A → • d , a will be added in I_0.

I_1 : goto (I_0, S)

S' → S•, $ We will carry second component as it is.

I_2 : goto (I_0, A)

S → A • a, $

I_3 : goto (I_0, b)

$S \rightarrow b \bullet Ac, \$$

$S \rightarrow b \bullet da, \$$

$A \rightarrow \bullet d, c$

$$\left[\because S \rightarrow b \bullet Ac, \$ \text{ and } FIRST(\beta a) = FIRST(c\,\$) = c \cdot \text{Hence second component of } A \rightarrow \bullet d \text{ is } c\right]$$

I_4 : goto (I_0, d)

$S \rightarrow d \bullet c, \$$

$A \rightarrow d \bullet, a$

I_5 : goto (I_2, a)

$S \rightarrow Aa \bullet, \$$

I_6 : goto (I_3, A)

$S \rightarrow bA \bullet c, \$$

I_7 : goto (I_3, d)

$S \rightarrow bd \bullet a, \$$

$A \rightarrow d \bullet, c$

I_8 : goto (I_4, c)

$S \rightarrow dc \bullet, \$$

I_9 : goto (I_6, c)

$S \rightarrow bAc \bullet, \$$

I_{10} : goto (I_7, a)

$S \rightarrow bda \bullet, \$$

In above set of canonical items no states are having common production rules. Hence we cannot merge these states. The same set of items will be considered for building LALR parsing table.

We will construct LALR parsing table using following rules.

1. If $[A \rightarrow \alpha \bullet a\beta, b]$ is in I_i and goto $(I_i, a) = I_j$ then action [i, a] = Shift j.
2. If there is a production $[A \rightarrow \alpha \bullet, a]$ in some state I_i then action [i, a] = reduce by $A \rightarrow \alpha$.
3. If there is a product $S' \rightarrow S \bullet, \$$ in I_i then action [i, $] = accept.

State	Action					goto	
	a	b	c	d	$	S	A
0		s3		s4		1	2
1					Accept		
2	s5						
3				s7			6
4	r5		s8				
5							
6			s9				
7	s10		r5				
8					r3		
9					r2		
10					r4		

Consider the input "bdc" for parsing with the help of above LALR parsing table.

Stack	Input buffer	Action
$0	bdc$	Shift 3
$0b3	dc$	Shift 7
$0b3**d7**	c$	Reduce by A → d
$0b3A6	c$	Shift 9
$**0b3A6c9**	$	Reduce by S → bAc
$0S1	$	Accept

Thus the input string gets parsed completely.

Example 3.9.4 *Show that the following grammar.*

$S \rightarrow Aa \mid bAc \mid Bc \mid bBa$

$A \rightarrow d$

$B \rightarrow d$

is LR (1) but not LALR(1). **GTU : Winter-14, 15, Marks 7**

Solution : We will number out the production rules in given grammar.

1) $S \rightarrow Aa$
2) $S \rightarrow bAc$
3) $S \rightarrow Bc$
4) $S \rightarrow bBa$

5) $A \rightarrow d$

6) $B \rightarrow d$

Now we will first construct canonical set of LR(1) items.

$S' \rightarrow \bullet S, \$$

Initially for the augmented grammar the second component is \$. As this rule $[S' \rightarrow \bullet S, \$]$ is matching with $[A \rightarrow \alpha \bullet X \beta, a]$ where α is ε, X is S. B is ε and a is \$. We will apply closure on S and add all the production rules deriving S. The second component in these production rules will be \$. This is because for $[S' \rightarrow \bullet S, \$]$ the $\beta = \varepsilon$ and a = \$ ∴ FIRST (β a), FIRST (ε \$) = FIRST (\$) = \$. Hence we will get

$S' \rightarrow \bullet S, \$$

$S \rightarrow \bullet Aa, \$$

$S \rightarrow \bullet bAc, \$$

$S \rightarrow \bullet Bc, \$$

$S \rightarrow \bullet bBa, \$$

Now we have got one rule $[S \rightarrow \bullet Aa, \$]$ which is matching with $[A \rightarrow a \bullet X\beta, a]$ and $[X \rightarrow \gamma]$. Hence we will add the closure on A. Hence $[A \rightarrow \bullet d]$ will be added in this list. Now the second component of $A \rightarrow \bullet d$ will be decided. As $[S \rightarrow \bullet Aa, \$]$ is matching with $[A \rightarrow A \bullet X \beta, a]$ having X as A, β as a and a as \$.

Then FIRST (β a) = FIRST (a\$) = FIRST (a) = a. Hence rule $[A \rightarrow \bullet d, a]$ will be added.

Now,

$S' \rightarrow \bullet S, \$$

$S \rightarrow \bullet Aa, \$$

$S \rightarrow \bullet bAc, \$$

$S \rightarrow \bullet Bc, \$$

$S \rightarrow \bullet bBa, \$$

$A \rightarrow \bullet d, a$

Now notice the rule $S \rightarrow \bullet Bc, \$$. It suggest to apply closure on B (As after dot B comes immediately) This rule is matching with $[A \rightarrow \alpha \bullet X \beta, a]$ and $[X \rightarrow \gamma]$. Hence we will add rule deriving B. Hence $B \rightarrow \bullet d$ will be added in the above list. Now

$S \rightarrow \bullet Bc, \$$ can be mapped with $[A \rightarrow \alpha \bullet X \beta, a]$.

Where $\alpha = \varepsilon$

$X = B$

$\beta = c$

$a = \$$

Hence FIRST (β a) = FIRST (c \$) = FIRST (c) = c.

Hence rule $[B \rightarrow \bullet d, c]$ will be added.

Hence finally I_0 will be -

I_0 : $S' \rightarrow \bullet S$

$S \rightarrow \bullet Aa, \$$

$S \rightarrow \bullet bAc, \$$

$S \rightarrow \bullet Bc, \$$

$S \rightarrow \bullet bBa, \$$

$A \rightarrow \bullet d, a$

$B \rightarrow \bullet d, c$

Continue with applying goto on each symbol.

I_1 : goto (I_0, S)
$S' \rightarrow S \bullet, \$$

There is no chance of applying closure or goto in this state. Hence it will have only one rule.

I_2 : goto (I_0, A)
$S' \rightarrow A \bullet a, \$$

Applied goto on A. But as after dot a terminal symbol comes we cannot apply any rule further. The second component is carried as it is.

I_3 : goto (I_0, b)
$S \rightarrow b \bullet Ac, \$$
$S \rightarrow b \bullet Ba, \$$
$A \rightarrow \bullet d, c$
$B \rightarrow \bullet d, a$

After dot A comes hence rule for $A \rightarrow \bullet d$ will be added. As $[S \rightarrow b \bullet Ac, \$]$ is matching with $[A \rightarrow \alpha \bullet X \beta, a$ and $X \rightarrow \gamma]$, FIRST (β a) = FIRST (cs) = FIRST (c) = c. The second component of $A \rightarrow \bullet d$ is c.

I_4 : goto (I_0, B)
$S' \rightarrow B \bullet c, \$$

The goto on B is applied and second component is carried as it is.

I_5 : goto (I_0, d)
$A \rightarrow d \bullet, a$
$B \rightarrow d \bullet, c$

The goto on d in state I_0 is applied with corresponding second components as it is.

Continuing in this fashion we will get further states.

I_6 : goto (I_2, a)
$S \rightarrow Aa \bullet, \$$

I_9 : goto (I_3, d)
$A \rightarrow d \bullet, c$
$B \rightarrow d \bullet, a$

I_7 : goto (I_3, A)
$S \rightarrow bA \bullet c, \$$

I_{10} : goto (I_4, c)
$S \rightarrow Bc \bullet, \$$

I_8 : goto (I_3, B)
$S \rightarrow bB \bullet a, \$$

I_{11} : goto (I_7, c)
$S \rightarrow bAc \bullet, \$$

I_{12} : goto (I_8, a)
$S \rightarrow bBa \bullet, \$$

Now using the above set of LR(1) items we will construct LR(1) parsing table as follows.

State	Action					goto		
	a	b	c	d	$	S	A	B
0		s3		s5		1	2	4
1					ACCEPT			
2	s6							
3				s9			7	8
4			s10					
5	r5		r6					
6					r1			
7			s11					
8	s12							
9	r6		r5					
10					r3			
11					r2			
12					r4			

We can parse the string "bda" using above constructed LR (1) parsing table as :

Stack	Input buffer	Action
$0	bda$	Shift 3
$0b3	da$	Shift 9

$0b3d9	a$	Reduce by B → d
$0b3B8	a$	Shift 12
$0b3B8a12	$	Reduce by S → bBa
$0S1	$	Accept

Now we will construct a set of LALR(1) items. In this construction we will simply merge the states deriving same production rules which differ in their second components only. In above set of LR(1) items state I_5 and I_9 are such states i.e.

I_5 : goto (I_0, d)
$A \rightarrow d \bullet, a$
$B \rightarrow d \bullet, c$

I_9 : goto (I_3, d)
$A \rightarrow d \bullet, c$
$B \rightarrow d \bullet, a$

We will form only one state by merging state 5 and 9 as

I_{59} : goto (I_0, d)
$A \rightarrow d \bullet, a / c$
$B \rightarrow d \bullet, a / c$

Hence the LALR(1) set of items are given as :

I_0 : $S' \rightarrow \bullet S, \$$
$S \rightarrow \bullet Aa, \$$
$S \rightarrow \bullet bAc, \$$
$S \rightarrow \bullet Bc, \$$
$A \rightarrow \bullet d, a$
$B \rightarrow \bullet d, c$

I_1 : goto (I_0, S)
$S' \rightarrow S \bullet, \$$

I_2 : goto (I_0, A)
$S \rightarrow A \bullet a, \$$

I_3 : goto (I_0, b)
$S \rightarrow b \bullet Ac, \$$
$S \rightarrow b \bullet Ba, \$$
$A \rightarrow \bullet d, c$
$B \rightarrow \bullet d, a$

I_6 : goto (I_2, a)
$S \rightarrow Aa \bullet, \$$

I_7 : goto (I_3, A)
$S \rightarrow bA \bullet c, \$$

I_8 : goto (I_3, B)
$S \rightarrow bB \bullet a, \$$

I_{10} : goto (I_4, c)
$S \rightarrow Bc \bullet, \$$

I_{11} : goto (I_7, c)
$S \rightarrow bAc \bullet, \$$

I_{12} : goto (I_8, a)
$S \rightarrow bBa \bullet, \$$

I_4 : goto (I_0, B)

S → B•c, $

I_{59} : goto (I_0, d)

A → d• , a / c

B → d• , a / c

The LALR parsing table will be -

State	Action					goto		
	a	b	c	d	$	S	A	B
0		s3		s5		1	2	4
1					ACCEPT			
2	s6							
3				s9			7	8
4			s10					
59	r5 r6		r5 r6					
6					r1			
7			s11					
8	s12							
10					r3			
11					r2			
12					r4			

The parsing table shows multiple entries in Action [59, a] and Action [59, c]. This is called **reduce/reduce conflict**. Because of this conflict we cannot parse input.

Thus it is shown that given grammar is LR(1) but not LALR(1)

Example 3.9.5 *Construct LALR parser table for the following grammar for CLR parser.* $S \rightarrow L = R,\ S \rightarrow R, L \rightarrow *R,\ R \rightarrow L.$ **GTU : Winter-16, Marks 7**

Solution : To derive the given grammar using LALR method we have to build the canonical collection of items using LR(1) items.

The LR(1) items are

I_0 : S' → • S, $
S → • L = R $
S → • R, $
L → • * R, = |$
L → • id, =|$
R → • L, $

I_1 : goto (I_0, S)
S' → S •, $

I_2 : goto (I_0, L)
S → L • = R, $
R → L • , $

I_3 : goto (I_0, R)
S → R • , $

I_4 : goto (I_0, *)
L → * • R, = |$
R → • L, = |$
L → • * R, = |$
L → • id, = |$

I_5 : goto (I_0, id)
L → id • , = |$

I_6 : goto (I_2, =)
S → L = • R $
R → • L, $
L → • * R, $
L → • id, $

I_7 : goto (I_4, R)
L → * R • = |$

I_8 : goto (I_4, L)
R → L •, = |$

I_9 : goto (I_6, R)
S → L = R • , $

I_{10} : goto (I_6, L)
R → L • , $

I_{11} : goto (I_6 *)
L → * • R, $
R → • L, $
L → • * R, $
L → • id, $

I_{12} : goto (I_6, id)
L → id •, $

I_{13} : goto (I_{11}, R)
L → * R •, $

From above set of items we have got

i) I_4 and I_{11} give same production but lookheads are different. Hence merge them to form I_{411}

ii) $I_5 = I_{12}$ Hence I_{512}

iii) $I_7 = I_{13}$ Hence I_{713}

iv) $I_8 = I_{10}$ Hence I_{810}

Therefore the set of items for LALR are

I_0 : S' → • S, $

S → • L = R, $

S → • R, $

L → • * R, =|$

L → • id, =|$

R → • L , $

I_{713} : L → * R • , = |$

I_{810} : R → L • , = |$

I_9 : S → L = R • , $

I_1 : S' → S • , $

I_2 : S → L • , = R, $

R → L • , $

I_3 : S → R • , $

I_{411} : L → * • R, =|$

R → L • , = |$

L → • * R, = |$

L → • Id, = |$

I_{512} L → id • = |$

I_6 : S → L = • R, $

R → • L, $

L → • * R, $

L → • id, $

The parsing table can be constructed as follows -

	Action				goto		
State	id	*	=	$	S	L	R
0	S5	S4			1	2	3
1				acc			
2			S6	r5			
3				r2			
4	S5	S4				8	7
5			r4	r4			
6	S12	S11				10	9
7			r3	r3			
8			r5	r5			
9				r1			
10				r5			
11	S12	S11				10	13
12				r4			
13				r3			

The LALR parsing table is

	Action				goto		
State	id	*	=	$	S	L	R
0	S512	S411			1	2	3
1				acc			
2			S6	r5			
3				r2			
411	S512	S411				810	713
512			r4	r4			
6	S512	S411				810	9
713			r3	r3			
810			r5	r5			
9				r1			

University Question

1. *Explain LALR parser in detail. Support your answer with example.*

GTU : Winter-15, Marks 7

3.10 Comparison of LR Parsers

GTU : Summer-16,19, Winter-18, Marks 7

It's a time to compare SLR, LALR and LR parser for the common factors such as size, class of CFG, efficiency and cost in terms of time and space.

Sr. No.	SLR parser	LALR parser	Canonical LR parser
1.	SLR parser is smallest in size.	The LALR and SLR have the same size.	LR parser or canonical LR parser is largest in size.
2.	It is an easiest method based on FOLLOW function.	This method is applicable to wider class than SLR.	This method is most powerful than SLR and LALR.
3.	This method exposes less syntactic features than that of LR parsers.	Most of the syntactic features of a language are expressed in LALR.	This method exposes less syntactic features than that of LR parsers.
4.	Error detection is not immediate in SLR.	Error detection is not immediate in LALR.	Immediate error detection is done by LR parser.
5.	It requires less time and space complexity.	The time and space complexity is more in LALR but efficient methods exist for constructing LALR parsers directly.	The time and space complexity is more for canonical LR parser.

Graphical representation for the class of LR family is as given below.

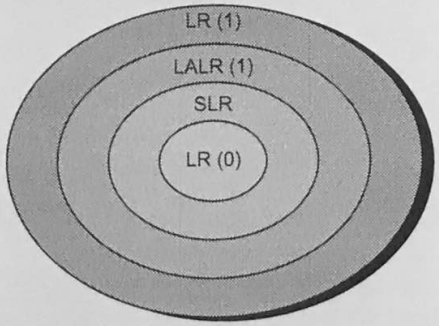


Fig. 3.10.1 Classification of grammars

Example 3.10.1 *Justify the statement "A class of grammar that can be parsed using LR methods is a proper subset of the class of grammars that can be parsed with predictive parser".*

Solution : Let us understand the term **proper subset.** Set A is a **proper** subset of B if and only if every element in A is also in B, and there exists **at least one element** in B that is **not** in A.

The Languages that can be parsed using predictive parser can also be parsed using LR methods. But there are some languages that can be parsed by LR methods and predictive parsers can not parse them. Hence it is said that the class of *grammar that can be parsed using LR methods is a proper subset of the class of grammars that can be parsed with predictive parser.*

Review Questions

1. *Differentiate SLR, Canonical LR and LALR. Also justify the statement " A class of grammar that can be parsed using LR methods is a proper subset of the class of grammars that can be parsed with predictive parser".* **GTU : Summer-16, Marks 7**
2. *Give the difference between SLR and CLR parser.* **GTU : Winter-18, Marks 3**
3. *Differentiate LR(1) and LALR(1) parsers.* **GTU : Summer-19, Marks 3**

3.11 Using Ambiguous Grammars

GTU : Winter-18, Marks 4

As we have seen various parsing methods in which if at all the grammar is **ambiguous** then it creates the **conflicts** and we cannot parse the input string with such ambiguous grammar. But for some languages in which arithmetic expressions are given ambiguous grammar are most compact and provide more natural specification as compared to equivalent unambiguous grammar. Secondly using ambiguous grammar we can add any new productions for special constructs.

While using ambiguous grammar for parsing the input string we will use all the disambiguating rules so that each time only one parse tree will be generated for that specific input. Thus ambiguous grammar can be used in controlled manner for parsing the input. We will consider some ambiguous grammar and let us try to parse some input string.

Using precedence and associativity to resolve parsing action conflicts

Consider an ambiguous grammar for arithmetic expression.

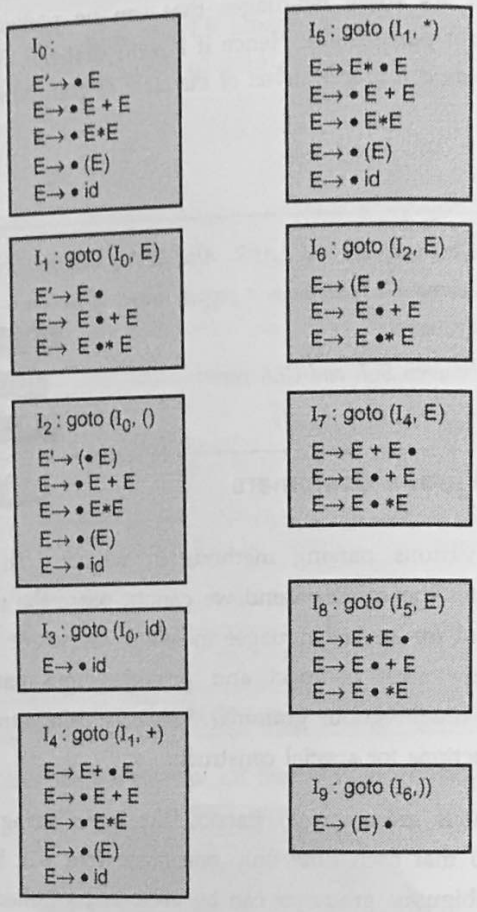
$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow id$

Now we will build the set of LR(0) items for this grammar.



Here FOLLOW(E) = {+, *,), $}.

We have computed FOLLOW (E) as it will be required while processing.

Now using the same rules of building SLR(0) parsing table we will generate the parsing table for above set of items.

State	Action						Goto
	id	+	*	(	)	$	E
0	s3			s2			1
1		s4	s5			Accept	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		s4 or r1	s5 or r1		r1	r1	
8		s4 or r2	s5 or r2		r2	r2	
9			r3		r3	r3	

As we can see in the parsing table the shift/reduce conflicts occur at state 7 and 8. We will try to resolve it, how? Let us consider one string **"id + id * id"**.

Stack	Input	Action with conflict resolution
$0	id + id * id$	Shift
$0id3	+ id * id$	Reduce by E → id
$0E1	+ id * id$	Shift
$0E1+4	id * id$	Shift
$0 E1+4id3	* id$	Reduce by E→ id
$0E1+4E7	*id$	The conflict can be resolved by shift 5 ←
$0 E1+4E7*5	id$	Shift
$0 E1+4E7*5id3	$	Reduce by E → id
$0 E1+4E7*5E8	$	Reduce by E → E * E
$0E1+4E7	$	Reduce by E → E + E
$0E1	$	Accept

As * has precedence over + we have to perform multiplication operation first. And for that it is necessary to push * on the top of the stack. The stack position will be

By this we can perform E * E first and then E+E. Therefore as you can see in the above table of parsing conflict can be resolved by assigning shift operation. Hence action [7, *] = s5.

Similarly if we consider the input **"id * id + id$"** for processing.

Stack	Input	Action with conflict resolution
$0	id *.id *+id$	Shift
$0id3	*id *id$	Reduce by E → id
$0E1	+id *id$	Shift
$0E1*5	id+id$	Shift
$0 E1*5id3	+id$	Reduce by E → id
$0E1*5E8	+id$	The conflict can be resolved by reduce by E → E * E ←
$0 E1	+id$	Shift
$0 E1+4	id$	Reduce by E → id
$0 E1+4id3	$	Reduce by E → E*E
$0E1+4E7	$	Reduce by E → E+E
$0E1	$	Accept

The conflict in action[8, +] = s4 or r2 can be resolved by allowing r2 action as we have to perform E * E operation first and then "id * id * id$" strings you can see that the conflicts for action[7, +] and action[8, *] can be resolved respectively. The resolution is action[7, +] = r1 and action[8, *] = r2.

Finally the ambiguous grammar without any conflict has following SLR parse table as -

State	Action						goto
	id	+	*	(	)	$	E
0	s3			s2			1
1		s4	s5			Accept	
2	s3			s2			
3		r4	r4		r4	r4	
4	s3			s2			
5	s3			s2			

6		s4	s5		s9	
7		r1	s5		r1	r1
8		r2	r2		r2	r2
9			r3		r3	r3

Example 3.11.1 *Construct SLR parsing table for the following grammar.*
$R \rightarrow R'|'R|RR|R^*|(R)|a|b$

Solution : Consider the grammar

$R \rightarrow R'|'R$

$R \rightarrow RR$

$R \rightarrow R^*$

$R \rightarrow (R)$

$R \rightarrow a$

$R \rightarrow b$

In this grammar, 'I' is an or operator.

The terminal symbols are = $\{1, *, (,), a, b\}$

The nonterminal symbols are = $\{R\}$

Now we will build the set of LR(0) items for this grammar will be

I_0 : $R' \rightarrow \bullet R$

$R \rightarrow \bullet R'|'R$

$R \rightarrow \bullet RR$

$R \rightarrow \bullet R*$

$R \rightarrow \bullet(R)$

$R \rightarrow \bullet a$

$R \rightarrow \bullet b$

I_1 : goto (I_0, R)

$R' \rightarrow R\bullet$

$R \rightarrow R\bullet'|'R$

$R \rightarrow R\bullet R$

$R \rightarrow R\bullet*$

$R \rightarrow \bullet R'|'R$

$R \rightarrow \bullet RR$

$R \rightarrow \bullet R*$

$R \rightarrow \bullet(R)$

$R \rightarrow \bullet a$

$R \rightarrow \bullet b$

I_2 : goto (I_0, ( )

$R \rightarrow (\bullet R)$

$R \rightarrow \bullet R'|'R$

$R \rightarrow \bullet RR$

$R \rightarrow \bullet R*$

$R \rightarrow \bullet(R)$

$R \rightarrow \bullet a$

$R \rightarrow \bullet b$

I_3 : goto (I_0, a)

$R \rightarrow a\bullet$

I_4 : goto (I_0, b)

$R \rightarrow b\bullet$

I_5 : goto (I_1,'|')

$R \rightarrow R'|'\bullet R$

$R \rightarrow \bullet R'I'R$

$R \rightarrow \bullet RR$

$R \rightarrow \bullet R*$

$R \rightarrow \bullet(R)$

$R \rightarrow \bullet a$

$R \rightarrow \bullet b$

I_6 : goto (I_1, R)

$R \rightarrow RR\bullet$

$R \rightarrow R\bullet'|'R$

$R \rightarrow R\bullet R$

$R \rightarrow R\bullet *$

$R \rightarrow \bullet R'|'R$

$R \rightarrow \bullet RR$

$R \rightarrow \bullet R*$

$R \rightarrow \bullet(R)$

$R \rightarrow \bullet a$

$R \rightarrow \bullet b$

I_7 : goto (I_1,*)

$R \rightarrow R*\bullet$

I_8 : goto (I_2, R)

$R \rightarrow (R\bullet)$

$R \rightarrow R\bullet'|'R$

$R \rightarrow R\bullet R$

$R \rightarrow R\bullet *$

$R \rightarrow \bullet RR$

$R \rightarrow \bullet R*$

$R \rightarrow \bullet(R)$

$R \rightarrow \bullet a$

$R \rightarrow \bullet b$

I_9 : goto (I_5, R)

$R \rightarrow R'|'R\bullet$

$R \rightarrow R\bullet'|'R$

$R \rightarrow R\bullet R$

$R \rightarrow R\bullet *$

$R \rightarrow \bullet R'|'R$

$R \rightarrow \bullet RR$

$R \rightarrow \bullet R*$

$R \rightarrow \bullet(R)$

$R \rightarrow \bullet a$

$R \rightarrow \bullet b$

I_{10} : goto (I_8,))

$R \rightarrow (R)\bullet$

The FOLLOW(R) = {'|', *,), a, b} is computed because it may be required in building SLR parsing table.

State	Action							goto
	'I'	*	(	)	a	b	$	R
0.			S2		S3	S4		1
1.	S5	S7			S3	S4	ACCEPT	6
2.			S2		S3	S4		8
3.	r5	r5		r5	r5	r5		
4.	r6	r6		r6	r6	r6		
5.			S2		S3	S4		9
6.	S5/r2	S7/r2	S2	r2	S3/r2	S4/r2		6
7.	r3	r3		r3	r3	r3		
8.	S5	S7	S2	S10	S3	S4		6
9.	S5/r1	S7/r1	S2	r1	S3/r1	S4/r1		6
10.	r4	r4		r4	r4	r4		

The above grammar is not SLR because the SLR parsing table contains shift reduce conflicts.

Review Question

1. List the different conflicts that occur in bottom up parsing and give examples for that.

GTU : Winter-18, Marks 4

3.12 Parser Generators

- Certain automation tools for parser generation are available. YACC is one such automatic tool for generating the parser program.

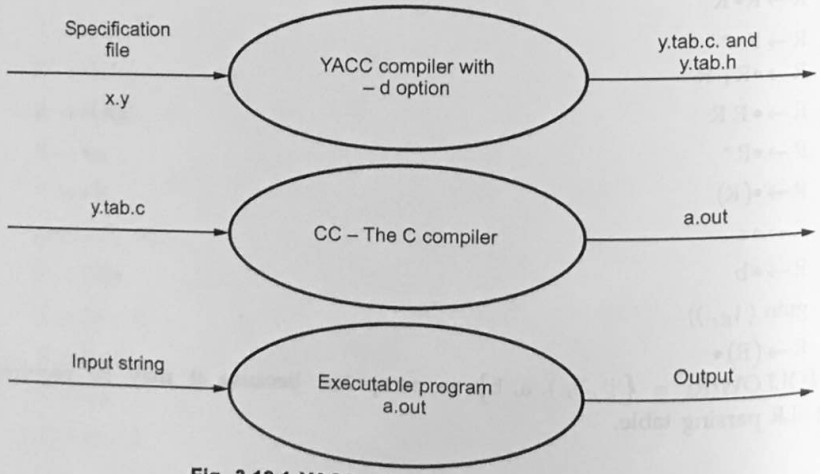


Fig. 3.12.1 YACC : Parser generator model

- YACC stands for **Yet Another Compiler Compiler** which is basically the utility available from UNIX.
- Basically YACC is **LALR parser** generator.
- The YACC can report conflicts or ambiguities (if at all) in the form of error messages.
- The LEX tool is for lexical analyzer. LEX and YACC work together to analyse the program syntactically.
- The typical YACC translator can be represented as shown in Fig. 3.12.1
- First we write a YACC specification file; let us name it as **x.y.** This file is given to the YACC compiler by UNIX command

  yacc x.y
- Then it will generate a parser program using your YACC specification file. This parser program has a standard name as **y.tab.c**. This is basically parser program in C generated automatically. You can also give the command with **– d** option

  yacc – d x.y

By – d option two files will get generated one is **y.tab.c** and other is **y.tab.h**. The header file y.tab.h will store all the tokens and so you need not have to create y.tab.h explicitly.

- The generated **y.tab.c** program will then be compiled by C compiler and generates the executable **a.out** file. Then you can test your YACC program with the help of some valid and invalid strings.
- Writing YACC specification program is the most logical activity. This specification file contains the context free grammar and using the production rules of context free grammar the parsing of the input string can be done by **y.tab.c.**

3.13 Automatic Generation of Parsers

First of all we will see the structure of YACC specification.

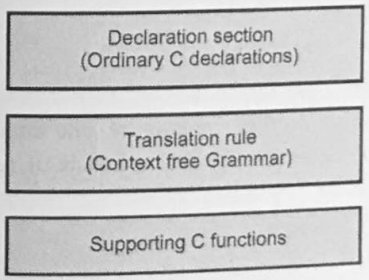


Fig. 3.13.1 Parts of YACC specification

The YACC specification file consists of three parts **declaration section, translation rule section** and **supporting C functions.**

The specification file with these sections can be written as

```
%{
/* declaration section */
%}
%%
 /* Translation rule section */
%%
/* Required C functions*/
```

1. **Declaration part :** In this section ordinary C declarations can be put. Not only this we can also declare grammar tokens in this section. The declaration of tokens should be within %{ and %}.
2. **The translation rule section :** It consists of all the production rules of context free grammar with corresponding actions. For instance

```
rule 1      action 1
rule 2      action 2
.
.
.
rule n      action n
```

If there are more than one alternatives to a single rule then those alternatives should be separated by | character. The actions are typical C statements. If CFG is

```
LHS →    alternative 1 |alternative 2 |.....|alternative n
Then
LHS :    alternative 1{ action 1}
       | alternative 2      { action 2}
         .
         .
         .
         alternative n      {action n}
         ;
```

3. **C functions section :** This section consists of one main function in which the routine yyparse() will be called. And it also consists of required C functions.

Programing Example : Write a YACC program for implementing desktop calculator.

Program : We will first create LEX program named calci.l then we will write program for YACC as calci.y. The extension to LEX program is .l and to YACC program is .y

LEX Program

/*Program name :calci.l*/

```
 %{
 #include "y.tab.h"  /*defines the tokens */
 #include <math.h>
 %}
 %%
      /*To recognize a valid number*/
([0-9]+|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?)  {yylval.dval = atof(yytext);
return NUMBER;}
      /*For log no | LOG no (log base 10)*/
log |
LOG {return LOG;}

      /*For ln no (Natural log)*/
ln    {return nLOG;}

      /*For sin angle*/
sin |
SIN  {return SINE;}

      /*For cos angle*/
cos |
COS {return COS;}

      /*For tan angle*/
tan |
TAN {return TAN;}

      /*For memory*/

mem          {return MEM;}

[ \t]  ;        /*Ignore white spaces*/

      /*End of input*/
\$    {return 0;}
      /*Catch the remaining and return a single character
        token to the parser*/
\n|.  return yytext[0];
%%
```

The YACC program

```
/*Program Name :calci.y */
%{
    double memvar;
%}

    /*To define possible symbol types*/
%union
{
    double dval;
}

    /*Tokens used which are returned by lexer*/
%token <dval> NUMBER
%token <dval> MEM
%token LOG SINE nLOG COS TAN

    /*Defining the precedence and associativity*/
%left '-' '+'                    /*Lowest precedence*/
%left '*' '/'
%right '^'
%left LOG SINE nLOG COS TAN   /*Highest precedence*/

    /*No associativity*/
%nonassoc UMINUS        /*Unary Minus*/

    /*Sets the type for non-terminal*/
%type <dval> expression

%%
    /*Start state*/
start:      statement '\n'
    |       start statement '\n'
    ;

            /*For storing the answer(memory)*/
statement: MEM '=' expression    {memvar = $3;}
    |       expression    {printf("Answer = %g\n",$1);}
    ;       /*For printing the answer*/
```



```
            /*For binary arithmetic operators*/
expression: expression '+' expression {$$ = $1 + $3;}
    |       expression '-' expression {$$ = $1 - $3;}
    |       expression '*' expression {$$ = $1 * $3;}
    |       expression '/' expression
            {       /*To handle divide by zero case*/
                    if($3 == 0)
                     yyerror("divide by zero");
                    else
                     $$ = $1 / $3;
            }
    |       expression '^' expression    {$$ = pow($1,$3);}
    ;

            /*For unary operators*/
expression: '-' expression  %prec UMINUS        {$$ = -$2;}
            /*%prec UMINUS signifies that unary minus should have
              the highest precedence*/
    |       '(' expression ')'              {$$ = $2;}
    |       LOG   expression                {$$ = log($2)/log(10);}
    |       nLOG  expression                {$$ = log($2);}
            /*Trignometric functions*/
    |       SINE   expression     {$$ = sin($2 * 3.141592654 / 180);}
    |       COS    expression     {$$ = cos($2 * 3.141592654 / 180);}
    |       TAN    expression     {$$ = tan($2 * 3.141592654 / 180);}
    |       NUMBER                {$$ = $1;}
    |       MEM                   {$$ = memvar;}
    ;       /*Retrieving the memory contents*/

%%
main()
{
    printf("Enter the expression: ");
    yyparse();
}
int yyerror(char *error)
{
fprintf(stderr,"%s\n",error);
}
```

Compiling and running of LEX and YACC programs

The output of the program can be obtained by following commands

```
[root@localhost]# lex calci.l
[root@localhost]# yacc –d calci.y
[root@localhost]# cc y.tab.c lex.yy.c -ll -ly -lm
[root@localhost]# ./a.out
Enter the expression : 2+2
Answer = 4

2 * 2 + 5 / 4
Answer = 5.25

mem = cos 45
sin 45 / mem
Answer = 1
ln 10
Answer = 2.30259
```

Note that if we use –d option then tab.h gets created automatically and we need not have to create explicitly

How to run lex and Yacc Programs together ?

lex calci. l ← will create lex.yy.c
yacc - d calci.y ← will creat y.tab.c
cc y.tab.c lex.yy.c – ll – ly – lm ← will compile both lex.yy.c and y.tab.c by linking various library file
./a .out ← will run executable file of program

Explanation of program

Declaration part -

%token is used to declare tokens in the YACC. In the above program token NUMBER can be declared as

%token <dval> NUMBER

The precedence and associativity can be declared in the above program %left, %right and %nonassoc declares the associativity of the operator being left associative, right associative or nonassociativity respectively.

The precedence can be declared in an increasing order.

The yylval used is the union of type double. YACC can associate data type with the token by using this yylval.

%token <dval> NUMBER

It means token NUMBER has the data type double. Any number of data types can be declared in union.

```
%left '-' '+'  /*Lowest precedence and left associative*/
%left '*' '/'
%right '^'    /*right associate*/
%left LOG SINE nLOG COS TAN    /*Highest precedence*/
      /*No associativity*/
```



```
%nonassoc UMINUS        /*Unary Minus*/
```

The operators on the same line has equal precedence. For instance '+' and '–' has the same precedence.

The shift/reduce conflict can be resolved by YACC with the help of these precedence rules. If the input token 'id' has more precedence then the shift action will be performed. If the precedence of production rule $A \rightarrow \alpha$ is greater than 'id' then reduce by $A \rightarrow \alpha$ will be performed. If there is same precedence then YACC checks for associativity. For left associativity shift action will be performed and for right associativity reduce action will be performed.

Rule section

In the rule section : (colon) is used to separate LHS and RHS of production rule. The termination of each rule is given by ' ; '. The $$ is used as attribute value at LHS of the grammar.

If there is a rule E+E then it has attribute values as $1+$3. Since E = $1, + is reserved for $2 and E = $3. To represent the RHS of the grammar $1, $2, ..., $n symbols are used. Finally the answer can be in $$ = $1.

Subroutine section

In the main important function **yyparse()** is called YACC invokes first **yyparse()** which inturn calls **yylex** when it requires tokens.

The routine **yyerror** is used to print the error message when an error is occurred in parsing of input.

Thus we have seen how an input string is parsed and syntactically checked.

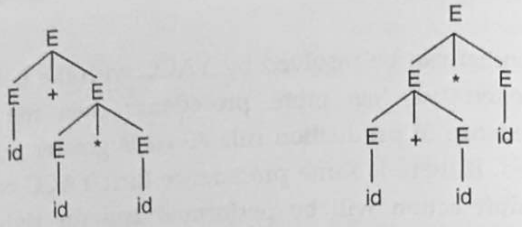
3.14 Short Questions and Answers

Q.1 Why lexical and syntax analyser are separated out ?

Ans : The lexical analyzer scans the input program and collects the tokens from it. On the other hand parser builds a parser tree using these tokens. These are two important activities and these activities are independently carried out by these two phases. Separating out these two phases has two advantages - Firstly it **accelerates** the process of **compilation** and secondly the **errors** in the source input can be **identified precisely.**

Q.2 What is ambiguous grammar ?

Ans : The ambiguous grammar is a grammar in which more than one parse trees can be generated for the same input. For example - The string id+id*id can be represented as -



(a) Parse tree 1 (b) Parse tree 2

Fig. 3.14.1

Q.3 What is recursive descent parsing ?

Ans : Recursive descent parsing is a kind of parsing in which the recursive procedures can be used for parsing the given input string. It is a top down parsing technique.

Q.4 What is the function of parser ?

Ans : There are two important functions that can be performed by the parser and those are - parsing of input string and generating error messages if there exists errors in the input.

Q.5 What is handle ?

Ans : "Handle of right sentential form γ is a production A->β and a position of γ where the string β may be found and replaced by A to produce the previous right sentential form in rightmost derivation of γ ".

Q.6 What is handle pruning ?

Ans : In bottom up parsing the process of detecting handle and using them in reduction is called handle pruning. For example -

Consider the grammar,

$E \rightarrow E+E$

$E \rightarrow id$

Now consider the string id + id + id and the rightmost derivation is

$E \underset{rm}{\Rightarrow} E + E$

$E \underset{rm}{\Rightarrow} E + E + E$

$E \underset{rm}{\Rightarrow} E + E+ id$

$E \underset{rm}{\Rightarrow} E + id + id$

$E \underset{rm}{\Rightarrow} id + id + id$

The bold strings are called handles.

Right sentential form	Handle	Production
id + id + id	id	E → id
E + id + id	id	E → id
E + E + id	id	E → id
E + E + E	E + E	E → E + E
E + E	E + E	E → E + E
E		

Q.7 What are the two important rules used in shift reduce parsing ?

Ans: Following are the two rules that can be used in shift reduce parsing -

- If the incoming operator has more priority than in stack operator then perform **shift**.
- If in stack operator has same or less priority than the priority of incoming operator then perform **reduce**.

Q.8 Explain the kernel and non kernel items

Ans : Kernel items - It is the collection of items S'-> • S and all the items whose dots are at the left end of the R.H.S of the rule.

Non Kernel items - It is the collection of all the items in which • are the left end of the R.H.S. of the rule.

Q.9 What is viable prefix ?

Ans : It is the set of prefixes in the right sentential form of a production A->α. This set can appear on the stack during shift/reduce action.

Q.10 What is the significance of lookahead operator?

Ans : The significance of lookahead symbols is that the parser can decide the reductions based on the k lookahead symbols. Hence this parsing procedure becomes an efficient one.

Q.11 What should the error handler in a parser do ?

Ans. : The error handler in-parser reports the syntactical error if any.

Q.12 Which of the parser-bottom-up or top-down parser-is called LR parser ? Why is it called LR ?

Ans. : The LR parsers are called bottom up parsers because in this type of parsing the input is parsed from bottom to top.

Q.13 Compare syntax tree and Parse tree.

Ans: Parse trees are much more detailed representation of the source language than that of syntax trees. Syntax trees are smaller than parse trees and these are much space and time efficient.

Q.14 Write the rule to eliminate left recursion in a grammar.

Ans: If the left recursive grammar is

$$\begin{matrix} A \to A\alpha \\ A \to \beta \end{matrix} \xrightarrow{\text{Eliminated as}} \begin{matrix} A \to \beta A' \\ A' \to \alpha A' \\ A' \to \varepsilon \end{matrix}$$

Q.15 Write a CF grammar to represent palindrome.

Ans : The context Free(CF) grammar to represent the palindrome is

CFG G=(V,T,P,S) where

V is set of nonterminal symbols = (S),

T is a set of terminal symbols =(a,b),

S is a start symbol

The set of production rules P = { S->aSa|bSb|a|b|∈ }

Q.16 What is the role of parser ?

Ans : Parser collects the tokens from the lexical analyzer in order to check the syntax of the source code. If any error is detected during the analysis of syntax, the syntax error messages are also displayed by the parser.

3.15 Multiple Choice Questions

Q.1 The syntax of the source language is checked by_____.

a lexical analyser
b syntax analysers
c semantic analysers
d code generators

Q.2 Which of the following is true about syntax analyzer ?

a It generates the syntactical errors if any
b It takes the tokens from the lexical analyzer
c It makes use of grammar for checking the syntax
d All of the above

Q.3 In compiler producing the parse tree is done by _____.

a scanner
b syntax analyzer
c code generator
d code optimizer.

Q.4 A grammar will be meaning less if _____.

a there is no terminal symbol on the right hand side.
b there is no ε symbol in the right hand side production
c the terminal set and non terminal set are not disjoint
d there are two non terminal symbols on the left hand side of the production.

Q.5 If w is a string of terminals and S and A are the non terminals then which of the following is left linear grammar ?

a S->Aw
b S->Aw|w
c S->wA|w
d S->wA

Q.6 If a is a terminal and S, P and Q are the non terminals then which of the following are the regular grammars?

a S-> ε, P->aS
b P->PaQ|aP
c S->PaQ|QaP
S->aP|a d

Q.7 The context free grammar can be recognized by _____.

a push down automata
b linear bounded automata
c finite automata
d regular expression

Q.8 Choose the correct statement.

a Syntax analyzer reports the logical errors as well.
b Lexical analyzer returns the tokens only after parser demands for it.
c Syntax analysis comes in synthesis phase of compilation.
d Syntax analysis checks the data type compatibility of the variables in the expression.

Q.9 Top down parser generates the _____.

a left most derivation
b right most derivation in reverse.
c right most derivation
d left most derivation in reverse.

Q.10 **Bottom up parser generates the _____.**

a left most derivation

b right most derivation in reverse.

c right most derivation

d left most derivation in reverse.

Q.11 **The recursive descent parsing is _____.**

a top down parsing method

b bottom up parsing method

c table driven parsing method

d None of the above.

Q.12 **The grammar E->E+E|E*E|id is _____.**

a unambiguous grammar

b ambiguous grammar

c ambiguous depends upon the input string

d unambiguous depends upon the input string.

Q.13 **Consider the grammar**

S->ABSc|Abc

BA->AB

Bb->bb

Ab->ab

Aa->aa

Which of the following string will be produced by it ?

a aabbcc b abbcc

c aabcc d abcc

Q.14 **Consider the grammar**

S->aB|bA

A->a|aS|bAA

B->b|bS|aBB

Which of the following string will be produced by it?

a aaabbabbba b bbabaaaa

c aaaaab d abab

Q.15 **Left factoring is the process of factoring out common _____.**

a suffixes of alternatives

b common symbols appearing anywhere in the rule

c prefixes of alternatives

d none of the above.

