

LECTURE NOTES

ON

COMPILER DESIGN

2018 – 2019

III B. Tech II Semester (JNTUA-R15)

Mrs. G. Indiravathi, Assistant Professor



**CHADALAWADA RAMANAMMA
ENGINEERING COLLEGE**

(AUTONOMOUS)

Chadalawada Nagar, Renigunta Road, Tirupati – 517 506

**Department of Computer Science
and Engineering**

UNIT -I

UNIT -I

PART A: INTRODUCTION

1. OVERVIEW OF LANGUAGE PROCESSING SYSTEM

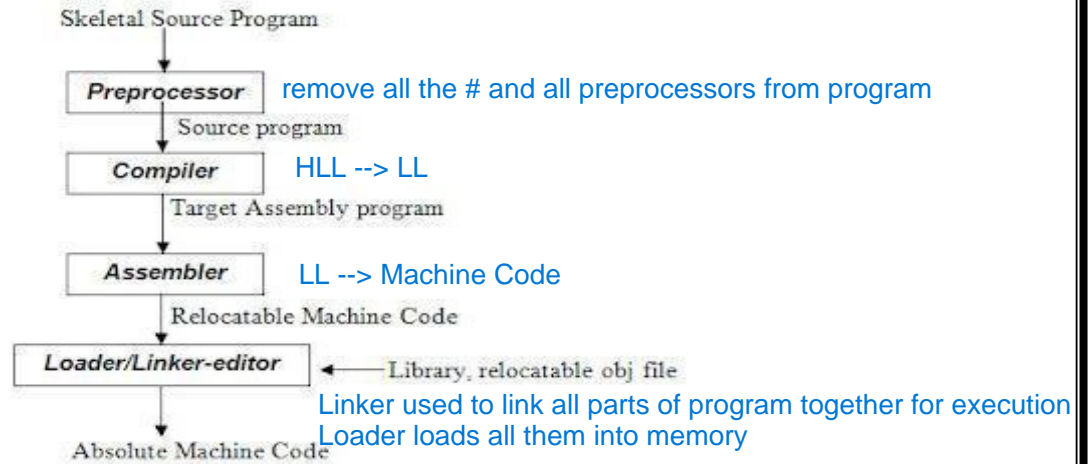


Fig 1.1 Language –processing System

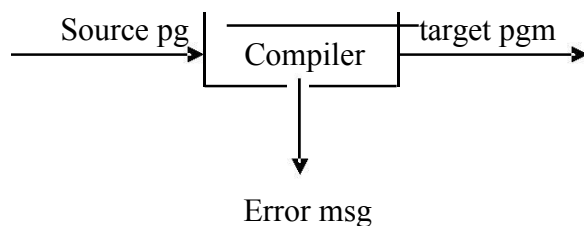
Preprocessor

A preprocessor produce input to compilers. They may perform the following functions.

1. **Macro processing:** A preprocessor may allow a user to define macros that are short hands for longer constructs.
2. **File inclusion:** A preprocessor may include header files into the program text.
3. **Rational preprocessor:** these preprocessors augment older languages with more modern flow-of-control and data structuring facilities.
4. **Language Extensions:** These preprocessor attempts to add capabilities to the language by certain amounts to build-in macro

Compiler

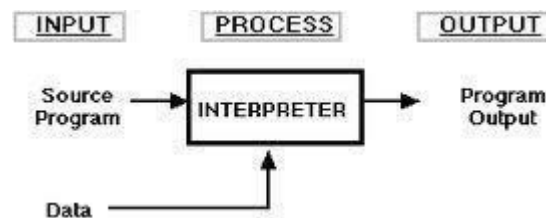
Compiler is a translator program that translates a program written in (HLL) the source program and translates it into an equivalent program in (MLL) the target program. As an important part of a compiler is error showing to the programmer.



Executing a program written in HLL programming language is basically of two parts. the source program must first be compiled translated into a object program. Then the results object program is loaded into a memory executed.

ASSEMBLER: programmers found it difficult to write or read programs in machine language. They begin to use a mnemonic (symbols) for each machine instruction, which they would subsequently translate into machine language. Such a mnemonic machine language is now called an assembly language. Programs known as assembler were written to automate the translation of assembly language into machine language. The input to an assembler program is called source program, the output is a machine language translation (object program).

INTERPRETER: An interpreter is a program that appears to execute a source program as if it were machine language.



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 various 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.

Loader and Link-editor:

Once the assembler produces an object program, that program must be placed into memory and executed. The assembler could place the object program directly in memory and transfer control to it, thereby causing the machine language program to be executed. This would waste core by leaving the assembler in memory while the user's program was being executed. Also the programmer would have to retranslate his program with each execution, thus wasting translation time. To overcome these problems of wasted translation time and memory, system programmers developed another component called loader.

“A loader is a program that places programs into memory and prepares them for execution.” It would be more efficient if subroutines could be translated into object form the loader could “relocate” directly behind the user's program. The task of adjusting programs so they may be placed in arbitrary core locations is called relocation. Relocation loaders perform four functions.

TRANSLATOR

A translator is a program that takes as input a program written in one language and produces as output a program in another language. Besides program translation, the translator performs another very important role, the error-detection. Any violation of the HLL specification would be detected and reported to the programmers. Important roles of a translator are:

- 1 Translating the hll program input into an equivalent ml program.
- 2 Providing diagnostic messages wherever the programmer violates specification of the hll.

TYPE OF TRANSLATORS:-

- Interpreter
- Compiler
- preprocessor

LIST OF COMPILERS

1. Ada compilers
 2. ALGOL compilers
 3. BASIC compilers
 4. C# compilers
 5. C compilers
 6. C++ compilers
 7. COBOL compilers
 8. Java compilers
-

2. PHASES OF A COMPILER:

A compiler operates in phases. A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation. The phases of a compiler are shown in below

There are two phases of compilation.

- Analysis (Machine Independent/Language Dependent)
- Synthesis (Machine Dependent/Language independent) Compilation process is partitioned into no-of-sub processes called 'phases'.

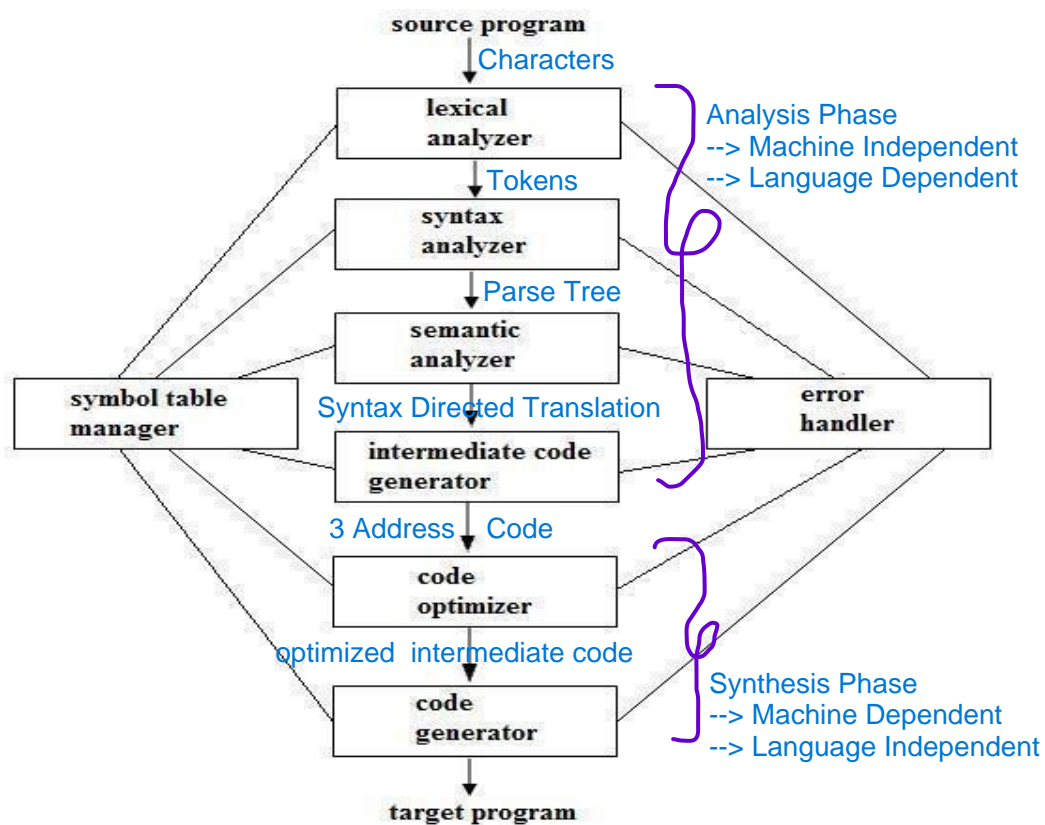


Fig 1.5 Phases of a compiler

1 Lexical Analysis:-

LA or Scanners reads the source program one character at a time, carving the source program into a sequence of automatic units called **tokens**.