Q.16 **A grammar is said to be operator precedence if _____.**

a there is no production on the right side is ε

b there should not be any production rule possessing two adjacent non-terminals at the right hand side

c the language produced by this grammar contains the operators.

d all of the above.

Q.17 **The grammar for generating the language $a^n b^n c^n$ is _____.**

a context free grammar b regular grammar

c context sensitive grammar d none of the above.

Q.18 **Consider the grammar**

S->AaAb|BbBa

A-> ε

B-> ε

Which of the following is the FOLLOW of B ?

a a b b

c ε d none of the above

Q.19 **Choose the correct statement -**

a SLR ≤ LR(1) ≤ LALR b SLR ≤ LL(1) ≤ LALR

c SLR ≤ LALR ≤ LR(1) d LL(1) ≤ LALR ≤ SLR

Q.20 **Missing operator is _____.**

a lexical error b syntactical error

c semantic error d logical error

Q.21 YACC is _____.

a lexical analyzer generator
b parser generator
c code generator
d none of the above.

Q.22 YACC generates_____.

a SLR parsing table
b LALR parsing table
c canonical parsing table
d predictive parsing table.

Q.23 The most powerful parser is _____.

a Canonical LR parser
b SLR parser
c LALR parser
d LL(1) parser

Answers Keys for Multiple Choice Questions

Q. 1	b	Q.2	d	Q.3	b	Q.4	a, c	Q.5	b
Q. 6	c	Q.7	a, b	Q.8	b	Q.9	a	Q.10	b
Q. 11	a	Q.12	b	Q.13	a	Q.14	a,d	Q.15	c
Q. 16	d	Q.17	c	Q.18	a, b	Q.19	c	Q.20	b
Q. 21	b	Q.22	b	Q.23	a				

❑❑❑

4 Syntax Directed Translation

Syllabus

Syntax-Directed Definitions, Construction of Syntax Trees, Bottom-Up Evaluation of S-Attributed Definitions, L-Attributed Definitions, syntax directed definitions and translation schemes.

Contents

4.1 Introduction

Simply syntax analysis is not sufficient for the language to get compiled we need something more than the syntax analysis. Hence semantic analysis is done to handle issues that are beyond the syntactic definitions. In this chapter we will learn how to do the static analysis using syntax-directed definition. We will also discuss how attributes are associated with the terminal and non-terminal symbols of the grammar and then learn how the evaluation takes place.

4.2 Syntax Directed Definitions (SDD)

GTU : Winter-11,12,13,15,16,17,18,19,20, Summer-14,15,17,18,20, Marks 7

While doing the static analysis of the language we use syntax-directed definitions. That means an augmented context free grammar is generated. In other words the set of attributes are associated with each terminal and non-terminal symbols. The attribute can be a string, a number, a type, a memory location or anything else.

The syntax-directed definition is a kind of abstract specification. The conceptual view of syntax-directed translation can be as shown in Fig. 4.2.1.

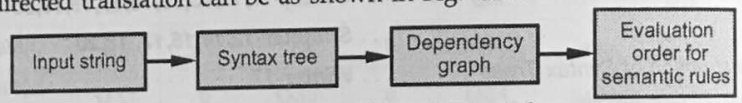


Fig. 4.2.1 Syntax-directed translation

Firstly we parse the input token stream and a syntax tree is generated. Then the tree is being traversed for evaluating the semantic rules at the parse tree nodes.

The implementation need not have to follow all the steps given in Fig. 4.2.1. In single pass implementation semantic rules can be evaluated during parsing without explicitly constructing a parse tree, or dependency graph. In such semantic evaluation, at the nodes of the syntax tree, values of the attribute are defined for the given input string. Such a parse tree containing the values of attributes at each node is called an **annotated or decorated the parse tree.**

Let us see the form of syntax-directed definition.

Definition : Syntax-directed definition is a generalization of context free grammar in which each grammar production $X \rightarrow \alpha$ is associated with it a set of semantic rules of the form $a := f(b_1, b_2, \ldots\ldots b_k)$, where a is an attribute obtained from the function f.

Attribute : The attribute can be a string, a number, a type, a memory location or anything else. Consider $X \rightarrow \alpha$ be a context free grammar and $a := f(b_1, b_2, \ldots\ldots b_k)$ where a is the attribute.

Then there are two types of attributes :

1. **Synthesized attribute** : The attribute 'a' is called synthesized attribute of X and $b_1, b_2, \ldots\ldots b_k$ are attributes belonging to the production symbols.

The value of synthesized attribute at a node is computed from the values of attributes at the children of that node in the parse tree.

2. **Inherited attribute** : The attribute 'a' is called inherited attribute of one of the grammar symbol on the right side of the production (i.e. α) and $b_1, b_2, \ldots\ldots b_k$ are belonging to either X or α .

The inherited attributes can be computed from the values of the attributes at the siblings and parent of that node.

1. Synthesized attribute

Let us see how to compute synthesized attributes.

Example :

Consider the context free grammar as

$S \rightarrow EN$

$E \rightarrow E + T$

$E \rightarrow E - T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow T/F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow$ digit

$N \rightarrow ;$

The syntax-directed definition can be written for the above grammar by using semantic actions for each production.

Production rule	Semantic actions
$S \rightarrow EN$	Print(E.val)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow E_1 - T$	$E.val := E_1.val - T.val$
$E \rightarrow T$	E.val:=T.val
$T \rightarrow T_1 * F$	$T.val := T_1.val \times F.val$

$T \rightarrow T_1 / F$	T.val:=T_1.val/F.val
$T \rightarrow F$	T.val:=F.val
$F \rightarrow (E)$	F.val:=E.val
$F \rightarrow$ digit	F.val:=digit.lexval
$N \rightarrow ;$	Can be ignored by lexical analyzer as ;is terminating symbol.

For the non-terminals E, T and F the values can be obtained using the attribute "val". Here "val" is a attribute and semantic rule is computing the value of val. (How ? that we will discuss shortly!)

The token digit has synthesized attribute *lexval* whose value can be obtained from lexical analyzer. In the rule S → EN, symbol S is the start symbol. This rule is to print the final answer of the expression.

- In syntax-directed definition, terminals have synthesized attributes only.
- Thus there is no definition of terminal. The synthesized attributes are quite often used in syntax-directed definition. The syntax-directed definition that uses only synthesized attributes is called **S-attributed definition**.
- In a parse tree, at each node the semantic rule is evaluated for annotating (computing) the S-attributed definition. This processing is in **bottom up** fashion i.e. from leaves to root.

Following steps are followed to **compute S-attributed definition**.

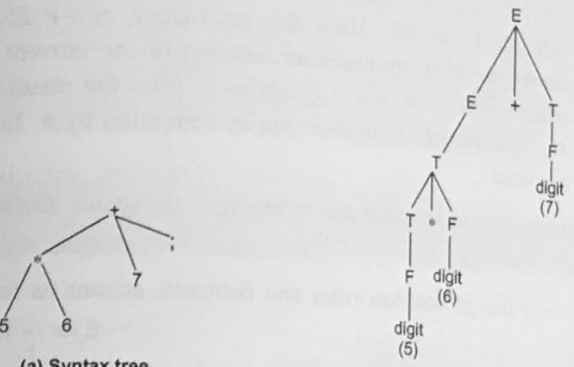
1. Write the syntax-directed definition using the appropriate semantic actions for corresponding production rule of the given grammar.
2. The annotated parse tree is generated and attribute values are computed. The computation is done in bottom up manner.
3. The value obtained at the root node is supposed to be the final output.

Let us take an input string for computing the S-attributed definition for the above given grammar.

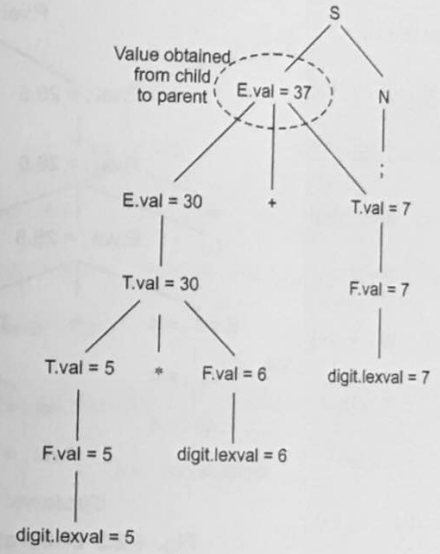
Example 4.2.1 *Construct parse tree, syntax tree and annotated parse tree for the input string is 5 * 6 + 7;*

Solution : Refer Fig. 4.2.2.

For the computation of attributes we start from the leftmost bottommost node. The rule **F → digit** is used in order to reduce digit to F. The semantic action that takes place here is **F.val: = digit.lexval**. The value of digit is obtained from lexical analyzer(here parser invokes the lexical analyzer to get the token value) which becomes the value of F. Hence **F.val=5**.



(a) Syntax tree (b) Parse tree



(c) Annotated parse tree

Fig. 4.2.2 Computation of S-attributed definition

Since T is the parent node of F and semantic action suggests that **T.val = F.val**. We can get the T.val=5. Thus the computation of S-attributes is done from children.

Then consider $T \rightarrow T_1 * F$ production; the corresponding semantic action is

$T.val = T_1.val \times F.val.$ Hence

$T.val = T_1.val \times F.val$

$= 5 * 6 = 30.$

Similarly, The combination of E_1.val + T.val becomes the E node.

∴ $E.val = E_1.val + T.val = 30 + 7$

$E.val = 37$

Here we get the E_1.val from left child of E and T.val from right child of E. Finally we acquire the value of E as 37. Then the production S → EN is applied to reduce E.val = 37. Then rule N→; indicates termination of the current expression. The semantic action associated with S → EN suggests us to print the result E.val. Hence the output will be 37. Thus S-attributed definition can be computed by a bottom-up fashion or using postorder traversal.

Example 4.2.2 *Construct a decorated parse tree according to the syntax directed definition for the following input statement : (4 + 7.5 * 3) / 2*

Solution : We will write the production rules and Semantic actions as follows :

Production rule	Semantic actions
S → E;	print (E.val)
E → E+ T	E.val : = E_1 · val + T · val
E → T	E.val : = T.val
T → T*P	T.val : = T_1 · val * P · val
T → P	T.val : = P.val
P → P / F	P.val : = P_1 · val / F · val
P → F	P.val : = F.val
F → (E)	F.val : = E.val
F → digit	F.val : = digit . lexval

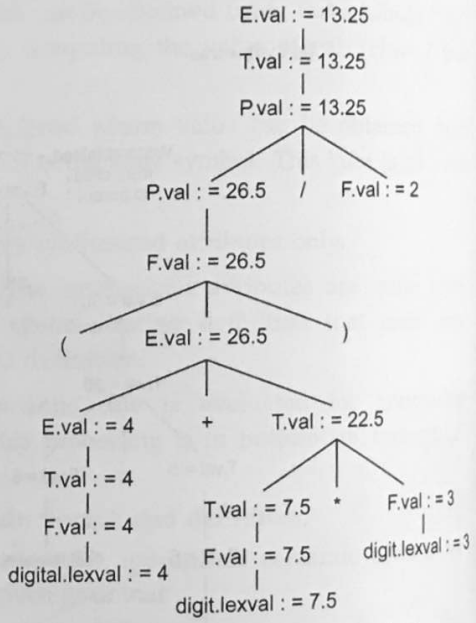


Fig. 4.2.3 Decorated parse tree

Example 4.2.3 *Give a syntax directed definition to differentiate expressions formed by applying the arithmetic operators + and * to the variable x and constants; expression : x * (3 * x + x * x).*

Solution : The syntax directed definition can be as given below :

Production rule	Semantic action
S → E;	Print (E.val)
E → E_1 + T	E.val : = E_1 · val + T · val
E → T	E.val : = T.val
T → T_1 * F	T.val : = T_i · val * F · val

T → F	T.val : = F.val
F → (E)	F.val : = E.val
F → digit	F.val : = digit.lexval
F → id	F.val : = id.entry

The annotated parse tree can be created as follows :

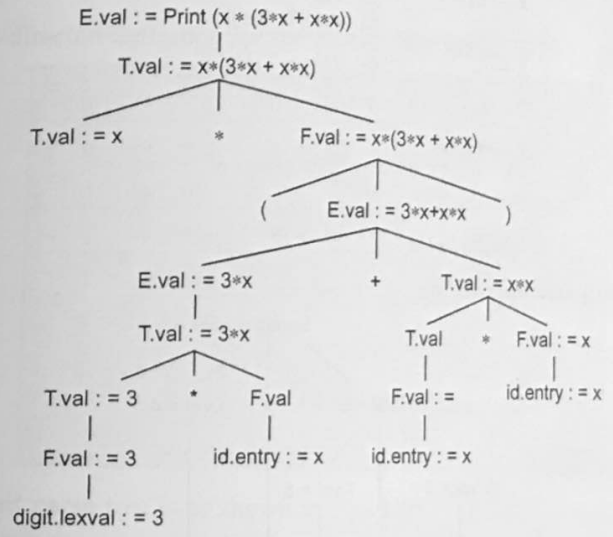


Fig. 4.2.4

Example 4.2.4 *Develop a syntax directed definition for following grammer*

E → TE'
E → +TE' |∈
T → (E)
T → id

GTU : Summer 17, Marks 7

Solution : The syntax directed definition is as follows.

Production rule	Semantic rule
E → TE′	E′•inh = T • val E′• val = E′•syn
E′ → +TE′	E'_1•inh = E′•inh + T • val E′•syn = E'_1•syn
E′ → ε	E′•syn = E′•inh
T → (E)	T • val = E•val
T → id	T • val = id•lexval

Example 4.2.5 *Give the translation scheme that converts infix to postfix notation. Generate the annotated parse tree for input string 3 – 5 + 4.* **GTU : Winter-17, Marks 3**

Solution : The translation scheme is

Production	Semantic Rule
$E \rightarrow E_1 + T$	$E_1 \cdot Val := E_1 \cdot Val \,\|\|\, T \cdot Val \,\|\|\, '+'$
$E \rightarrow E_1 - T$	$E_1 \cdot Val := E_1 \cdot Val \,\|\|\, T \cdot Val \,\|\|\, '-'$
$E \rightarrow T$	$E_1 \cdot Val := T \cdot Val$
$T \rightarrow 0$	$T \cdot Val := '0'$
$T \rightarrow 1$	$T \cdot Val := '1'$
...	...
$T \rightarrow 9$	$T \cdot Val := '9'$

The annotated parse tree will be

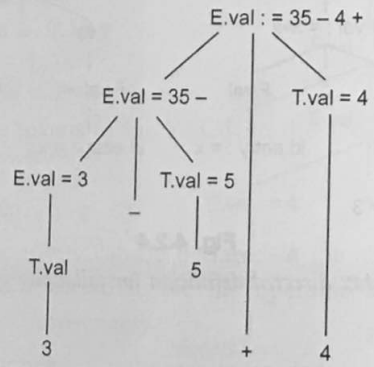


Fig. 4.2.5 : Annotated parse tree

2. Inherited attribute

The value of inherited attribute at a node in a parse tree is defined using the attribute values at the **parent** or **siblings**. Consider an example and let us compute the inherited attributes.

Annotate the parse tree for the computation of inherited attributes for the given string : int a, b, c; the grammar is as given below.

$S \rightarrow T\ L$

$T \rightarrow$ int

$T \rightarrow$ float

$T \rightarrow$ char

$T \rightarrow$ double

$L \rightarrow L_1$, id

$L \rightarrow$ pid.

For the string *int a, b, c* we have to distribute the data type **int** to all the identifiers **a, b** and **c**; such that **a** becomes **integer, b** becomes **integer** and **c** becomes **integer.** Following steps are to be followed :

1. Construct the syntax-directed definition using semantic action.
2. Annotate the parse tree with inherited attributes by processing in top-down fashion.

The syntax-directed definition for the above given grammar is

Production rule	Semantic actions
$S \rightarrow T\ L$	L.in:=T.type
$T \rightarrow$ int	T.type:=integer
$T \rightarrow$ float	T.type:=float
$T \rightarrow$ char	T.type:=char
$T \rightarrow$ double	T.type:=double
$L \rightarrow L_1$, id	L_1.in:= L.in Enter_type(id.entry, L.in)
$L \rightarrow$ id	Enter_type(id.entry, L.in)

The annotated parse tree is as shown in Fig. 4.2.5.

The value of L nodes is first obtained from T.type(sibling). The T.type is basically lexical value obtained as int or float or char or double. Then the L nodes give the type of the identifiers a, b and c. The computation of type is done in top-down manner or in preorder traversal. Using function Enter_type the type of identifiers a, b and c is inserted in the symbol table at corresponding id.entry (The id.entry is the address of corresponding identifier in the symbol table).

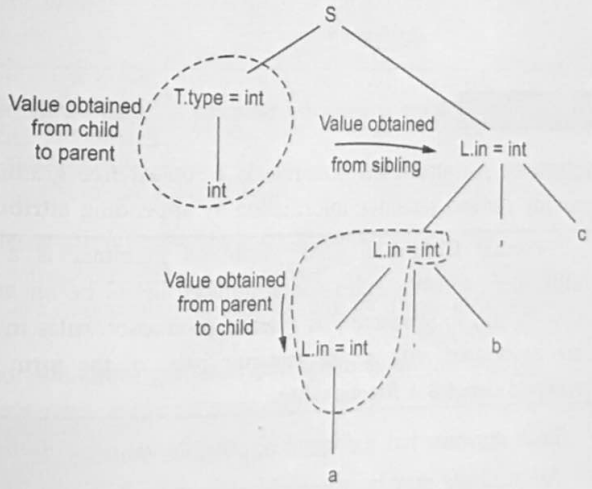
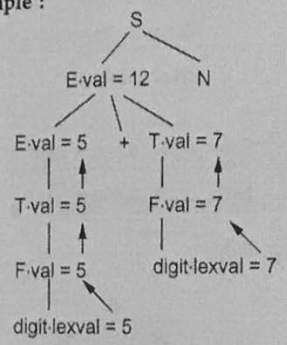
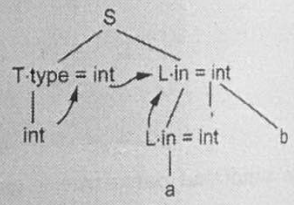


Fig. 4.2.6 Annotated parse tree

Sr. No.	Synthesized translation	Inherited translation
1.	The attribute 'a' is called synthesized attribute of X and $b_1, b_2, ..., b_k$ are attributes belonging to the production symbols. When $x \rightarrow \alpha$ is a rule associated with $a := f^*(b_1, b_2, ... b_k)$	The attribute 'a' is called inherited attribute of one of the grammar symbol on the right side of the production and $b_1, b_2, ..., b_k$ are belonging to either X or production α. When $x \rightarrow \alpha$ is a rule associated with $a := f^*(b_1, b_2, ... b_k)$
2.	In synthesized translation the synthesized attributes are computed.	In inherited translation, the inherited attributes are computed.
3.	The synthesized attributes are computed using **values of children.**	The inherited attributes are computed using values of **parent** and **sibling nodes.**
4.	Information is passed up in syntax tree.	Information is passed down in syntax tree.
5.	Computing synthesized attributes is **simple.**	Computing inherited attributes is **complex.**
6.	**Example :** 	**Example :** 

Example 4.2.6 *What is attributed Grammar ? Explain with suitable example.*

Solution : An attributed grammer is a context free grammar that has been extended to provide context sensitive information by appending **attributes** to its non-terminals.

Formal Definition - An attributed grammar is a context free grammar with attributes, semantic rules and condition. Let G be an attribute grammar denoted by $G = (N, \Sigma, P, S)$ where P is a set of production rules in the form $A \rightarrow \alpha$. These rules are associated with set of semantic rules of the form $b := f(a_1, a_2, a_3, ..., a_n)$ where f denotes a functionality.

- Each attribute has a domain of possible values.
- An attribute may be assigned values from its domain during parsing.
- The attributes can be evaluated in assignments or conditions.

There are two types of attributes

1) Synthesized attributes 2) Inherited attribute

An example of synthesized attribute

Production rule	Semantic action
$S \rightarrow EN$	Print (E.val)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	E.val = T.val
$T \rightarrow$ digit	T.val = digit.level
$N \rightarrow ;$	ignore

3. Dependency graph

The directed graph that represents the interdependencies between synthesized and inherited attributes at nodes in the parse tree is called dependency graph.

For the rule X→YZ the semantic action is given by X.x := f(Y.y, Z.z) then

Synthesized attribute is X.x and X.x depends upon attributes Y.y and Z.z.

Algorithm for Constructing dependency graph

```
for (each node n in the parse tree) do
{
    for (each attribute a of grammar at node n) do
    {
        for the attribute a, build a node of dependency graph.
        // Constructing all the nodes of graph.
    }
}
for (each node n in the parse tree) do
{
    for (each semantic rule b : = f (c1 , c2,... ck)
        which is associated with productions) do
        for ( i := 1) to k) do
        Consturct edge from node ci to b.
}
```

Example 4.2.7 *Design the dependency graph for the following grammar.*

$E \rightarrow E_1 + E_2$

$E \rightarrow E_1 * E_2$

Solution : The semantic rules for the above grammar is as given below.

Production rule	Semantic rule
$E \rightarrow E_1 + E_2$	$E \cdot val := E_1 \cdot val + E_2 \cdot val$
$E \rightarrow E_1 * E_2$	$E \cdot val := E_1 \cdot val \times E_2 \cdot val$

The dependency graph is as shown in Fig. 4.2.7.

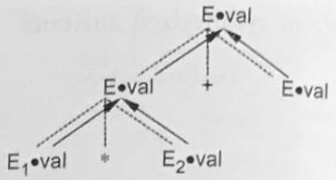


Fig. 4.2.7 Dependency graph

The synthesized attributes can be represented by •val. Hence the synthesized attributes are given by E•val, E_1•val and E_2•val. The dependencies among the nodes is given by solid arrows. The arrows from E_1 and E_2 show that value of E depends upon E_1 and E_2. We have represented parse tree using dotted lines.

Example 4.2.8 *Design the dependency graph for the following grammar.*

$S \rightarrow T\ List$
$T \rightarrow int$
$T \rightarrow float$
$T \rightarrow char$
$T \rightarrow double$
$List \rightarrow List_1, id$
$List \rightarrow id$

OR

Write grammar to declare variables with data type int, float or char. Also develop a syntax directed definition for that. Draw the dependency graph for same.

GTU : Summer-17, Marks 7

Solution : The dotted line is for representing the parse tree.

The semantic rules for the above grammar is as given below.

Production rule	Semantic actions
S → T List	List.in:=T.type
T → int	T.type:=integer
T → float	T.type:=float
T → char	T.type:=char
T → double	T.type:=double
List → $List_1$, id	$List_1$.in:=List.in Enter_type(id.entry, List.in)
List → id	Enter_type(id.entry, List.in)

The dependency graph is as shown in Fig. 4.2.8.

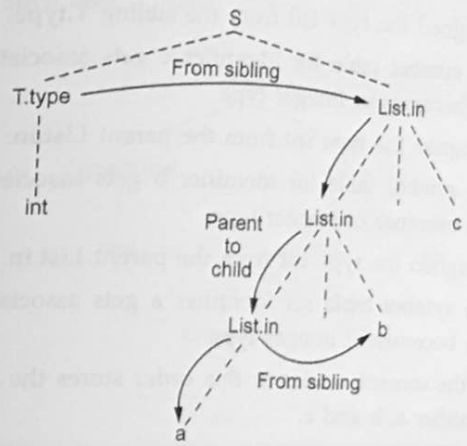


Fig. 4.2.8 Dependency graph

The dependencies among the nodes can be shown be solid arrows. In the above drawn dependency graph how the values can be inherited from the parent or sibling node is shown clearly. Hence the name for the attributes is inherited attributes.

4. Evaluation order

The **topological sort** of the dependency graph decides the evaluation order in a parse tree. In deciding evaluation order the semantic rules in the syntax-directed definitions are used. Thus the translation is specified by **syntax-directed definitions.** Therefore the precise definition of syntax-directed definition is required.

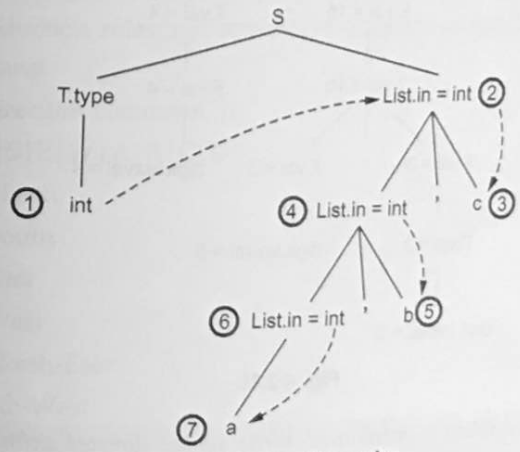


Fig. 4.2.9 Evaluation order

The evaluation order can be decided as follows.

1. The type **int** is obtained from lexical analyzer by analyzing the input token.

2. The **List.in** is assigned the type **int** from the sibling **T.type**.
3. The entry in the symbol table for identifier **c** gets associated with the type int. Hence variable **c** becomes of integer type.
4. The **List.in** is assigned the type **int** from the parent **List.in**.
5. The entry in the symbol table for identifier b gets associated with the type int. Hence variable **b** becomes of integer type.
6. The **List.in** is assigned the type **int** from the parent **List.in**.
7. The entry in the symbol table for identifier a gets associated with the type int. Hence variable **a** becomes of integer type.

Thus by evaluation the semantic rules in this order stores the type **int** in the symbol table entry for each identifier a, b and c.

Example 4.2.9 *Write a syntax directed definition for desk calculator. Justify whether this is an S-attributed definition or L-attributed definition. Using this definition draw annotated parse tree for 3*5+4n.* **GTU : Winter-11,15, Marks 7**

Solution : Syntax Directed Definition : Refer section 4.2 (1).

The annonated parse tree for 3 * 5 + 4n will be

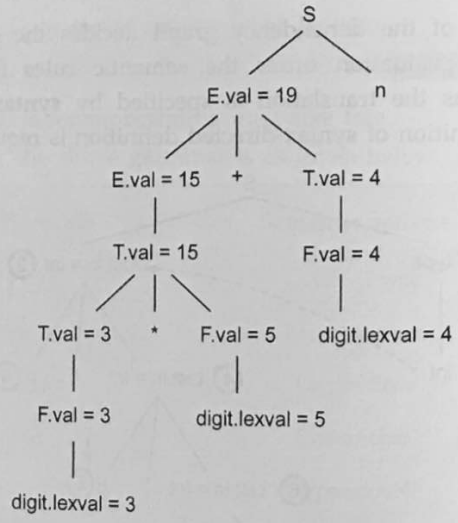


Fig. 4.2.10

Example 4.2.10 *What is inherited attribute ? Write syntax directed definition with inherited attributes for type declaration for list of identifiers. Show annotated parse tree for the sentence real id_1, id_2, id_3.* **GTU : Winter-11, Marks 7**

Solution : Inherited attributes - Refer section 4.2(2). The annotated parse tree real id_1, id_2, id_3 is -

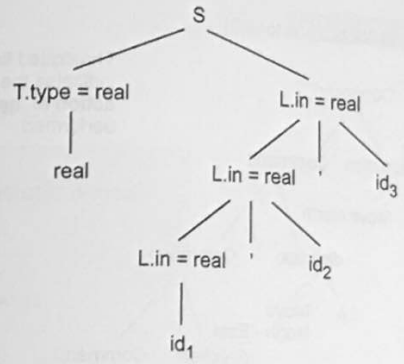


Fig. 4.2.11

Example 4.2.11 *A robot is to be moved to a unit step in a direction specified as a command given to it. The robot moves in the direction North, South, East, West on receiving N, S, E,W command respectively and in the direction North-East, North-West, South-East, South-West on receiving A, B, C, D commands respectively. The current position of the robot is initialized to (0,0) Cartesian coordinates on receiving command Start. Write production rules for producing sequence of commands and semantic rules for knowing position of a robot after receiving a sequence of commands. Draw annotated parse tree for following sequence :*

Start N N A A C C N **GTU : Winter-12, Marks 7**

Solution : The production rules and semantic actions are as given below -

Start → Command

Command→ direction command |ε

direction → N|S|E|W|A|B|C|D

N → Move North

S → Move South

E → Move East

W → Move West

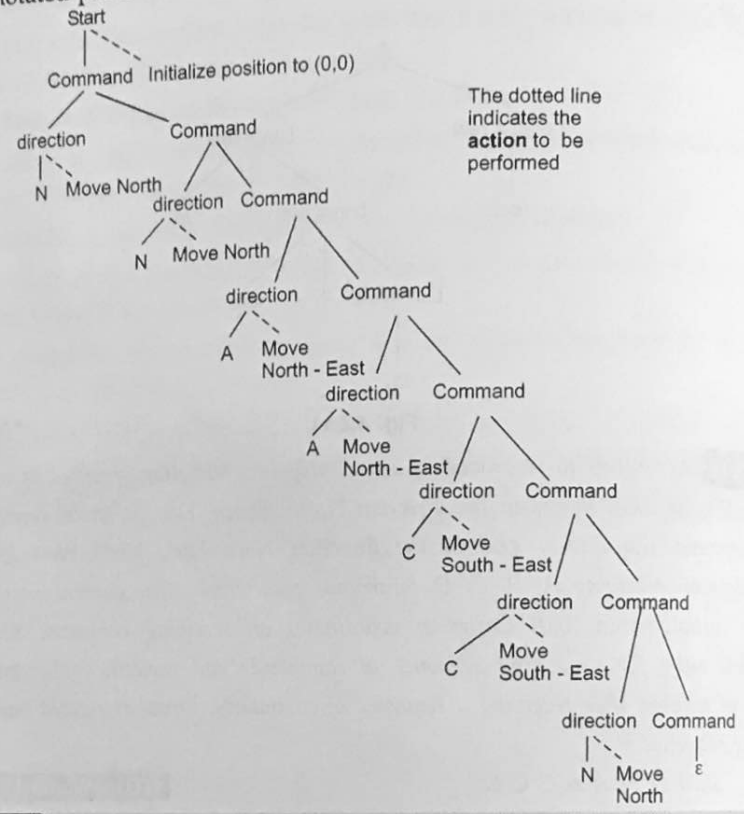
A → Move North-East

B → Move North-West

C → Move Sorth-East

D → Move South-West

The annotated parse tree can be as given below -



Example 4.2.12 *Write a syntax directed definition of a simple desk calculator and draw an annotated parse tree for 4*3 + 2*5 n.* **GTU : Summer-14, Marks 6**

Solution : Refer section 4.2(1)

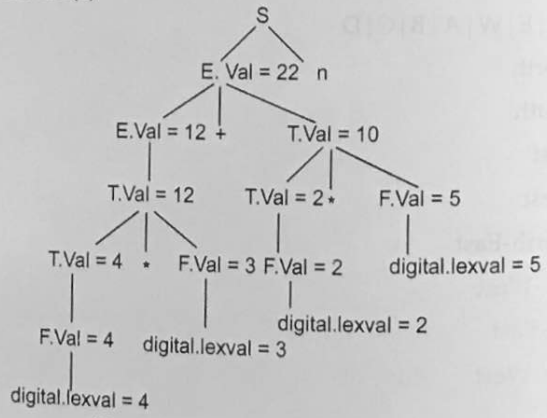


Fig. 4.2.12 Annotated parse tree

Example 4.2.13 *Write a context free grammar for the arithmetic expressions. Develop a syntax directed definition for the grammar. Draw an annotated parse tree for the input expression : (3*2+2) * 4* **GTU : Summer-15, Winter-20, Marks 7**

Solution :

Syntax directed definition

Production Rule	Semantic Actions
$S \rightarrow E$	Print (E• val)
$S \rightarrow E_1 + T$	E• val := E_1• Val + T• Val
$E \rightarrow T$	E• val := T• val
$T \rightarrow T_1 * F$	T• val := T_1• Val × F• Val
$T \rightarrow F$	T• val := F• Val
$F \rightarrow (E)$	F• Val := E• val
$F \rightarrow$ digit	F• Val := digit• lexval

The annotated parse tree will be shown in Fig. 4.2.13.

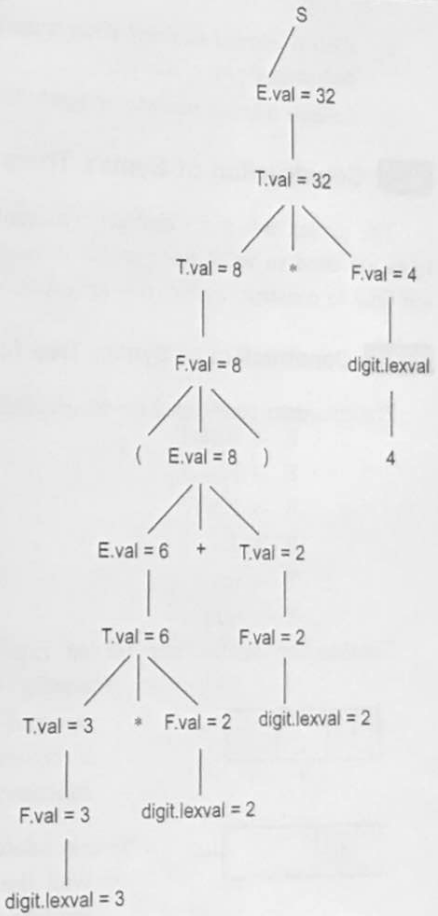


Fig. 4.2.13 Annotated parse tree

Review Questions

1. *What is attributed grammar ? Which phase of the compilation process does it facilitate ? Explain with example.* **GTU : Summer-12, Winter-19, Marks 3**
2. *Explain synthesized attribute* **GTU : Winter-12, Marks 3**
3. *What is inherited attribute ? Explain with suitable example* **GTU : Winter-13, Marks 6**
4. *Differentiate synthesized and inherited attributes.* **GTU : Summer-14, Winter-18, Marks 4**
5. *Discuss synthesized and inherited attributes using suitable grammar* **GTU : Summer-15, Marks 7**

6. *Write S-attributed syntax directed definition for simple desk calculator. Draw annotated parse tree for any valid input.* **GTU : Winter-16, Marks 7**
7. *Discuss synthesized attributes and inherited attributes in details.* **GTU : Summer-17, Marks 7,Winter-20, Marks 3**
8. *What is inherited attribute? Write syntax directed definition with inherited attributes for type declaration for list of identifiers* **GTU : Summer-18, Marks 7**
9. *Compare inherited attributes vs. synthesized attributes* **GTU : Summer-20, Marks 4**

4.3 Construction of Syntax Trees

GTU : Winter-17, Marks 7

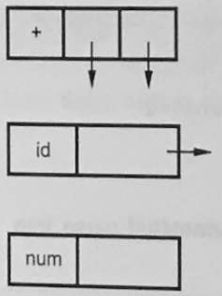
The syntax tree is an abstract representation of the language constructs. The syntax trees are used to write the translation routines using syntax-directed definitions. Let us see how to construct syntax tree for expression and how to obtain translation routines.

4.3.1 Construction of Syntax Tree for Expression

The grammar considered for the expression is

$E \rightarrow E_1 + T$
$E \rightarrow E_1 - T$
$E \rightarrow E_1 * T$
$E \rightarrow T$
$T \rightarrow id$
$T \rightarrow num$

Constructing syntax tree for an expression means translation of expression into postfix form. The nodes for each operator and operand is created. Each node can be implemented as a record with multiple fields. Following are the functions used in syntax tree for expression.



1. **mknode(op,left,right) :** This function creates a node with the field operator having operator as label, and the two pointers to left and right.
2. **mkleaf(id,entry) :** This function creates an identifier node with label id and a pointer to symbol table is given by 'entry'.
3. **mkleaf(num,val) :** This function creates node for number with label num and val is for value of that number.

Example 4.3.1 *Construct the syntax tree for the expression x*y–5+z.*

Solution :

Step 1 : Convert the expression from infix to postfix xy*5–z+.

Step 2 : Make use of the functions mknode(),mkleaf(id,ptr) and mkleaf(num,val).

Step 3 : The sequence of function calls is given.

Postfix expression xy*5 – z+

Symbol	Operation
x	p_1=mkleaf(id, ptr to entry x)
y	p_2= mkleaf(id, ptr to entry y)
*	p_3=mknode(*,p_1,p_2)
5	p_4= mkleaf(num,5)
–	p_5=mknode(–, p_3,p_4)
z	p_6= mkleaf(id, ptr to entry z)
+	p_7= mknode(+, p_5,p_6)

Consider the string x*y–5+z and let us draw the syntax tree.

The syntax-directed definition for the above grammar is as given below.

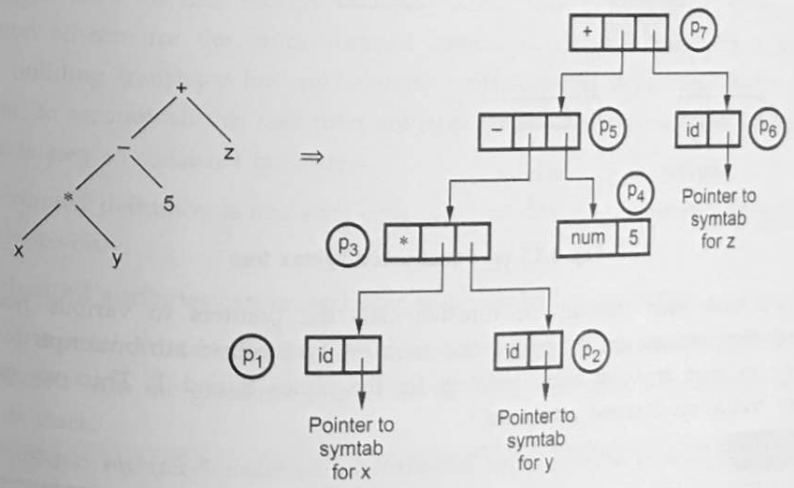


Fig. 4.3.1 Syntax tree

Production rule	Semantic operation
$E \rightarrow E_1 + T$	E.nptr:=mknode('+',E_1.nptr,T.nptr)
$E \rightarrow E_1 - T$	E.nptr:=mknode('−',E_1.nptr,T.nptr)
$E \rightarrow E_1 * T$	E.nptr:=mknode('*',E_1.nptr,T.nptr)
$E \rightarrow T$	E.nptr:=T.nptr
$T \rightarrow id$	E.nptr:=mkleaf(id,id.ptr_entry)
$T \rightarrow num$	T.nptr:=mkleaf(num,num.val)

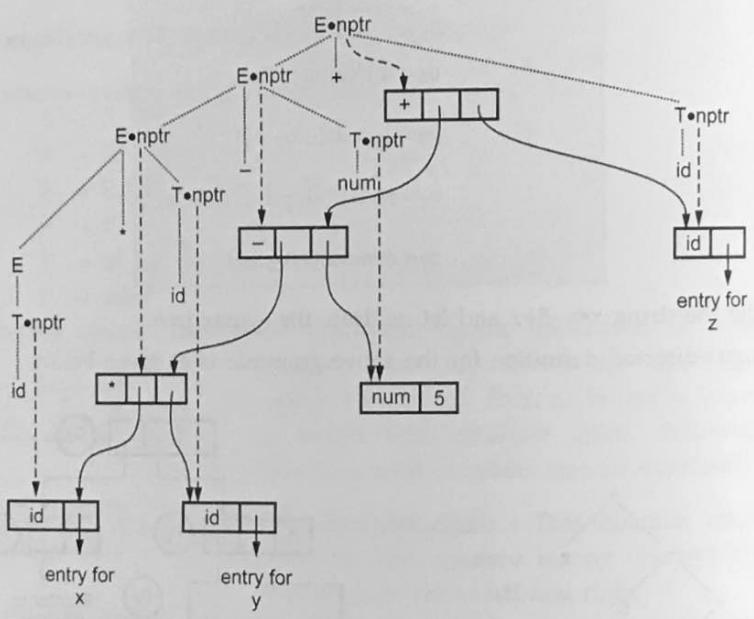


Fig. 4.3.1 (a) Constructed syntax tree

As we have seen that in the function calls the pointers to various nodes are generated. Such pointer are p_1, p_2, p_3 and so on. A synthesized attribute nptr for E and T is used to keep track of these pointers for the nodes E and T. Thus we get **nptr, ptr_entry, val** as synthesized attributes.

Example 4.3.2 *Define syntax tree. What is s-attributed definition ? Explain construction of syntax tree for the expression a − 4 + c using SDD.* **GTU : Winter-17, Marks 7**

Solution :

Syntax tree : Refer section 4.3.

S-attributed definition : Refer sections 4.2 and 4.3.

The syntax tree is :

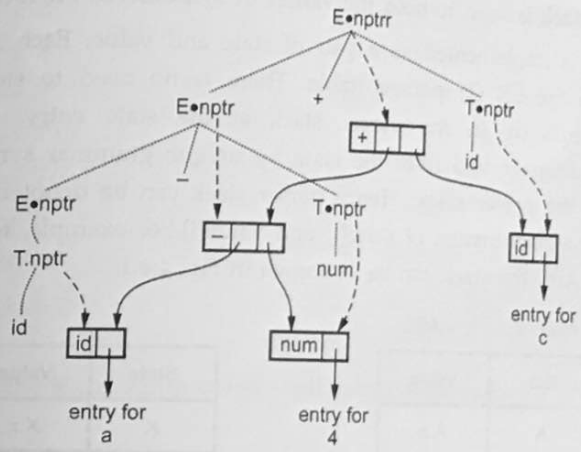


Fig. 4.3.2 Syntax tree

4.4 Bottom Up Evaluation of S-Attributed Definitions

We have already discussed how to use syntax-directed definitions to specify translations. Now in this section we will discuss how to implement syntax-directed translation scheme for the syntax-directed definitions. Hence a translator is built. The task of building translator for any arbitrary syntax-directed definition is very difficult. However, to accomplish this task there are large classes of syntax-directed definitions for which it is easy to construct translators.

- S-attributed definition is one such class of syntax-directed definition with synthesized attributes only.
- Synthesized attributes can be evaluated using the bottom-up parser.
- The purpose of stack is to keep track of values of the synthesized attributes associated with the grammar symbol on its stack. This stack is commonly known as **parser stack.**

4.4.1 Synthesized Attributes on the Parser Stack

1. A translator for S-attributed definition is implemented using **LR parser generator.**
2. **A bottom up** method is used to parse the input string.
3. **A parser stack** is used to hold the **values of synthesized attribute.**

The stack is implemented as a pair of state and value. Each state entry is the pointer to the LR (1) parsing table. There is no need to store the grammar symbol implicitly in the parser stack at the state entry. But for ease of understanding we will refer the state by unique grammar symbol that is been placed in the parser stack. Hence parser stack can be denoted as stack[i]. And stack[i] is a combination of state[i] and value[i].For example, for the production rule X → ABC the stack can be as shown in Fig. 4.4.1.

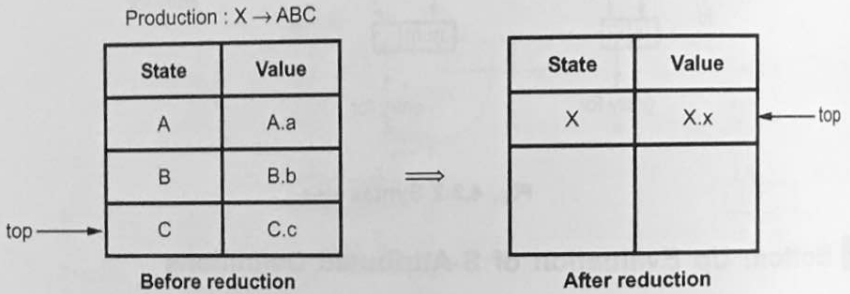


Fig. 4.4.1 Parser stack

The top symbol on the stack is pointed by pointer top.

Production rule	Semantic action
X → ABC	X.x = f(A.a,B.b,C.c)

Before reduction the states A, B and C can be inserted in the stack along with the values A.a, B.b and C.c.The top pointer of value[top] will point the value C.c similarly B.b is in value[top – 1] and A.a is in value[top – 2].After reduction the left hand side symbol of the production i.e. X will be placed in the stack along with the value X.x at the top. Hence after reduction value[top] = X.x.

4. After reduction top is decremented by 2 the state covering X is placed at the top of state[top] and value of synthesized attribute X.x is put in value[top].

5. If the symbol has no attribute then the corresponding entry in the value array will be kept undefined.

Example 4.4.1 *For the following given grammar construct the syntax-directed definition and generate the code fragment (translator) using S-attributed definition.*

$S \rightarrow EN$
$E \rightarrow E + T$
$E \rightarrow E - T$
$E \rightarrow T$
$T \rightarrow T*F$
$T \rightarrow T/F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow digit$
$N \rightarrow ;$

Also evaluate the input string 2*3+4; with parser stack using LR parsing method.

Solution :

- The syntax-directed definition for the given grammar can be written as follows.

Production rule	Semantic actions
S → EN	Print(E.val)
E → E_1 + T	E.val=E_1.val+T.val
E → E_1 – T	E.val:=E_1.val–T.val
E → T	E.val:=T.val
T → T_1*F	T.val:=T_1.val X F.val
T → T_1/F	T.val:=T_1.val/F.val
T → F	T.val:=F.val
F → (E)	F.val:=E.val
F → digit	F.val:=digit.lexval
N → ;	Can be ignored by lexical analyzer as ; is terminating symbol.

- The LR parser table can be generated.

- To evaluate the attributes the code fragment can be generated by using the parser stack. The appropriate reduction of each production and corresponding code fragment is as given below.

Production rule	Code fragment
$S \rightarrow EN$	Print(value[top])
$E \rightarrow E_1 + T$	value[top]:=value[top–2]+value[top]
$E \rightarrow E_1 - T$	value[top]:=value[top–2]–value[top]
$E \rightarrow T$	
$T \rightarrow T_1 * F$	value[top]:=value[top–2]*value[top]
$T \rightarrow T_1 / F$	value[top]:=value[top–2]/value[top]
$T \rightarrow F$	
$F \rightarrow (E)$	value[top]:=value[top–1]
$F \rightarrow$ digit	
$N \rightarrow ;$	

- The sequence of moves made by the parser for the input 2*3+4; are as given below.

Input string	State	Value	Production rule used
2*3+4;	–	–	
*3+4;	2	2	
*3+4;	F	2	F → digit
*3+4;	T	2	T → F
3+4;	T *	2 –	
+4;	T * 3	2 – 3	
+4;	T * F	2 – 3	F → digit
+4;	T	6	T → T*F

+4;	E	6	E→T
4;	E +	6 –	
;	E + 4	6 – 4	
;	E + 4	6 – 4	
;	E + F	6 – 4	F → digit
;	E + T	6 – 4	T → F
;	E	10	E → E + T
	E;	10 –	←
	EN	10	
	S	10	S → EN

On seeing the first input symbol 2, initially the symbol 2 is recognized as digit and the parser shifts F in the state stack. F corresponding to digit and the semantic action F.val = digit.lexval will be implemented and the value[top] becomes = 2. In the next move parser reduces by T→F. As no code fragment is associated with this production the value[top] and state[top] is left unchanged. Continuing in this fashion the evaluation of the input string is done and the parser halts successfully when it reaches to state[top] = S the start sate.

In this way the bottom-up evaluation of S-attributed definitions is done.

Example 4.4.2 *Write S - attributed grammar to convert the given grammar with infix operators to prefix operators.*

$L \rightarrow E$, $E \rightarrow E + T$, $E \rightarrow E - T$, $E \rightarrow T$,

$T \rightarrow T * F$, $T \rightarrow T/ F$, $T \rightarrow F$

$F \rightarrow F \uparrow P$, $F \rightarrow P$, $P \rightarrow (E)$, $P \rightarrow id$

Solution : The grammar that contains all the syntactic rules along with the semantic rules having synthesized attributes only, is called s - attributed grammar. Such a grammar for converting infix operators to prefix is given by using the 'val' as s - attribute.

Production Rule	Semantic Rule
$L \rightarrow E$	L · val : = E · val
$E \rightarrow E + T$	E · val : = '+' E · val T · val
$E \rightarrow E - T$	E · val : = ' - ' E · val T · val
$E \rightarrow T$	E · val : = T · val
$T \rightarrow T * F$	T · val : = ' * ' T · val F · val
$T \rightarrow T/ F$	T · val : = ' / ' T · val F · val
$T \rightarrow F$	T · val : = F · val
$F \rightarrow F \uparrow P$	F · val : = ' ↑ ' F · val P · val
$F \rightarrow P$	F · val : = P · val
$P \rightarrow (E)$	P · val : = E · val
$P \rightarrow id$	P · val : = id · lexval

4.5 L-Attributed Definition

The syntax-directed definition can be defined as the L-attributed for the production rule $A \rightarrow X_1X_2...X_n$ where the inherited attribute X_k is such that $1 \leq k \leq n$. The production $A \rightarrow X_1X_2...X_n$ is such that

1) It depends upon the attributes of the symbol $X_1, X_2, ...X_{j-1}$ to the left of X_j.

2) It also depends upon the inherited attribute A.

Note that because of these two conditions every S-attributed definition is also L-attributed definition.

Let us discuss one example of L-attributed definitions.

Example 4.5.1 *Check whether the given SDD (Syntax-Directed Definition) is L-attributed or not.*

$A \rightarrow PQ$ $\quad P.in:=p(A.in)$

$\quad Q.in:=q(P.sy)$

$\quad A.sy:=f(Q.sy)$

$A \rightarrow XY$ $\quad Y.in:=y(A.in)$

$\quad X.in:=x(Y.sy)$

$\quad A.sy:=f(X.sy)$

Solution : The attributes **in** and **sy** represent the inherited and synthesized attributes respectively. The given syntax-directed definition is not L-attributed definition.

Production rule	Semantic action	Class of attribute
$A \rightarrow PQ$	P.in := p(A.in)	L-attribute
	Q.in:= q(P.sy)	L-attribute
	A.sy:= f(Q.sy)	L-attribute
$A \rightarrow XY$	Y.in:= y(A.in)	L-attribute
	X.in:= x(Y.sy)	Not L-attribute
	A.sy:= f(X.sy)	L-attribute

Because here value of left symbol (X) is dependent upon value of right symbol (i.e. Y)

This is because of the definition X.in:=x(Y.sy). This semantic action suggests that value of **X.in** depends upon value of **Y.sy**. That means value of left symbol is dependant on the value of the right symbol. This **violates** the 1st rule of the L-attributed definition. *[Logically also while parsing the scan is done from left to right and not from right to left!!]* Thus X.in :x(Y.sy) is **not L-attribute**.

Example 4.5.2 *Explain why every S-attributed definition is L-attributed.*

Solution : Following are the restrictions that can be applied on inherited attributes of X_k where $1 \leq k \leq n$ and $A \rightarrow X_1 X_2 \ldots X_n$.

i) Suppose $A \rightarrow X_1 X_2 \ldots X_{j-1} \cdot X_j \ldots X_n$ is a production then attribute of X_j should depend upon attributes of $X_1 X_2 \ldots X_{j-1}$.

ii) The attributes of X_k can be dependant upon inherited attributes of A.

Because of these restrictions on inherited attributes every S-attributed definition is L-attributed.

Example 4.5.3 *Differentiate between S-attributed grammar and L-attributed grammar.*

Solution :

No.	S-attributed grammar	L-attributed grammar
1.	This is a class of attributed grammar having no inherited attributes but only synthesized attributes.	This is the class of grammar in which the attributes to be evaluated in one left-to-right traversal of the abstract syntax tree. This is a class of attributed grammar in which inherited attributes can be evaluated.

2.	The S-attributed grammar can be incorporated in both top down parsing and bottom up parsing.	The L-attributed grammar is incorporated in using top down parsing.
3.	The YACC family can be broadly considered as S-attributed grammar.	Many programming languages are L-attributed. Special types of compilers, the narrow compilers, are based on some form of L-attributed grammar.
4.	S-attributed grammar can be L-attributed grammar.	L-attributed grammar can not be S-attributed.
5.	Example - Refer example 4.4.2.	Example - Refer example 4.5.1.

4.6 Syntax Directed Definitions and Translation Schemes

GTU : Summer-12, Winter-14,15,18,20, Marks 7

As we have seen that the attributes are specified for each grammar symbol of the given production. These attributes are used to evaluate the expression along the process of parsing. And exactly at the same time semantic rules are applied.

- During the process of parsing the evaluation of attribute takes place by consulting the semantic action enclosed in { } at the right of the grammar symbol. This process of execution of code fragment semantic actions from the syntax-directed definition is called **syntax-directed translation**. Thus the execution of syntax-directed definition can be done by **syntax-directed translation scheme**.
- A translation scheme generates the output by executing the semantic actions in an **ordered manner**.
- This processing is using **depth first traversal**.

Example 4.6.1 *Give the translation scheme that coverts infix to postfix form for the following grammar. Also generate the annotated parse tree for input string 2+6+1.*

$E \rightarrow E+T$ {print('+')}
$E \rightarrow T$
$T \rightarrow 0$ {print('0')}
$T \rightarrow 1$ {print('1')}
$T \rightarrow 2$ {print('2')}
$T \rightarrow 3$ {print('3')}
$T \rightarrow 4$ {print('4')}
$T \rightarrow 5$ {print('5')}
$T \rightarrow 6$ {print('6')}
$T \rightarrow 7$ {print('7')}
$T \rightarrow 8$ {print('8')}
$T \rightarrow 9$ {print('9')}

Solution : Let us convert this grammar into non left recursive form

$E \rightarrow TP$

$T \rightarrow 0|1|2|3|4|5|6|7|8|9$

$P \rightarrow +TP \mid \varepsilon$

The translation scheme for this grammar is as given below.

Productions	Semantic actions
$E \rightarrow T\ P$	
$T \rightarrow 0$	{print('0')}
$T \rightarrow 1$	{print('1')}
$T \rightarrow 2$	{print('2')}
$T \rightarrow 3$	{print('3')}
$T \rightarrow 4$	{print('4')}
$T \rightarrow 5$	{print('5')}
$T \rightarrow 6$	{print('6')}
$T \rightarrow 7$	{print('7')}
$T \rightarrow 8$	{print('8')}
$T \rightarrow 9$	{print('9')}
$P \rightarrow +TP \mid \varepsilon$	{print('+')} P | ε

The annotated parse tree for the string 2+6+1 is as given in Fig. 4.6.1.

The translation scheme proceeds using depth first traversal. The annotated parse tree shows the semantic actions associated appropriate node. Thus this tree shows the translation scheme for mapping of **infix form to postfix form.** The associated semantic actions can be shown using dotted lines.

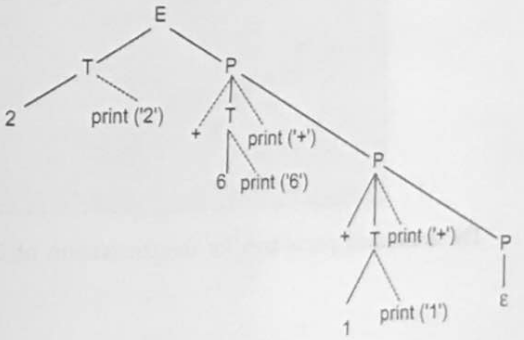


Fig. 4.6.1 Annotated parse tree

Example 4.6.2 *Write syntax directed defination for translating following grammer for postfix notation. Also draw annotated parse tree for 9–5+2.*

expr–> expr+term
expr–> expr–term
term–> 0|1|...|9

GTU : Summer-12, Winter-20, Marks 7

Solution :

SDD for translating to postfix : Refer example 4.6.1.

Annotated parse tree,

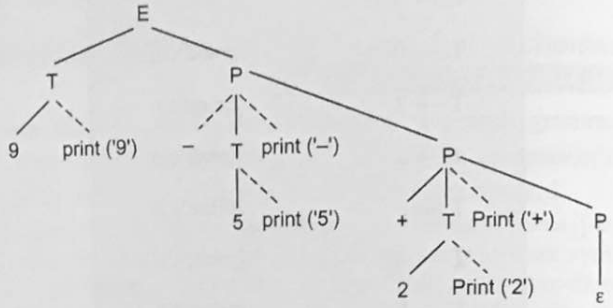


Fig. 4.6.2 Annotated parse tree for 9 – 5 + 2

Example 4.6.3 *Construct a syntax-directed translation scheme that translates arithmetic expressions from infix into postfix notation. Show the application of your scheme to the string "3*4+5*2".*

GTU : Winter-14, Marks 7

Solution : The translation scheme is as given below -

Production	Semantic action
E → E + T	{Print ('+')}
E → T	{null}
T → T * F	{Print ('*') }
T → F	{null}
F → num	{Print (num · val) }

The annotated parse tree for the conversion of '3 * 4 + 5 * 2' to postfix notation is

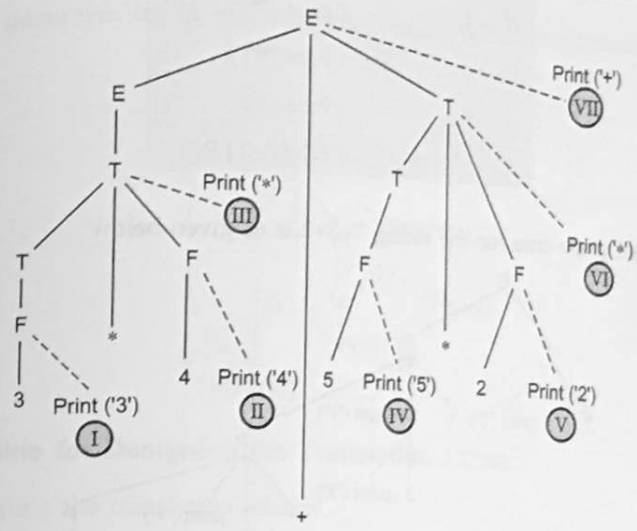


Fig. 4.6.3

Example 4.6.4 *Give the translation scheme that converts infix to postfix expression for the following grammar and also generate the annotated parse tree for input string 7+3+2*

$E \rightarrow E+T, E \rightarrow T, \ T \rightarrow 0|1|2|3|4|5|6|7|8|9$

GTU : Winter-15, Marks 7

Solution : Let us convert this grammar into a form in which the left recursion is removed

$E \rightarrow T\ P$

$T \rightarrow 0|1|2|3|4|5|6|7|8|9$

$P \rightarrow +\ T\ P|\varepsilon$

The translation scheme will be

Production	Semantic Actions
E → TP	
T → 0	{ Print ('0') }
T → 1	{ Print ('1') }
T → 2	{ Print ('2') }
T → 3	{ Print ('3') }
T → 4	{ Print ('4') }
T → 5	{ Print ('5') }
T → 6	{ Print ('6') }

T → 7	{ Print ('7') }
T → 8	{ Print ('8') }
T → 9	{ Print ('9') }
P → +TP | ε	{ Print ('+') } P | ε

The annotated prase tree for the string 7+3+2 is as given below

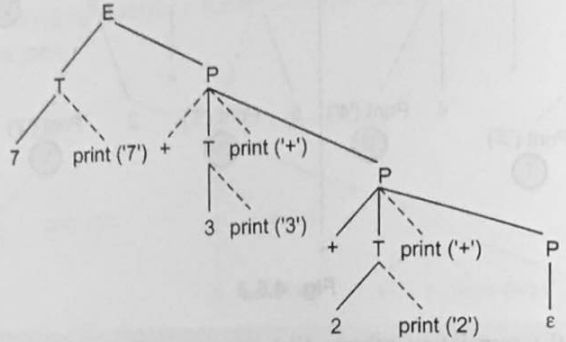


Fig. 4.6.4

Example 4.6.5 *Give the translation scheme that converts infix to postfix expression for the following grammar and also generate the annotated parse tree for input string "id+id*id"*

E → E+T | T

T → T*F | F

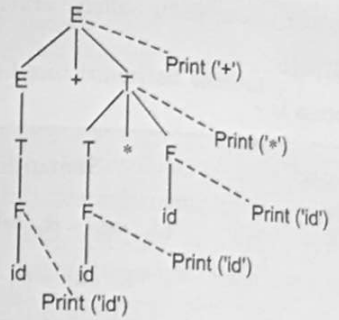
F → id

GTU : Winter-18, Marks 4

Solution : Translation scheme for given grammar is

Production	Semantic action
E → E + T	{ Print (' + ') }
E → T	{ null }
T → T * F	{ Print (' * ') }
T → F	{ null }
F → id	{ Print (' id ') }

Annotated parse tree for id + id * id which converts infix expression to psotfix is as follows -



4.6.1 Guideline for Designing the Translation Scheme

While designing the translation scheme -

- We have to follow one restriction and that is for every semantic action if it refers to some attribute then that attribute value must be computed before that attribute gets referred.
- Computation of synthesized attribute is very easy. The synthesized attribute can be computed as follows :

Production Rule	Semantic Action
$E \rightarrow E_1 * T$	{ E.val:=E_1.val × T.val }

Here **val** is synthesized attribute.

- For the computation of both the inherited and synthesized attributes
  - i) In any semantic action the computation of inherited attribute for the symbol on the right side of the production must be done before that symbol. That means if there is a production A→XY then computation of **X.in** must be done already.
  - ii) In the semantic action there should not be any reference to the synthesized attribute of the symbol that is right. That means for the production $A \rightarrow B_1B_2$ the semantic action $\{B_1.s:=B_2.s\}$ is invalid as computation of attribute for B_1 is based on computation of the attribute of B_2.
  - iii) For a computation of synthesized attribute for a non-terminal on the left, first do all the computations of the attributes that refer this non terminal. Typically the semantic action containing the computation of all such attributes is placed at the end of right side of the production.

Such a design of translation scheme helps for the implementation of the syntax-directed definition.

Example 4.6.6 *Explain the translator process using suitable example.*

$R \rightarrow R' \mid R \mid RR \mid R* \mid (R) \mid a \mid b$

Solution : The translator process is -

Production rules	Semantic rules
$R \rightarrow R' \mid 'R$	{ R · val : = R_1 · val ' | ' R_2 · val }
$R \rightarrow RR$	{ R · val : = R_1 · val R_2 · val }
$R \rightarrow R*$	{ R · val : = R· val ' * ' }
$R \rightarrow (R)$	{ R · val : = R_1 · val }
$R \rightarrow a$	{ R · val : = a }
$R \rightarrow b$	{ R · val : = b }

The annotated parse tree for a|bc will be

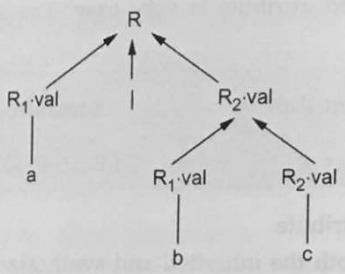


Fig. 4.6.5 Annoted parse tree

4.7 Short Questions and Answers

Q.1 What is a translation scheme ?

Ans. : During the process of parsing the evaluation of attributes take place by the semantic actions. The semantic actions are defined along with the context free grammar. The syntax directed translation scheme defines the code which is to be executed based on these semantic actions.

Q.2 Mention the two rules for type checking.

Ans. : i) Each identifier must be declared before the use

ii) The use of identdifier must be within the scope.

iii)An identifier must not have multiple declarations at a time within the same scope.

Q.3 Mention the role of semantic analysis.

Ans. : The semantic analysis is carried out in order to get the precise meaning of programming constructs. The data type of the identifier is obtained by type checking system of semantic analysis phase.

Q.4 What are synthesized and inherited attributes ?

Ans. :

1. **Synthesized attribute** : The attribute 'a' is called synthesized attribute of X and $b_1, b_2, \ldots\ldots b_k$ are attributes belonging to the production symbols.
   The value of synthesized attribute at a node is computed from the values of attributes at the children of that node in the parse tree.
2. **Inherited attribute** : The attribute 'a' is called inherited attribute of one of the grammar symbol on the right side of the production (i.e. α) and $b_1, b_2, \ldots\ldots b_k$ are belonging to either X or α .

Q.5 What is dependency graph ?

Ans. : The directed graph that represents the interdependencies between synthesized and inherited attributes at nodes in the parse tree is called dependency graph.

For the rule X→YZ the semantic action is given by X.x := f(Y.y, Z.z) then

Synthesized attribute is X.x and X.x depends upon attributes Y.y and Z.z.

Q.6 While creating syntax tree for expression which functions are used ?

Ans. : The **mknode** and **mkleaf** are the functions that are used while creating syntax tree for expression. The **mknode** function is used for creating an intermediate node. The **mkleaf** function is used for creating the leaf nodes.

Q.7 What are S-attributed definitions ?

Ans. : The grammar that contains all the syntactic rules along with the semantic rules having synthesized attributes only, is called s - attributed definitions

Q.8 What are L-attributed definitions ?

Ans. : The syntax-directed definition can be defined as the L-attributed for the production rule $A \rightarrow X_1 X_2 \ldots X_n$ where the inherited attribute X_k is such that $1 \leq k \leq n$. The production $A \rightarrow X_1 X_2 \ldots X_n$ is such that

1) It depends upon the attributes of the symbol $X_1, X_2, \ldots X_{j-1}$ to the left of X_j.

2) It also depends upon the inherited attribute A.

Q.9 What is syntax directed translation scheme ?

Ans. : A translation scheme generates the output by executing the semantic actions in an **ordered manner**.Thus the execution of syntax-directed definition can be done by **syntax-directed translation scheme**.

Q.10 What is the advantages of Syntax Directed Translation(SDT) ?

Ans. : Syntax directed translation is a scheme that indicate the order in which semantic rules are to be evaluated. The main advantage of SDT is that it helps in deciding evaluation order. The evaluation of semantic actions associated with SDT may generate code, save information in symbol table, or may issue error messages.

Q.11 Explain why every S-attributed definition is L-attributed.

Ans. : Following are the restrictions that can be applied on inherited attributes of X_k where $1 \le k \le n$ and $A \rightarrow X_1 X_2 \cdots X_n$.

i) Suppose $A \rightarrow X_1 X_2 \cdots X_{j-1} \cdot X_j \cdots X_n$ is a production then attribute of X_j should depend upon attributes of $X_1 X_2 \cdots X_{j-1}$.

ii) The attributes of X_k can be dependant upon inherited attributes of A.

Because of these restrictions on inherited attributes every S-attributed definition is L-attributed.

4.8 Multiple Choice Questions

Q.1 The augmented context free grammar is generated during _____.

a type checking
b syntax analysis
c syntax directed translation scheme
d none of the above

Q.2 Evaluation of semantic rules _____.

a generates the code
b save information in symbol table
c issue error messages
d all above is true

Q.3 Synthesized attribute is computed at _____.

a child node of the parse tree
b parent node of the parse tree
c root node of the parse tree
d sibling node of the parse tree

Q.4 Inherited attribute is computed at _____.

a child node of the parse tree
b parent node of the parse tree
c root node of the parse tree
d sibling node of the parse tree

Q.5 The graph which represents the interdependency between synthesized and inherited attributes is _____.

a flow graph
b directed graph
c dependency graph
d Direct Acyclic Graph(DAG)

Q.6 For deciding the evaluation order from the dependency graph _____.

a topological sort is used
b quick sort is used
c bubble sort is used
d heap sort is used

Q.7 Functions in semantic rule will often be written as _____.

a program code
b expression
c production rule
d none of the above

Q.8 A parse tree for S-attributed grammar can be annotated by evaluating it in _____.

a top to bottom up manner
b sibling to node manner
c bottom up manner
d none of the above

Q.9 Which of the following data structure is used evaluating the synthesized attributes ?

a array
b stack
c queue
d linked list

Q.10 Translator for the S-attributed definition can often be implemented with the help of _____.

a LEX
b semantic analyser
c LR parser generator
d both B and C

Q.11 The attributes of L-attributes definitions is evaluated in _____.

a breadth First Manner
b depth First Manner

Q.12 In semantic analysis following task is done _____.

a separation of tokens from the source program
b building of syntax tree
c type checking
d all of the above

Q.13 Coercion is _____.

a implicit type conversion

b explicit type conversion

c implicit type conversion depending upon the expression

d none of the above

Q.14 Checking the scope of the variable is done in _____.

a syntax analysis

b semantic Analysis

c lexical analysis

d during the execution of the program

Q.15 Semantic errors can be detected at _____.

a compile time

b run time

c both at compile time and run time

d none of the above

Answer Keys for Multiple Choice Questions

Q.1	c	Q.2	d	Q.3	a
Q.4	b, d	Q.5	c	Q.6	a
Q.7	b	Q.8	c	Q.9	b
Q.10	c	Q.11	b	Q.12	c
Q.13	a	Q.14	b	Q.15	c

❑❑❑

5 Error Recovery

Syllabus

Error Detection & Recovery, Ad-Hoc and Systematic Methods.

Contents

5.1 Error Detection and Recovery

In this phase of compilation all possible errors made by the programmer are detected and they are reported to the user in the form of messages. This process of locating errors and reporting to user is called **error handling process**.

Functions of error handler

1. The error handler should identify all possible errors from the source code.
2. It should report these errors with appropriate messages to the user / programmer. The error messages should be informative so that the programmer can correct those.
3. The error handler should repair the errors at that instance in order to continue processing of the program.
4. The error handler should recover from errors in order to detect as many errors as possible.

5.2 Ad-Hoc and Systematic Methods

GTU : Summer-12,14,15,Winter-11,13,14,17,20, Marks 7

Parser employs various strategies to recover from syntactic errors. These strategies are i) Panic mode ii) Phrase level iii) Error productions iv) Global correction.

No one strategy is universally acceptable but can be applied to a broad domain. These strategies are given in detail as below -

i) Panic mode

- This strategy is used by most parsing methods.
- This is simple to implement.
- In this method on discovering error, the parser discards input symbol one at a time. This process is continued until one of a designated set of synchronizing tokens is found. Synchronizing tokens are delimiters such as semicolon or end . These tokens indicate an end of input statement.
- Thus in panic mode recovery a considerable amount of input is skipped without checking it for additional errors.
- This method guarantees not to go in infinite loop.
- If there are less number of errors in the same statement then this strategy is a best choice.

ii) Phrase level recovery

- In this method, on discovering error parser perform local correction on remaining input.

- It can replace a prefix of remaining input by some string. This actually helps the parser to continue its job.
- The local correction can be replacing comma by semicolon, deletion of extra semicolons or inserting missing semicolon; The type of local correction is decided by compiler designer.
- While doing the replacement a care should be taken for not going in an infinite loop.
- This method is used in many error-repairing compilers.
- The drawback of this method is it finds difficult to handle the situations where the actual error has occured before the point of detection.

iii) Error production

- If we have a knowledge of common errors that can be encountered then we can incorporate these errors by augmenting the grammar of the corresponding language with error productions that generate the erroneous constructs.
- If error production is used then during parsing, we can generate appropriate error message and parsing can be continued.
- This method is extremely difficult to maintain. Because if we change the grammar then it becomes necessary to change the corresponding error productions.

iv) Global production

- We often want such a compiler that makes very few changes in processing an incorrect input string.
- We expect less number of insertions, deletions, and changes of tokens to recover from erroneous input.
- Such methods increase time and space requirements at parsing time.
- Global production is thus simply a theoretical concept.

Review Questions

1. *How can panic mode and phrase level recovery be implemented in LR parsers ? Consider the expression grammar E->E+E | E*E | (E) | id. Prepare the SLR parsing table with error detection and recovery routines.*
   GTU : Winter-11, Marks 7
   [Ans. : Refer sections 5.3 and 3.11]
2. *Explain : Error recovery strategies in compiler in brief.*
   GTU : Summer-12,14, Winter-14, Marks 7
3. *Explain how panic mode recovery can be implemented.*
   GTU : Winter-13, Marks 7; Winter-17,20, Marks 4

4. *List the errors generated by the syntax analysis phase. Discuss error handling methods in the syntax analysis phase.* **GTU : Summer-15, Marks 7**

5.3 Short Questions and Answers

Q.1 Explain the term error handling.

Ans. : Error handling is process of locating errors and reporting them to users.

Q.2 What are the two types of errors that occur commonly ?

Ans. : Globally there are two types of errors : Compile time and run time errors.

Q.3 Enlist lexical phase errors.

Ans. : Typical lexical phase errors are -Spelling errors. Hence get incorrect tokens. For example -

i) Exceeding length of identifier or numeric constants.

ii) Appearance of illegal characters.

iii) Normally due to typing mistakes the wrong spellings may appear in the program.

Q.4 Give examples of syntactical errors.

Ans. : Syntax errors appear during syntax analysis phase of compiler.

Typical phase errors are :

i) Errors in structure ii) Missing operators iii) Unbalanced parenthesis.

Q.5 What are semantic errors ?

Ans. : Semantic errors are those errors which get detected during semantic analysis phase.Typical errors in this phase are :

i) Incompatible types of operands.

ii) Undeclared variables.

iii) Not matching of actual arguments with formal arguments.

Q.6 Enlist various strategies to recover from syntactic errors.

Ans. : Parser employs various strategies to recover from syntactic errors. These strategies are

i) Panic mode ii) Phrase level

iii) Error productions iv) Global correction.

Q.7 What is panic mode strategy to recover from syntactic error ?

Ans. : In this method on discovering error, the parser discards input symbol one at a time. This process is continued until one of a designated set of synchronizing tokens is found.

❑❑❑

6 Intermediate Code Generation

Syllabus

Different Intermediate Forms, Syntax Directed Translation Mechanisms And Attributed Mechanisms And Attributed Definition.

Contents

6.1 Introduction to Intermediate Code

The task of compiler is to convert the source program into machine program. This activity can be done directly, but it is not always possible to generate such a machine code directly in one pass. Then, typically compilers generate an easy to represent form of source language which is called **intermediate language**. The generation of an intermediate language leads to efficient code generation.

6.1.1 Benefits of Intermediate Code Generation

There are certain benefits of generating machine independent intermediate code :

1. A compiler for different machines can be created by attaching different backend to the existing front ends of each machine.
2. A compiler for different source languages (on the same machine) can be created by proving different front ends for corresponding source languages to existing back end.
3. A machine independent code optimizer can be applied to intermediate code in order to optimize the code generation.

The role of intermediate code generator in compiler is depicted by Fig. 6.1.1.

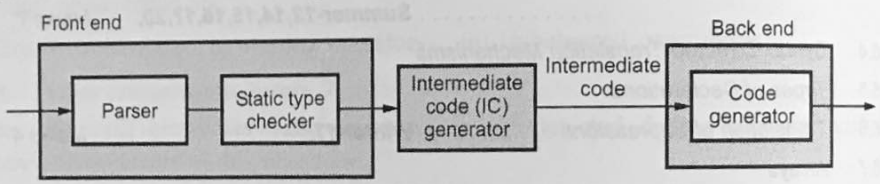


Fig. 6.1.1 Intermediate code generator in compiler

6.1.2 Properties of Intermediate Languages

Following are the properties of intermediate languages.

1. The intermediate language is an **easy form** of source language which can be generated efficiently by the compiler.
2. The generation of intermediate language should lead to efficient code generation.
3. The intermediate language should act as **effective mediator** between front and back end.
4. The intermediate language should be **flexible** enough so that optimized code can be generated.

Review Questions

1. *List out the benefits of using machine independent intermediate forms.*
2. *Write the properties of intermediate languages.*

6.2 Variants of Syntax Trees

GTU : Summer-12, Winter-11,13,16,17,20, Marks 7

There are mainly three types of intermediate code representations :

1. Abstract syntax tree
2. Polish notation
3. Three address code.

These three representations are used to represent the intermediate languages. Let us discuss each with the help of examples :

1. Abstract syntax tree

The natural hierarchical structure is represented by syntax trees. Directed Acyclic Graph or DAG is very much similar to syntax trees but they are in more compact form.

The code being generated as intermediate should be such that the remaining processing of the subsequent phases should be easy.

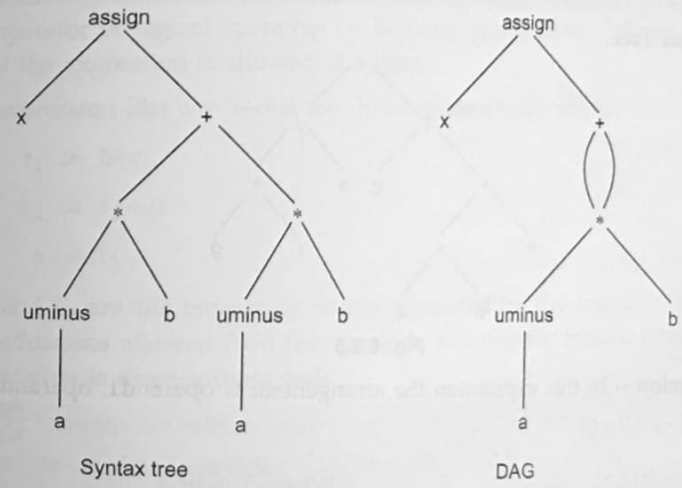


Fig. 6.2.1 Syntax tree and DAG

Consider the input string x:= –a*b + –a*b for the syntax tree and DAG representation.

2. Polish notation

Basically, the linearization of syntax trees is polish notation. In this representation, the operator can be easily associated with the corresponding operands. This is the most natural way of representation in expression evaluation.

The polish notation is also called as **Prefix notation** in which the operator occurs first and then operands are arranged.

For example : $(a + b)*(c - d)$ can be written as

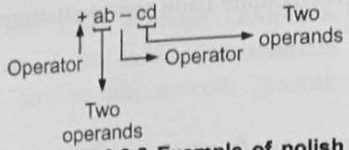


Fig. 6.2.2 Example of polish notation

There is a **reverse polish notation** which is used using **postfix** or Posix representation.

Consider that the input expression is -

x:= –a*b + –a*b, then the required posix form is -

x a – b * a – b * + :=

Example 6.2.1 *Construct syntax tree and postfix notation for the following expression.*

$(a+(b*c))\uparrow d-e/(f+g)$

Solution : Syntax Tree

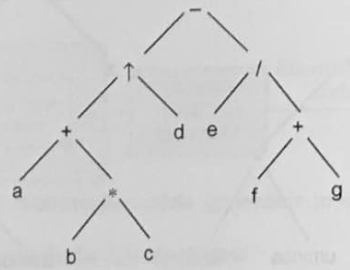


Fig. 6.2.3

Postfix Expression – In this expression the arrangement is operand1, operand2 operator.

$(a+(b*c))\uparrow d-e/(f+g)$

$(a+T_1)\uparrow d-e/(f+g)$ Where $T_1 = bc*$

$T_2\uparrow d-e/(f+g)$ Where $T_2 = aT_1+$

$T_3-e/(f+g)$ Where $T_3 = T_2d\uparrow$

T_3-e/T_4 Where $T_4 = fg+$

T_3-T_5 Where $T_5 = eT_4/$

T_6 Where $T_6 = T_3T_5-$

Now backward substituting the values of temporary variables -

T_6

T_3T_5-

$T_3\ e\ T_4\ /-$

$T_3\ e\ fg+/-$

$T_2d\uparrow e\ fg+/-$

$aT_1+d\uparrow e\ fg+/-$

$abc*+d\uparrow e\ fg+/-$ is the postfix expression

3. Three address code

In three address code form at the most three addresses are used to represent any statement. The general form of three address code representation is –

$$a := b\ op\ c$$

where a,b or c are the operands that can be names, constants, compiler generated temporaries and op represents the operator. The operators can be fixed or floating point arithmetic operator or logical operators on Boolean valued data. Only single operation at right side of the expression is allowed at a time.

For the expression like a = b+c+d the three address code will be

$$t_1 := b+c$$

$$t_2 := t_1 + d$$

$$a = t_2$$

Here t_1 and t_2 are the temporary names generated by the compiler. There are at the most three addresses allowed (two for operands and one for result). Hence, the name of this representation is three-address code.

Example 6.2.2 *Translate the arithmetic expression a * – (b + c) into*

1. Syntax tree 2. Postfix notation 3. Three address code

GTU : Winter-11, 17, Marks 7

Solution : 1) Syntax tree for a * – (b + c) :

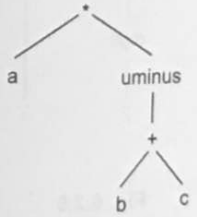


Fig. 6.2.4

2) Pastfix notation :

a * – (b + c)

a * – T_1 T_1 = bc +

T_2 T_2 = a T_1 – *

Substituting in backward direction,

a T_1 – *

a bc + – * is postfix notation

3) Three address code :

T_1 : = b + c

T_2 : = uminus T_1

T_3 = a * T_2

Example 6.2.3 *What is the difference between parse tree and syntax tree ? Draw the parse tree for following expression : a = a + a * b + a * b * c – a / b + a * b and write three address code for it.*

GTU : May-12, Marks 4

Solution : Both parse trees and syntax trees are hierarchical data structures. But the syntax trees are abstract i.e. in syntax tree each node simply shows operator and children node represents operands. On the other hand, the parse tree represents the translation that took place for the expression.

For example - Consider an expression 2 + 3 * 4. Following are the two trees syntax tree and parse tree that represents the expression.

The syntax tree is also called as **abstract syntax tree** whereas the parse tree is called **concrete syntax tree**.

Consider the production rules

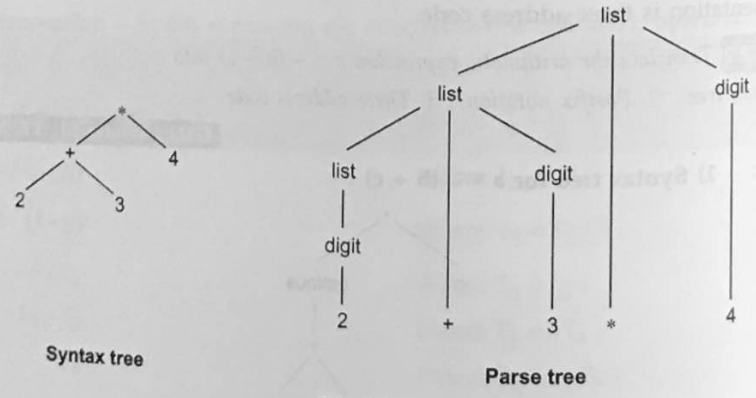


Fig. 6.2.5

The parse tree will then be as shown in Fig. 6.2.6.

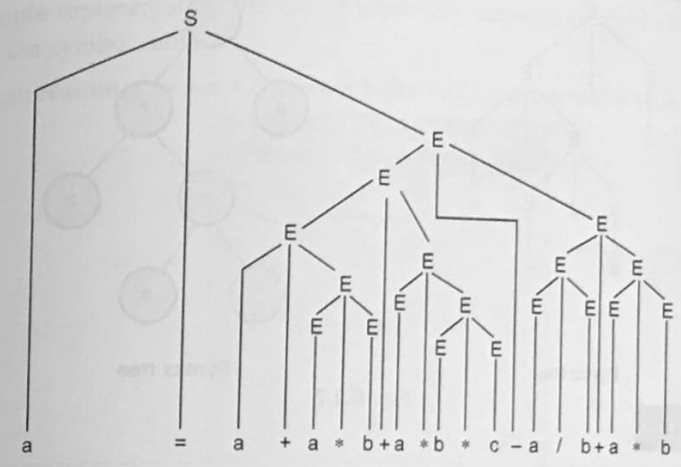


Fig. 6.2.6

S → a = E

E → E + E

E → E * E

E → E – E

E → E / E

E → a | b | c

Example 6.2.4 *What is the difference between parse tree and syntax tree ? Write appropriate grammar and draw parse as well as syntax tree for a*(a–a^a).*

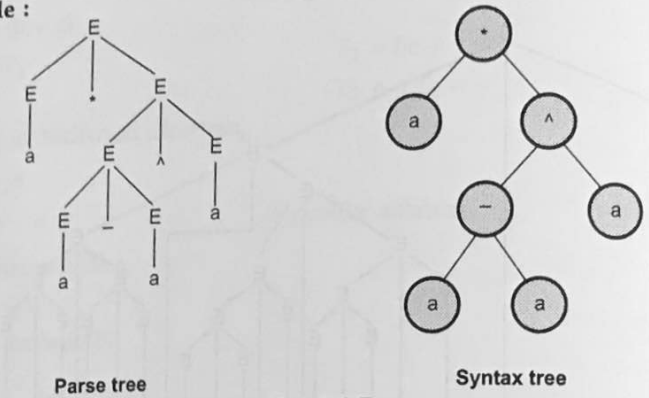
GTU : Winter-13, Marks 7

Solution : Parse tree represents concrete syntax of the grammar whereas syntax tree represents the abstract syntax of the grammar.

Parse trees have interior nodes as non-terminals and leaf nodes as terminals.

In syntax tree interior nodes are operators and leaves are operands.

For example :



Parse tree Syntax tree

Fig. 6.2.7

Review Question

1. *What is intermediate code ? What is its importance ? Discuss various representations of three address code.* **GTU : Winter-16,20, Marks 7**

6.3 Three Address Code

GTU : Winter-11,14,15,18,19,20, Summer-12,14,15,16,17,20, Marks 7

Three address code is an abstract form of intermediate code that can be implemented as a record with the address fields. There are three representations used for three address code such as quadruples, triples and indirect triples.

Quadruple representation

The quadruple is a structure with at the most four fields such as op, arg1, arg2, result. The op field is used to represent the internal code for operator, the arg1 and arg2 represent the two operands used and result field is used to store the result of an expression.

For example : Consider the input statement x := – a * b + – a * b

The three address code is

t_1 := uminus a
t_2 := t_1 * b
t_3 := – a
t_4 := t_3 * b
t_5 := t_2 + t_4
x := t_5

Quadruple

	Op	Arg1	Arg2	result
(0)	uminus	a		t_1
(1)	*	t_1	b	t_2
(2)	uminus	a		t_3
(3)	*	t_3	b	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		x

Triples

In the triple representation the use of temporary variables is avoided by referring the pointers in the symbol table.

For the expression x := – a * b + – a * b the triple representation is as given below

Number	Op	Arg1	Arg2
(0)	uminus	a	
(1)	*	(0)	b
(2)	uminus	a	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	:=	x	(4)

Indirect triples

In the indirect triple representation the listing of triples is been done. And listing pointers are used instead of using statements.

Number	Op	Arg1	Arg2
(0)	uminus	a	
(1)	*	(11)	b
(2)	uminus	a	
(3)	*	(13)	b
(4)	+	(12)	(14)
(5)	:=	x	(15)

	Statement
(0)	(11)
(1)	(12)
(2)	(13)
(3)	(14)
(4)	(15)
(5)	(16)

6.3.1 Merits and Demerits of Quadruple, Triple and Indirect Triples

- In the quadruple representation using temporary names the entries in the symbol table against those temporaries can be obtained.
- The advantage with quadruple representation is that one can quickly access the value of temporary variables using symbol table. Use of temporaries introduces the level of indirection for the use of symbol table in quadruple representation.