2 Syntax Analysis:-

The second stage of translation is called syntax analysis or parsing. In this phase expressions, statements, declarations etc... are identified by using the results of lexical analysis. Syntax analysis is aided by using techniques based on formal grammar of the programming language.

3 Semantic Analysis Check whether it follows the rules of lang.

It checks for the semantic consistency, also keep track of identifiers, their types and expressions.

Eg: `sum = a+b;`
`int a; double Sum; char b;`

--> Here the code is Syntactically Right but Semantically wrong as their are different data types

Properties of intermediate code

- It should be easy to produce.
- It should be easy to translate into target program.

three address code.
t1 = inttofloat (5)
t2 = id3
* t1
t3 = id2 + t2
id1 = t3

4 Intermediate Code Generations:-

An intermediate representation of the final machine language code is produced. This phase bridges the analysis and synthesis phases of translation.

--> represent of final m/c code produced

5 Code Optimization:-

This is optional phase described to improve the intermediate code so that the output runs faster and takes less space.

6 Code Generation:-

The last phase of translation is code generation. A number of optimizations to Reduce the length of machine language program are carried out during this phase. The output of the code generator is the machine language program of the specified computer.

Intermediate instructions are translated into a sequence of machine instructions

Table Management (or) Book-keeping:- to store all the information about identifiers used in the program

This is the portion to keep the names used by the program and records essential information about each. The data structure used to record this information called a

'Symbol Table'. Whenever an identifier is detected in any of the phases, it is stored in the symbol table

Error Handlers:-

It is invoked when a flaw error in the source program is detected. The output of LA is a stream of tokens, which is passed to the next phase, the syntax analyzer or parser. The SA groups the tokens together into syntactic structure called as **expression**. Expression may further be combined to form statements. The syntactic structure can be regarded as a tree whose leaves are the token called as parse trees.

The parser has two functions. It checks if the tokens from lexical analyzer, occur in pattern that are permitted by the specification for the source language. It also imposes on tokens a tree-like structure that is used by the sub-sequent phases of the compiler.

Example, if a program contains the expression **A+/B** after lexical analysis this expression might appear to the syntax analyzer as the token sequence **id+/id**. On seeing the /, the syntax analyzer should detect an error situation, because the presence of these two adjacent binary operators violates the formulations rule of an expression.

Syntax analysis is to make explicit the hierarchical structure of the incoming token stream by identifying which parts of the token stream should be grouped.

Example, (A/B*C has two possible interpretations.)

1- divide A by B and then multiply by C or

2- multiply B by C and then use the result to divide A.

Each of these two interpretations can be represented in terms of a parse tree.

Intermediate Code Generation:-

The intermediate code generation uses the structure produced by the syntax analyzer to create a stream of simple instructions. Many styles of intermediate code are

possible. One common style uses instruction with one operator and a small number of operands. The output of the syntax analyzer is some representation of a parse tree. The intermediate code generation phase transforms this parse tree into an intermediate language representation of the source program.

Code Optimization:-

This is optional phase described to improve the intermediate code so that the output runs faster and takes less space. Its output is another intermediate code program that does the same job as the original, but in a way that saves time and / or spaces.

/* 1, Local Optimization:-

There are local transformations that can be applied to a program to make an improvement. For example,

If $A > B$ goto $L2$

Goto $L3$ $L2 :$

This can be replaced by a single statement If $A < B$ goto $L3$

Another important local optimization is the elimination of common sub-expressions

$A := B + C + D$

$E := B + C + F$

Might be evaluated as

$T1 := B + C$

$A := T1 + D$

$E := T1 + F$

Take this advantage of the common sub-expressions $B + C$.