- Whereas, in triple representation the pointers are used. By using pointers one can access directly the symbol table entry.
- The quadruple representation is beneficial for code optimization.
- In indirect triple list of all references to computations is made separately and stored. Thus indirect triple and quadruple representations are similar as far as their utility is concerned. But indirect triple saves some amount of space as compared with quadruple representation.

Comparison between Quadruple, Triple and Indirect Triple

Quadruple	Triple	Indirect triple
The format of quadruple is (operator, operand1, operand2, result)	The format of triple is (operator, operand1, operand2)	The format of triple is (operator, operand1, operand2) but it makes use of two tables.
In this representation, the entries in the symbol table against the temporaries can be obtained. One can quickly access the value of temporary variables using symbol table.	In triple representation the **pointers** are used. By using pointers one can access directly the symbol table entry.	In indirect triple list of all references to computations is made separately and stored. Thus indirect triple and quadruple representations are similar as far as their utility is concerned. But indirect triple **saves some amount of space** as compared with quadruple representation.
The **quadruple** representation is beneficial for **code optimization.**	The **use of pointer** allows to **access the symbol table** entries quickly.	During the code optimization the **task of moving the instructions around** and **deleting the instructions** is involved. These are the easy tasks for quadruples but difficult for triples.

Example 6.3.1 *Construct triples of an expression* $a * -(b+c)$

Solution : Step 1 : We will first write the three address code for given expression

$t_1 := a$

$t_2 := b$

$t_3 := t_2 + c$

$t_4 := \text{uminus } t_3$

$t_5 := t_1 * t_4$

Step 2 : Design quadruple representation from the three address code obtained in step 1.

Location	OP	Avg. 1	Avg. 2	result
(0)		a		t_1
(1)		b		t_2
(2)	+	t_2	c	t_3
(3)	uminus	t_3		t_4
(4)	*	t_1	t_4	t_5

Step 3 : Create triple representation from quadruple representation obtained in step 2.

Location	OP	Avg. 1	Avg. 2
(0)		a	
(1)		b	
(2)	+	(1)	c
(3)	uminus	(2)	
(4)	*	(0)	(3)

Triple representation

Example 6.3.2 *Write the quadruples, triple and indirect triples for the expression :*
$-(a*b)+(c+d)-(a+b+c+d)$ **GTU : Winter-15, Marks 7**

Solution : The three address code can be -

$t_1 := a * b$

$t_2 := \text{uminus } t_1$

$t_3 := c + d$

$t_4 := t_2 + t_3$

$t_5 := a + b$

$t_6 := t_5 + t_3$

$t_7 := t_4 - t_6$

Quadruple

Location	Operator	Operand 1	Operand 2	result
(1)	*	a	b	t_1
(2)	uminus	t_1		t_2
(3)	+	c	d	t_3
(4)	+	t_2	t_3	t_4
(5)	+	a	b	t_5
(6)	+	t_5	t_3	t_6
(7)	–	t_4	t_6	t_7

Triple

Location	Operator	Operand 1	Operand 2
(1)	*	a	b
(2)	uminus	(1)	
(3)	+	c	d
(4)	+	(2)	(3)
(5)	+	a	b
(6)	+	(5)	(3)
(7)	–	(4)	(6)

Indirect Triple

Location	Operator	Operand 1	Operand 2
(11)	*	a	b
(12)	uminus	(11)	
(13)	+	c	d
(14)	+	(12)	(13)
(15)	+	a	b
(16)	+	(15)	(13)
(17)	–	(14)	(16)

	Statement
(1)	(11)
(2)	(12)
(3)	(13)
(4)	(14)
(5)	(15)
(6)	(16)
(7)	(17)

Example 6.3.3 *Translate the expression $-(a+b)*(c+d)+(a+b+c)$ into 1. Quadruples 2. Triples 3. Indirect triples.* **GTU : Winter-11,20, Marks 7**

Solution : The three address code can be -

$t_1 : = a + b$

$t_2 : =$ uminus t_1

$t_3 : = c + d$

$t_4 : = t_2 * t_3$ /* – (a + b) * (c + d) */

$t_5 : = t_1 + c$ /* a + b + c */

$t_6 : = t_4 + t_5$ /* – (a + b) * (c + d) + (a + b + c) */

Quadruple :

Location	operator	operand 1	operand 2	Result
(1)	+	a	b	t_1
(2)	uminus	t_1		t_2
(3)	+	c	d	t_3
(4)	*	t_2	t_3	t_4
(5)	+	t_1	c	t_5
(6)	+	t_4	t_5	t_4

Triple :

Location	operator	operand 1	operand 2
(1)	+	a	b
(2)	uminus	t_1	
(3)	+	c	d
(4)	*	(2)	(3)
(5)	+	(1)	c
(6)	+	(4)	(5)

Example 6.3.4 *Convert the following into quadruple, triple and indirect triple forms : –(a+b) *(c–d)* **GTU : Summer-14, Marks 6**

Solution : The three address code is

$t_1 : = a + b$

$t_2 := \text{uminus } t_1$

$t_3 := c - d$

$t_4 := t_2 * t_3$

Quadruple

Location	Operator	Operand 1	Operand 2	Result
(1)	+	a	b	t_1
(2)	uminus	t_1		t_2
(3)	–	c	d	t_3
(4)	*	t_2	t_3	t_4

Triple

Location	Operator	Operand 1	Operand 2
(1)	+	a	b
(2)	uminus	(1)	
(3)	–	c	d
(4)	*	(2)	(3)

Indirect triple

Location	Operator	Operand 1	Operand 2
(11)	+	a	b
(12)	uminus	(11)	
(13)	–	c	d
(14)	*	(12)	(13)

	Statement
(1)	(11)
(2)	(12)
(3)	(13)
(4)	(14)

Example 6.3.5 *Convert the following statement into triple, indirect triple and quadruple forms. A = (B + C) $ E + (B + C) * F* **GTU : Summer-15, Marks 7**

Solution : Quadruple

Location	Operator	Operand 1	Operand 2	result
(1)	+	B	C	t_1
(2)	$	t_1	E	t_2

(3)	*	t_1	F	t_3
(4)	+	t_2	t_3	t_4
(5)	=	t_4		A

Triple

Location	Operator	Operand 1	Operand 2
(1)	+	B	C
(2)	$	(1)	E
(3)	*	(1)	F
(4)	+	(2)	(3)
(5)	=	(4)	A

Indirect Triple

Location	Operator	Operand 1	Operand 2
(11)	+	B	C
(12)	$	(11)	E
(13)	*	(11)	F
(14)	+	(12)	(13)
(15)	=	(14)	A

	Statement
(1)	(11)
(2)	(12)
(3)	(13)
(4)	(14)
(5)	(15)

Example 6.3.6 *Translate the expression – (a * b) + (c * d) + (a * b * c) into 1) Quadruples 2) Triples 3) Indirect triples* **GTU : Summer-16, Marks 7**

Solution : The three address code can be

$t_1 := a * b$

$t_2 := \text{uminus } t_1$

$t_3 := c * d$

$t_4 := t_2 + t_3$

$t_5 := t_1 * c$

$t_6 := t_4 + t_5$

Quadruple

Location	Operator	Operand 1	Operand 2	Result
(1)	*	a	b	t_1

(2)	uminus	t_1		t_2
(3)	*	c	d	t_3
(4)	+	t_2	t_3	t_4
(5)	*	t_1	c	t_5
(6)	+	t_4	t_5	t_6

Triple

Location	Operator	Operand 1	Operand 2
(1)	*	a	b
(2)	uminus	(1)	
(3)	*	c	d
(4)	+	(2)	(3)
(5)	*	(1)	c
(6)	+	(4)	(5)

Indirect Triple

Location	Operator	Operand 1	Operand 2
(11)	+	a	b
(12)	uminus	(11)	
(13)	*	c	d
(14)	+	(12)	(13)
(15)	*	(11)	c
(16)	+	(14)	15

	Statement
(1)	(11)
(2)	(12)
(3)	(13)
(4)	(14)
(5)	(15)
(6)	(16)

Example 6.3.7 *Write three address code for x=*y; a=&x;*

Solution :

100: t_1 := ptr_of y

101: x := t_1

102: t_2 := addr_of x

103: a := t_2

Example 6.3.8 *Place the above generated 3-address code in Triplets and Indirect Triplets.*

Solution : Quadruple

Location	Operator	Operand1	Operand2	Result
(0)	ptr_of	y		t_1
(1)	:=	t_1		x
(2)	addr_of	x		t_2
(3)	:=	t_2		a

Triple

Location	Operator	Operand1	Operand2
(0)	ptr_of	y	
(1)	:=	x	(0)
(2)	addr_of	x	
(3)	:=	a	(2)

Indirect Triple

Location	Operator	Operand1	Operand2
(11)	ptr_of	y	
(12)	:=	x	(0)
(13)	addr_of	x	
(14)	:=	a	(2)

	Statement
(0)	(11)
(1)	(12)
(2)	(13)
(3)	(14)

Example 6.3.9 *What is importance of intermediate code ? Discuss various representations of three address code using the given expression. a = b * – c + b * – c.*

GTU : Winter-17, Marks 7

Solution : Importance : Refer section 6.1.1

Three address code

t1 : = uminus c

t2 : = a * t1

t3 : = t2 + t2

a : = t3

Quadruple

Location	Operator	Operand 1	Operand 2	Result
(1)	uminus	c		t1
(2)	*	a	t1	t2
(3)	+	t2	t2	t3
(4)	: =	t3		a

Triple

Location	Operator	Operand 1	Operand 2
(1)	uminus	c	
(2)	*	a	(1)
(3)	+	(2)	(2)
(4)	: =	(3)	a

Example 6.3.10 *Translate following arithmetic expression (a * b) + (c + d) – (a + b) into 1] Quadruples 2] Triple 3] Indirect Triple* **GTU : Winter-19, Marks 7**

Solution : The given expression is

(a * b) + (c + d) – (a + b)

The three address code is,

t_1 : = a * b

t_2 : = c + d

t_3 := t_1 + t_2

t_4: = a + b

t_5: = t_3 – t_4

Quadruples :

Location	Operator	Operand 1	Operand 2	Result
(1)	*	a	b	t_1
(2)	+	c	d	t_2
(3)	+	t_1	t_2	t_3
(4)	+	a	b	t_4
(5)	–	t_3	t_4	t_5

Triple :

Location	Operator	Operand 1	Operand 2
(1)	*	a	b
(2)	+	c	d
(3)	+	(1)	(2)
(4)	+	a	b
(5)	–	(3)	(4)

Indirect Triple :

Location	Operator	Operand 1	Operand 2
(11)	*	a	b
(12)	+	c	d
(13)	+	(11)	(12)
(14)	+	a	b
(15)	–	(13)	(14)

LOC	Statement
(1)	(11)
(2)	(12)
(3)	(13)
(4)	(14)
(5)	(15)

Example 6.3.11 *Write three address code for a: = a+a*b+a*b*c–a/b+a*b* **GTU : Summer-20, Marks 4**

Solution: The three address code is as follows -

T1 = a*b

T2 = a/b

T3 = a+T1

T4 = T3+T1

T5 = T4*c

T6 = T5–T2

T7 = T6+T1

Review Questions

1. *What is intermediate form of the code ? What are the advantages of it ? What are generally used intermediate forms ? Write N-Tuple notation for : (a+b)*(c+d)–(a+b+c).*
   GTU : Summer-12, Marks 7
2. *Explain quadruple, triple and indirect triple with suitable example.*
   GTU : Winter-14, Marks 7
3. *What is intermediate code ? What is its importance ? Discuss various representations of three address code.*
   GTU : Summer-15,17, Winter-18, Marks 7

6.4 Syntax Directed Translation Mechanisms

For obtaining the three address code the SDD translation scheme or semantic rules must be written for each source code statement. There are various programming constructs for which the semantic rules can be defined. Using these rules the corresponding intermediate code in the form of three address code can be generated. Various programming constructs are -

1. Declarative statement
2. Assignment statement
3. Arrays
4. Boolean expressions
5. Control statement
6. Switch case
7. Procedure calls

6.5 Types and Declarations

In the declarative statements the data items along with their data types are declared.

For example :

S → D	{offset:=0}
D → id:T	{enter_tab(id.name,T.type,offset); offset:=offset+T.width)}
T→ integer	{T.type:=integer; T.width:=4}
T → real	{T.type:=real; T.width:=8}
T→ array[num] of T_1	{T.type:=array(num.val,T_1.type) T.width:=num.val X T_1.width }
T → *T_1	{T.type:=pointer(T.type) T.width:=4}

- Initially, the value of offset is set to zero. The computation of offset can be done by using the formula offset = offset + width.
- In the above translation scheme **T.type, T.width** are the synthesized attributes. The type indicates the data type of corresponding identifier and width is used to indicate the memory units associated with an identifier of corresponding type. For instance integer has width 4 and real has 8.
- The rule D → id:T is a declarative statement for id declaration. The enter_tab is a function used for creating the symbol table entry for identifier along with its type and offset.
- The width of array is obtained by multiplying the width of each element by number of elements in the array.
- The width of pointer type is supposed to be 4.

6.6 Translation of Expressions

GTU : Winter 17, Marks 4

The assignment statement mainly deals with the expressions. The expressions can be of type integer, real, array and record. In this section we will see how to write the syntax directed translation scheme for generation of three address code for assignment statement containing arithmetic expressions.

Example 6.6.1 *Obtain the translation scheme for obtaining the tree address code for the grammar -*

$$S \rightarrow id := E$$
$$E \rightarrow E_1 + E_2$$
$$E \rightarrow E_1 * E_2$$
$$E \rightarrow - E_1$$
$$E \rightarrow (E_1)$$
$$E \rightarrow id$$

Solution : Here we will use the translation scheme which in turn will generate the three address code.

Production Rule	Semantic actions

S → id:= E	{ id_entry:=look_up(id.name);
	if id_entry ≠ nil then
	append(id_entry ':=' E.place)
	else error; /* id not declared*/
	}

$E \rightarrow E_1 + E_2$	{ E.place:=newtemp(); append(E.place':='E_1.place '+' E_2.place) }
$E \rightarrow E_1 * E_2$	{ E.place:=newtemp(); append(E.place':='E_1.place '*' E_2.place) }
$E \rightarrow -E_1$	{ E.place := newtemp(); append(E.place':=' 'uminus' E_1.place) }
$E \rightarrow (E_1)$	{ E.place:=E1.place }
$E \rightarrow$ id	{ id_entry:=look_up(id.name); if id_entry ≠ nil then append(id_entry ':=' E.place) else error; /* id not declared*/ }

- The look_up returns the entry for id.name in the symbol table if it exists there.
- The function append is for appending the three address code to the output file. Otherwise an error will be reported.
- **newtemp()** is the function for generating new temporary variables.
- E.place is used to hold the value of E.

Example 6.6.2 *Obtain translation scheme and draw the annotated parse tree for the following statement*

$x:=(a+b)*(c+d)$

Solution :

Production Rule	Semantic action for Attribute evaluation	Output
$E \rightarrow$ id	E.place := a	
$E \rightarrow$ id	E.place := b	
$E \rightarrow E_1+E_2$	E.place := t_1	t_1 := a+b
$E \rightarrow$ id	E.place := c	
$E \rightarrow$ id	E.place := d	

$E \rightarrow E_1 + E_2$	E.place := t_2	t_2 := c+d
$E \rightarrow E_1 * E_2$	E.place := t_3	t_3 := (a+b)*(c+d)
$S \rightarrow$ id := E		x := t_3

The annotated parse tree can be drawn as follows -

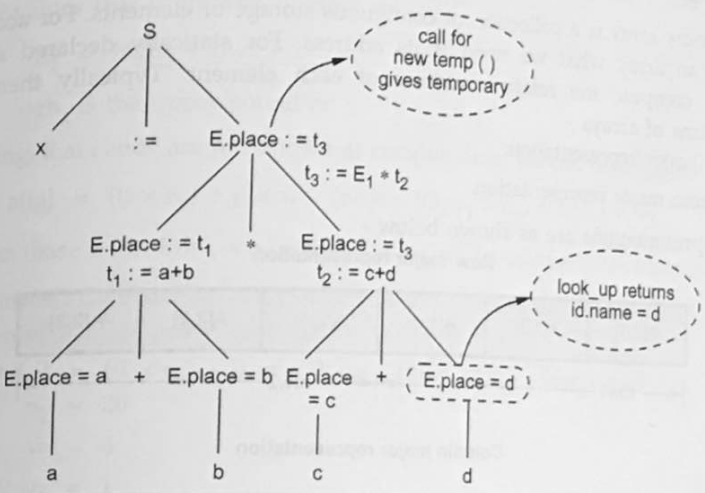


Fig. 6.6.1 Annotated parse tree for generation of three address code

Example 6.6.3 *Write down the translation scheme to generate code for assignment statement. Use the scheme for generating three address code for the assignment statement g: = a+b – c*d.*

Soluiton : Refer section 6.6.

Production rule	Semantic action	Output
$E \rightarrow$ id	E.place := a	
$E \rightarrow$ id	E.place := b	
$E \rightarrow E_1 + E_2$	E.place := t_1	t_1 := a + b
$E \rightarrow$ id	E.place := c	
$E \rightarrow$ id	E.place := d	
$E \rightarrow E_1 * E_2$	E.place := t_2	t_2 = c * d
$E \rightarrow E_1 - E_2$	E.place := t_3	$t_3 = t_1 - t_2$
$S \rightarrow$ id := E		g := t_3

Review Question

1. *Write syntax directed definition to produce three address code for expression containing the operators =, +, -, unary minus,( ) and id.* **GTU : Winter-17, Marks 4**

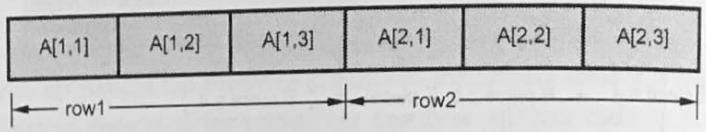
6.7 Arrays

As we know array is a collection of contiguous storage of elements. For accessing any element of an array what we need is its address. For statically declared arrays it is possible to compute the relative address of each element. Typically there are two representations of arrays -

1. Row major representation.
2. Column major representation.

These representations are as shown below -

Row major representation



Column major representation

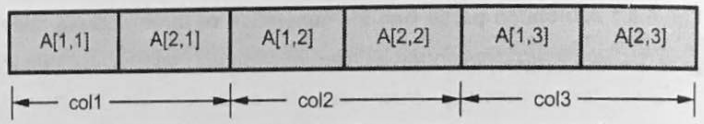


Fig. 6.7.1 Row major and column major representation

To compute the address of any element ;

Let base is the address at a[ ] and w is the width of the element (required memory units) then to compute i^{th} address of a[ ]

base+(i – low) × w

where low is lower bound on subscript. Here a[low]=base

$$\text{base}+(i - \text{low}) \times w = \text{base} + i \times w - \text{low} \times w$$

$$= i \times w + (\text{base} - \text{low} \times w)$$

Let c = base – low × w is computed at compile time. Then the relative address of a[i] can be computed as,

c + (i × w)

Similarly, for calculation of relative address of two dimensional array we need to consider i and j subscripts. Considering, row major representation we will compute the relative address for a[i,j] using following formula :

$$a[i,j] = \text{base} + ((i - \text{low}_1) \times n_2 + (j - \text{low}_2)) \times w$$

where, low_1 and low_2 are the two lower bounds on values of i and j and n_2 is number of values that j can take. In other words;

$$n_2 = \text{high}_2 - \text{low}_2 + 1$$

where, high_2 is the upper bound on j.

Assuming, that i and j are not known at compile time, we can re-write the formula as :

$$a[i,j] = ((i \times n_2) + j) \times w + (\text{base} - ((\text{low}_1 \times n_2) + \text{low}_2) \times w)$$

The term $(\text{base} - ((i - \text{low}_1 \times n_2) + \text{low}_2) \times w)$ can be computed at compile time.

For example : Consider an array A of size 10 × 20 assuming $\text{low}_1 = 1$ and $\text{low}_2 = 1$. The computation of A[i,j] is possible by assuming that w = 4 as :

$$\mathbf{A[i,j] = ((i \times n_2) + j)\ w + (base - ((low_1 \times n_2) + low_2) \times w)}$$

Given, $n_2 = 20$

$w = 4$

$\text{low}_1 = 1$

$\text{low}_2 = 1$

$$A[i,j] = ((i \times 20) + j) \times 4 + (\text{base} - ((1 \times 20) + 1) \times 4$$

$$\mathbf{A[i,j] = 4 \times (20i + j) + (base - 84)}$$

The value base – 84 can be computed at compile time.

The generalized formula for multi-dimensional array by considering row major representation is -

$$a[i_1, i_2, i_3 \ldots i_k] = ((\ldots((i_1 n_2 + i_2) n_3 + i_3) \ldots) n_k + i_k) \times w +$$

$$\text{base}(\ldots((\text{low}_1 n_2 + \text{low}_2) n_3 + \text{low}_3) \ldots) n_k + \text{low}_k) \times w$$

where any value of j for n

$$n_j = \text{high}_j - \text{low}_j + 1$$

and computation of the term $\text{base} (\ldots((\text{low}_1 n_2 + \text{low}_2) n_3 + \text{low}_3) \ldots) n_k + \text{low}_k) \times w$ is done at compile time, after getting the base address.

Let us now discuss the translation scheme for generating three address code for array references. We have to consider the context free grammar for array references :

$$L \rightarrow \text{id [List]} \mid \text{id}$$

$$\text{List} \rightarrow \text{List, E} \mid \text{E}$$

List represents the list of index having limit of n_j and E could be any arithmetic expression. This grammar should support any dimensional array reference. Hence we will modify the grammar and rewrite it as :

$L \rightarrow [List] \mid id$

$List \rightarrow List, E \mid id\,[E$

That means array name is associated with leftmost index expression (i.e. List).The complete grammar for array reference can be given as :

$S \rightarrow L := E$

$E \rightarrow E + E$

$E \rightarrow (E)$

$E \rightarrow L$

$L \rightarrow List\ ]$

$L \rightarrow id$

$List \rightarrow List_1, E$

$List \rightarrow id\ [E$

The translation scheme for generating three address code is given by using appropriate semantic actions.

Rule No.	Production rule	Semantic rule

1)	S → L: = E	{if L.offset := NULL /* L is **id** only */ then
		append(L.place ':=' E.place)
		else
		append(L.place '[' L.offset']'':='E.place)
		}

2)	$E \rightarrow E_1 + E_2$	{E.place := newtemp(); append(E.place ':=' E_1.place'+' E_2.place) }
3)	$E \rightarrow (E_1)$	{E.place := E_1.place }

4)	E → L	{if L.offset = NULL then
		E.place:=L.place
		else
		{
		E.place := newtemp()
		append(E.place ':=' L.place'[' L.offset']')
		}
		}

5)	L → List]	{L.place :=newtemp()
		L.offset:=newtemp()
		append(L.place ':=' c(List.array));
		append(L.offset ':=' List.place '*' Size(List.array))
		}

6)	L → id	{L.place := id.place; L.offset := NULL }

7)	$List \rightarrow List_1, E$	{t:=newtemp()
		dim:=$List_1$.ndim+1;
		append(t ':='$List_1$.place '*' Limit($List_1$.array,dim));
		append(t ':='t '+' E.place)
		List.array := $List_1$.array;
		List.place := t;
		List.ndim :=dim;
		}

8)	List → id [E	{List.array:=id.place List.place:=E.place List.ndim :=1 }

- For the rule 1) if L is simple name then normal assignment is done otherwise the indexed assignment is done at specified location in L. The attribute *offset* will give the relative address for the corresponding entry in the symbol table. And the attribute *place* will refer to a pointer to symbol table entry for that name. Here append is a function used to generate the three address code and append it to output file.
- The rule 2) is for arithmetic operation, in which the newtemp() function is used to generate the temporary variable t_1, t_2... The value of E can be denoted by E.place and computed using E_1.place + E_2.place.
- For the rule E → L we want r-value of L. Hence in the semantic actions computation of r-value of L can be done. If L.offset is NULL then it indicates that L is simple id. Otherwise we will use indexing such as L.offset to obtain the E.place.
- To represent the first term (of an expression for a $[i_1, i_2, ..., i_k]$ i.e. $((... (i_1 n_2 + i_2) n_3 + i_3) ...) n_k + i_k) \times w$) L.offset is used and here L.offset is new temporary. Similarly the second term.
- Base $(... ((low_1 n_2 + low_2) n_3 + low_3) ...) n_k + low_k) \times w$ can be represented by L.place. The function c(List.array) is used to obtain the L.place. The function Size(List.array) is used to return the value of width w.

- The L.offset = NULL indicates simple name of id.
  In rule (7) the List.ndim indicates number of index expressions (dimensions).Function Limit($List_1$.array, dim) returns n_j. The n_j means number of elements along j^{th} dimension of array. The E.place denotes the temporary value obtained by computing value from index expression.
- Consider formula a $[i_1, i_2, i_3 \ldots i_k]$ = $((\ldots((i_1 n_2 + i_2) n_3 + i_3)\ldots) n_k + i_k) \times w +$ base $(\ldots((low_1 n_2 + low_2) n_3 + low_3)\ldots) n_k + low_k) \times w$

The List for expressions can be produced by

$$(\ldots((i_1 n_2 + i_2) n_3 + i_3 \ldots) n_m + i_m$$

Using recurrence relation

$$e_1 = i_1$$

$$e_m = e_m{-}1 \; n_m + i_m$$

By rule (7) $List_1$.place corresponds to $e_m{-}1$ and List.place corresponds to e_m

$\therefore$ $e_m = e_m{-}1 \; n_m$

List.place := $List_1$.place * Limit($List_1$.array, dim)

In rule (8) The value of E is finalized.Hence E.place holds the value of expression E and value of E for having dimension m = 1.

Example for Understanding

Example 6.7.1 *Generate the three address code for the expression x:=A[i,j] for an array 10×20. Assume $low_1 = 1$ and $low_2 = 1$.*

Solution : Given that

$low_1 = 1$ and $low_2 = 1$

$n_1 = 10$, $n_2 = 20$

$$A[i, j] = ((i \times n_2) + j) \times w + (\text{base} - ((low_1 \times n_2) + low_2) \times w)$$

$$A[i, j] = ((i \times 20) + j) \times 4 + (\text{base} - ((1 \times 20) + 1) \times 4$$

$$A[i, j] = 4 \times (20i + j) + (\text{base} - 84)$$

The three address code for this expression can be,

```
t1 := i * 20
t1 := t1 + j
t2 := c   /*computation of c = base – 84 */
t3 := 4 * t1
t4 := t2 [t3]
x := t4
```


The annotated parse tree can be drawn as follows

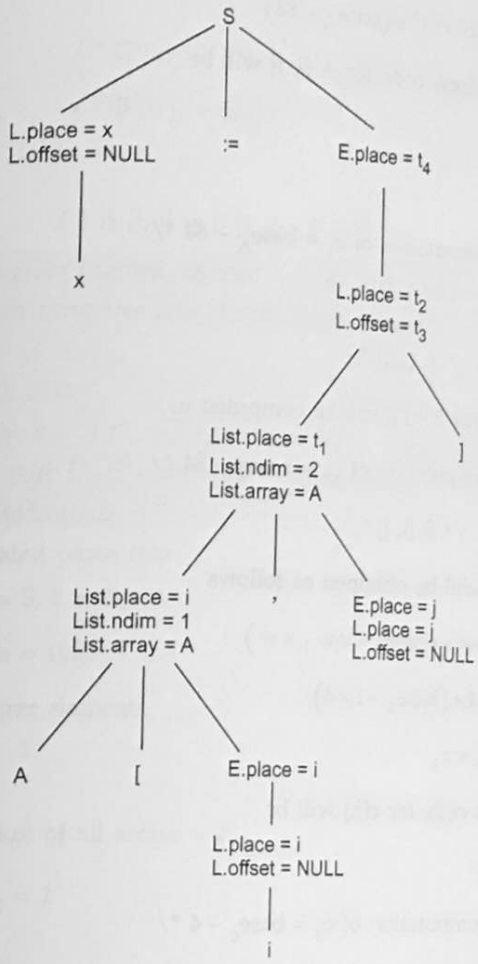


Fig. 6.7.2 Annotated parse tree for x := A[i, j]

Example 6.7.2 *Translate the following integer array operation into three address code. A [i, j] : = B [i, j] + C [k] where A and B are of size 10 x 20 and C contains 50 elements.*

Solution : We will assume $low_1 = 1$ and $low_2 = 1$ and it is given that $n_1 = 10$ and $n_2 = 20$. Integer element

$\therefore$ w = 4 bytes

$$A[i, j] = ((i \times n_2) + j) \times w + (\text{base}_A - ((low_1 \times n_2) + low_2) \times w)$$

$$= ((i \times 20) + j) \times 4 + (\text{base}_A - ((1 \times 20) + 1) \times 4)$$

$$= 4 \times (20\,i + j) + (base_A - 84)$$

Hence the three address code for A [i, j] will be

$t_0 := i * 20$

$t_1 := t_0 + j$

$t_2 := c_1$ /* computation of $c_1 = base_A - 84$ */

$t_3 := 4 * t_1$

$t_4 := t_2[t_3]$ /* A [i, j] */

Similarly the value for B [i, j] can be computed as

$t_5 := c_2$ /* computation of $c_2 = base_B - 84$ */

$t_6 := t_5[t_3]$ /* B [i, j] */

The value for C [k] will be obtained as follows

$$k \times w + (base_c - low_1 \times w)$$

$$= k \times 4 + (base_c - 1 \times 4)$$

$C[k] = 4\,k + c_3$

Hence three address code for c[k] will be

$t_7 := 4 * k$

$t_8 := c_3$ /* computation of $c_3 = base_c - 4$ */

$t_9 := t_8[t_7]$

Hence the 3 address code for given expression -

$t_0 := j * 20$

$t_1 := t_0 + j$

$t_3 := 4 * t_1$

$t_5 := c_2$

$t_6 := t_5[t_3]$ /* B [i, j] */

$t_7 := 4 * k$

$t_8 := c_3$

$t_9 := t_8[t_7]$ / * C [k] * /

$t_{10} := t_6 + t_9$ / * B [i, j] + C[k] * /

$t_2 := c_1$

$t_2[t_3] := t_{10}$ / * A [i, j] := B [i, j] + C [k] * /

Example 6.7.3 *For the given program fragment A [i ,j] : = B [i, k] do the following :*

i) Draw the annotated parse tree with the translation scheme to convert to three address code.

ii) Write three address code.

iii) Determine the address of A [3, 5] where all are integer arrays with size of A as 10 × 10 and B as 10 × 10 with k = 2 and the start index position of all arrays is at 1. (Assume the base addresses).

Solution : i) The annotated parse tree -

Given that : i = 3, j = 5, k = 2.

Assume base address = 1000

The array stores integer elements.

$\therefore \quad w = 2$

The start index position of all arrays = 1.

$\therefore \quad low_1 = low_2 = 1$

$n_1 = n_2 = 10$

$\therefore \quad A[i, j] = ((i * n_2) + j) * w + (base - ((low_1 * n_2) + low_2) * w)$

$= ((i * 10) + j) * w + (base - ((1 * 10) + 1) * 2)$

$A[i, j] = 2 * (10\,i + j) + (base - 42)$

Similarly,

$B[i, k] = 2 * (10\,i + k) + (base - 42)$

ii) The three address code for this expression is,

$t_1 := i * 10$

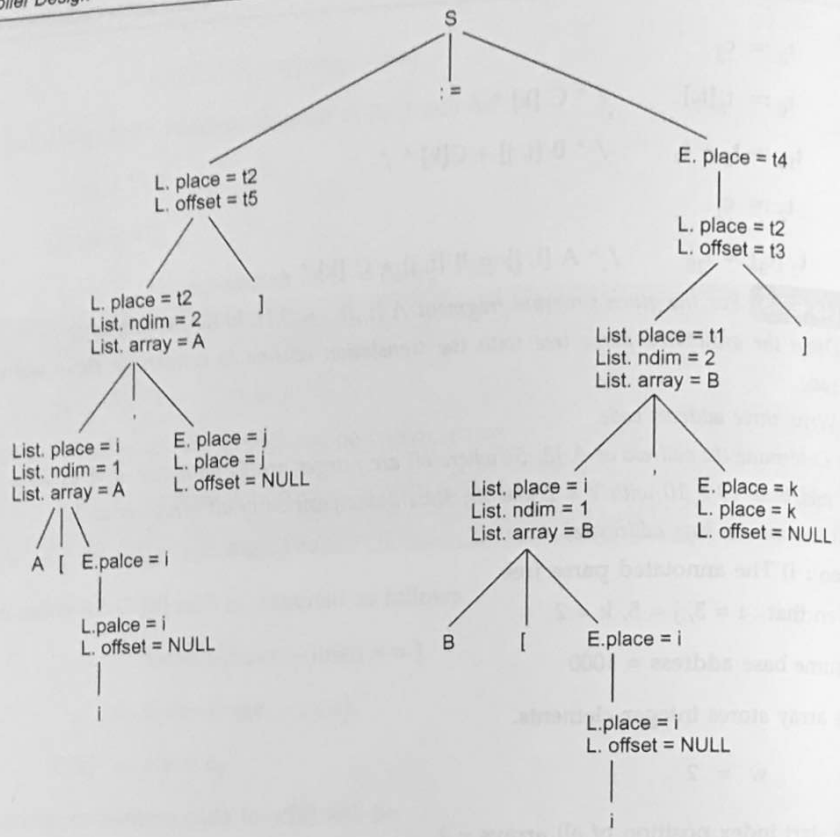


Fig. 6.7.3

$t_1 := t_1 + k$

$t_2 := c$ /* base – 42 is c */

$t_3 := 2 * t_1$

$t_4 := t_2[t_3]$ /* B [i, k] */

$t_5 := i * 10$

$t_5 := t_5 + j$

$t_5 := 2 * t_5$

$t_2 [t_5] := t_4$ /* A [i, j] = t_2 [t_5] */

iii) The formula used for determining the address of B [i, k] using row major representation is,

$$B[i, k] = ((i * n_2) + k) * w + (base - ((low_1 * n_2) + low_2) * w)$$
$$= 2 * (10 i + k) + (base - 42)$$
$$= 2 * (10 * 3 + 2) + (1000 - 42)$$
$$B[3, 2] = 1022$$

As the address,

$$A[i, j] := B[i, k] \text{ Hence}$$
$$\mathbf{A[i, j] = 1022}$$

Example 6.7.4 *Generate intermediate code for the following code segment along with required syntax directed translation scheme.*

$S := S + a [i] [j]$

Solution : Syntax Directed Translation Scheme : Refer table on page 6 - 24.

Intermediate Code in the Form of Three Address Code :

Assumption : Assume array a of 10 × 20 size with $low_1 = low_2 = 1$. Hence

$t_1 := i * 20$

$t_1 := t_1 + j$

$t_2 := c$ /* Constant c = $base_a$ – 84 */

$t_3 := 4 * t_1$

$t_4 := t_2 [t_3]$

$t_5 := S + t_4$

$S := t_5$

6.8 Boolean Expressions

Normally there are two types of Boolean expressions used :

1. For computing the logical values.
2. In conditional expressions using if-then-else or while-do.

Consider the Boolean expression generated by following grammar :

$E \rightarrow E$ **OR** E

$E \rightarrow E$ **AND** E

$E \rightarrow$ **NOT** E

$E \rightarrow (E)$

$E \rightarrow$ **id relop id**

$E \rightarrow$ TRUE

$E \rightarrow$ FALSE

Here the relop is denoted by ≤, ≥, ≠, <, >. The OR and AND are left associate. The highest precedence is to NOT then AND and lastly OR.

6.8.1 Numerical Representation

The translation scheme for Boolean expressions having numerical representation is as given below :

Translation scheme for generation of **3 address code for Boolean expression**

Production rule	Semantic rule
$E \rightarrow E_1$ **OR** E_2	{ E.place:=newtemp() append(E.place ':=' E_1.place 'OR' E_2.place) }
$E \rightarrow E_1$ **AND** E_2	{ E.place:=newtemp() append(E.place ':=' E_1.place 'AND' E_2.place) }
$E \rightarrow$ **NOT** E_1	{ E.place:=newtemp() append(E.place ':='"NOT' E_1.place) }
$E \rightarrow (E_1)$	{ E.place := E_1.place }
$E \rightarrow id_1$ **relop** id_2	{ E.place := newtemp() append('if' id_1.place relop.op id_2.place 'goto' next_state+3); append(E.place':=' '0'); append('goto' next_state +2); append(E.place := '1') }
$E \rightarrow$ TRUE	{ E.place := newtemp(); append(E.place ':=' '1') }
$E \rightarrow$ FALSE	{ E.place := newtemp() append(E.place ':=' '0') }

The function append generates the three address code and newtemp() is for generation of temporary variables. For the semantic action for the rule -

$E \rightarrow id_1$ relop id_2 contains next_state which gives the index of next three address statement in the output sequence. Let us take an example and generate the three address code using above translation scheme -

p > q AND r < s OR u > v

100: if p > q goto 103
101: t_1:=0
102: goto 104
103: t_1:=1
104: if r < s goto 107
105: t_2:=0
106: goto 108
107: t_2:=1
108: if u > v goto 111
109: t_3:=0
110: goto 112
111: t_3:=1
112: t_4:= t_1 AND t_2
113: t_5:= t_4 OR t_3

In this three address code we have used goto to jump on some specific statement. This method of evaluation is called **"short-circuit"**. The AND has higher precedence over OR hence at location 112 the ANDing is performed and then in the immediate next statement ORing is performed.

6.8.2 Flow of Control Statements

In this section we will discuss the translation of Boolean expression into three address code.The control statements are if-then-else and while-do. The grammar for such statements is as shown below

$S \rightarrow$ if E then S_1
| if E then S_1 else S_2
| while E do S_1

While generating three address code -

- To generate new symbolic label the function new_label() is used.
- With the expression E.true and E.false are the labels associated.

S.code and E.code is for generating three address code.

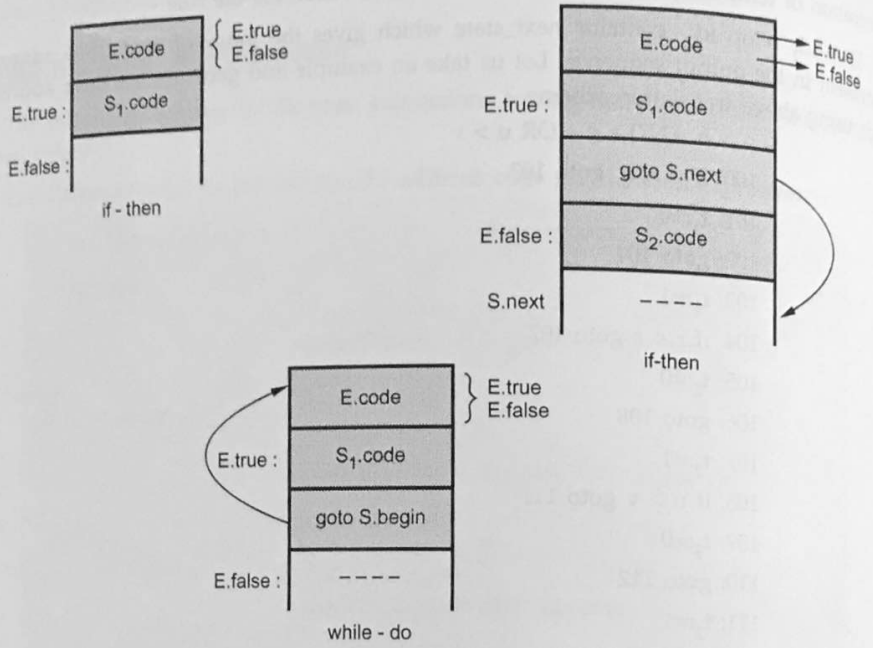


Fig. 6.8.1

The short circuit code is as follows.

$S \rightarrow$ if E then S_1

E.true := new_label()

E.false := S.next

S_1.next := S.next

S.code := E.code || gen_code(E.true':') || S_1.code

In the above translation scheme || is used to concatenate the strings. The function gen_code is used to evaluate the non quoted arguments passed to it and to concatenate complete string. The S.code is the important rule which ultimately generates the three address code.

Consider the statement is if a < b then a=a+5 else a=a+7

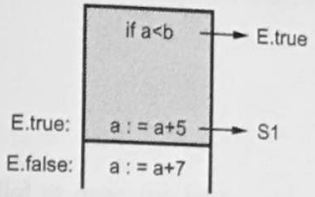


Fig. 6.8.2

The three address code for if-then is -

100 : if a < b goto L1

101 : goto 103

102 : L1: a=a+5 /*E.true*/

103: a:=a+7

Here E.code is "if a < b " L1 denotes E.true and and E.false is shown by jumping to line 103(i.e. S.next)

Similarly,

$S \rightarrow$ if E then S_1 else S_2

E.true := new_label()

E.false := new_label()

S_1.next := S.next

S_2.next := S.next

S.code := E.code || gen_code(E.true ':') ||

S_1.code || gen_code('goto', S.next) ||

gen_code (E.false ':') || S_2.code

$S \rightarrow$ while E do S_1

S.begin := new_label()

E.true := new_label()

E.false := S.next

S_1.next := S.begin

S.code := gen_code(S.begin' :') || E.code ||

gen_code(E.true':')|| S_1.code || gen_code('goto',S.begin)

Examples for Understanding

Example 6.8.1 *Generate the three address code for*

```
while (i<10)
{
  x=0;
i=i+1;
}
```

Solution : The three address code can be generated as follows

100. L1: if i<10 goto L2
101. goto Lnext
102. L2: x=0
103. i=i+1
104. goto L1
105. Lnext:

Here we use the translation scheme for while. ⟹

S.begin	=	new_label() = L1
E.true	=	new_label() = L2
E.code	=	"if i<10 goto "
E.false	=	S.next = Lnext
S_1.code	=	x = 0; i = i + 1;

Example 6.8.2 *Construct 3 address code for the following*

If $\left[(a < b) \text{ and } ((c > d) \text{ or } (a > d))\right]$ *then*

z = x + y * z

else z = z + 1

Solution : Three address code

100 : if a < b goto 102
101 : goto 110
102 : if c < d goto 106
103 : goto 104
104 : if a > d goto 106
105 : goto 110
106 : t_1 := x+y
107 : t_2 := t_1 * z
108 : z := t_2

109 : goto 112
110 : t_3 := z+1
111 : z := t_3
112 : Stop

Examples with Solution

Example 6.8.3 *Consider the following while-statement-while A > B and A <= 2* B – 5 do*

A : A+B ;

i) Construct the parse tree for the above given while-statement

ii) Write the intermediate code for while-statement.

Solution : i) Parse Tree

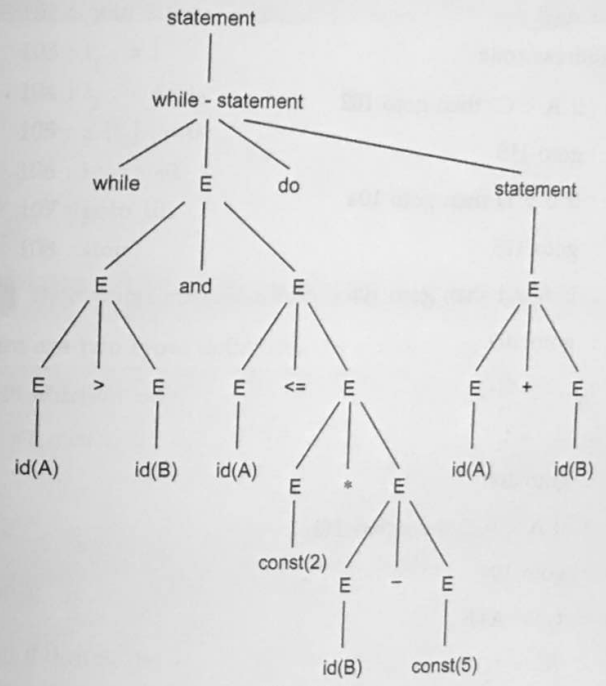


Fig. 6.8.3

ii) Intermediate Code

L_1: if A > B goto L_2
goto L_{end}
L_2: t_1 := 2 * B

$t_2 := t_1 - 5$

if $A <= t_2$

goto t_4

L_4: A := A + B

goto t_1

L_{end} : stop

Example 6.8.4 *Generate the three address code for the following program fragment -*

While (A < C and B > D) *do*

if A = 1 then C = C+1

else while A < = D do

A = A+B

Solution : Three address code

100 : if A < C then goto 102

101 : goto 115

102 : if B > D then goto 104

103 : goto 115

104 : if A = 1 then goto 106

105 : goto 109

106 : t_1 : = C+1

107 : C : = t_1

108 : goto 100

109 : if A < = D then goto 111

110 : goto 100

111 : t_2 : = A+B

112 : A : = t_2

113 : goto 109

114 : goto 100

115 : •

Example 6.8.5 *Translate the executable statements of the following C program into three-address code :*

```
Main ( )
{
    int i ;
    int a [10] ;
    i = 1 ;
    while (i ≤ 10)
        {
            a [i] = 0;
            i = i + 1 ;
        }
}
```

Solution :

100 : i : = 1

101 : if i <= 10 then goto 103

102 : goto 108

103 : t_1 : = i

104 : t_2 : = 4 * t_1

105 : a [t_2] : = 0

106 : i : = i +1

107 : goto 101

108 : stop

Example 6.8.6 *Give syntax directed definition for if-else statement.*

Solution : There are two types definitions for if-else statement.

	Production rule	Semantic rule
1.	S → if E then S_1	E • true : = new_label () E • false : = S.next S_1 • next : = S.next S • code : = E • code || gen_code(E • true ' : ')|| S_1 • code
2.	S → if E then S_1 else S_2	E • true : = new_label () E • false : = new_label () S_1 • next : = S • next S_2 • next : = S • next S • code : = E • code || gen_code(E • true ' : ') || S_1 • code || gen_code ('go to', S • next) || gen_code (E • false ' : ') || S_2 • code

Example 6.8.7 *Generate a three address code for the following segment of code.*

```
C = 0
do
{
    if (a < b) then
    x++ ;
    else
    x--;
    C++;
} while ( C < 5)
```

Solution : The three address code is :

```
C : = 0
L1 : if a < b goto L2
     goto L3
L2 : t1 : = x + 1
     x : = t1
     goto L4
L3 : t1 : = x - 1
     x : = t1
     goto L4
L4 : t2 : = C + 1
     C : = t2
if t2 < 5 goto L1
     goto L5
L5 : .
```

Example 6.8.8 *Write a three address code for following C statement*

```
for (i = 1 ; i < = 10 ; i ++)
{
a [i] = x * 5 ;
}
```

Solution : The 3 address code is

```
          i : = 1
Loop : t1 := x * 5
          t2 : = & a
          t3 : = sizeof(int)
          t4 : = t3 * i
          t5 := t2+t4
          *t5 : = t1
          i : = i+1
          if i <= 10 goto Loop
```


Example 6.8.9 *Translate the following C fragment into three address code.*

```
int i
int a [10] [10]
i = 0
while (i < 10)
{
  a [i] [i] = 1;
   i++;
}
```

Solution : We consider the values stored in 2-D array is in row major form. Assuming 4 byte allocation per word a [i] [j] will be at location [addr(a) + (10*i+j)*4]

$\therefore$ a [i] [i] = addr (a) + 44 * i

The 3 address code then will be -

```
          i = 0
L1:   if i < 10 goto L2
          t1 = 44*i
       a[t1] = 1
          t2 = i+1
          i = t2
          goto t1
L2:   .
```

Example 6.8.10 *Translate the following statement to quadruple :*

if a > b then x = a + b
else x = a – b

Solution : Three address code

Quadruple :

```
100 : if a > b then  goto 102
101 : goto 105
102 : t1 : = a+b
103 : x : = t1
104 : goto 108
105 : t1 : = a–b
106 : x : = t1
107 : goto 108
108 : .
```

	Operator	Operand 1	Operand 2	Result
100	>	a	b	goto 102
101		goto 105		
102	+	a	b	t1
103	:=	t1		x
104		goto 108		
105	–	a	b	t1
106	:=	t1		x
107		goto 108		

Example 6.8.11 *Write syntax directed translation for the flow-of-control statement - i) If - then ii)If - then - else iii) While and iv) For using the translation, convert the following statement to three address code :*

```
if (x> 10) then
        while (a > 10)
        y = x+a
else if (y > 100)    y = a
```

Solution : Refer section 6.8.2.

Three address code

```
100  :    if x>10 goto 102
101  :    goto 106
102  :    if a>10 goto 104
103  :    goto 109
104  :    y : = x+a
105  :    goto 102
106  :    if y>100 goto 108
107  :    goto 109
108  :    y : = a
```

Example 6.8.12 *Generate intermediate code for the following code segment along with the required syntax directed translation scheme.*

```
while (i < 10)
    if (i % 2 = = 0)
        Evensum = evensum + i
    Else
        Oddsum = oddsum + i
```

Solution : Syntax directed translation scheme : Refer section 6.8.2.

Three Address Code

```
100 : if i < 10 goto 102
101 : goto 108
102 : if i % 2 == 0 goto 104
103 : goto 106
104 : EvenSum : = evensum + i
105 : goto 100
106 : Oddsum : = oddsum + i
107 : goto 100
108 : .
```


6.9 Type Checking

6.9.1 Type System

The type analysis and type checking is an important activity done in the semantic analysis phase. The need for type checking is -

i) To detect the **errors** arising in the expression due to incompatible operand.

ii) To generate **intermediate code** for expressions and statements. Typically language supports two types of data types - basic and constructed.

The basic data types are - integer, character, real, Boolean, enumerated data type. And arrays, record(structure), set and pointer are the constructed types. The constructed data types are build using basic data types.

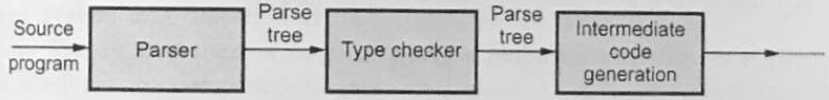


Fig. 6.9.1 Role of type checker

6.9.1.1 Type Expression

- The systematic way of expressing type of language construct is called type expression. Thus the type expression can be defined as -

1) The basic type is called **type expression**. Hence int, char, float, double, enum are type expressions.

2) While performing type checking two special basic types are needed such as **type_error** and **void**. The type_error is for reporting error occurred during type checking and void indicates that there is no type (NULL) associated with the statement.

3) The type name is also a type expression. For example -

```
typedef int *INT_PTR
```

This statements defines the type name INT_PTR as type expression which is actually a integer pointer.

4) The **type constructors** are also type expressions. The type constructors are array, product, struct, product, pointer, function.

- Each of these type expressions are explained with the help of an example as follows -

i) **Array** - An array (I,T) is a type expression where I is an index set which is usually an integer and the data type of the array elements is given by T. The type expression is :

```
array (I,T)
```

For example - int arr[20];

The type expression for an array 'arr' having the elements from 0, 1,...19 is

```
array(0,1,..19,int)
```

ii) **Product** - The type expression for the cartesian product is given by T1 × T2 where T1 and T2 are the two data types. Always × is assumed to be left associative.

iii) **Struct** - The structure given by keyword **struct** is also a type expression. The structure is applied as a product of the type of the fields (members) of the structure. Such a product is a type expression in struct.

For example -

```
struct stud {
        char name[10];
        float marks;
      }
struct stud student[10];
```

Here the type name is stud having the type expression as :

struct ((name × array(0,1...9,char)) × (marks float))

Thus the type expression for student is given as :

array(0,1,...9,stud)

iv) **Pointers** - The type expression for pointer is given as pointer(T) where T is a data type.

For example -

```
float *xyz;
```

then type expression for identifier xyz is given as :

pointer(float)

v) **Function** - The type expression for function is given by domain-range. In the sense that the type expression for function is $D \rightarrow R$.

For example -

```
int sum(int a,int b)
```

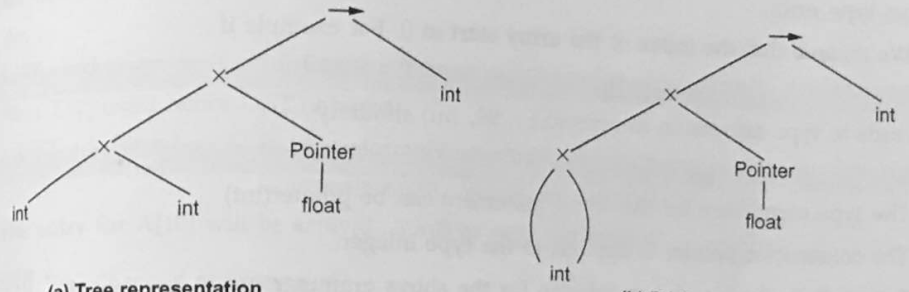
The type expression for sum is -

int × int → int

- The type expression can be represented using tree or DAG (Directed Acyclic Graph).
- Let us take an example for the same :

```
int maxi(int a,int b,float *c)
```


The type expression for function maxi is -



(a) Tree representation (b) DAG representation

Fig. 6.9.2

A **type system** is a collection of rules for assigning type expressions to the language constructs. The type system is implemented by type checker. The type checker finds whether the types are compatible to each other or not by finding the corresponding type expressions.

6.9.2 Specification of Simple Type Checker

In this section we will write a **type checker** for the simple language construct. In this language construct the type of the identifier must be declared before the use of that identifier. The type checker is a **translation scheme** in which the type of each expression from the types of subexpressions is obtained. The type checker can decide the types for arrays, pointers, statements and functions.

Let us consider a grammar for source language.

$S \rightarrow D;E$

$D \rightarrow D;D \mid T\ LIST$

$T \rightarrow char \mid int \mid float$

$LIST \rightarrow L_1\ [num] \mid *L_1 \mid id$

$E \rightarrow literal \mid num \mid id \mid E\ mod\ E \mid E[E] \mid *E \mid E\ op\ E \mid E(E)$

The type checker ensures following things -

i) Each identifier must be declared **before the use.**

ii) The use of identifier must be **within the scope.**

iii) An identifier must **not** have **multiple definitions** at a time within the same scope.

The given grammar has basic data types as char, int, float and for reporting the errors type_error.

We assume that the index of the array start at 0. For example if

```
int arr[100];
```

leads to type expression as array(0,1 ...99, int) similarly

```
int *ptr;
```

The type expression for the above statement can be pointer(int)

The constructor pointer is applied to the type integer.

Let us write the translation scheme for the above grammar.

Production rule	Type expression
S → D;E	This means all declarations before expressions
D → T LIST	LIST.type:=T.type
LIST → id	{addtype(id.entry,LIST.type)}
T → char	{T.type:=char}
T → int	{T.type:=int}
T → float	{T.type:=float}
LIST → *L_1	{L_1.type:= pointer(LIST.type)}
LIST → L_1[num]	{L_1.type:=array(0...num.val–1,LIST.type}

Type checker

This translation scheme is given for declaration of identifiers. For the rule

D →T LIST the data type is saved in the symbol table entry for identifier. For adding the entry of type for the identifier in the symbol table the function addtype(id.entry,LIST.type) is used. If T.type=int then the type of that identifier is int or if T.type=char then the type of that identifier is char and so on. For the declaration of pointer type variable, consider if

```
char *xyz;
```

now the pointer(char) will be the data type of identifier xyz. In the first declaration because of D → T LIST, the data type of LIST=char.

And LIST→*L_1 gives

L_1.type = pointer(LIST.type)
= pointer(char)

hence pointer(char) entry will be added in the symbol table for identifier xyz.

Similarly for

int A[10];

As,

LIST → id LIST.type = T.type

Here T.type=int hence LIST.type=int

∴ LIST→ L_1[num] L_1.type:={array(0...num.val–1),LIST.type}

L_1.type={array(0...9),int} as num.val=10

The entry for A[10] will be array(0...9,int) as type expression.

6.9.2.1 Type Checking of Expression

After obtaining the type expression it becomes convenient to write the semantic rules. These are the semantic rules for type checking of expression.

```
E → literal    {E.type:= char}
E → num        {E.type:= int}
```

Here as E can be literal or num, the data types associated with them can be char or int respectively.

```
E → id    {E.type = look_up(id.entry)
```

The function look_up(id.entry) can be used to obtain the data type of the id. The look_up function reads the symbol table for id entry and thereby it obtains the type of identifier.

```
E → E1 mod E2 { E.type: = if E1.type = int and E2.type = int
                          then int
                          else
                          type_error
                          }
```

In the mod operation, if type of E_1 and type of E_2 is int then only type of E is int. Otherwise the special data type type_error will raise the signal for reporting error in the type.

In the same way while performing binary operation the data types are decided.

```
E → E1 op E2 {E.type:= if E1.type:= int and E2.type:= int
                       then int
                       else if E1.type:= float and E2.type:= float
                            then float
                       else if E1.type:= char and E2.type:= char
                            then char
                       else
                            type_error
                       }
```

In the binary operation, if type of E_1 and type of E_2 is int then only type of E is int. Similarly the type of E_1 and E_2 should be float then only type of E will be float. Also the type of E_1 and E_2 should be char then only type of E will be char. Otherwise, the special data type type_error will raise the signal for reporting error in the type.

For the array reference

```
E → E1[E2]    { E.type : = if E2.type:= int and E1.type:=array(s,t) then t
              else
              type_error
              }
```

Here s is the set of index and t is the data type of the array. Hence the type returned for expression E will be t.

For example -

```
int x[10],z;
z=x[3]
  ↑
  int
```

value of x[3] is int hence z is also integer.

As E_1.type = array(0,...2,int) hence

E_1.type = int

Hence E.type = int

Similarly,

```
char p[5],q;
q = p[1]
  ↑
  int
```

Array p is of character type hence q has a data type char.

For the pointer type variable -

```
E → *E1      {    E.type:=if E1.type:=pointer(t) then t
             else
                  type_error
             }
```

For the function call.

```
E → E1(E2)  {    E.type:= if E2.type:= s and E1.type:=s → t then t
                 else
                 E1.type =s → t then t
                 else type_error
            }
```


In the function call $E_1(E_2)$ the s is the data type for E_2 and t is the return type of $E_1(E_2)$. Hence E.type is t.

6.9.2.2 Type Checking of Statements

Usually, the language constructs like statements do not have values. And therefore the special type **void** is used. The type_error is the basic type used to raise the error signal. Consider the grammar as -

$S \rightarrow id:=E$

$S \rightarrow$ if (E) S_1

$S \rightarrow$ while (E) S_1

$S \rightarrow S_1;S_2$

First of all we will write the translation scheme for checking the type of statements.

Production rule	Semantic rule
$S \rightarrow id:=E$	{S.type:= if id.type=E.type then void else type_error }
$S \rightarrow$ if (E) S_1	{S.type:=if E.type:=boolean(True) then S_1.type else type_error }
$S \rightarrow$ while (E) S_1	{S.type:=if E.type:=boolean(True) then S_1.type else type_error }
$S \rightarrow S_1;S_2$	{S.type:=if S_1.type:=void and S_2.type = void then void else type_error }

The semantic actions for **S → id:=E** tells us that the type of LHS and RHS of the assignment statement should be same.

The statements **S → if (E) S_1** and **S → while (E) S_1** tells us that these are **conditional** and **while** statements and the type of E should be boolean i.e. either true or false.

For the last statement, if the subexpression has type void then the sequence of statements has the type void.

If any **mismatch** in the type occurs then the basic type **type_error** reports the error.

Thus the simple type checker is a combination of production rules and semantic rules.

Review Questions

1. *Explain with some suitable example simple type checker.*
2. *Explain the specification of simple type checker.*

6.10 Short Questions and Answers

Q.1 List out the benefits of using machine independent intermediate forms.

Ans. : There are certain benefits of generating machine independent intermediate code :

1. A compiler for different machines can be created by attaching different backend to the existing front ends of each machine.
2. A compiler for different source languages (on the same machine) can be created by proving different front ends for corresponding source languages to existing back end.
3. A machine independent code optimizer can be applied to intermediate code in order to optimize the code generation.

Q.2 Write the properties of intermediate language.

Ans. : Following are the properties of intermediate languages :

1. The intermediate language is an **easy form** of source language which can be generated efficiently by the compiler.
2. The generation of intermediate language should lead to efficient code generation.
3. The intermediate language should act as **effective mediator** between front and back end.
4. The intermediate language should be flexible enough so that optimized code can be generated.

Q.3 What are the types of three address statements ?

Ans. : There are three types of intermediate code representations :

1. Abstract syntax tree 2. Polish notation 3. Three address code.

Q.4 What are the different ways of implementing the three address code ?

Ans. : The implementation of three address code can be done using :

1. Quadruple representation 2. Triple 3. Indirect Triple.

Q.5 Write down the equation for arrays.

Ans. : The equation for two dimensional arrays is :

$$A[i,j] = ((i * n2) + j) * w + (base - ((low1 * n2) + low2) * w)$$

Q.6 Why are quadruples preferred over triples in an optimizing compiler ?

Ans. : In quadruple representation, the values of temporaries can be accessed using the symbol table. Hence the quadruples are preferred over triples in optimizing compiler.

Q.7 List out the motivations for back patching.

Ans. : i) In a Boolean expression suppose there are two expression E1 && E2. If the expression E1 is false then there is no need to compute rest of the expression. But if we generate the code for the expression E1 we must know where to jump on encountering the false E1 expression. But this can be accomplished by the backpatching technique.

ii) For the flow of control statements(such as if, else, while), the jumping location can be identified using the backpatching.

Q.8 What is the idea behind generating three address codes during compilation ?

Ans. : The intermediate language is an easy form of source language which can be generated efficiently by the compiler. It leads to efficient code the generation and it acts as an effective mediator between front end and back end. Hence it is necessary to generate three address codes during compilation.

Q.9 Translate the arithmetic expression a*–(b+c) into syntax tree and postfix notation.

Ans. :

a)

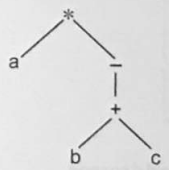


Fig. 6.10.1 Syntax tree

b) Postfix Notation

$a * - (b + c)$

$a * - T_1$ $\quad T_1 = bc+$

T_2 $\quad T_2 = a\ T_1 - *$

By backward substitution

$a\ T_1 - *$

a bc + – *

is a postfix notation

Q.10 What are the types of intermediate languages ?

Ans. : Various types of intermediate languages are

1. Abstract Syntax Tree 2. Polish Notations 3. Three Address Code.

Q.11 What is the intermediate code representation for the expression a or b and not c ?

Ans. : The expression

a or b and not c

is a boolean expression. We can generate an intermediate code for such expression using three address code form. The three address code will be -

t_1 : = a or b

t_2 : = not c

t_3 : = t_1 and t_2

Q.12 What are the various methods of implementing three address statements ?

Ans. : The three address code can be implemented using following methods.

1) **Quadruple :** The quadruple is a structure with at the most four fields such as OP, arg1, arg2 and result.

2) **Triples :** In triple representation the use of temporary variables is avoided by referring the pointers in the symbol table.

3) **Indirect triples :** In indirect triple representation the listing of triples is been done. And listing pointers are used instead of using statements.

6.11 Multiple Choice Questions

Q.1 Which of the following is not an intermediate code form ?

a Syntax Trees b Postfix notations

c Three Address code d Triples

Q.2 The three address code contains _____.

a exactly three address b at the most three addresses

c no operator d no temporary variables

Q.3 While producing the target code from three address statements _____.

a temporary variables will be discarded

b temporary variables will be used as it is

c temporary variables will be assigned with some run time memory location

d none of these

Q.4 x[i]:=y is a _____.

a Unary Operations b Binary Operations

c Ternary operation d None of these

Q.5 Evaluation of postfix expression requires _____.

a operator stack

b operand stack

c operator stack and operand stack both

d none of the above

Q.6 In the intermediate code, the non terminals generating, are called _____.

a marker non terminal b null terminals

c empty production d none of these

Q.7 The relative address of A[i][j] in row major form having low1 and low 2 as lower bounds can be calculated using formula _____.

a base+((i-low2)*n1+j-low1)*w

b base+((j-low1)*n1+i-low2)*w

c base+((i-low2)*n2+j-low1)*w

d base+((i-low1)*n2+j-low2)*w

Answer Keys for Multiple Choice Questions

Q.1	d	Q.2	b
Q.3	c	Q.4	c
Q.5	b	Q.6	a
Q.7	d		

❑❑❑

Notes

7 Run Time Environments

Syllabus

Source Language Issues, Storage Organization. Stack Allocation of Space, Access to Nonlocal Data on the Stack, Heap Management.

Contents

7.1 Source Language Issues

There are various language features that affect the organization of memory. The organization of data is determined by answering the following questions. The source language issues are -

- **Does the source language allow recursion ?**
  While handling the recursive calls there may be several instances of recursive procedures that are active simultaneously. Memory allocation must be needed to store each instance with its copy of local variables and parameters passed to that recursive procedure. But the number of active instances is determined by run time.
- **How the parameters are passed to the procedure ?**
  There are two methods of parameter passing: call by value and call by reference. The allocation strategies for each of these methods are different.

  Some languages support passing of procedures itself as parameter or a return value of the procedure itself could be a procedure.
- **Does the procedure refer nonlocal names ? How ?**
  Any procedure has a access to its local names. But a language should support the method of accessing non local names by procedures.
- **Does the language support the memory allocation and deallocation dynamically ?**
  The dynamic allocation and deallocation of memory brings the effective utilization of memory.

There is a great effect of these source language issues on run time environment. Hence run-time storage management is very important.

7.2 Storage Organization

- The compiler demands for a block of memory to operating system. The compiler utilizes this block of memory for running (executing) the compiled program. This block of memory is called run time storage.
- The run time storage is subdivided to hold code and data such as :
  i) The generated target code
  ii) Data objects
  iii) Information which keeps track of procedure activations.
- The size of generated code is fixed. Hence the target code occupies the statically determined area of the memory. Compiler places the target code at the lower end of the memory.

The amount of memory required by the data objects is known at the compiled time and hence data objects also can be placed at the statically determined area of the memory. Compiler prefers to place the data objects in the statically determined

area because these data objects then can be compiled into target code. For example in FORTRAN all the data objects are allocated statically. Hence the static data area is on the top of code area. The subdivision of run time memory as shown by following figure.

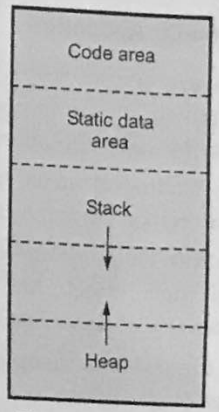


Fig. 7.2.1 Run time storage organization

- The counterpart of control stack is used to manage the active procedures. Managing of active procedures means that when a call occurs then execution of activation is interrupted and information about status of the stack is saved on the stack. When the control returns from the call this suspended activation is resumed after storing the values of relevant registers. Also the program counter is set to the point immediately after the call. This information is stored in the stack area of run time storage. Some data objects which are contained in this activation can be allocated on the stack along with the relevant information.
- The heap area is the area of run time storage in which the other information is stored. For example memory for some data items is allocated under the program control. Memory required for these data items is obtained from this heap area. Memory for some activation is also allocated from heap area.
- The size of stack and heap is not fixed it may grow or shrink interchangeably during the program execution.
- Pascal and C need the run time stack.

7.3 Storage Allocation Strategies

GTU : Winter-13,14,18,20, Summer-18,19, Marks 4

As we have already discussed that the run time storage is divided into :

1. Code area 2. Static data area
3. Stack area 4. Heap area

There are three different storage allocation strategies based on this division of run time storage. The strategies are -

1. **Static allocation** - The static allocation is for all the data objects at compile time.
2. **Stack allocation** - In the stack allocation a stack is used to manage the run time storage.
3. **Heap allocation** - In heap allocation the heap is used to manage the dynamic memory allocation.

Let us discuss each of these strategies in detail.

7.3.1 Static Allocation

a. The size of data objects is known at compile time. The names of these objects are bound to storage at compile time only and such an allocation of data objects is done by static allocation.

b. The binding of name with the amount of storage allocated do not change at run time. Hence the name of this allocation is static allocation.

c. In static allocation the compiler can determine the amount of storage required by each data object. And therefore it becomes easy for a compiler to find the addresses of these data in the activation record.

d. At compile time compiler can fill the addresses at which the target code can find the data it operates on.

e. FORTRAN uses the static allocation strategy.

Limitations of static allocation

1. The static allocation can be done only if the size of data object is known at compile time.
2. The data structures can not be created dynamically. In the sense that, the static allocation can not manage the allocation of memory at run time.
3. Recursive procedures are not supported by this type of allocation.

7.3.2 Stack Allocation

a. Stack allocation strategy is a strategy in which the storage is organized as stack. This stack is also called **control stack**.

b. As activation begins the activation records are pushed onto the stack and on completion of this activation the corresponding activation records can be popped.

c. The locals are stored in the each activation record. Hence locals are bound to corresponding activation record on each fresh activation.

d. The data structures can be created dynamically for stack allocation.

Limitations of stack allocation

The memory addressing can be done using pointers and index registers. Hence this type of allocation is slower than static allocation.

7.3.3 Heap Allocation

a. If the values of non local variables must be retained even after the activation record then such a retaining is not possible by stack allocation. This limitation of

stack allocation is because of its Last In First Out nature. For retaining of such local variables heap allocation strategy is used.

b. The heap allocation allocates the continuous block of memory when required for storage of activation records or other data object. This allocated memory can be deallocated when activation ends. This deallocated (free) space can be further reused by heap manager.

c. The efficient heap management can be done by
   i) Creating a linked list for the free blocks and when any memory is deallocated that block of memory is appended in the linked list.
   ii) Allocate the most suitable block of memory from the linked list. i.e. use best fit technique for allocation of block.

7.3.4 Comparison between Static, Stack and Heap Allocation

Sr. No.	Static allocation	Stack allocation	Heap allocation
1.	Static allocation is done for all data objects at compile time.	In stack allocation, stack is used to manage runtime storage.	In heap allocation, heap is used to manage dynamic memory allocation.
2.	Data structures can not be created dynamically because in static allocation compiler can determine the amount of storage required by each data object.	Data structures and data objects can be created dynamically.	Data structures and data objects can be created dynamically.
3.	**Memory allocation :** The names of data objects are bound to storage at compile time.	**Memory allocation :** Using Last In First Out (LIFO) activation records and data objects are pushed onto the stack. The memory addressing can be done using index and registers.	**Memory allocation :** A contiguous block of memory from heap is allocated for activation record or data object. A linked list is maintained for free blocks.
4.	**Merits and limitations :** This allocation strategy is simple to implement but supports static allocation only. Similarly recursive procedures are not supported by static allocation strategy.	**Merits and limitations :** It supports dynamic memory allocation but it is slower than static allocation strategy. Supports recursive procedures but references to non local variables after activation record can not be retained.	**Merits and limitations :** Efficient memory management is done using linked list. The deallocated space can be reused. But since memory block is allocated using best fit, holes may get introduced in the memory.

Review Questions

1. *Explain stack allocation in brief* — **GTU : Winter-13, 14, 18, Marks 3**
2. *Write differences between stack and heap memory allocation.* — **GTU : Summer-18, Marks 4**
3. *Explain static storage allocation.* — **GTU : Summer-19, Marks 3**
4. *What are the limitations of static storage allocation.* — **GTU : Winter-20, Marks 3**

7.4 Storage Allocation Space

GTU : Winter-11,12,13,14,17,20, Summer-12,15,17, Marks 7

7.4.1 Activation Record

- The activation record is a block of memory used for managing information needed by a **single execution of a procedure.**
- FORTRAN uses the static data area to store the activation record where as in PASCAL and C the activation record is situated in stack area. The contents of activation record are as shown in the Fig. 7.4.1.
- Various fields of activation record are as follows -

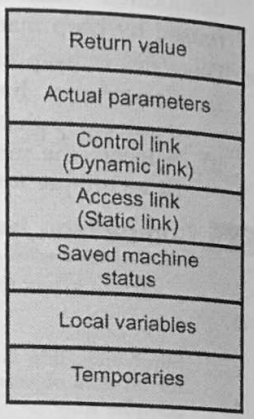


Fig. 7.4.1 Model of activation record

1. **Temporary values -** The temporary variables are needed during the evaluation of expressions. Such variables are stored in the temporary field of activation record.
2. **Local variables -** The local data is a data that is local to the execution of procedure is stored in this field of activation record.
3. **Saved machine registers -** This field holds the information regarding the status of machine just before the procedure is called. This field contains the machine registers and program counter.
4. **Control link -** This field is optional. It points to the activation record of the calling procedure. This link is also called dynamic link.
5. **Access link -** This field is also optional. It refers to the non local data in other activation record. This field is also called static link field.
6. **Actual parameters -** This field holds the information about the actual parameters. These actual parameters are passed to the called procedure.
7. **Return values -** This field is used to store the result of a function call.

- The size of each field of activation record is determined at the time when a procedure is called.

Example 7.4.1 *By taking the example of factorial program explain how activation record will look like for every recursive call in case of factorial (3).*

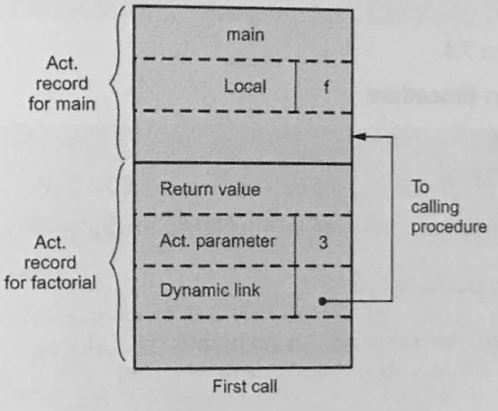
OR

Distinguish between the source text of a procedure and its activation at run time.

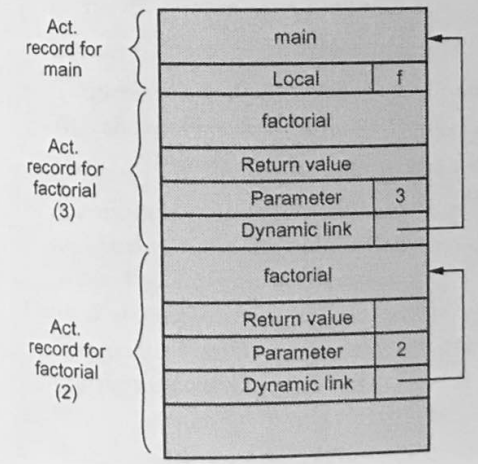
Solution :

```
main()
{
    int f;
    f = factorial (3);
}
int factorial (int n)
{
    if (n==1)
        return 1;
    else
        return (n*factorial (n-1));
}
```

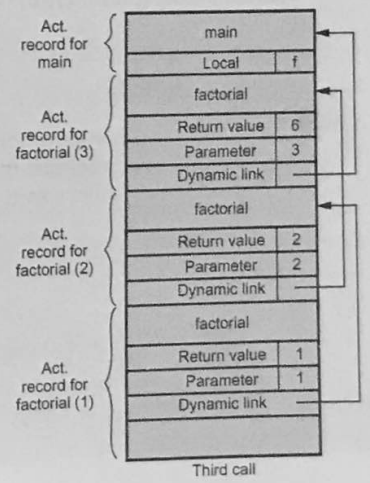
Step 1 :



Step 2 :



Step 3 :



Activation tree : Activation tree is a data structure used for specifying the flow of control among the procedures. In an activation tree -

i) Each node shows an activation of procedure.

ii) For showing activation of main program root is used.

iii) If lifetime of 'x' occurs before the lifetime of 'y' then x becomes left child of y.

iv) If control flows from x to y then 'x' is parent node of 'y'.

Example 7.4.2 *Explain in detail stack based allocation of space. Justify use of activation trees and activation records in stack based allocation. Write a simple program to implement a sort procedure. Draw the activation tree when the numbers 9, 8, 7, 6, 5, 4, 3, 2, 1 are sorted. What is the largest number of activation records that ever appear together on the stack ?*

Solution : Refer section 7.4.

Implementation of sort procedure -

```
int  Arr [10]
void readArray ()
{
      /* Reads the 9 integers into Arr[1] to Arr[9] */
}
int partition (int m, int n)
{
      /* Partitions the array Arr [m,n] into
            Arr [m ... p – 1] and Arr [p + 1 ... n]
      */
}
void quicksort (int m, int n)
{
            int i;
            if (m>n)
      {
                  i = partition (m,n);
                  quicksort (m, i–1);
                  quicksort (i + 1, n);
      }
}
main ()
{
      readArray ();
      a[0] = – 999
      quicksort (9,1);
}
```


Activation tree

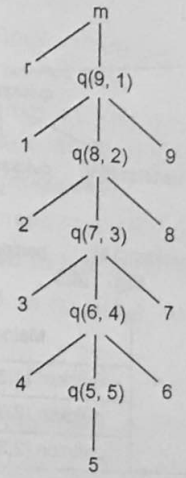


Fig. 7.4.2 Activation Tree for quick sort

Example 7.4.3 *Explain activation tree and control stack.* **GTU : Winter-11, Marks 4**

Solution : Activation tree :

- The activation tree is a graphical representation which describes the flow of control over the procedure call-chains.
- Activation tree shows the way control enters and leaves activations for one run of a program.
- Each call is represented by a node.
- The root of activation tree represents the activation of main procedure.
- If function A calls function B then node A can be represented as parent of node B. This shows that control flows from A to B.
- Node A is to the left of node B if a terminates before B.
- **For example :** In the program of quick sort, the procedures ReadArray, quicksrt and partition are the subroutines which are called. The activation tree for them is - (Refer Fig. 7.4.3 on next page)

Control stack :

- The control stack is a data structure used at run-time to keep track of live procedure activations.
- When the lifetime of a procedure begins then push a node onto the stack and POP it at the end of its lifetime.

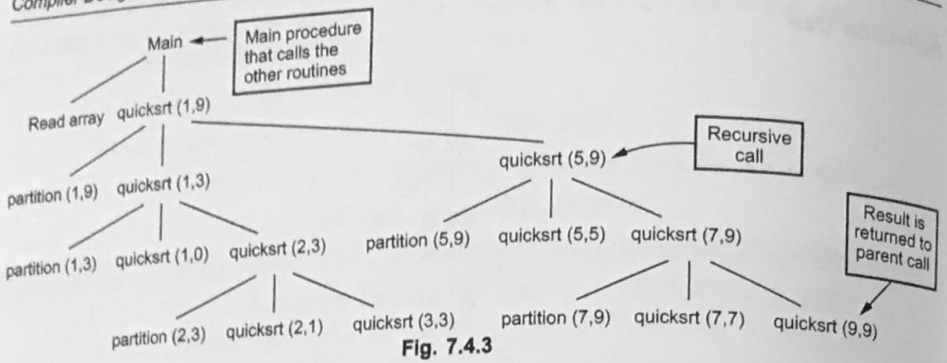


Fig. 7.4.3

- **For example :**

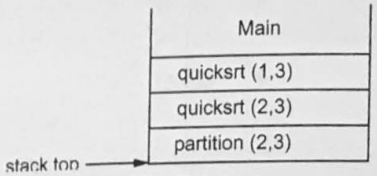


Fig. 7.4.4

Review Questions

1. *Explain the structure of an activation record with all its components.* **GTU : Summer-12, Marks 4**
2. *Explain activation record. How is task divided between calling and called program for stack updating ?* **GTU : Winter-12, Marks 7**
3. *Explain activation record organization in brief.* **GTU : Winter-13, 14, Marks 3**
4. *Explain the term "activation record" in detail.* **GTU : Summer-15, Marks 7**
5. *Explain activation record and activation tree in brief.* **GTU : Summer-17, Marks 7**
6. *What is activation record ? Explain stack allocation of activation records using example.* **GTU : Winter-17, Marks 7**
7. *What is activation tree ?* **GTU : Winter-17, Marks 4**
8. *What is activation record ?* **GTU : Winter-20, Marks 3**

7.5 Block Structure and Non Block Structure Storage Allocation

GTU : Winter-11, Summer-14,16, Marks 7

The **block** is a sequence of statements containing the local data declarations and enclosed within the delimiters.

For example :

```
{
 Declaration statements
 ...
}
```


The delimiters mark the beginning and end of the block. The blocks can be in nesting fashion that means block B2 completely can be inside the block B1.

The scope of declaration in a **block structured language** is given by most closely nested loop or static rule. It is as given below.

The declarations are visible at a program point are :

i) The declarations that are made locally in the procedure.

ii) The names of all enclosing procedures.

iii) The declarations of names made immediately within such procedures.

Local variable : A variable declared in block b is called local variable of block b.

Non local variable : A variable of an enclosing block that is accessible within block b is called non local variable of block b.

To understand the concept of scope let us see one example

Example 7.5.1 *Obtain the Local and Non Local variables from the following piece of code.*

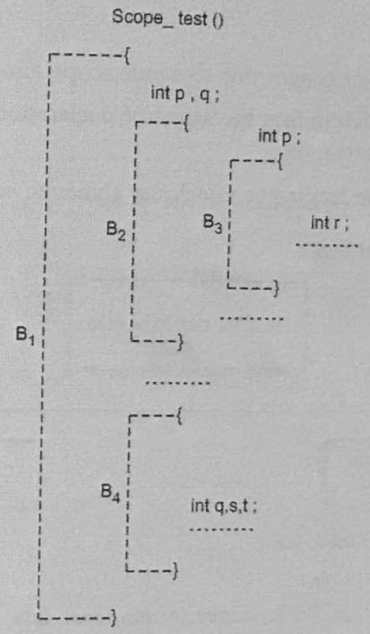


Fig. 7.5.1

Solution : To denote the scope of the variable we will use name of the variable followed by the block name.

Block	Local variable	Non Local variable
B1	pB1,qB1	-

B2	pB2	pB1,qB1
B3	rB3	pB2,qB1
B4	qB4,sB4,tB4	pB1,qB1

7.5.1 Access to Non Local Names

A procedure may sometimes refer to variables which are not local to it. Such variables are called as **non local variables**. For the non local names there are two types of scope rules that can be defined: **static and dynamic**

1. Static scope rule

The static scope rule is also called as **lexical scope**. In this type the scope is determined by examining the program text. PASCAL, C and ADA are the languages that use the static scope rule. These languages are also called as block structured languages.

2. Dynamic scope rule

For non block structured languages this dynamic scope allocation rules are used.

The dynamic scope rule determines the scope of declaration of the names at run time by considering the current activation.

LISP and SNOBOL are the languages which use dynamic scope rule.

7.5.1.1 Static Scope or Lexical Scope

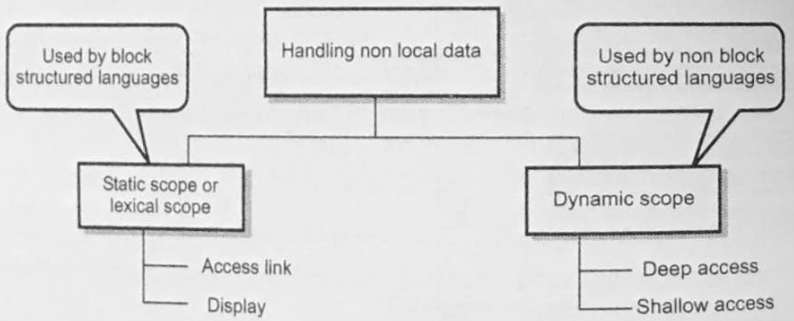


Fig. 7.5.2 Access to non local data

In this section we will first discuss the static scope rule. First of all let us understand few terminologies.

Block

The block is a sequence of statements containing the local data declarations and enclosed within the delimiters.

For example :

```
{
Declaration statements
...
}
```

The delimiters mark the beginning and end of the block. The blocks can be in nesting fashion that means block B2 completely can be inside the block B1.

The scope of declaration in a block structured language is given by most closely nested loop or static rule. It is as given below.

The declarations are visible at a program point are :

i) The declarations that are made locally in the procedure.

ii) The names of all enclosing procedures.

iii) The declarations of names made immediately within such procedures.

To understand the static scope consider one example.

Example 7.5.2 *Obtain the static scope of the declarations made in the following piece of code.*

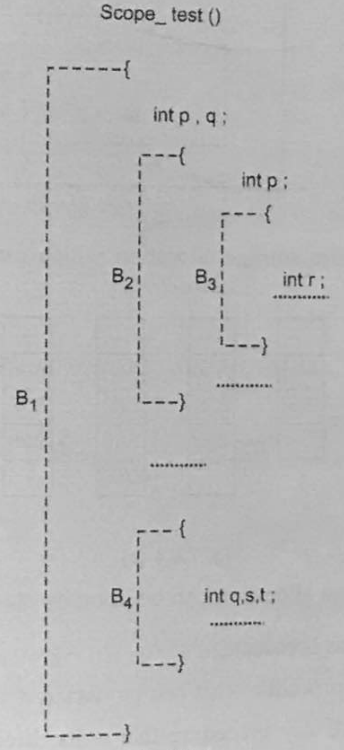
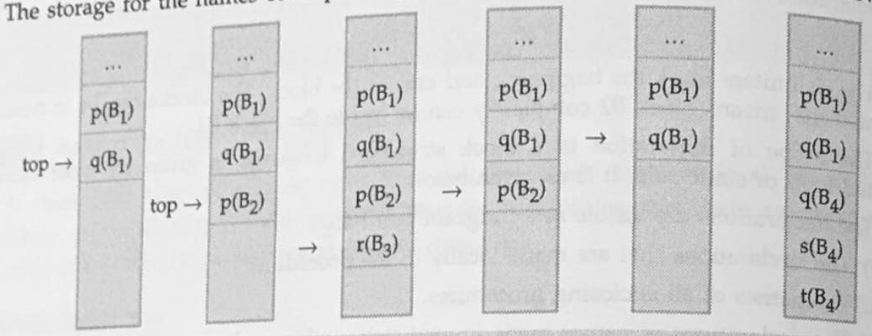


Fig. 7.5.3

Solution : The storage can be allocated for a complete procedure body at one time.

The storage for the names corresponding to particular block can be as shown below.



Example 7.5.3 *Explain the behaviour of stack, used for allocation of storage, for the execution of each statement of the block structured program :*

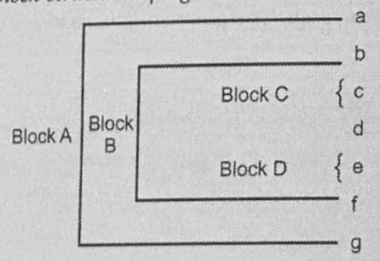


Fig. 7.5.4

Solution : The block structure storage allocation which can be done using stack is as given below.

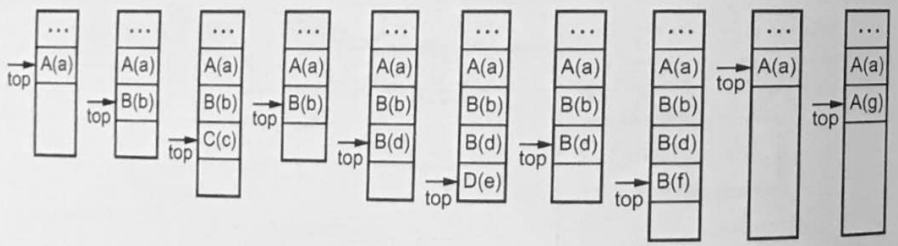


Fig. 7.5.4 (a)

Thus block structure storage allocation can be done by stack.

7.5.1.2 Lexical Scope for Nested Procedure

- **Nested procedure** is a procedure that can be declared within another procedure.
- A procedure pi, can call any procedure that is its direct ancestor or older siblings of its direct ancestor.

- The nested procedures can be as shown below.
  Procedure main
  procedure P1
  procedure P2
  procedure P3
  procedure P4
- **Nesting depth** - Nesting depth of a procedure is used to implement lexical scope. The nesting depth can be calculated as follows.
  i) The nesting depth of main program is 1.
  ii) Add 1 to depth each time when a new procedure begins.
  iii) Subtract 1 from depth each time when you exit from a nested procedure.
  iv) The variable declared in specific procedure is associated with nesting depth.

For example : Consider the following piece of code

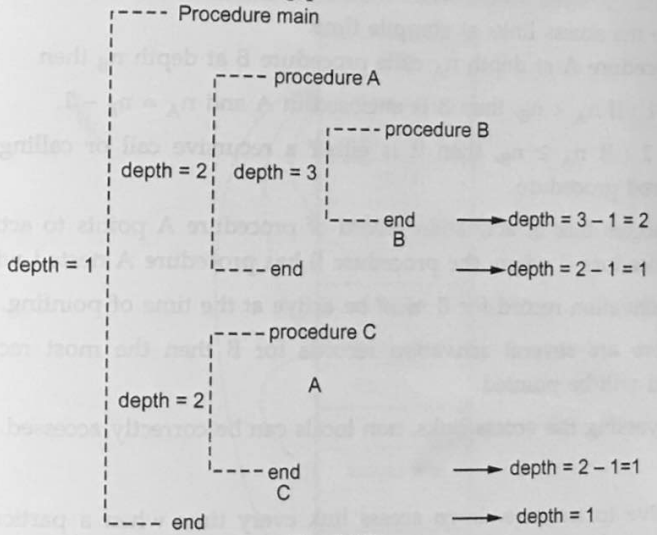


Fig. 7.5.5

- The lexical scope can be implemented using access link and displays.

1. Access link

The implementation of lexical scope can be obtained by using pointer to each activation record. These pointers are called access links. If procedure p is nested within a procedure q then access link of p points to access link of most recent activation record of procedure q.

For example : Consider following piece of code and the run time stack during execution of the program.

```
program test;
    var a : int;
        procedure A;
            var d : int;
                begin a : = 1, end;
        procedure B(i : int);
            var b : int;
                procedure C;
                    var k : int;
                    begin A; end;
                        begin
                        if (i<>0) then B(i–1)
                        else C;
                        end
                begin B(1); end;
```

- To set up the access links at compile time.
  i) If procedure A at depth n_A calls procedure B at depth n_B then

  Case 1 : If $n_A < n_B$, then B is enclosed in A and $n_A = n_B - 1$.

  Case 2 : If $n_A \geq n_B$, then it is either a recursive call or calling a previously declared procedure.

  ii) The access link of activation record of procedure A points to activation record of procedure B where the procedure B has procedure A nested within it.

  iii) The activation record for B must be active at the time of pointing.

  iv) If there are several activation records for B then the most recent activation record will be pointed.

Thus by traversing the access links, non locals can be correctly accessed.

2. Displays

It is expensive to traverse down access link every time when a particular non local variable is accessed. To speed up the access to non locals can be achieved by maintaining an array of pointers called display.

In display -

a) An array of pointers to activation record is maintained.

b) Array is indexed by nesting level.

c) The pointers point to only accessible activation record.

d) The display changes when a new activation occurs and it must be reset when control returns from the new activation.

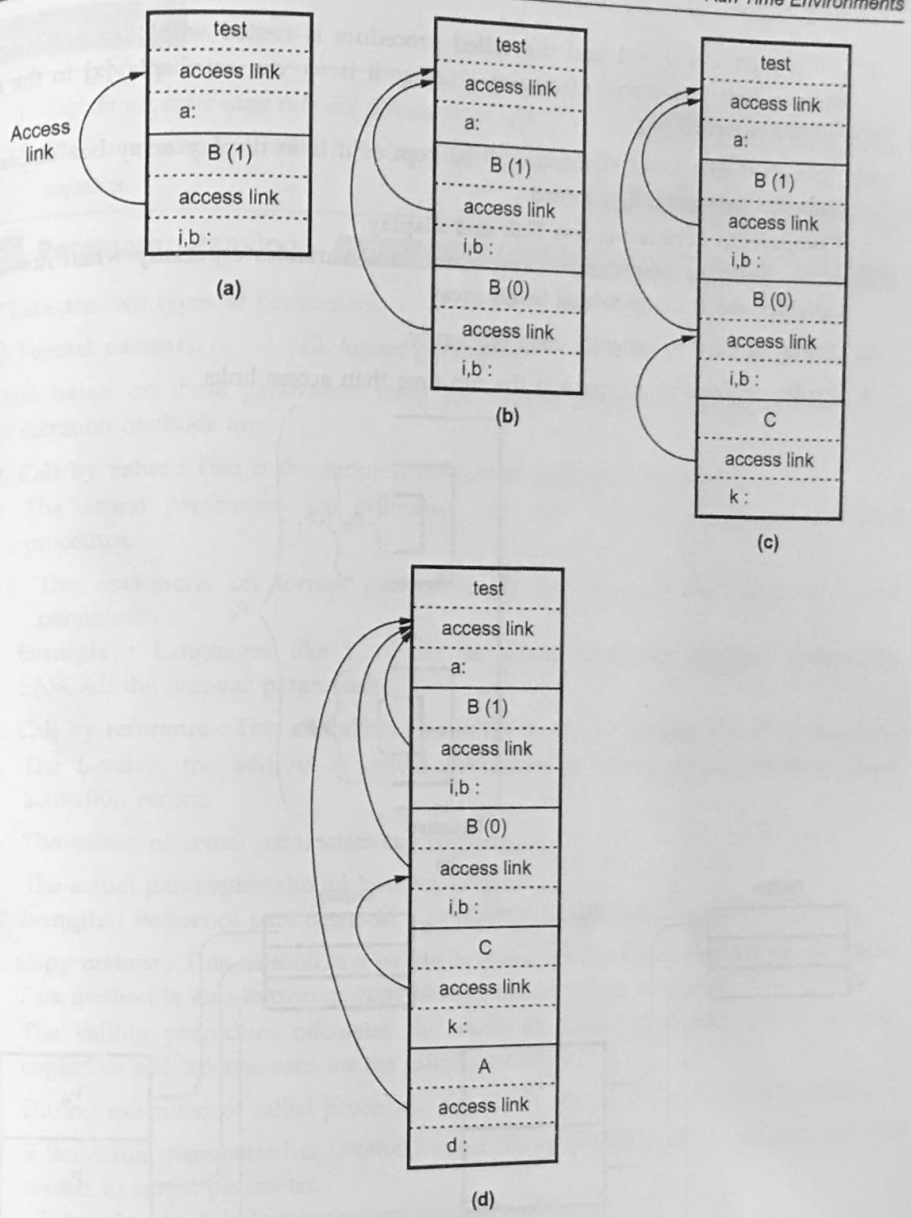


Fig. 7.5.6 Access link

For example

Consider the scope of procedures as shown below

Nesting depth y calls a procedure at depth x then

i) $y < x$ then $x = y + 1$ and the called procedure is nested within caller. Then by keeping first y elements of display array as it is we can set display[x] to the new activation record.

ii) $y \geq x$ then first $x - 1$ elements will be kept as it is in display array but display[x] points to new activation record.

- **Comparison between access link and display**

1. Access link take more time to access non local variables especially when non local variables are at many nested levels away.
2. Display is used to generate code efficiently.
3. Display requires more space at the run time than access links.

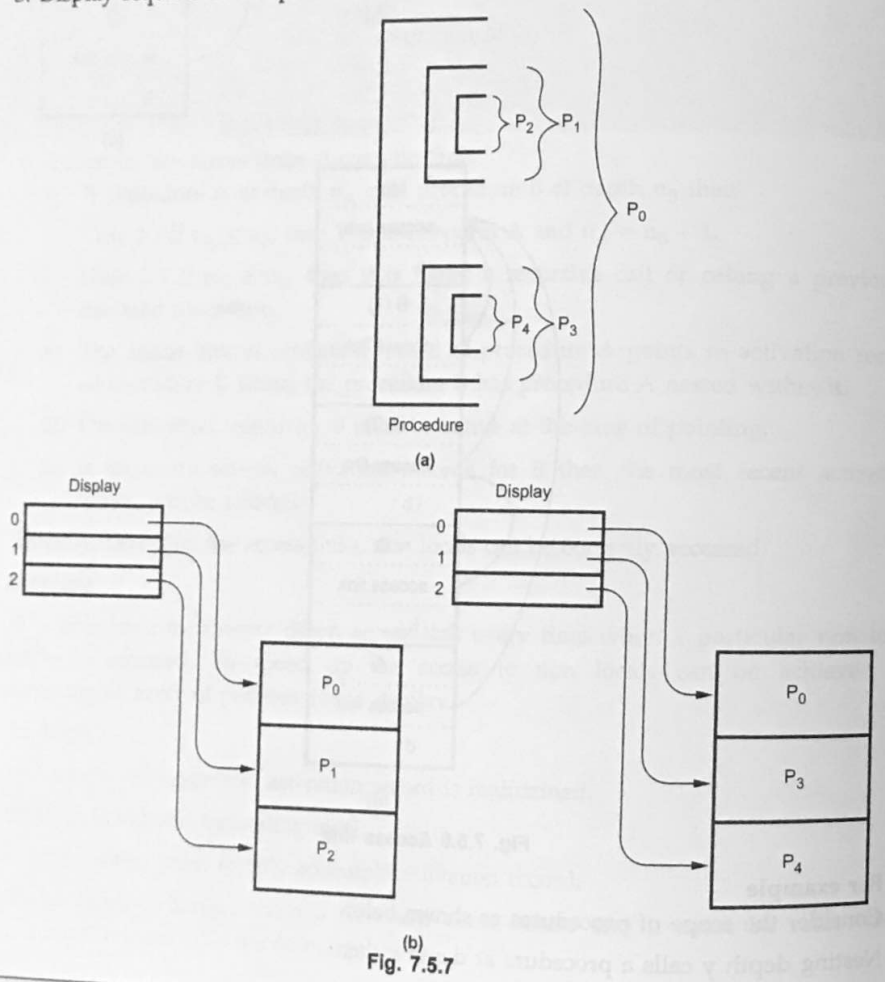


Fig. 7.5.7

Review Questions

1. *Explain the static scope rule and dynamic scope rule.* **GTU : Winter-11, Marks 3**
2. *What is an activation record ? Explain how they are used to access various local and global variables.* **GTU : Summer-14, 16, Marks 7**

7.6 Parameter Passing

GTU : Winter-12,16,17,18,20, Summer-18,19,20, Marks 7

There are two types of parameters.

i) Formal parameters **ii) Actual parameters**

And based on these parameters there are various parameter passing methods, the most common methods are,

1. **Call by value :** This is the simplest method of parameter passing.
   - The actual parameters are evaluated and their r-values are passed to called procedure.
   - The operations on formal parameters do not changes the values of actual parameter.

   Example : Languages like C, C++ use actual parameter passing method. In PASCAL the non-var parameters.
2. **Call by reference** : This method is also called as call by address or call by location.
   - The L-value, the address of actual parameter is passed to the called routines activation record.
   - The values of actual parameters can be changed.
   - The actual parameters should have an L-value.

   Example : Reference parameters in C++, PASCAL'S var parameters.
3. **Copy restore** : This method is a hybrid between call by value and call by reference. This method is also known as copy-in-copy-out or values result.
   - The calling procedure calculates the value of actual parameter and it is then copied to activation record for the called procedure.
   - During execution of called procedure, the actual parameters value is not affected.
   - If the actual parameter has L-value then at return the value of formal parameter is copied to actual parameter.

   Example : In Ada this parameter passing method is used.
4. **Call by name :** This is less popular method of parameter passing.
   - Procedure is treated like macro. The procedure body is substituted for call in caller with actual parameters substituted for formals.

- The actual parameters can be surrounded by parenthesis to preserve their integrity.
- The local names of called procedure and names of calling procedure are distinct.
- **Example :** ALGOL uses call by name method.

For example : Consider the following piece of code along with the output.

```
procedure exchange (m,n:integer);
    var t : integer;
    begin
        t := m;
        m := n;
        n := t;
    end;


...
i := 1
a[i] := 50 { a:array [1....10] of integer}
print (i,a[i]);
exchange{i,a[i]);
print (i,a[1]);
```

The output for all the above discussed method is

Call by value		Call by reference		Copy restore		Call by name	
1	50	1	50	1	50	1	50
1	50	50	1	50	1	Error	

The error occurs because

t := i ∴ t := 1

i := a[i] ∴ i := 50 as a[i] = 50

a[i] := t ∴ a[50] = t = 1 then index is out of bounds.

Review Questions

1. *Explain various parameter passing methods.* **GTU : Winter-12, Summer-20, Marks 7.**
2. *Explain the following parameter passing methods.*
   1. Call by value 2. Call by reference 3. Copy restore 4. Call by name.
   GTU : Winter-16,17,18, Summer-18,19, Marks 4, Summer-20, Winter-20, Marks 7

7.7 Heap Management

Heap is the portion of the memory that can remain allocated for indefinite time. The memory created using dynamic allocation technique makes use of heap memory. For example C++ or Java programmer can make use of **new** to allocate memory or **delete** to deallocate the memory from heap.

7.7.1 Memory Manager

- The **task of memory manager** is to keep track of all the free space that can be available in the heap. Following are the two important functions of memory manager -

1. **Allocation :** When a program requests for allocation of memory for some object or variable then the memory manager allocates a chunk of memory to the object/variable. Sometimes the memory manager makes use of the free space available in the heap memory in order to satisfy the request. If still the required free space is not available then it increases the size of heap storage by getting some bytes of memory from virtual memory of operating system.
2. **Deallocation :** The memory manager returns the deallocated memory to the pool of free storage maintained in the heap memory. This free space is used by the memory manager to satisfy the request for memory allocation.

- Following are some **desired properties of memory manager -**

1. **Space efficiency :** The memory manager must minimize the requirement of total heap space required by the program. For that matter, the larger programs are allowed to run in fixed virtual address space. The reduction in **fragmentation** also helps in achieving the space efficiency.
2. **Program efficiency :** By using the heap memory appropriately the program efficiency can be achieved. The object that are required more frequently must be placed in the accessible region, this will help to run the program faster. The principle of locality is used to achieve the program efficiency.
3. **Low overhead :** Memory allocation and deallocation are the most frequently performed operations in the program. If these operations are executed efficiently then it reduces the overhead in program execution.

7.7.2 Memory Hierarchy

- Compiler optimization is very much dependent upon the memory management technique.
- The efficiency of program is determined by - 1) Number of instructions getting executed and 2) The amount time required by each instruction for execution. The

time taken by the instruction to execute greatly depends upon how much time it takes to access different parts of the memory.

- The significant difference in memory access time is due to **limitations of hardware technologies**. Quite often there are small storage with fast access and large storage with slow access. To make use of the efficient memory access all the modern computers arrange in memory in **hierarchy.** Fig. 7.7.1 represents the memory hierarchy.

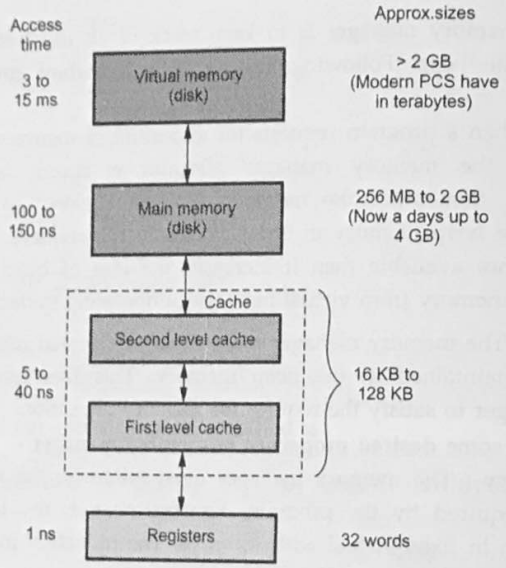


Fig. 7.7.1 Memory hierarchy

- The processors has several registers which can be controlled by the software. Then the levels of cache memory come. It is there in kilobytes to several megabytes.The next level has the physical main memory which is made up of hundreds of megabytes to gigabytes of dynamic RAM. Finally the virtual memory which has several gigabytes of memory.
- During the memory access, the data is looked in the lower level of memory and if it is not found there then the next level is accessed and so on. This simplifies the programming task and the same program can work effectively on various machines with different memory configurations.

7.7.3 Locality in Programs

- Many programs spent lot of time in executing small amount of code by accessing very little amount of memory. Thus exhibits the high degree of **locality.**

- Programs has two types of localities - temporal locality and spatial locality.
- The program has **temporal locality** if the locations it access are likely to accessed again within in short period of time. The program has the **spatial locality** if the locations close to the locations it access are likely to be accessed within a short time.
- Following are some reasons why the programs spent most of its time in executing very small amount of code -

1. Most of the programs contain the instructions that are never executed. Programs make use of libraries that use small amount of provided functionalities. Many times there are changes in the requirements of the systems or the legacy systems get evolved.
2. Typically, small amount of code is often executed. For example the code for performing input and output operations or exceptional cases.
3. Many programs spend a lot of time in executing the inner loops or the recursive calls.

- The **locality** makes use of memory hierarchy for increasing the efficiency in program execution. For instance by placing the most common data in fast but small storage and less frequently needed data in slow and large storage the memory access time can be reduced.
- We can improve the temporal and special locality of data access in the program by changing the order of computation. For example we can bring the data from slow level(main memory) to fast level(disk) and perform the computation on this data. This makes the execution faster.

7.7.4 Fragmentation

- Due to frequent memory allocation and deallocation the heap memory becomes fragmented. That is the memory contains many free slots as well as many chunks of allocated memory. The **free slots** in the memory are called **holes.**
- When a memory allocation request comes, the memory manager places the request into large enough hole. This will satisfy the request but at the same time it creates one more small hole.

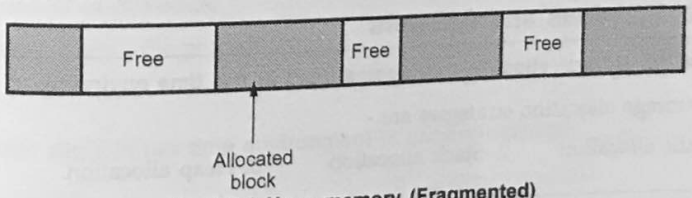


Fig. 7.7.2 Heap memory (Fragmented)

- With the deallocation request, the freed space is returned to the pool of free space. If memory allocation of deallocation is not done carefully many small noncontiguous holes in the memory may get created. This leads to fragmentation.

Strategies to Reduce Fragmentations

1) Best and Next Fit

- In the **best fit** strategy, the smallest hole which is large enough for the object that demands for allocation of memory is used. In this method the memory manager searches the entire list of free slots and finds the hole which is the best possible that can be used for allocating the memory for the target object. Although this strategy produces best allocation space, this method is less efficient as it requires more time in searching the appropriate hole and create many small holes.
- Another strategy is the **First fit.** In this strategy, the free list of holes is simply scanned to find the large enough hole to hold the object. This strategy is more efficient than the best fit. However due to this strategy the small holes tend to accumulate at the beginning of the free list. The problem of small holes accumulating is solved with **next fit strategy** which starts each search where the last one left off. It wraps around to the beginning when the end of the list is reached

2) Managing and Coalescing Free Space

Due allocation of memory randomly many holes are created in the memory. Calescing is the technique by which many free spaces are combined together to form a large chunk of free space. The advantage of this strategy is that the fragmentation gets reduced and a large free chunk is available for allocating the space for the object of large size(since many small chunks can not hold the large object)

Review Questions

1. *Explain in detail, the strategy for reducing fragmentation in heap memory.*
2. *Explain the desirable properties of memory manager.*

7.8 Short Questions and Answers

Q.1 Enlist the storage allocation strategies used in run time environment.

Ans. : The storage allocation strategies are -

1. Static allocation 2. Stack allocation 3. Heap allocation.

Q.2 What is the purpose of control stack used in run time storage organization ?

Ans. : The control stack is used to manage the active procedures. Managing the active procedures means that when a call occurs then the execution of activation is interrupted and information about the status of the control flow is saved. When the control returns from the call this suspended activation is resumed after storing the values of relevant registers.

Q.3 What are various ways to pass the parameters to the function ?

Ans. : Various ways to pass the parameters to the function are -

1. **Call by value :** In this method actual value of the parameter is passed.
2. **Call by Reference :** In this method the address of actual parameter is passed.
3. **Copy-restore :** This method is hybrid between call by value and call by reference.
4. **Call by name** : In this method procedure is treated like Macro.

Q.4 Suggest a suitable approach for computing the hash function.

Ans. : Using hash function we should obtain exact locations of name in symbol table. The hash function should result in uniform distribution of names in symbol table. The hash function should be such that there will be minimum number of collisions. Collisions is such a situation where hash function results in same location for storing the names.

Q.5 What are the limitations of static allocation ?

Ans. : 1. The static allocation can be done only if the size of data object is known at compile time.

2. The data structures can not be created dynamically. In other words, the static allocation can not manage the allocation of memory at run time.
3. Recursive procedures are not supported by this type of allocation.

7.9 Multiple Choice Questions

Q.1 Activation of procedure refers to _____ .

a. invoking the procedure
b. execution of procedure
c. returning from the procedure
d. all of the above

Q.2 Control stack in run time environment is used to manage _____ .

a. data object
b. active procedures
c. target code
d. none of the above

Q.3 Compiler places target code at _____ .

a lower end of memory
b upper end of memory
c anywhere in the memory depending upon the code
d none of the above

Q.4 Recursive procedures are not supported by _____ .

a stack allocation
b heap allocation
c static allocation
d code area

Q.5 The access link used in activation record is a_____.

a static link
b dynamic link
c control link
d none of the above

Q.6 The control link used in activation record is a_____.

a static link
b dynamic link
c control link
d none of the above

Q.7 In C the activation record is stored in_____.

a code area
b static area
c stack area
d heap area

Q.8 In FORTRAN the activation record is stored in_____.

a code area
b static area
c stack area
d heap area

Q.9 In ability of handling recursive procedure handling is the limitation of_____.

a static area
b stack area
c heap area
d none of the above

Q.10 The dangling reference occurs when _____.

a reference is assigned to some address containing data
b reference is assigned to something which is deallocated.
c referece is assigned to some address containing another address.
d none of the above

Q.11 _____ performs the garbage collection.

a C
bPASCAL
c LISP
dJAVA

Answer Keys for Multiple Choice Questions

Q.1	b	Q.2	b
Q.3	a	Q.4	c
Q.5	a	Q.6	b
Q.7	c	Q.8	b
Q.9	a	Q.10	b
Q.11	c, d		

❑❑❑

Notes

8 Code Generation and Optimization

Syllabus

Issues in the Design of a Code Generator, The Target Language, Addresses in the Target Code, Basic Blocks and Flow Graphs, Optimization of Basic Blocks, A Simple Code Generator, Machine dependent optimization, Machine independent optimization Error detection of recovery.

Contents

8.1 Code Generation

Code generation is the final phase in the process of compilation. It takes intermediate code as an input and generates target machine code as output. The position of code generator in compilation process is illustrated by following Fig. 8.1.1.

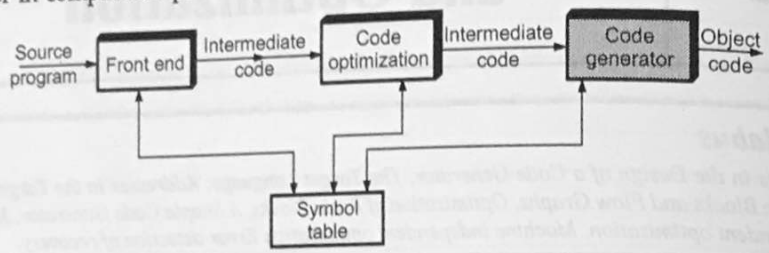


Fig. 8.1.1 Position of code generator in compiler

8.2 Issues in the Design of a Code Generator

GTU : Winter-11,12,13,14,15,16,17,18,19,20,Summer-12,18, Marks 7

Let us discuss some common issues in design of code generator.

• Input to the code generator

The code generation phase takes intermediate code as input. This intermediate code along with the symbol table information is used to determine the runtime addresses of the data objects. These data objects are denoted by the names in the intermediate representation. The intermediate code may be in any form such as three address code, quadruple, posix notation or it may be represented using graphical representations such as syntax trees or DAGs. The intermediate code generated by the front end should be such that the target machine can easily manipulate it, in order to generate appropriate target machine code. In the front end of compiler necessary type checking and type conversion needs to be done. The detection of the semantic errors should be done before submitting the input to the code generator. The code generation phase requires the complete error free intermediate code as input.

• Target programs

The output of code generator is target code. Typically, the target code comes in three forms such as : absolute machine language, relocatable machine language and assembly language.

The advantage of producing target code in absolute machine form is that it can be placed directly at the fixed memory location and then can be executed immediately. The benefit of such target code is that small programs can be quickly compiled.

For example : The WATFIV and PL/C are the compilers which produce the absolute code as output.

The advantage of producing the relocatable machine code as output is that the subroutines can be compiled separately. Relocatable object modules can be linked together and loaded for execution by a linking loader. This offers great flexibility of compiling the subroutines separately. If the target machine can not handle the reloacation automatically then the compiler must provide explicit relocation information to the loader in order to link the separately compiled subroutine segments.

The advantage of producing assembly code as output makes the code generation process easier. The symbolic instructions and Macro facilities of assembler can be used to generate the code. It is advantageous to have assembly language as output for the machines with small memory.

However, out of these three forms of output target code it is always preferable to have relocatable machine code as target code.

• Memory management

Both the front end and code generator performs the task of mapping the names in the source program to addresses to the data objects in run time memory. The names used in the three address code refer to the entries in the symbol table. The type in a declaration statement determines the amount of storage(memory) needed to store the declared identifier. Thus using the symbol table information about memory requirements code generator determines the addresses in the target code.

Similarly if the three address code contains the labels then those labels can be converted into equivalent memory addresses. For instance if a reference to *'goto j'* is encountered in three address code then appropriate jump instruction can be generated by computing the memory address for label j.

• Instruction selection

The uniformity and completeness of instruction set is an important factor for the code generator. The selection of instruction depends upon the instruction set of target machine. The speed of instruction and machine idioms are two important factors in selection of instructions. If we do not consider the efficiency of target code then the instruction selection becomes a straightforward task.

For each type of three address code the code skeleton can be prepared which ultimately gives the target code for the corresponding construct.

For example x:=a+b then the code sequence that can be generated as

```
MOV a,R0  /* loads the a to register R0 */
ADD b,R0  /* performs the addition of b to R0 */
```

```
MOV R0,x /* stores the contents of register R0 to x */
```

In the above example the code is generated line by line. Such a line by line code generation process generates the poor code because the redundancies can be achieved by subsequent lines and those redundancies can not be considered in the process of line by line code generation.

For example

```
x:= y+z
a:=x+t
```

The code for the above statements can be generated as follows :

```
MOV y, R0
ADD z, R0
MOV R0, x
MOV x,R0
ADD t,R0
MOV R0,a
```

The above generated code is a poor code because MOV R0,a is not used and statement MOV a,R0 is redundant. Hence the efficient code can be

```
MOV y, R0
ADD z, R0
ADD t,R0
MOV R0,a
```

The quality of generated code is decided by its speed and size. Simply line by line translation of three address code into target code leads to correct code but it can generate unacceptably non efficient target code.

- **Register allocation**

If the instruction contains register operands then such a use becomes shorter and faster than that of using operands in the memory. Hence while generating a good code efficient utilization of register is an important factor. There are two important activities done while using registers.

1. **Register allocation -** During register allocation,select appropriate set of variables that will reside in registers.
2. **Register assignment -** During register assignment, pick up the specific register in which corresponding variable will reside.

Obtaining the optimal (minimum) assignment of registers to variable is difficult.

Certain machines require register pairs such as even odd numbered registered for some operands and results.

For example in IBM systems integer multiplication requires register pair.

Consider the three address code

```
t1:=a+b
t1:= t1*c
t1:= t1/d
```

The efficient machine code sequence will be

```
MOV a,R0
ADD b,R0
MUL c,R0
DIV d,R0
MOV R0,t1
```

- **Choice of evaluation order**

The evaluation order is an important factor in generating an efficient target code. Some orders require less number of registers to hold the intermediate results than the others. Picking up the best order is one of the difficulty in code generation. Mostly, we can avoid this problem by referring the order in which the three address code is generated by semantic actions.

- **Approaches to code generation**

The most important factor for a code generation is that it should produce the correct code. With this approach of code generation various algorithms for generating code are designed.

Review Questions

1. *Write the generic issues in the design of code generators*
   GTU : Winter-11,13,16, Marks 7
2. *Discuss the factors affecting the target code generation.* **GTU : May-12, Marks 3**
3. *Describe code generator design issues.*
   GTU : Winter-12,17,18,19,20, Marks 4, Winter-14,15, Summer-18, Marks 7

8.3 The Target Language

For designing the good code generator it is necessary to have prior knowledge of target machine and instruction set used for this target machine.

In this chapter we will assume that the target machine code is a register machine like minicomputers. Specifically following assumptions are made for code generation.

- We will assume that in the target computer addresses are given in bytes and four bytes form a word.
- There are n general purpose registers R0,R1,....Rn–1.

- The two address instruction is of the form.

op source destination

where op is an opcode and source and destination are data fields.

For instance :

MOV - moves from source to destination.

ADD - add source to destination.

SUB - substracts source from destination.

The source and destination are specified by registers and memory locations

- The addressing modes used are as follows

Addressing mode	Form	Address	Added cost
absolute	M	M	1
register	R	R	0
indexed	c(R)	c + contents(R)	1
indirect register	*R	contents(R)	0
indirect indexed	*c(R)	contents(c+contents(R))	1
literal	#c	c	1

If we have absolute or resister addressing mode we can use M or register for source or destination.

For example, the instruction MOV R1,M stores the contents of register R_1 into memory location M.

For the indexed addressing mode the address offset c from the value of register R0 can be written as

MOV 7(R1), M

Means it strores the value contents (7+ contents(R1) to the memory location M.

The last two addressing modes represent the indirect addresses indicated by *.

For the instruction MOV *7(R0),M

It stores the value contents(contents (7+ contents(R0)) to the memory location M.

In the literal addressing mode the source becomes constant.

For example MOV #5,R0

By this instruction we can store the constant 5 into register R0.

8.3.1 Cost of the Instruction

The instruction cost can be computed as one plus cost associated with the source and destination addressing modes given by "added cost"

Instruction	Cost	Interpretation
MOV R0,R1	1	Cost of register mode +1=0+1=1
MOV R1,M	2	Use of memory variable +1=1+1=2
SUB 5(R0),*10(R1)	3	Use of first constant +use of second constant +1 =3

Example 8.3.1 *Compute the cost of following set of instructions.*

```
MOV a,R0
ADD b,R0
MOV R0,c
```

Solution : The cost of this set is 6 because

```
MOV a,R0          2
ADD b,R0          2
MOV R0,c          2
                  ---
Total cost  =     6
```

Example 8.3.2 *Compute the cost of following set of instructions*

```
MOV *R1,*R0
ADD *R2,*R0
```

Solution : The cost of this set is 2 because

```
MOV *R1,*R0     1
ADD *R2,*R0     1
                ---
Total cost  =   2
```

Example 8.3.3 *Consider the following code sequence*

i) MOV B, R0
ADD C, R0
MOV R0, A
ii) MOV B, A
ADD C, A

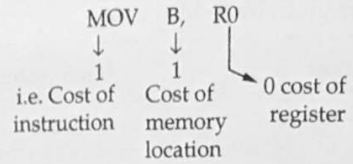
Calculate cost of above instructions.

Solution : i) MOV B, R0

ADD C, R0

MOV R0, A

For instruction



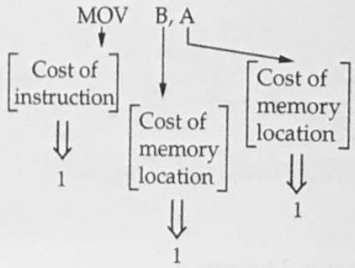
∴ MOV B, R0 → cost = 1 + 1 + 0 = 2

ADD C, R0 → cost = 1 + 1 + 0 = 2

MOV R0, A → cost = 1 + 0 + 1 = 2

Total cost = 6

ii)



∴ MOV B, A → cost = 1 + 1 + 1 = 3

ADD C, A → cost = 1 + 1 + 1 = 3

Total cost = 6

8.4 Basic Blocks and Flow Graphs

GTU : Winter-13, Summer-19, Marks 7

The basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching.

An example of the basic block is as shown below.

```
t1 := a*5
t2 := t1+7
t3 := t2-5
t4 := t1+t3
t5 := t2+b
```


8.4.1 Some Terminologies used in Basic Blocks

- **Define and use** - The three address statement a := b+c is said to define a and to use b and c.
- **Live and dead** - The name in the basic block is said to be live at a given point if its value is used after that point in the program. And the name(variable) in the basic block is said to be dead at a given point if its value is never used after that point in the program.

8.4.2 Algorithm for Partitioning into Blocks

Any given program can be partitioned into basic blocks by using following algorithm. We assume that an intermediate code is already generated for the given program.

1. First determine the **leaders** by using following rules.
   a) The **first statement** is a leader.
   b) Any target statement of conditional or unconditional **goto** is a leader.
   c) Any statement that immediately **follow a goto** or unconditional goto is a leader.
2. The basic block is formed starting at the leader statement and ending just before the next leader statement appearing.

Example 8.4.1 *Consider the following program code for computing dot product of two vectors a and b of length 10 and partition it into basic blocks.*

```
prod = 0;
i = 1;
do
{
 prod = prod + a[i] * b[i];
 i=i+1;
}while(i<=10);
```

Solution : First we will write the equivalent three address code for the above program.

1. prod := 0
2. i := 1
3. t_1 := 4 * i
4. t_2 := a[t_1] /* computation of a[i] */
5. t_3 := 4 * i
6. t_4 := b[t_3] /* computation of b[i] */
7. t_5 := t_2 * t_4
8. t_6 := prod+t_5

9. prod := t_6
10. t_7 := i+1
11. i := t_7
12. if i< = 10 goto (3)

According to the algorithm

Statement 1 is a leader by rule 1(a).

Statement 3 is also leader by rule 1(b).

Hence, statement 1 and 2 form the basic block. Similarly statement 3 to 12 form another basic block.

Block 1

1. prod := 0
2. i := 1

Block 2

3. t_1 := 4 * i
4. t_2 := a[t_1]
5. t_3 := 4 * i
6. t_4 := b[t_3]
7. t_5 := t_2 * t_4
8. t_6 := prod+t_5
9. prod := t_6
10. t_7 := i + 1
11. i := t_7
12. if i<=10 goto (3)

8.4.3 Flow Graph

A flow graph is a directed graph in which the flow control information is added to the basic blocks.

- The nodes to the flow graph are represented by **basic blocks**.
- The block whose leader is the first statement is called **initial block.**
- There is a directed edge from block B_1 to block B_2 if B_2 immediately follows B_1 in the given sequence. We can say that B_1 is a predecessor of B_2.

For example, consider the three address code as

1. prod := 0
2. i := 1
3. t_1 := 4 * i
4. t_2 := a[t_1] /* computation of a[i] */
5. t_3 := 4 * i
6. t_4 := b[t_3] /* computation of b[i] */
7. t_5 := t_2 * t_4
8. t_6 := prod+t_5
9. prod := t_6
10. t_7 := i+1
11. i := t_7
12. if i<=10 goto (3)

The flow graph for the above code can be drawn as follows.

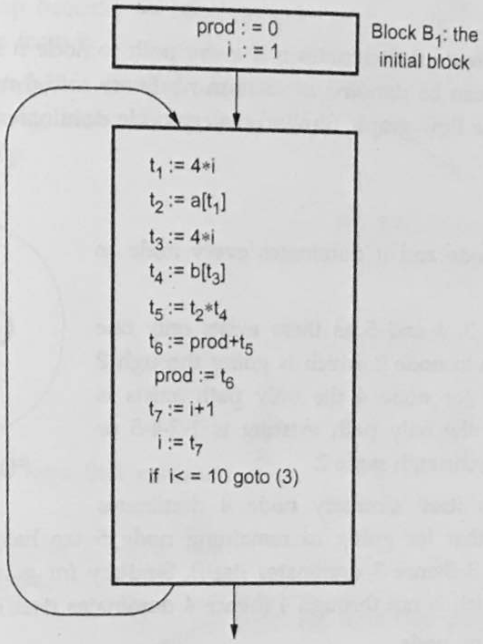


Fig. 8.4.1 Flow graph

In this flow graph block B_1 is a initial block.

Loop

Loop is a collection of nodes in the flow graph such that,

i) All such nodes are strongly connected. That means always there is a path from any node to any other node within that loop.

ii) The collection of nodes has unique entry. That means there is only one path from a node outside the loop to the node inside the loop.

iii) The loop that contains no other loop is called inner loop.

Review Questions

1. *Write down the algorithm for basic blocks.* **GTU : Winter-13, Marks 7**
2. *Define basic block with simple example.* **GTU : Summer-19, Marks 3**

8.5 Loops in Flow Graph

GTU : Summer-17, Marks 7

Let us get introduced with some common terminologies being used for loops in flow graph.

1. Dominators

In a flow graph, a node d dominates n if every path to node n from initial node goes through d only. This can be denoted as 'd dom n'. Every initial node dominates all the remaining nodes in the flow graph. Similarly every node dominates itself.

For example :

In the flow graph,

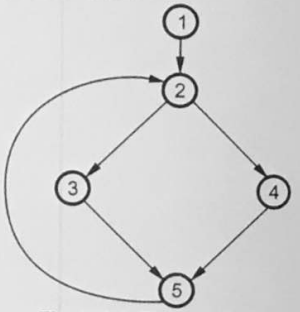


Fig. 8.5.1 Flow graph

Node 1 is initial node and it dominates every node as it is a initial node.

Node 2 dominates 3, 4 and 5 as there exists only one path from initial node to node 2 which is going through 2 (it is 1-2-3). Similarly for node 4 the only path exists is 1-2-4 and for node 5 the only path existing is 1-2-4-5 or 1-2-3-5 which is going through node 2.

Node 3 dominates itself similarly node 4 dominates itself. The reason is that for going to remaining node 5 we have another path 1-4-5 which is not through 3 (hence 3 dominates itself). Similary for going to node 5 there is another path 1-3-5 which is not through 4 (hence 4 dominates itself).

Node 5 dominates no node.

2. Natural loops

Loop in a flow graph can be denoted by n → d such that d dom n. These edges are called back edges and for a loop there can be more than one back edges. If there is p → q then q is a head and p is a tail. And head dominates tail.

For example :

The loops in the above graph can be denoted by 4 → 1 i.e. 1 dom 4. Similarly 5 → 4 i.e. 4 dom 5.

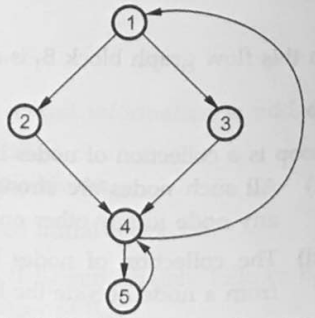


Fig. 8.5.2 Flow graph with loops

The natural loop can be defined by a back edge n → d such that there exists a collection of all the nodes that can reach to n without going through d and at the same time d also can be added to this collection.

For example :

6→1 is a natural loop because we can reach to all the remaining nodes from 6.

By observing all the predecessors of node 6 we can obtain a natural loop. Hence 2-3-4-5-6-1 is a collection in natural loop.

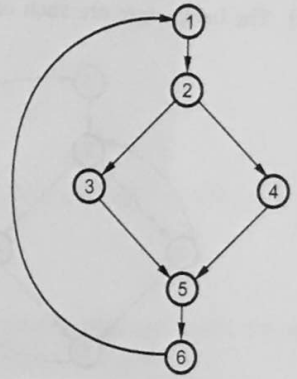


Fig. 8.5.3 Flow graph of natural loop

3. Inner loops

The inner loop is a loop that contains no other loop.

Here the inner loop is 4 → 2 that means edge given by 2-3-4.

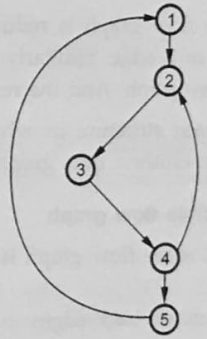


Fig. 8.5.4 Flow graph for inner loop

4. Pre-header

The pre-header is a new block created such that successor of this block is the header block. All the computations that can be made before the header block can be made before the pre-header block.

The pre-header can be as shown in Fig. 8.5.5.

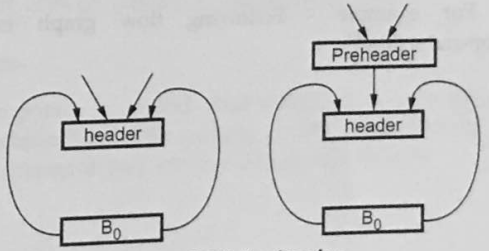


Fig. 8.5.5 Pre-header

5. Reducible flow graphs

The reducible graph is a flow graph in which there are two types of edges forward edges and backward edges. These edges have following properties,

i) The forward edge form an acyclic graph.

ii) The back edges are such edges whose head dominates their tail.

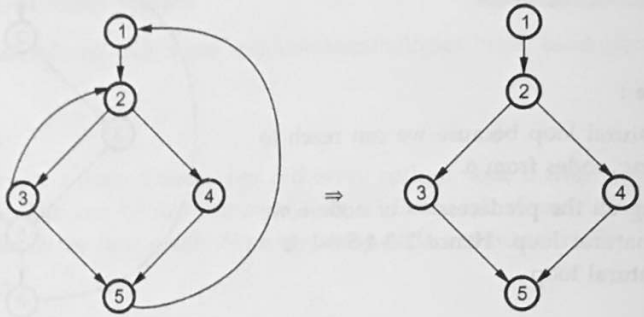


Fig. 8.5.6 Flow graph

The above flow graph is reducible. We can reduce this graph by removing the back edge from 3 to 2 edge. Similarly by removing the backedge from 5 to 1 we can reduce the above flow graph. And the resultant graph is a cyclic graph.

The program structure in which there is exclusive use of if-then, while-do or goto statements generates a flow graph which is always reducible.

6. Non-reducible flow graph

A non reducible flow graph is a flow graph in which -

i) There are no back edges

ii) Forward edges may produce cycle in the graph.

For example - Following flow graph is non-reducible.

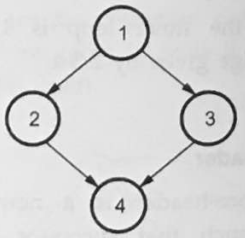


Fig. 8.5.7

Example 8.5.1 *Construct DAG for a + a * (b – c) + (b – c)* d. also generate three address code for same.*

GTU : Summer-17, Marks 7

Solution : The three address code is

$t_1 = b - c$

$t_2 = a * t_1$

$t_3 = a + t_3$

$t_4 = t_1 * d$

$t_5 = t_3 + t_4$

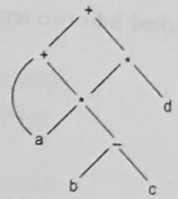


Fig. 8.5.8 DAG for a + a * (b – c) + (b – c) * d

8.6 Next Use Information

The next-use information is a collection of all the **names** that are useful for next subsequent statement in a block. The **use of a name** is defined as follows.

Consider a statement,

x := i

j := x op y

that means the statement j uses value of x.

The next-use information can be collected by making the backward scan of the programming code in that specific block. Suppose the three address statement is as given below.

L: a := b op c then the steps in backward scan are -

i) The currently found information in symbol table regarding the next use and liveness of a, b and c is associated with the statement L.

ii) In the symbol table set 'a' to "not live" and "no next use".

iii) Set 'b' and 'c' to live and next uses of b and c in symbol table.

8.6.1 Storage for Temporary Names

For the distinct names each time a temporary is needed. And each time a space gets allocated for each temporary. To have optimization in the process of code generation we can pack two temporaries into the same location if they are not live simultaneously.

Consider three address code as,

$t_1 := a * a$

$t_2 := a * b$

$t_3 := 4 * t_2$

$t_4 := t_{1+}t_3$

$t_5 := b * b$

$t_6 := t_{4+}t_5$

This can be packed into two temporaries as follows.

$t_1 := a * a$

$t_2 := a * b$

$t_2 := 4 * t_2$

$t_1 := t_{1+}t_2$

$t_2 := b * b$

$t_1 := t_{1+}t_2$

Many times the temporaries can be packed into registers rather than memory locations.

8.7 The DAG Representation of Basic Blocks

GTU : Winter-12,18, Summer-19, Marks 7

The directed acyclic graph is used to apply transformations on the basic block. To apply the transformations on basic block a DAG is constructed from three address statement.

A DAG can be constructed for the following type of labels on nodes

1) Leaf nodes are labeled by identifiers or variable names or constants. Generally leaves represent r-values.
2) Interior nodes store operator values.

The DAG and flow graphs are two different pictorial representations. Each node of the flow graph can be represented by DAG because each node of the flow graph is a basic block.

Example 8.7.1 *Consider*

```
sum = 0;
for (i-0;i<=10;i++)
sum = sum+a[i];
```

Solution : The three address code for above code is

(1) sum :=0

(2) i;=0

(3) $t_1 := 4*i$

(4) $t_2 := a[t_1]$

(5) $t_3 := sum+t_2$

(6) sum := t_3

(7) $t_4 = i+1;$

(8) i:= t_4

(9) if i<=10 goto (3)

We can partition above code into basic blocks as follows :

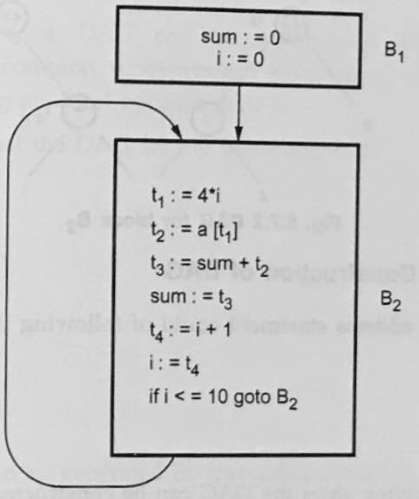


Fig. 8.7.1 Basic blocks

Now let us consider block B_2 for construction of DAG by numbering it.

(1) $t_1 := 4 * i$

(2) $t_2 := a[t_1]$

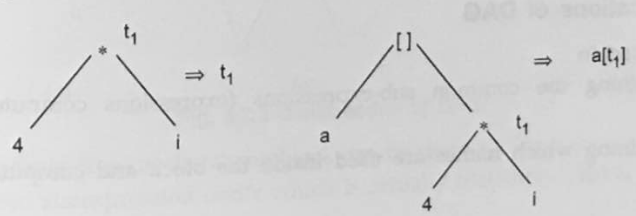
(3) $t_3 := sum + t_2$

(4) sum := t_3

(5) $t_4 := i + 1$

(6) i := t_4

(7) if i<10 goto (1)



Continuing in this fashion,

At node <= we have specified the goto 1 condition by mentioning statement number (1).

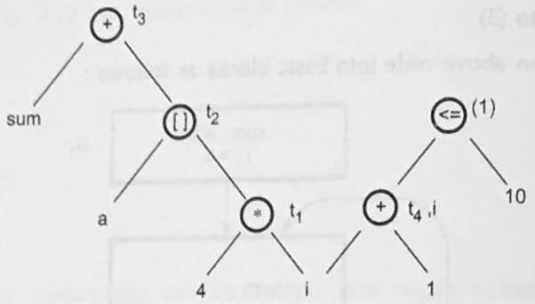


Fig. 8.7.2 DAG for block B_2

8.7.1 Algorithm for Construction of DAG

We assume the three address statement could of following types

Case (i) x:=y op z

Case (ii)x:=op y

Case (iii) x:=y

With the help of following steps the DAG can be constructed

Step 1 : If y is undefined then create node(y). Similarly if z is undefined create a node(z).

Step 2 : For the case(i) create a node(op) whose left child is node(y) and node(z) will be the right child. Also check for any common subexpressions. For the case(ii) determine whether is a node labeled op,such node will have a child node(y). In case(iii) node n will be node(y).

Step 3 : Delete x from list of identifiers for node(x). Append x to the list of attached identifiers for node n found in 2.

8.7.2 Applications of DAG

The DAG is used in

1. Determining the common sub-expressions (expressions computed more than once).
2. Determining which names are used inside the block and computed outside the block.

3. Determining which statements of the block could have their computed value outside the block.
4. Simplifying the list of quadruples by eliminating the common sub-expressions and not performing the assignment of the form x:=y unless and until it is a must.

8.7.3 DAG based Local Optimization

As mentioned earlier, a DAG can be constructed for a block and certain transformations such as common subexpression elimination; dead code elimination can be applied for performing the local optimization.

For example : Construct the DAG for the following block

```
a := b*c
d := b
e := d*c
b := e
f := b + c
g := f + d
```

The optimized code can be generated by traversing the DAG.

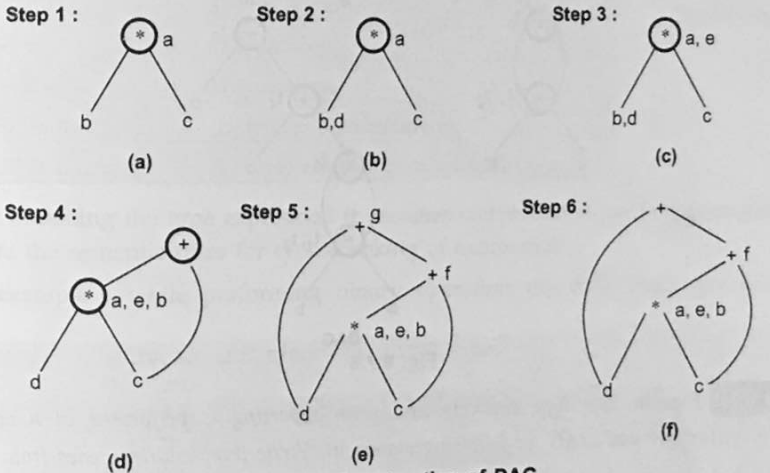


Fig. 8.7.3 Construction of DAG

The local optimization on the above block can be done.

1. A common subexpression e=d*c which is actually b*c(since d:=b) is eliminated.
2. A dead code for b=e is eliminated.

The optimized code and basic block is

```
a := b * cs
d := b
f := a + c
g := f + d
```

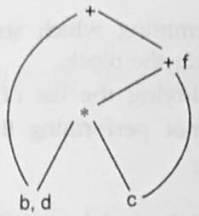


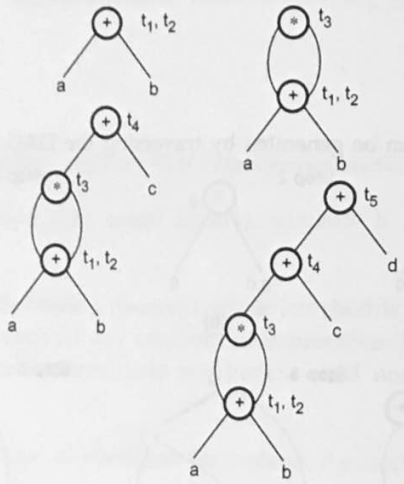
Fig. 8.7.4 Optimized code and DAG

Example 8.7.2 *Explain how following expression can be converted in a DAG* $a+b*(a+b)+c+d$.

Solution : We will build the three address code for given expression.

$t_1 := a + b$
$t_2 := a + b$
$t_3 := t_1 * t_2$
$t_4 := t_3 + c$
$t_5 := t_4 + d$

Now the DAG can be constructed in step by step manner.



DAG

Fig. 8.7.5

Example 8.7.3 *Explain how type checking and error reporting is performed in a compiler. Draw syntax tree and DAG for following statement. Write three address codes from both.*
$a = (a + b * c) \wedge (b * c) + b * c$

GTU : Winter-12, Marks 7

Solution : The type checker is a **translation scheme** in which the type of each expression from the types of subexpressions is obtained. The type checker can decide the types for arrays, pointers, statements and functions.

The type checker ensures following things -

i) Each identifier must be declared **before the use.**

ii) The use of identifier must be **within the scope.**

iii) An identifier must **not** have **multiple definitions** at a time within the same scope.

The given grammar has basic data types as char, int, float and for reporting the errors **type_error.**

We assume that the index of the array start at 0. For example if

```
int arr[100];
```

leads to **type expression** as array(0,1...99,int) similarly

```
int *ptr;
```

The **type expression** for the above statement can be pointer(int)

Let us write the translation scheme for the above grammar.

Production rule	Type expression
$S \rightarrow D;E$	This means all declarations before expressions
$D \rightarrow T$ LIST	LIST.type:=T.type
LIST $\rightarrow$ id	{addtype(id.entry,LIST.type)}
$T \rightarrow$ char	{T.type:=char}
$T \rightarrow$ int	{T.type:=int}
$T \rightarrow$ float	{T.type:=float}
LIST $\rightarrow *L_1$	{L_1.type := pointer(LIST.type)}
LIST $\rightarrow L_1$[num]	{L_1.type:=array(0...num.val-1,LIST.type}

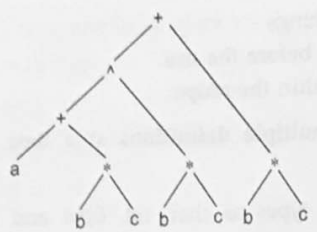
} Type checker

After obtaining the type expression it becomes convenient to write the semantic rules. These are the semantic rules for type checking of expression.

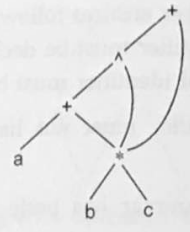
For example : while performing binary operation the data types are decided as follows

```
E → E1 op E2    {E.type:= if E1.type:= int and E2.type:= int
                 then int
                 else if E1.type:= float and E2.type:= float
                 then float
                 else if E1.type:= char and E2.type:= char
                 then char
                 else type_error
```

The **type_error** routine is used to report the error that occur during type checking.



Syntax tree



DAG

Three address code for syntax tree	Three address code for DAG
$t_1 := b*c$	$t_1 := b*c$
$t_2 := a+t_1$	$t_2 := a+t_1$
$t_3 := b*c$	$t_3 := t_2 \wedge t_1$
$t_4 := t_2 \wedge t_3$	$t_4 := t_3 + t_1$
$t_5 := b*c$	
$t_6 := t_4 + t_5$	

Example 8.7.4 *Draw syntax tree and DAG for the statement*

$x = (a+b)*(a+b+c)*(a+b+c+d)$ **GTU : Winter-18, Marks 7**

Solution : Syntax Tree :

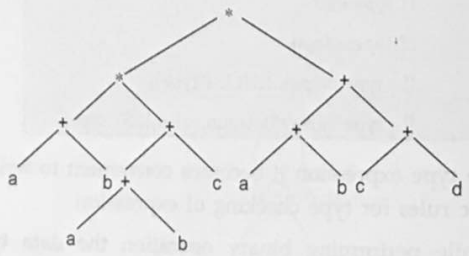


Fig. 8.7.6

DAG :

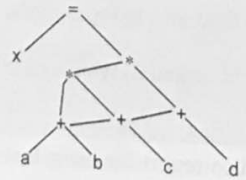


Fig. 8.7.7

Example 8.7.5 *Construct syntax tree and DAG for following expression.*

$a = (b + c + d) * (b + c - d) + a$ **GTU : Summer-19, Marks 7**

Solution : a = (b + c + d) * (b + c – d) + a

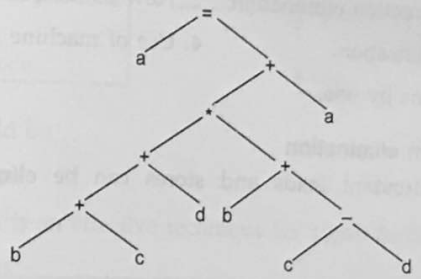


Fig. 8.7.8 Syntax tree

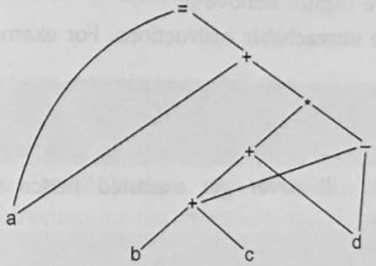


Fig. 8.7.9 DAG

8.8 Machine Dependent Optimization

GTU : Summer-14,16,19, Winter-11,16, Marks 7

Machine dependent optimization involves transformations that take into consideration, the properties of the target machine like registers and special machine instruction sequences available. These techniques are applied on generated target code.

Definition

Peephole optimization is a simple and effective technique for locally improving target code. This technique is applied to improve the performance of the target program by examining the short sequence of target instructions and replacing these instructions by shorter or faster sequence. [Note that : a peephole – a small window is moved over the target code and transformations can be made. And hence is the name !]

8.8.1 Characteristics of Peephole Optimization

The peephole optimization can be applied on the target code using following characteristic.

1. Redundant instruction elimination.
2. Flow of control optimization.
3. Algebraic simplification.
4. Use of machine idioms.

Let us discuss each one by one.

1. Redundant instruction elimination

- Especially the redundant loads and stores can be eliminated in this type of transformations.

For example :

```
MOV R0,x
MOV x,R0
```

We can eliminate the second instruction since x is already in R0. But if (MOV x, R0) is a label statement then we cannot remove it.

- We can eliminate the unreachable instructions. For example, following is a piece of C code.

```
sum=0
if(sum)
 printf("%d",sum);
```

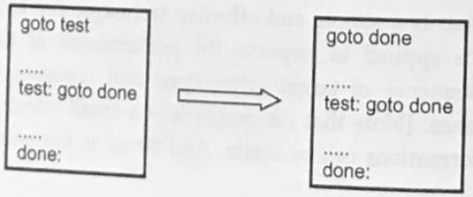
Now this if statement will never get executed hence we can eliminate such a unreachable code. Similarly

```
int fun(int a,int b)
{
c=a+b;
return c;
printf("%d",c); /* unreachable code and hence can be eliminated */
}
```

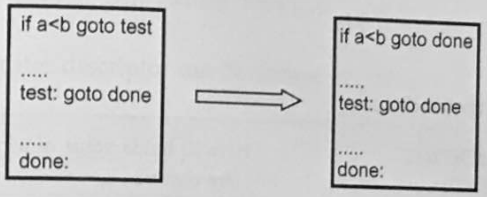
2. Flow of control optimization

Using peephole optimization unnecessary jumps on jumps can be eliminated.

For example,



Thus we reduce one jump by this transformation.



Another example could be

3. Algebraic simplification

Peephole optimization is an effective technique for algebraic simplification.

The statements such as

```
x := x + 0
```

or

```
x := x * 1
```

can be eliminated by peephole optimization.

4. Reduction in strength

Certain machine instructions are cheaper than the other. In order to improve the performance of the intermediate code we can replace these instructions by equivalent cheaper instruction. For example, x^2 is cheaper than x * x. Similarly addition and subtraction is cheaper than multiplication and division. So we can effectively use equivalent addition and subtraction for multiplication and division.

5. Machine idioms

The target instructions have equivalent machine instructions for performing some operations. Hence we can replace these target instructions by equivalent machine instructions in order to improve the efficiency. For example, some machines have auto-increment or auto-decrement addressing modes that are used to perform add or subtract operations.

Review Questions

1. *Explain peephole optimization.* **GTU : Nov.-11, Summer-14,16, Winter-16, Marks 7**
2. *Explain various methods of peephole optimization.* **GTU : Summer-19, Marks 7**

8.9 A Simple Code Generator

A this section we will discuss the method of generating target code from three address statement.

- In this method computed results can be kept in registers as long as possible.

For example

x := a+b;

The corresponding target code is

ADD b,R_i	Here R_i holds value of a Here cost =2

OR

MOV b,R_1 ADD R_1,R_0	Here R_0 holds value of a Here cost =2

- The code generator algorithm uses descriptors to keep track of register contents and addresses for names.
  1. A **register descriptor** is used to keep track of what is currently in each register. The register descriptors show that initially all the registers are empty. As the code generation for the block progresses the registers will hold the values of computations.
  2. The **address descriptor** stores the location where the current value of the name can be found at run time. The information about locations can be stored in the symbol table and is used to access the variables.
- The modes of operand addressability are as given below.

  S is used to indicate value of operand in storage.

  R is used to indicate value of operand in register.

  IS indicates that the address of operand is stored in storage i.e. indirect accessing.

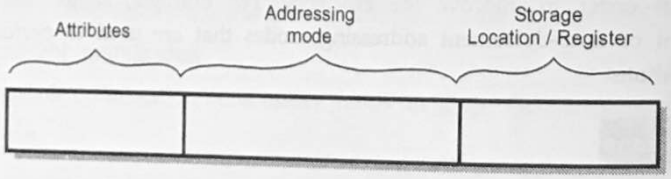


Fig. 8.9.1 Address descriptor

  IR indicates that the address of operand is stored in register i.e. indirect accessing.

- The address descriptor has following fields.

The attributes mean type of the operand. It generally refers to the name of temporary variables.

The addressing mode indicates whether the addresses are of type 'S','R','IS','IR'.

The third field is location field which indicates whether the address is in storage location or in register.

- Similarly the register descriptor can be shown as below.

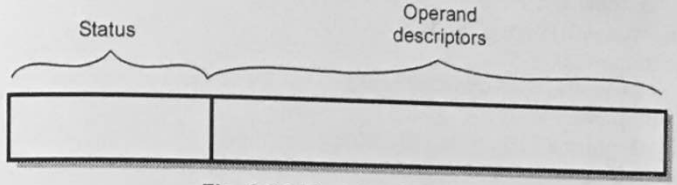


Fig. 8.9.2 Register descriptor

By using register descriptors we can keep track of the registers which are currently occupied. The status field is of Boolean type which is used to check whether the register is occupied with some data or not. When the status field holds the value 'True' then operand descriptors fields contains the pointer to the operand descriptor who is having the latest value in the register.

Algorithm for code generation using GETREG

Read the expression in the form of Operator, operand1 and operand2 and generate code using following algorithm -

```
Gen_Code(operator,operand1,operand2)
{
    if (operand1.addressmode = 'R')
    {
        if (operator = '+')
        Generate('ADD operand2,R0');
        else if(operator = '-')
            Generate('SUB operand2,R0');
        else if(operator = '*')
            Generate('MUL operand2,R0');
        else if(operator = '/')
            Generate('DIV operand2,R0');
    }
        else  if (operand2.addressmode = 'R')
    {
        if (operator = '+')
        Generate('ADD operand1,R0');
        else if(operator = '-')
            Generate('SUB operand1,R0');
        else if(operator = '*')
            Generate('MUL operand1,R0');
        else if(operator = '/')
```

```
            Generate('DIV operand1,R0');
    }
        else
    {
        Generate('MOV operand2,R0);
            if (operator = '+')
        Generate('ADD operand2,R0');
        else if(operator = '-')
            Generate('SUB operand2,R0');
        else if(operator = '*')
            Generate('MUL operand2,R0');
        else if(operator = '/')
            Generate('DIV operand2,R0');

    }
```

Example

We will generate code for following expression

$x := (a + b) * (c - d) + ((e/f) * (a + b))$

The corresponding three address code can be given as

$t_1 := a+b$

$t_2 := c-d$

$t_3 := e/f$

$t_4 := t_1 * t_2$

$t_5 := t_3 * t_1$

$t_6 := t_4 + t_5$

Using the simple code generation algorithm the sequence target code can be generated as

Three address code	Target code sequence	Register descriptor	Operand descriptor
$t_1 := a+b$	MOV a, R_0 ADD b, R_0	Empty R_0 contains t_1	t_1 R R_0
$t_2 := c-d$	MOV c, R_1 SUB d, R_1	R_1 contains c R_1 contains t_2	t_2 R R_1
$t_3 := e/f$	MOV e, R_2 DIV f, R_2	R_2 contains e R_2 contains t_3	t_3 R R_2

$t_4 := t_1 * t_2$	MUL R_0, R_1	R_0 contains t_1 R_1 contains t_2 R_1 contains t_4	t_4 R R_1
$t_5 := t_3 * t_1$	MUL R_2,R_0	R_2 contains t_3 R_0 contains t_1 R_0 contains t_5	t_5 R R_0
$t_6 := t_4 + t_5$	ADD R_1,R_0	R_1 contains t_4 R_0 contains t_5 R_0 contains t_6	t_6 R R_0

Example 8.9.1 *Generate the code sequence using code generation algorithm for the following expression*

$W := (A-B)+(A-C)+(A-C)$

Solution : We will write the 3 address code for given expression

$t_1 := A - B$

$t_2 := A - C$

$t_3 := t_1 + t_2$

$t_4 := t_3 + t_2$

$W := t_4$

Three address code	Target code sequence	Register descriptor	Operand/ address descriptor
$t_1 := A - B$	MOV A, R_0 SUB B, R_0	Empty R_0 contains t_1	t_1 is in R_0
$t_2 := A - C$	MOV A, R_1 SUB C, R_1	R_1 contains A R_1 contains t_2	t_2 is in R_1
$t_3 := t_1 + t_2$	ADD R_1, R_0	R_0 contains t_3	t_3 in R_0
$t_4 := t_3 + t_2$ $W := t_4$	ADD R_1, R_0 MOV R_0, W	R_0 contains t_4 R_0 contains W	W is in R_0

8.10 Register Allocation and Assignment

As we know the use of register operands instead of memory operands is always the faster and shorter. This also means that proper use of registers help in generating the good code. There are various strategies for deciding how to use these registers and what values are to be stored in the registers. In other words certain strategies are adopted by

the compiler for register allocation and assignment (as we know that register handling in code generation can be done using register allocation and assignment) .

- The most commonly used strategy to register allocation and assignment is to assign specific values to specific registers. For instance for base addresses separate set of registers can be used. Similarly for storing the stack pointers again a separate set of registers can be assigned; arithmetic computations can be done using separate set of registers. Remaining set of registers are used by the compiler for suitable purposes.
- The advantage of this approach is that the design process of compiler for code generation becomes simplified.
- The disadvantage of this method is that the design of compiler becomes complicated because of restrictive use of registers. At the same time certain set of registers remain totally unused over substantial portions of code and some set of registers get overloaded. But this disadvantage can be tolerable by most of the compiler and this approach can be adopted in most of the computing environment.

Now we will discuss various strategies used in register allocation and assignment and those are

1. Global register allocation.
2. Usage count.
3. Register assignment for outer loop.
4. Graph coloring for register assignment.

8.10.1 Global Register Allocation

While generating the code the registers are used to hold the values for the duration of single block. All the live variables are stored at the end of each block. For the variables that are used consistently we can allocate specific set of registers. Hence allocation of variables to specific registers that is consistent across the block boundaries is called **global register allocation**.

Following are the strategies adopted while doing the global register allocation.

- The global register allocation has a strategy of storing the most frequently used variables in fixed registers throughout the loop.
- Another strategy is to assign some fixed number of global registers to hold the most active values in each inner loop.
- The registers not already allocated may be used to hold values local to one block.
- In certain languages like C or Bliss programmer can do the register allocation by using register declaration.

8.10.2 Usage Count

The usage count is the count for the use of some variable x in some register used in any basic block. The usage count gives the idea about how many units of cost can be saved by selecting a specific variable for global register allocation. The approximate formula for usage count for the Loop L in some basic block B can be given as

$$\sum_{\text{block B in L}} (\text{use}(x,B) + 2^* \text{ live}(x,B))$$

where use(x,B) is number of times x used in block B prior to any definition of x and live(x,B) =1 is x is live on exit from B; otherwise live(x) = 0.

For example :

Consider a block B1, B2, B3, B4 and count the usage count for block B in following loop L

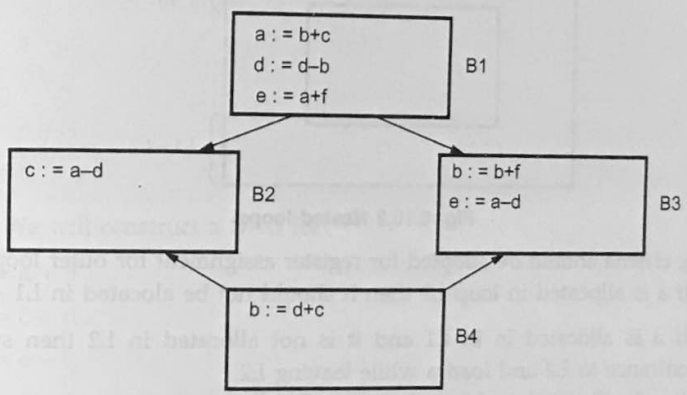


Fig. 8.10.1

The usage count for block B1 for variable a is

use(a,B1) = 0 ∵ a is defined in B1 before use.

2*live(a,B1) = 2 ∵ a is live on exit of B1 hence live(a,B1)=1

∴ (use(a,B1) + 2* live(a,B)) = 2

The usage count for block B2 and B3 for variable a is

use(a,B2) = use(a,B2) = 1 ∵ a is used in B1 and B2 before definition.

2*live(a,B1) = 2*live(a,B2) = 0 ∵ a is not live on exit of B1 and B2

∴ $\sum_{\text{B in L}} \text{use}(a,B) = 2$

$$\therefore \quad \sum_{B \text{ in } L} use(a,B) + 2 * live(a,b) = 2 + 2$$
$$= 4$$

Hence the usage count of a is 4. That means compiler can save 4 units of cost by selecting a for the global register allocation.

8.10.3 Register Assignment for Outer Loop

Consider that there are two loops L1 is outer loop and L2 is an inner loop. And allocation of variable a is to be done to some register. The approximate scenario is as given below -

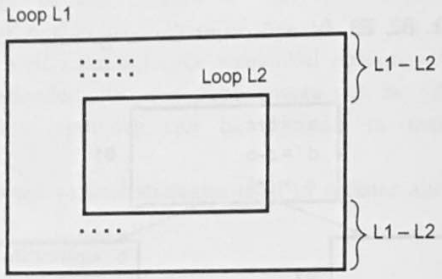


Fig. 8.10.2 Nested loops

Following criteria should be adopted for register assignment for outer loop

1) If a is allocated in loop L2 then it should not be alocated in L1 – L2.
2) If a is allocated in in L1 and it is not allocated in L2 then store a on a entrance to L2 and load a while leaving L2.
3) If a is allocated in L2 and not in L1 then load a on entrance of L2 and store a on exite from L2.

8.10.4 Graph Coloring for Register Assignment

As we encounter the variable, the register is allocated for it. Thus we go on allocating the registers for the variables that we visit. But a time may come when a register is needed for computation but all the registers are occupied. In such a situation we may need to make some registers free for reusability. Again from the heap of allocated registers which register is can be freed is a big question. To solve this problem a graph coloring technique is used.

The graph coloring works in two passes. The working is as given below

1. In the first pass the specific machine instruction is selected for register allocation. For each variable a symbolic register is allocated.

2. In the second pass the register inference graph is prepared. In register inference graph each node is a symbolic registers and an edge connects two nodes where one is live at a point where other is defined.
3. Then a graph coloring technique is applied for this register inference graph using k-color. The k-colors can be assumed to be number of assignable registers. In graph coloring technique no two adjacent nodes can have same color. Hence in register inference graph using such graph coloring principle each node (actually a variable) is assigned the symbolic registers so that no two symbolic registers can interfere with each other with assigned physical registers.

8.11 More Examples on Code Generation

Example 8.11.1 *Compute DAG for following three address statements. Considering this DAG as an example, explain the process of code generation from DAG.*

$t_1 = a + b$

$t_2 = c + d$

$t_3 = e - t_2$

$t_4 = t_1 - t_3$

Solution : We will construct a DAG for

$t_1 = a + b$

$t_2 = c + d$

$t_3 = e - t_2$

$t_4 = t_1 - t_3$

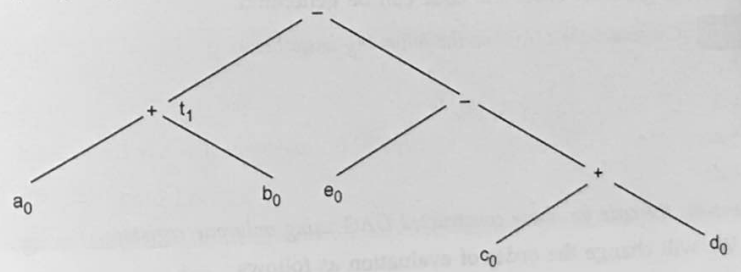


Fig. 8.11.1

For the above order of evaluation we get the code as [Assuming two registers R_0 and R_1 are available]

MOV a, R_0

ADD b, R_0

MOV c, R_1

ADD d, R_1

MOV R_0, t_1

MOV e, R_0

SUB R_1, R_0

MOV t_1, R_1

SUB R_0, R_1

MOV R_1, t_4

If we change the sequence of computation as :

$t_2 = c + d$

$t_3 = e - t_2$

$t_1 = a + b$

$t_4 = t_1 - t_3$

Then the generated code will be

MOV c, R_0

ADD d, R_0

MOV e, R_1

SUB R_0, R_1

MOV a, R_0

ADD b, R_0

SUB R_1, R_0

MOV R_0, t_4

Thus by rearranging the order the code can be generated.

Example 8.11.2 *Construct the DAG for the following basic block*

*d := b * c*

e := a + b

*b := b * c*

a := e – d

Then generate the code for above constructed DAG using only one register.

Solution : We will change the order of evaluation as follows

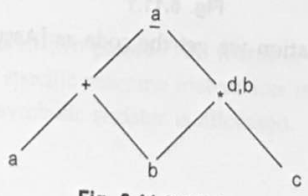


Fig. 8.11.2 DAG

d := b * c

e := a + b

a := e – d

b: = d

The three code can be generated as follows

t_1 := b * c

t_2 := a + b

$t_2 := t_2 - t_1$

$t_3 := t_1$

The code can be generated using one register as follows

MOV b, R0

MOV R0, t_1 → t_1:=b

ADD a,R0 → R0:=a+b

MUL c, t_1 → t1:=b*c

SUB t_1, R0 → R0:=(a+b)–(b*c)

MOV R0,a

Example 8.11.3 *Generate code for the following C program using any code generation algorithm.*

```
main()
{
 int i;
 int a[10];
 while(i<=10)
  a[i]=0;
}
```

Solution : First of all we will generate three address codes for the given C code.

Label 1 : if i<=10 goto Label 2

goto Label 3

Label 2 : t_1 := i*4

t_2 := addr(a)

$t_2[t_1]$:= 0

goto Label1

Label 3 : stop

The code generation using simple code generation algorithm for the corresponding code is

```
MOV i,R0
Label 1 : CMP R0,#10
          JLE Label2
          JMP Label3
Label 2 : MOV #0,a(R0)
          JMP Label1
Label 3 : EXIT
```

Example 8.11.4 *Construct a DAG for the given expression*

*(a – b) + c * (d / e)*

Also generate the code for the same.

Solution :

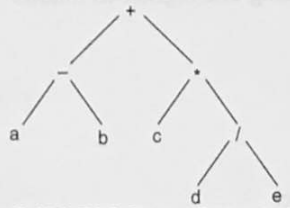


Fig. 8.11.3 DAG for a – b + c * (d/e)

The code being generated is

```
MOV c,R0
MOV d,R1
DIV R1,e        R1:=(d/e)
MUL R1,R0       R0:= c*(d/e)
MOV R1,a          R1:=a
SUB b,R1        R1:=R1-b i.e a-b
ADD R1,R0       R0:= R1+R0
```

Example 8.11.5 *Generate a code for following statements*

a = b + c ;

d = a + e ;

Solution : The code will be

```
MOV b, R0
ADD c, R0
MOV R0, a
MOV a, R0      } Redundant code, it can be removed
ADD e, R0
MOV R0, d
```


Then we get,

```
MOV b, R0
ADD c, R0
ADD e, R0
MOV R0, d
```

Example 8.11.6 *For the following expression obtain optimal code using*

i) only two registers

ii) only one register

(a + b) – (c – (d + e))

Solution : First of all we will write a three address code for given expression

$t_1 = a + b$

$t_2 = d + e$

$t_3 = c - t2$

$t_4 = t1 - t3$

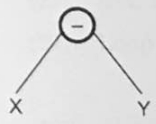
The generated code using simple code generation algorithm will be

```
i)      MOV d, R0
        ADD e,  R0
        MOV c, R1
        SUB R0,R1
        MOV a,  R0
        ADD b, R0
        SUB R1, R0
ii)     MOV d, R0
        ADD e,  R0
        MOV R0, t1
        MOV c, R0
        SUB t1, R0
        MOV R0, t2
        MOV a, R0
        ADD b, R0
        SUB t2, R0
```

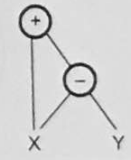
Example 8.11.7 *Show the DAG for statement Z = X – Y + X * Y * U – V / W + X + V.*

Solution :

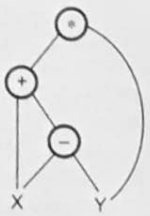
Step 1



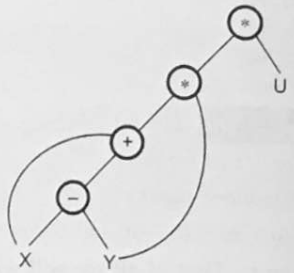
Step 2 : X – Y + X



Step 3 : X – Y + X * Y

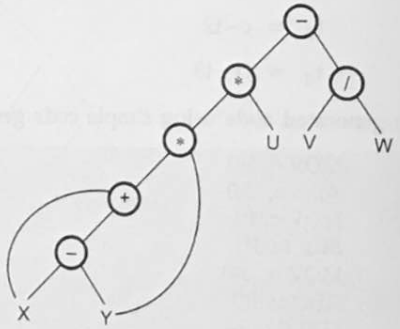


Step 4 : X – Y + X * Y * U

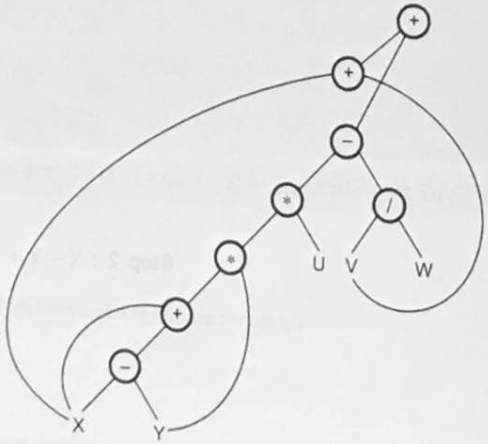


Step 5 : V / W

Step 6 : X – Y + X * Y * U – V/ W



Step 7 : X – Y + X * Y * U – V/ W + X + V



Example 8.11.8 *Generate code for following c statements*

i) x = $f(a)+f(a)+f(a)$ *ii)* x = $f(a)/g(b,c)$

iii) x = $f(f(a))$ *iv)* x = + + $f(a)$

Solution : i) f(a) denotes a call to the function. This call is made thrice. And then addition of their return values is done. The result is finally stored in variable x.

We will first write a three address code for given statement.

```
        t1 : = 0
        t2 : = 1
        t3 : = 3
L1 :    if t2 <= t3 goto Loop
        goto End
Loop :  parama
        call f,1
        return t4
        t1 : = t1 + t4
        t2 : = t2 + 1
        goto L1
End :
```

The code will be

```
(10)  MOV #0, R0
(11)  MOV #1, R1
(12)  MOV #3, R2
(13)  CMP R1, R2
(14)  CJ < = (16)
(15)  HALT
(16)  MOV a, R3  /* parameter a is in register R3 */
(17)  GOTO 100        /* At location 100 procedure f is defined */.
(18)  MOV AH, R4 /* return val from function f is in AH */
(19)  ADD R4, R0  /* it is stored in R4 then added to R0 *\
(20)  MOV R0, x /* R0 value is stored in x */
(21)  INC R1
(22)  Loop (13)
 :
(100) /* procedure for f*/
```

ii) $x = f(a) / g(b, c)$

Param a
call f, 1
return t_1
Param b
Param c
call g, 2 /* 2 parameter to g */
return t_2
$x = t_1 / t_2$.

The code will be

(1) MOV a, R_0
(2) GOTO 100
(3) MOV AX, R_1 /* return value of f(a) in R_1 */
(4) MOV b, R_2
(5) MOV c, R_3
(6) GOTO 200
(7) MOV AX, R_4 /* return value of g(b, c) in R_4 */
(8) DIV R_1, R_4 /* f(a)/g(b ,c)*/
(9) MOV R_4 x. /* store result in x */
⋮
(100) /* procedure for f */
⋮
(200) /* procedure for g */
⋮

iii) Three address code for x=f(f(a))

param a
call f, 1
return t_1
param t_1
call f, 1
return t_2
$x := t_2$

The code will be

(1) MOV a, R_0 /* parameter a in R_0 */
(2) GOTO 100 /* at location 100 f(a) is defined */
(3) MOV AX, R_1 /* return addres of AX is stored in R */
(4) MOV R_1, R_2 /* Take parameter R_1 in register R_2 */
(5) GOTO 200 /* at 200 location procedure f(f(a)) */
(6) MOV AX, R_3 /* return value in R_3 */
(7) MOV R_3, x /* finally in x */
⋮
(100) /* procedure for f(a) */
⋮
(200) /* f(f(a)) */
⋮

iv) $x = + + f(a)$

param a
call f, 1
return t_1
$t_1 = t_1 + 1$
$x := t_1$

The code will be

(1) MOV a, R_0 /* parameter a is in R_0 */
(2) GOTO 100 /* At location 100 f(a) is defined */
(3) MOV AX, R_1 /* MOV return value in R1 */
(4) ADD # 1, R_1
(5) MOV R_1, x /* Final result in variable x */
⋮
(100) /* procedure for f(a) */

Example 8.11.9 *Generate code for following statements for the target machine (target machine is a byte addressable machine with 4 bytes to a word and N general purpose registers). Assuming all variables are static. Assume 3 registers are available.*

(a) $x = a[I+1]$ *(b)* $a[I] = b[c[I]]$
(c) $a[I][J] = b[I][k] * c[k][J]$ *(d)* $a[I] = a[I] + b[J]$

Solution : (a) x = a [I]+1

```
MOV  I, R0
MOV a(R0), R1
ADD #1, R1
MOV  R1, x.
```

(b) a [I] = b [c [I]]

```
MOV I, R0
MOV c(R0), R1          /* c [ I ] */
MOV b (R1), R2         /* b [c [I ] ] */
MOV  R2, a ( R0)
```

(c) a [I][J] = b [I][k]*C [k][J]

The two dimensional array can be represented by one dimensional array.

For instance : Consider elements 10, 20, 30, 40, 50, 60, 70, 80

In 2D array i = 2, j = 3

	1	2	3	4
1	10	20	30	40
2	50	60	70	80

Two dimensional array

1	2	3	4	5	6	7	8
10	20	30	40	50	60	70	80

↑

In 2D array i = 2, j = 3

Hence in row major representation

A[i] [j] = Base address + (j * row-size + i) × Element size

Element size = 4 bytes.

Row-size = 2

Then A [i] [j] = Base address + (2 * j+ i) * 4.

The code will be

```
MOV J, R0
MUL #2, R0
ADD  k, R0
MUL #4, R0
```



```
MOV c(R0), R1          /* c[k] [J] */
MOV k, R0
MUL #2, R0
ADD I, R0
MUL #4, R0
MUL b (R0), R1         /*  R1 = b [I] [k] * c[k] [J] */
MOV J, R0
MUL #2, R0
ADD I, R0
MUL #4, R0
MOV R1, a(R0)
```

(d) a [I] = a [I]+b [J]

```
MOV I, R0
MOV a(R0) , R1
MOV J, R0
ADD b(R0),  R1
MOV I, R0
MOV a(R0), R2
MOV R1, R2
```

8.12 Machine Independent Optimization

Code optimization is required to produce an efficient target code. Following are the **advantages** of code optimization

(1) The intermediate code can be optimized to produce efficient object code.

(2) The code optimization allows faster execution of source program by making minor modifications.

The classification of optimization can be done in two categories machine dependant optimization and machine independent optimization.

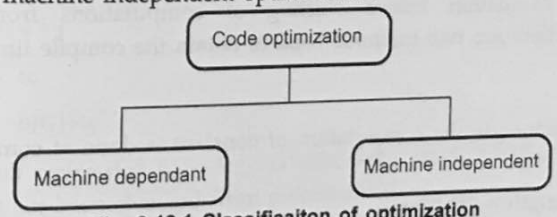


Fig. 8.12.1 Classificaiton of optimization

The machine dependant optimization is based on characteristics of the **target machine** for the instruction set used and addressing modes used for the instructions to produce the efficient target code.

The machine independent optimization is based on the characteristics of the **programming languages** for appropriate programming structure and usage of efficient arithmetic properties in order to reduce the execution time.

The **machine dependant optimization** can be achieved using following criteria -

1. Allocation of sufficient number of resources to improve the execution efficiency of the program.
2. Using immediate instructions wherever necessary.
3. The use of intermix instructions. The intermixing of instruction along with the data increases the speed of execution.

The **machine independent optimization** can be achieved using following criteria -

1. The code should be analyzed completely and use alternative equivalent sequence of source code that will produce a minimum amount of target code.
2. Use appropriate program structure in order to improve the efficiency of target code.
3. From the source program eliminate the unreachable code.
4. Move two or more identical computations at one place and make use of the result instead of each time computing the expressions.

8.13 Few Selected Optimizations

GTU : Winter-12,14,15,16,17,18,19,20, Summer-12,18, Marks 7

The optimization can be done locally or globally. If the transformation is applied on the same basic block then that kind of transformation is done locally otherwise transformation is done globally. Generally the local transformations are done first.

While applying the optimizing transformations the semantics of the source program should not be changed.

8.13.1 Compile Time Evaluation

Compile time evaluation means shifting of computations from run time to compilation time. There are two methods used to obtain the compile time evaluation.

1. Folding

In the folding technique the computation of constant is done at compile time instead of execution time.

For example : length = (22/7) * d

Here folding is implied by performing the computation of 22/7 at compile time instead of execution time.

2. Constant propagation

In this technique the value of variable is replaced and computation of an expression is done at the compilation time.

For example :

```
pi = 3.14;
 r = 5;
Area = pi * r * r
```

Here at the compilation time the value of pi is replaced by 3.14 and r by 5 then computation of 3.14 * r * r is done during compilation.

8.13.2 Common Sub Expression Elimination

The common sub expression is an expression appearing repeatedly in the program which is computed previously. Then if the operands of this sub expression do not get changed at all then result of such sub expression is used instead of recomputing it each time.

For example :

$t_1 := 4 * i$

$t_2 := a[t_1]$

$t_3 := 4 * j$

$t_4 := 4 * i$

$t_5 := n;$

$t_6 := b[t_4]+t_5$

The above code can be optimized using the common sub expression elimination as,

$t_1 := 4 * i$

$t_2 := a[t_1]$

$t_3 := 4 * j$

$t_5 := n;$

$t_6 := b[t_1]+t_5$

The common sub expression $t_4 := 4 * i$ is eliminated as its computation is already in t_1 and value of i is not been changed from definition to use.

8.13.3 Variable Propagation

Variable propagation means use of one variable instead of another.

For example :

```
x = pi;
...
area = x * r * r;
```

The optimization using variable propagation can be done as follows,

```
area = pi * r * r;
```

Here the variable x is eliminated. Here the necessary condition is that a variable must be assigned to another variable or some constant.

8.13.4 Code Movement

There are two basic goals of code movement -

1. To reduce the size of the code i.e. to obtain the space complexity.
2. To reduce the frequency of execution of code i.e. to obtain the time complexity.

For example :

```
for(i=0;i<=10;i++)
{
 x=y*5;
 ...
 k=(y*5) + 50;
}
```

This can be optimized as

```
temp = y*z;
for(i=0;i<=10;i++)
{
 x=z;
 ...
 k=z + 50;
}
```

The code for y * 5 will be generated only once. And simply the result of that computation is used wherever necessary.

Loop invariant computation

The loop invariant optimization can be obtained by moving some amount of code outside the loop and placing it just before entering in the loop. This method is also called code motion.

For example :

```
while(i<=Max-1)
{
 sum= sum+a[i];
}
```

The above code can be optimized by removing the computation of Max-1 outside the loop. The optimized code can be,

```
n= Max-1;
while(i<=n)
{
 sum=sum+a[i];
}
```

8.13.5 Strength Reduction

The strength of certain operators is higher than others. For instance strength of * is higher than +. In strength reduction technique the higher strength operators can be replaced by lower strength operators.

For example :

```
for(i=1;i<=50;i++)
{
...
count = i × 7;
...
}
```

Here we get the count values as 7, 14, 21 and so on upto less than 50.

This code can be replaced by using strength reduction as follows -

```
temp=7
for(i=1;i<=50;i++)
{
...
count = temp;
temp = temp+7;
}
```

Thus we get the value of count as 7, 14, 21 and so on upto less than 50.

The induction variable is integer scalar identifier used in the form of

v = v ± constant, Here v is a induction variable.

The strength reduction is not applied to the floating point expressions because such a use may yield different results.

8.13.6 Dead Code Elimination

A variable is said to be **live** in a program if the value contained into it is used subsequently. On the other hand, the variable is said to be dead at a point in a program if the value contained into it is never been used. The code containing such a variable supposed to be a dead code. And an optimization can be performed by eliminating such a dead code.

For example :

```
i=j;
...
x=i+10;
....
```

The optimization can be performed by eliminating the assignment statement i = j. This assignment statement is called **dead assignment**.

Another example :

```
i=0;
if(i=1)
{
 a=x+5;
}
```

Here **if** statement is a dead code as this condition will never get satisfied hence, if statement can be eliminated and optimization can be done.

Review Questions

1. *Explain various code optimization techniques.*
   GTU : May-12, Marks 4, Winter-14,18,20, Marks 7
2. *Explain any three types of optimization techniques.*
   GTU : Winter-12, Winter-15,16,19, Summer-18, Marks 7
3. *Explain function preserving transformations with example.* **GTU : Winter-17, Marks 7**

8.14 Loop Optimization

GTU : Winter-17, Marks 3

The code optimization can be significantly done in **loops** of the program. Specially inner loop is a place where program spends large amount of time. Hence if number of instructions are less in inner loop then the running time of the program will get decreased to a large extent. Hence loop optimization is a technique in which code optimization performed on inner loops. The loop optimization is carried out by following methods -

1. Code motion.
2. Induction variable and strength reduction.

3. Loop invariant method.
4. Loop unrolling.
5. Loop fusion.

Let us discuss these methods with the help of example.

1. Code motion

Code motion is a technique which moves the code outside the loop. Hence is the name. If there lies some expression in the loop whose result remains unchanged even after executing the loop for several times, then such an expression should be placed just before the loop (i.e. outside the loop). Here before the loop means at the entry of the loop.

For example :

```
while (i<=Max-1)
{
      sum=sum+a[i];
}
```

The above code can be optimized by removing the computation of Max-1 outside the loop. Hence the optimized code can be -

```
n = Max-1;
while (i<=n)
{
   sum=sum+a[i];
}
```

2. Induction variables and reduction in strength

A variable x is called an induction variable of loop L if the value of variable gets changed every time. It is either decremented or incremented by some constant.

For example :

Consider the block given below.

B1

```
i := i+1
t1:= 4*j
t2:= a[t1]
if t2 < 10 goto B1
```

In above code the values of i and t_1 are in locked state. That is, when value of i gets incremented by 1 then t_1 gets incremented by 4. Hence i and t_4 are induction variables.

When there are two or more induction variables in a loop, it may be possible to get rid of all but one.

Reduction in strength

The strength of certain operators is higher than others.

For example : Strength of * is higher than +. In strength reduction technique the higher strength operators can be replaced by lower strength operators.

For example :

```
for(i=1 ; i<=50; i++)
{
    ...
    count = i * 7;
    ...
}
```

Here we get values of count as 7, 14, 21 and so on upto 50.

This code can be replaced by using strength reduction as follows -

```
temp=7
for(i=1; i<=50; i++)
{
    ...
    count = temp
    temp = temp+7;
}
```

The replacement of multiplication by addition will speed up the object code. Thus the strength of operation is reduced without changing the meaning of above code.

The strength reduction is not applied to the floating point expressions because such a use may yield different results.

3. Loop invariant method

In this optimization technique the computation inside the loop is avoided and thereby the computation overhead on compiler is avoided. This ultimately optimizes code generation.

For example :

```
for i:=0 to 10 do begin
 k:=i+a/b;
 ...
 ...
 ...
end;
```

can be written as

```
t:=a/b;
for i:=0 to 10 do begin
 k:=i+t;
```



```
 ...
 ...
 ...
end;
```

4. Loop unrolling

In this method the number of jumps and tests can be reduced by writing the code two times.

For example :

```
int i=1;
while(i<=100)
{
    a[i]=b[i];
    i++;
}
```

Can be written as →

```
int i=1;
while(i<=100)
{
    a[i]=b[i];
    i++;
    a[i]=b[i];
    i++;
}
```

5. Loop fusion

In loop fusion method several loops are merged to one loop

For example :

```
for i:=1 to n do
for j:=1 to m do
a[i,j]:=10
```

Can be written as →

```
for i:=1 to n*m do
a[i]:=10
```

Example 8.14.1 *Construct basic blocks and data flow graph and identify loop invarint statements :*

```
for (i = 1 to n)
{
    j = 1
    while (j ≤ n)
    {
        A = B * C/D
        j = j + 1 ;
    }
}
```

Solution :

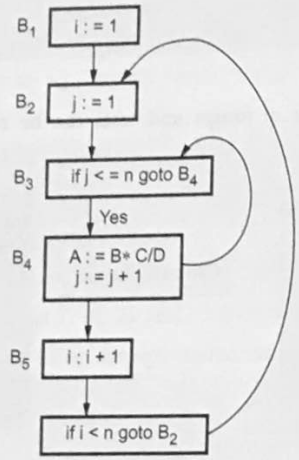


Fig. 8.14.1

The loop invariant statement is

```
while (j < = n)                          t := C/D
{                                        while (j < = n)
                                         {
A = B * C/D        Optimized as          A = B * t
...                                      ...
...                                      ...
```

Review Question

1. *Explain three loop optimization techniques with example.* **GTU : Winter-17, Marks 3**

8.15 Short Questions and Answers

Q.1 What are the properties of object code generation phase ?

Ans. : Following are the properties of object code generation -

- **Correctness -** It should produce a correct code and do not alter the purpose of source code.
- **High quality -** It should produce a high quality object code.

- **Efficient use of resources of the target machine -** While generating the code it is necessary to know the target machine on which it is going to get generated. By this the code generation phase can make an efficient use of resources of the target machine. For instance memory utilization while allocating the registers or utilization of arithmetic logic unit while performing the arithmetic operations.
- **Quick code generation -** This is a most desired feature of code generation phase. It is necessary that the code generation phase should produce the code quickly after compiling the source program.

Q.2 What are various forms of object code ?

Ans. : Various forms of object code are

1. Absolute code 2. Relocatable code 3. Assembler code

1. **Absolute code -** Absolute code is a machine code that contains reference to actual addresses within program's address space. The generated code can be placed directly in the memory and execution starts immediately.
2. **Relocatable code -** Producing a relocatable machine language program as output allows subprograms to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution with the help of a linking loader.
3. **Assembler code -** Producing an assembly language program as output makes the process of code generation somewhat easier. We can generate symbolic instructions and use the macro facilities of the assembler to help in generation of code.

Q.3 Explain various addressing modes used in code generation.

Ans. :

Addressing mode	Form	Address	Added cost
absolute	M	M	1
register	R	R	0
indexed	c(R)	c + contents(R)	1
indirect register	R	contents(R)	0
indirect indexed	c(R)	contents(c+contents(R))	1
literal	#c	c	1

Q.4 How do you calculate the cost of an instruction ?

Ans. : The instruction cost can be computed as one plus cost associated with the source and destination addressing modes given by "added cost".

Instruction	Cost	Interpretation
MOV R0,R1	1	Cost of register mode +1=0+1=1
MOV R1,M	2	Use of memory variable +1=1+1=2
SUB 5(R0),10(R1)	3	Use of first constant +use of second constant +1=3

Q.5 What is the use of temporary names in code generation process ?

Ans. : The temporaries are used to store the distinct values of the operands during the code generation. It optimizes the compilation process. Sometimes these temporaries can be reused during evaluation of expression.

Q.6 What are the uses of register and address descriptors in code generation ?

Ans. : The register descriptor is used to keep track of the contents within the registers. The address descriptor stores the location where the current value of the name can be found at run time.

Q.7 What is the need of code optimization ?

Ans. : The code optimization is used to produce the efficient target code.

Q.8 What are the criteria that needs to be considered while applying the code optimization techniques ?

Ans. : There are two important issues that need to be considered while applying the techniques for code optimization and those are,

1. The semantic equivalence of the program must not be changed. In simple words the meaning of program must not be changed.
2. The improvement over the program efficiency must be achieved without changing the algorithm of the program.

Q.9 What are the transformation techniques that can be applied in code optimization ?

Ans. : Following are some commonly used transformations that can be applied during the code optimization -

1. Constant folding
2. Constant propagation
3. Common sub expression elimination
4. Variable propagation
5. Code movement
6. Strength reduction
7. Dead code elimination
8. Loop invariant computation
9. Loop optimization

Q.10 What is code motion ?

Ans. : Code motion is an optimization technique in which amount of code in a loop is decreased. This transformation is applicable to the expression that yields the same

result independent of the number of times the loop is executed. Such an expression is placed before the loop.

For example -

```
while (i < Max – 1)
{
 sum = sum + a [i];
}
```

→

```
n = Max – 1
while (i < = n)
{
 sum = sum +a [i];
}
```

Q.11 Explain loop fusion or loop jamming.

Ans. : In loop fusion method several loops are merged into one loop.

For example

```
for i:=1 to n do          for i:=1 to nm do
for j:=1 to m do          a[i]:=10
a[i, j]:=10
```

Q.12 What is dead code elimination ?

Ans. : Dead code elimination is an optimization technique in which the code containing the variable whose value is never used is removed.

Q.13 What is constant folding ?

Ans. : Constant folding is an optimization technique in which the computation of constant is done at compile time instead of execution time.

Q.14 Define loop unrolling with example

Ans. : The loop unrolling is the loop optimization technique in which the number of jumps and tests can be reduced by writing the code two times.

For example -

```
int i = 1;
while(i<=100)
{
a[i]=b[i];
i++;
}
```

→

```
int i = 1;
while(i<=50)
{
a[i]=b[i];
i++;
a[i]=b[i];
i++;
}
```

Q.15 What is an optimizing compiler ?

Ans. : The optimizing compiler is a compiler that take minimum amount of time to execute the program and also takes less amount of memory for executing the program,

Q.16 Name the techniques in loop optimization

Ans. : Various techniques in loop optimization are -

1. Code motion
2. Induction variable and strength reduction
3. Loop invariant method
4. Loop unrolling
5. Loop fusion

8.16 Multiple Choice Questions

Q.1 Following is a form of an object code _____.

a Three address code
b Polish notation
c Relocatable code
d None of the above

Q.2 Following are the properties of code generation _____.

a Producing correct code
b Generation of high quality code
c Utilization of resources of target machine
d None of the above

Q.3 _____ is a form of object code produced by WATFIV.

a Absolute code
b Relocatable code
c Three address code
d Assembler code

Q.4 Code generation take _____ as input.

a Source code
b Assembly language code
c Intermediate code
d None of the above

Q.5 Selection of appropriate instruction is an important factor in _____.

a code optimization phase
b code generation phase
c syntax analysis phase
d lexical analysis phase

Q.6 Register allocation is an important factor in _____.

a code optimization phase

b code generation phase
c syntax analysis phase
d lexical analysis phase

Q.7 Match the following

a	Absolute	1. *c(R)
b	Indirect register	2. c(R)
c	Indexed	3. *R
d	Indirect indexed	4.M

A. (A-4),(B-3)(C-2),(D-1) B. (A-1),(B-2)(C-4),(D-3)
C. (A-3),(B-1)(C-2),(D-4) D. (A-2),(B-4)(C-1),(D-3)

Q.8 Consider the following code _____.

```
MOV a,R0
ADD b, R0
MOV R0,c
```

The cost is

a 3
b 4
c 5
d 6

Q.9 Following techniques are based on function preserving transformations _____.

a common sub-expression elimination
b code motion
c reduction in strength
d induction variable elimination.

Q.10 Following techniques are based on loop optimization _____.

a common sub-expression elimination
b code motion
c copy propagation
d dead code elimination.

Q.11 Reduction in strength means _____.

a removing loop invariant computations
b replacing run time computation by compile time computation

c removing common sub-expression elimination

d replacing costly operation by cheaper one.

Q.12 Basic block can be analyzed by _____.

a DAG

b flow graph

c graph that involves cycle

d none of these

Q.13 The loops can be identified using _____.

a dominators

b reducible graphs

c depth first ordering

d all of the above.

Q.14 Peephole optimization is _____.

a applied to a small part of the code

b can be applied to the portion of the code that is not contiguous

c it can be applied repeatedly to get the optimized code

d all of the above.

Q.15 Data flow equations can be computed using _____.

a available expression

b reaching definition

c live variable analysis

d all of the above.

Answer Keys for Multiple Choice Questions

Q.1	c	Q.2	a,b,c
Q.3	a	Q.4	c
Q.5	a,b	Q.6	b
Q.7	a	Q.8	d
Q.9	a	Q.10	b
Q.11	d	Q.12	a
Q.13	d	Q.14	d
Q.15	d		

❑❑❑

9 Instruction - Level Parallelism

Syllabus

Processor Architectures, Code - Scheduling Constraints, Basic - Block Scheduling, Pass structure of assembler.

Contents

9.1 Processor Architectures

- While achieving the instruction level parallelism, processor is expected to issue several operations in a single clock cycle. In fact, machine issues just one operation per clock and yet achieve instruction level parallelism using concept of pipelining.

9.1.1 Instruction Pipelines and Branch Delays

- Instruction pipelining is a mechanism in which every new instruction is fetched at every clock and at the same time the preceding instructions are getting processed through pipeline.
- Every single instruction undergoes through five different stages.
- These stages are
  1. **Fetch(IF),**
  2. **Decode(ID),**
  3. **Execute(EX),**
  4. **Memory Access(MEM), and**
  5. **Write(WB).**
- That means, first of all the instruction is fetched, then it is decoded, executes the operation, access the memory and result is written back.
- Following figure shows how five instructions are getting executed using instruction pipelining mechanism.

CLOCK \ Instruction	I_1	I_2	I_3	I_4	I_5
1	IF				
2	ID	IF			
3	EX	ID	IF		
4	MEM	EX	ID	IF	
5	WB	MEM	EX	ID	IF
6		WB	MEM	EX	ID
7			WB	MEM	EX
8				WB	MEM
9					WB

Fig. 9.1.1 Instruction processing using instruction pipelining

- As shown in the above Fig. 9.1.1, at the first clock tick, the first instruction is fetched from the memory. Then the first instruction is decoded at the next clock tick and at the same clock tick, second instruction is fetched from the memory. At the third clock tick, the first instruction is executed, second instruction is getting decoded and the third instruction is fetched from the memory.
- In this way, multiple instructions are processed simultaneously using pipelining mechanism.
- **Branch instructions** are especially **problematic** because until they are fetched, decoded and executed, the processor does not know which instruction will execute next.
- When a branch instruction is encountered, the instruction pipeline is emptied and branch target is fetched. While handling the branch instruction the delay get introduced as branch target needs to be fetched. These delays cause "hiccups" in the instruction pipeline.

9.1.2 Pipelined Execution

- Instruction pipelining is possible only when the instruction is not dependant on the result of other instruction to proceed.
- There are some instructions that take several clocks to execute. For instance, **memory load** is one such operation that takes several clocks to execute.
- Even if a processor can issue only one operation per clock, several operations might be in their execution stages at the same time. So all the instructions can not be fully pipelined.
- Floating point add and multiplication can be fully pipelined. But floating point division can not be fully pipelined as this operation is complex.
- Most general purpose processors dynamically detect dependences between consecutive instructions.
- Some processors especially those embedded in hand-held devices, leave the dependence checking to the software in order to keep hardware simple and power consumption low.
- The compiler inserts "no-op" instructions in the code if necessary to assure that the results are available when needed.

9.1.3 Multiple Instruction Issue

- Multiple instructions can be issued at every clock tick. In the instruction pipeline mechanism, multiple instructions can be processed. For that purpose the arrangement of execution of multiple instructions can be made by multiplying

number of available stages in the pipeline by the number of instructions issue width.

- Just similar to instruction pipeline mechanism, multiple instructions issue can be handled hardware or software parallelism.
- The machines that rely on **software** to manage parallelism for multiple instructions issue are called **Very Long Instruction Word(VLIW) machines**.
- The machines that rely on **hardware** to manage parallelism for multiple instruction issue are called **superscalar machines**.
- The **VLWI machines** have wider instruction words to decode the operation in a single clock. In these machines the compiler decides which operations are to be issued in parallel and encode the information.
- **Superscalar machine** on the other hand have regular instruction set with ordinary sequential execution mechanism. Superscalar machines automatically detect dependencies among the instructions and issue them as operands.

Review Question

1. *Explain how instruction Processing occurs using Instruction Pipelining*

9.2 Code - Scheduling Constraints

- Code scheduling is a form of code optimization which is applied on the machine code generated by the code generator. The code-scheduling has three types of constraints -
  1) **Control dependence constraints :** All the operations executed in the original program must be executed in the optimized program.
  2) **Data dependence constraints :** The operations must generate the same results as the corresponding ones in the original program.
  3) **Resource constraints :** The scheduler must not oversubscribe the resources on the machine.

9.2.1 Data Dependence

- Data dependence is based on two basic operations read and write. The order of execution of these operations determine the three types of data dependence -
  1) **True dependence :** It is **read after write**. If a write is followed by a read of the same location, the read depends on the value written; In other words, a read of 'x' must continue to follow previous write of x. such dependence is known as a **true dependence**.

  2) **Anti dependence :** It is **write after read**. If write operation is performed after the read operation to the same location, we say that there is an anti-dependence from the read to the write. The write does not depend on the read operation, but if the write operation occurs before the read, then the read operation will pick up the wrong value. In other words, a write of 'x' must continue to follow previous reads of x.
  3) **Output dependence :** It is **write after write**. Two writes to the same location share an output dependence. If the dependence is violated, the value of the memory location written will have the wrong value after both operations are performed. **Anti-dependence** and **output dependences** are referred to as storage-related dependences. These are **not "true" dependences** and can be eliminated by using different locations to store different values.

9.2.2 Finding Dependences Among Memory Access

- To find the data dependence when some operations are performed, we need to check if the operations are performed on the same memory locations or not.
- Data dependence is generally undecidable at compile time. Consider following code fragment for finding the data dependence.

```
1) x = 10
2) *ptr = 20
3) y = x
```

- There can be **output dependence** between the statements (1) and (2) as these are write after write statements.
- There can be **true dependence** between the statement (1) and (3) as these are read after write statements.
- Data dependence analysis is highly sensitive to the programming languages used in the program. For the type-unsafe languages like C and C++ where pointer can be cast to point to any kind of object, sophisticated analysis is necessary to prove independence between any pair of pointer based memory access.

9.2.3 Tradeoff between Register Usage and Parallelism

- Tradeoff means when one entity is used less then other entity needs to be used more and vice versa.
- The typical tradeoff occurs between the **number of registers** used in intermediate code representation and the **instruction level parallelism**. Following example represents how the usage of number of registers used affect the use of parallel instructions.

1) Consider following code

```
LD t1, a    //t1=a
ST b, t1    //b=t1
LD t2,c     //t2=c
ST d,t2     //d=t2
```

Now suppose we use only one register R0 for assigning the t1 and t2 then we can not execute the above set of instructions in parallel, in fact this code sequence need to be executed sequentially. Thus minimizing the number of registers will affect the parallelism.

2) Following example illustrates that to achieve the instruction level parallelism, we need to have large number of registers.

Consider the expression **(a * b) + c + (d + e)**

Now the above computation is possible with the help of three registers

```
LD r1 , a // r1 = a
LD r2 , b // r2 = b
MUL r1 , r1 , r2 // r1=r1*r2
LD r2 , c // r2 = c
ADD r1 , r1 , r2 // r1=r1+r2
LD r2 , d // r2= d
LD r3 , e // r3= e
ADD r2 , r2 , r3 //r2 = r2+r3
ADD r1 , r1 , r2 // r1 = r1+r2
```

- But when we use only three registers to accomplish above expression evaluation, we have to execute the above code serially, the instruction level parallelism is not possible over here. To achieve the instruction level parallelism, we must use multiple number of registers to store the results of subexpressions. It can be achieved as follows -

Execution of tasks in Parallel

Task 1	Task 2	Task 3	Task 4	Task 5
r1 = a	r2 = b	r3 = c	r4 = d	r5 = e
r6 = r1 * r2	r7 = r4 + r5			
r8 = r6 + r3 r9 = r8 + r7				

9.2.4 Phase ordering between Register Allocation and Code Scheduling

- There are two important activities that influence each other - **Register allocation** and **code scheduling**.

- If registers are allocated before scheduling, then the resulting code tend to have many storage dependences which cause problems to code scheduling. On the other hand, if the code scheduling is done before register allocation then instruction level parallelism can not be achieved. In such a situation what a compiler should do first whether register allocation or code scheduling?
- In order to address this issue we must consider the characteristics of program being compiled. There are two types of applications - **numeric applications** and **non numeric applications**.
- The non numeric applications do not require high degree of parallelism. Hence it is possible to have code scheduling first and assign registers to temporary variables.
- In case of numeric applications, there are large number of expressions. So we can adopt hierarchical approach for these applications. That means, starting with the **inner most loop**, instructions are scheduled first with the help of pseudo-registers. Physical register allocation is applied after scheduling. This process is repeated for the code in the **outer loop**.

9.2.5 Control Dependence

- An instruction i1 is said to be control dependent on instruction i2 if outcome of i2 determines whether i1 is to be executed.
- For example - Consider following code statement

```
if(x>y)
        a=x
else
b=x
```

- Here the statements a=x and b=x is control dependent on the condition 'x>y'
- Similarly in the following while statement

```
while(c)
        S;
```

- The body S is control dependent on c

9.2.6 Speculative Execution Support

- Speculative execution is an optimization technique in which a processor performs a series of tasks before it is prompted to, in order to have the information ready if it is required at any point.

- There are three commonly used techniques that are used for speculative execution support -

1) **Prefetching :**
   - Prefetching is a technique in which data is fetched from memory to cache before it is used.
   - The prefetch instruction indicates that the data being fetched from memory to cache is most likely to be used in near future.

2) **Poison Bits :**
   - Poison bit is a kind of **tag** marked to indicate that the instruction is speculative.
   - Each register on the machine is accompanied with a poison bit.
   - If illegal memory access occurs then the processor does not raise the exception immediately but it simply sets the poison bit of the destination register.
   - An **exception** is raised only if the contents of the register with marked poison bit is used.

3) **Predicated Execution**
   - Predicated instruction is like a normal instruction with some extra operand. The instruction is executed only if the predicate is found to be true.
   - For example -

Consider following code

```
if(x==0)
y=z * t
```

If we use the registers R1, R2, R4 and R5 for the variables x,y,z and t respectively then the machine code for the above code fragment is

```
(1)   MUL R3, R4, R5
(2)   CMOVZ R2,R3,R1
```

The first instruction is Multiplication operation in which R4*R5= z*t and the result is stored in register R3.

The CMOVZ is a predicate instruction. The meaning of this instruction is contents of R3 register are moved to R2(i.e. for variable y) if R1(i.e. meant for x) is zero. Thus if the condition x=0 gets true then only multiplication operation will be performed.

- By using the predicate execution, the series of control dependence can be replaced by data dependence.
- The predicate instruction is costly because the data needs to be fetched and decoded for execution of this instruction. This activity requires resources to be utilized. Hence predicate execution should not be used aggressively unless machine has sufficient number of resources.

9.2.7 Basic Machine Model

- The basic machine model is represented as M={R,T} where
  - **R** stands for hardware resources. It is denoted as a vector R={r1, r2, ...} where ri is the number of units available of the i^{th} kind of resource. The types of resources are - memory access units, Arithmetic Logic Units and floating point functional units.
  - **T** stands for type of operation. Various types of operations are arithmetic operations, loads and stores.
- Each operation has a set of input operands, set of output operands and resource requirements.
- Associated with each input operand is an **input latency** indicating when the input value must be available. The input operand has zero latency, meaning that the values are needed immediately at the clock when the operation is issued.
- Similarly associated with each output operand is the output latency which shows that the result is available when the operation gets started.
- There exists a two dimensional table called resource reservation table RT_t. The width of this table is number of different kinds of resources available and length is the duration over which resources are used by the operation.
- The entry $RT_t[i,j]$ is the number of units of jth resource used by the operation type t. Here i indicates the clocks.
- For any t, i and j the $RT_t[i,j]$ must be less than or equal to R[j] i.e. number of resources of type j that the machine has.
- Every machine operation occupy only one unit of resource at the time an operation is issued. But some operations may have more than one functional units. For instance - in multiply and add operation , the multiplication is done at first clock and the addition is done at the second clock.
- In case of the fully pipelined operations, the operations are performed at every clock.

Review Questions

1. *What is phase ordering between register allocation and code scheduling.*
2. *Explain the techniques used in support of speculative execution of the instructions.*
3. *What are the three types of data dependence that occur ?*
4. *Explain the basic machine model used in instruction level parallelism.*

9.3 Basic - Block Scheduling

In this section we will get introduced with the simple scheduling algorithm associated with basic block. The basic block contains finite number of machine instructions. This is **list scheduling algorithm**.

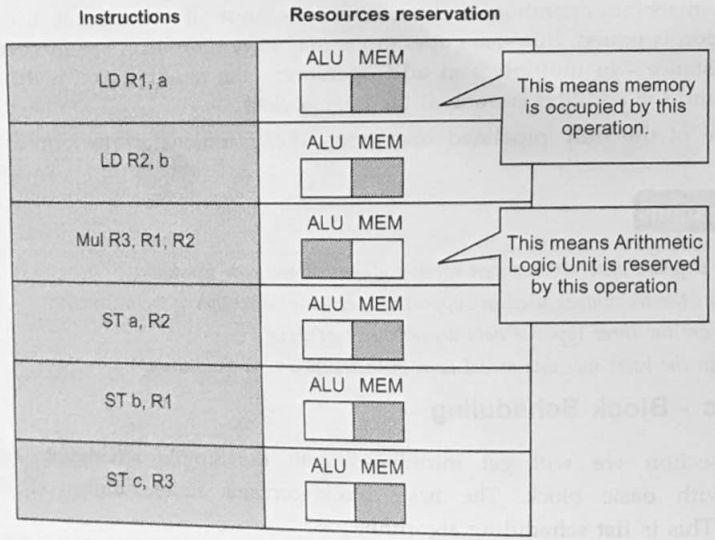
9.3.1 Data Dependence Graph

- The data dependence graph is a collection of **nodes(N) and edges(E)**. It is denoted as G = (N,E).
- The **nodes** represent the **machine instructions**. The meaning of **edge i->j** is that edge is present among the nodes when instruction j has **data dependence** on instruction i
- In the data dependence graph the **edge is labelled with minimum delay interval** between when(i) may initiate and when (j) may initiate. The delay is measured in clock cycles.
- For simplicity we consider a machine model that executes two operations .
  - One type of operation is basic arithmetic operation(ALU). It has following syntax

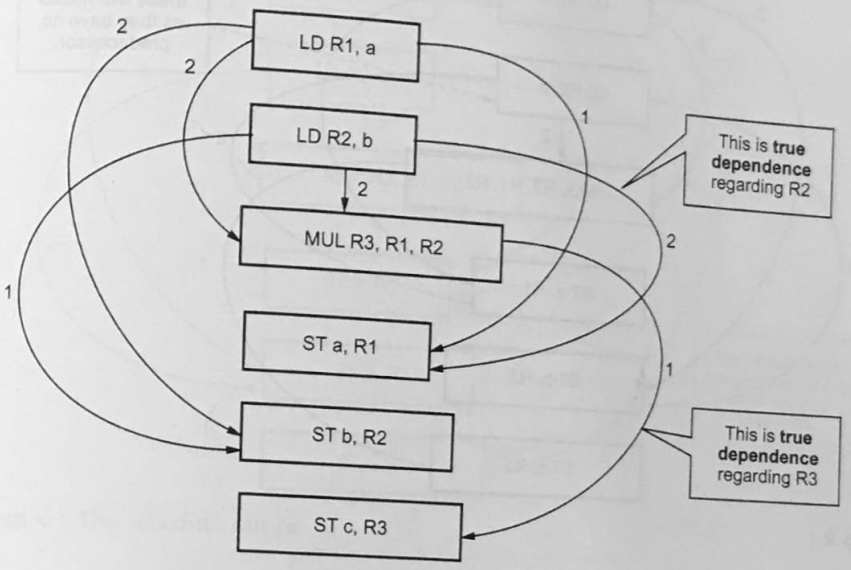
    Op dest, src1, src2
  - Another type of operation is either load or store operation. It has following syntax:

    LD dest, addr
    ST addr, src

    The load operation takes **two clocks** and store requires **one clock** to complete.
  - The arithmetic operation requires one unit of ALU and LD/ST requires one unit of MEM(memory buffer)
- To represent the data dependence graph consider following code fragment.



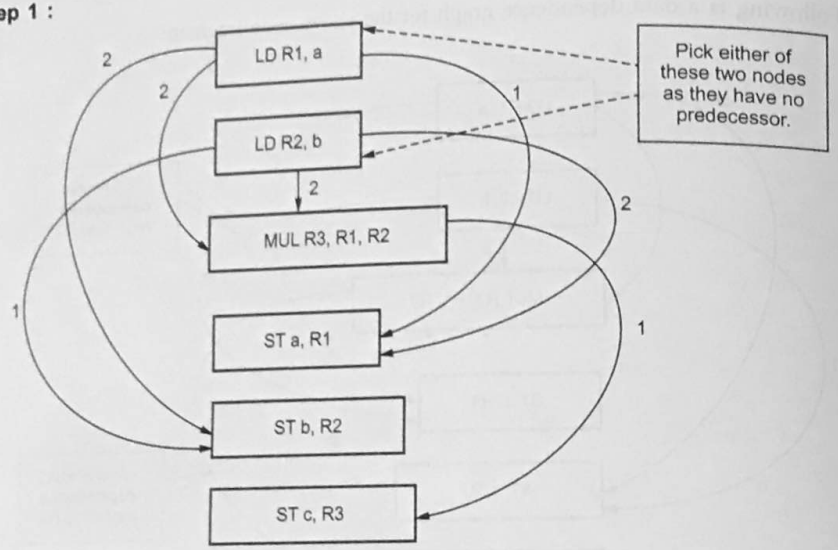
Following is a data dependence graph for the above code fragment



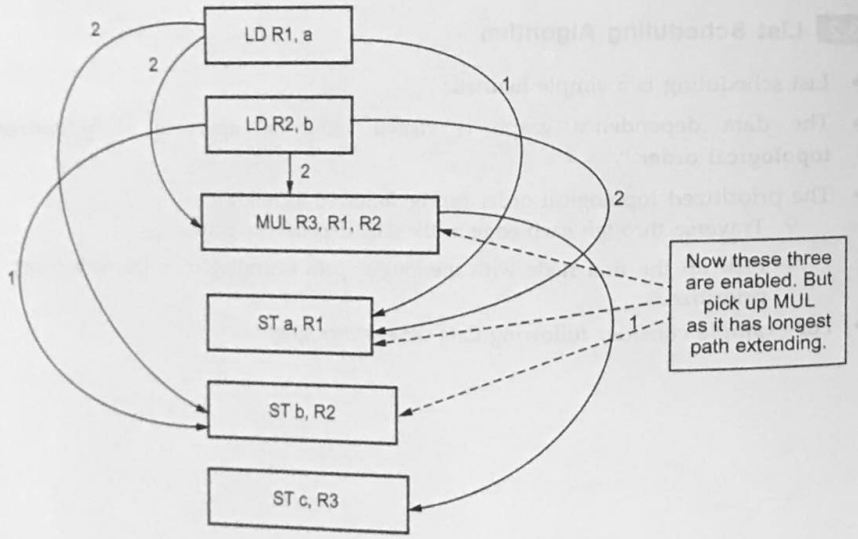
9.3.2 List Scheduling Algorithm

- List scheduling is a simple heuristic.
- The data dependence graph is visited using the approach of **"prioritized topological order "**.
- The prioritized topological order can be followed as follows -
  - Traverse through each edge in the data dependence graph.
  - Pick up the first node with the longest path extending from the node being prioritized.
- For example consider following data dependence graph

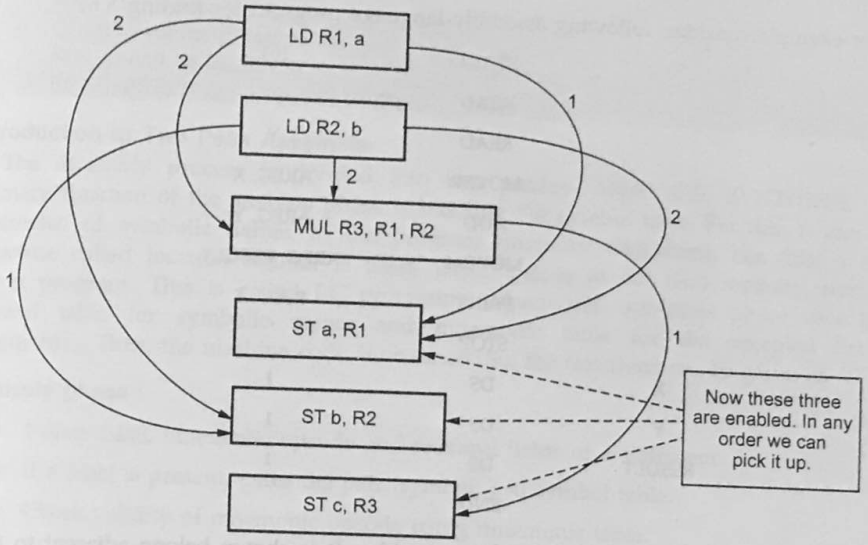
Step 1 :



Step 2 :



Step 3 :



Step 4 : The schedule can be

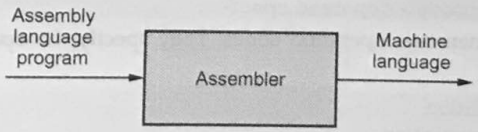
LD R1,a	
LD R2,b	
ST b, R1	
MUL R3,R1,R2	ST a, R2
ST c, R3	

Review Question

1. *Explain basic block scheduling with suitable example.*

9.4 Pass Structure of Assembler

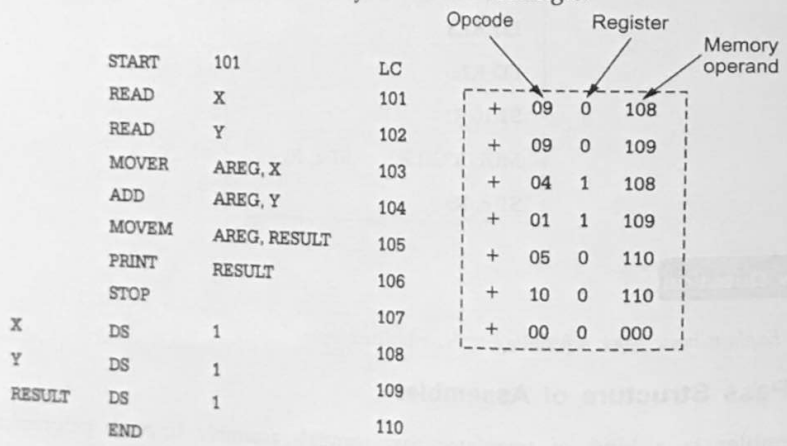
Assembler is a kind of translator that converts assembly language program into machine language.



For example consider, following assembly language program for finding X+Y

```
        START     101
        READ      X
        READ      Y
        MOVER     AREG, X
        ADD       AREG, Y
        MOVEM     AREG, RESULT
        PRINT     RESULT
        STOP
X       DS        1
Y       DS        1
RESULT  DS        1
        END
```

The machine code is generated by the assembler. It is shown belong adjacent to the source code. The machine code is marked by the dotted rectangle.



	START	101	LC			
	READ	X	101	+	09	0	108
	READ	Y	102	+	09	0	109
	MOVER	AREG, X	103	+	04	1	108
	ADD	AREG, Y	104	+	01	1	109
	MOVEM	AREG, RESULT	105	+	05	0	110
	PRINT	RESULT	106	+	10	0	110
	STOP		107	+	00	0	000
X	DS	1	108				
Y	DS	1	109				
RESULT	DS	1	110				
	END						

An assembly level statement has the following format:

[label] <opcode><operand spec>

opcode represents mnemonic operation codes. They specify the operation.

For example -

STOP: stops execution
ADD: Arithmetic operation
SUB : Arithmetic Operation

MULT: Arithmetic Operation
MOVER: register[] memory
MOVEM: memory' register.
COMP: sets condition codes
BC: Branch on condition
READ, PRINT: Reading and printing.

Introduction to Two Pass Assembler

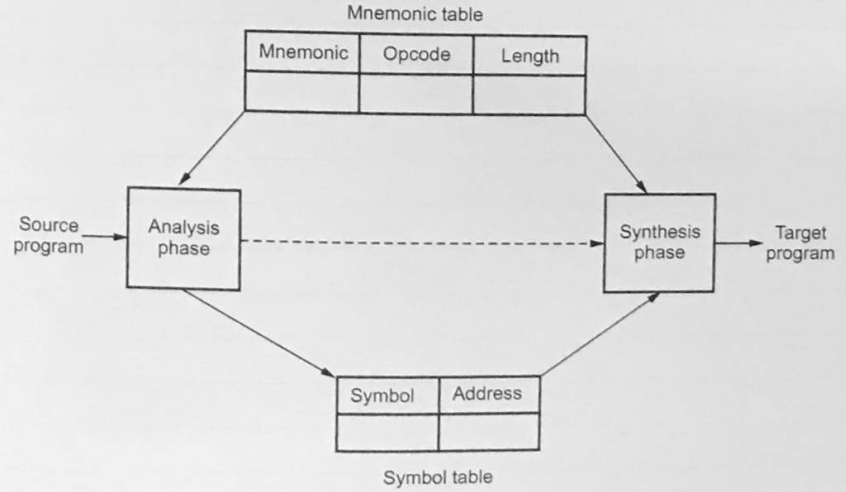
The assembly process is divided into two phases- ANALYSIS, SYNTHESIS. The primary function of the analysis phase is building the symbol table. For this, it uses the addresses of symbolic names in the program (memory allocation). For this, a data structure called location counter is used, which points to the next memory word of target program. This is called LC processing. Meanwhile, synthesis phase uses both symbol table for symbolic names and mnemonic table for the accepted list of mnemonics. Thus, the machine code is obtained. So, the functions can be given as :

Analysis phase :

- Isolate label, mnemonic opcode and operand fields of a statement.
- If a label is present, enter the pair (symbol,) to symbol table.
- Check validity of mnemonic opcode using mnemonic table.
- Perform LC processing.

Synthesis phase :

- Obtain machine code for the mnemonic from the mnemonic table.
- Obtain address of memory operand from symbol table. Synthesize the machine instruction. The phases can be represented as :



Functions of Two pass Assembler

The two pass assembler performs the following functions. It performs some function in pass 1 and some functions in pass 2.

Pass 1

1) Assign address to all statements in the assembly language program.
2) Save the address with label for use in pass 2.
3) Define symbols and literals.
4) Determine the length of machine instructions
5) Keep track of location counter.
6) Process some assembler directions or operations.

Pass 2

1) Perform processing of assembler directives which are not done during the pass 1.
2) Generate the object program.

Review Question

1. *What is assembler ? Explain the two pass assembler with suitable block diagram.*

❑❑❑

Notes