Loop Optimization:-

Another important source of optimization concerns about **increasing the speed of loops**. A typical loop improvement is to move a computation that produces the same result each time around the loop to a point, in the program just before the loop is entered.*/

Code generator :-

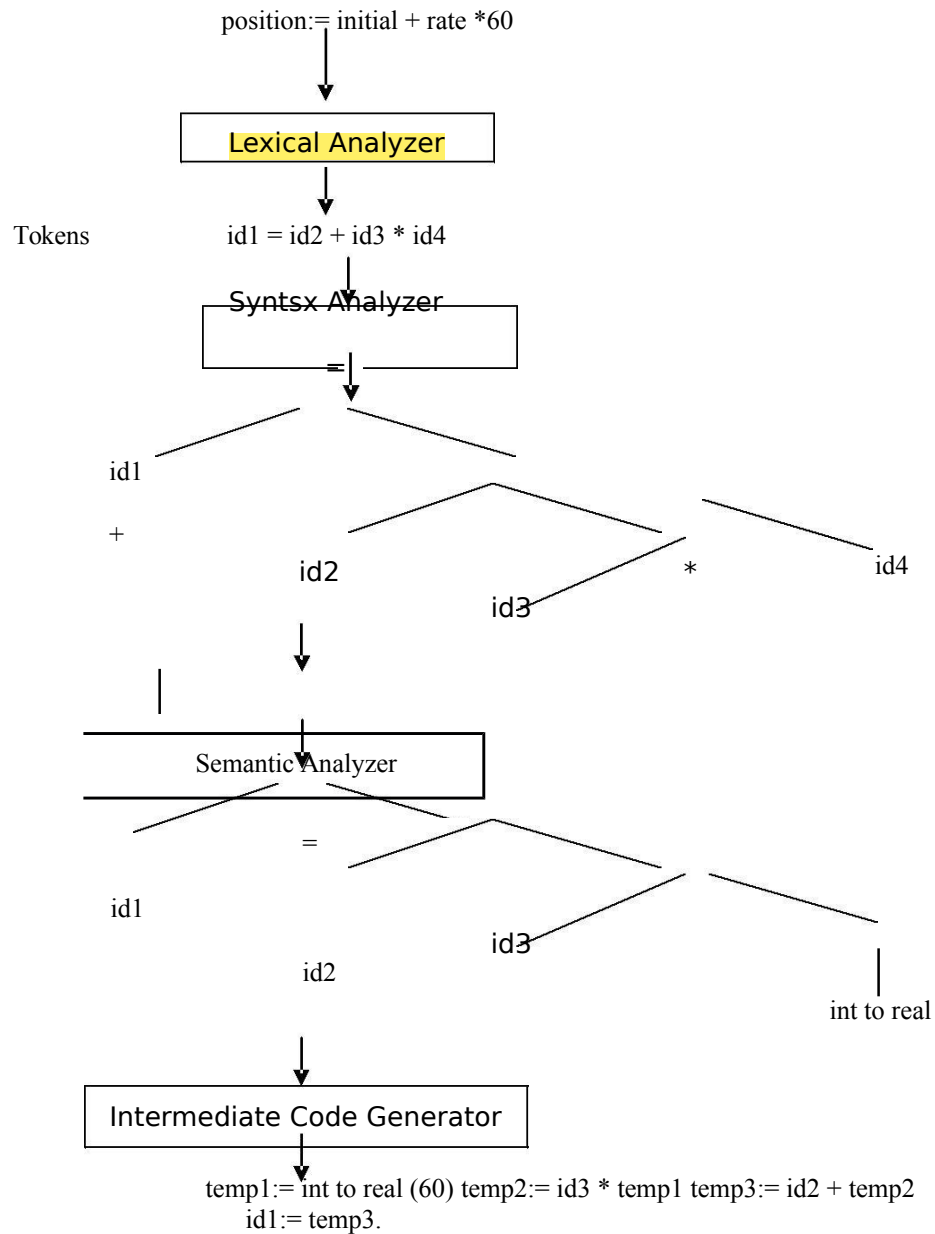
C produces the object code by deciding on the memory locations for data, selecting code to access each data and selecting the registers in which each computation is to be done. Many computers have only a few high speed registers in which computations can be performed quickly. A good code generator would attempt to utilize registers as efficiently as possible.

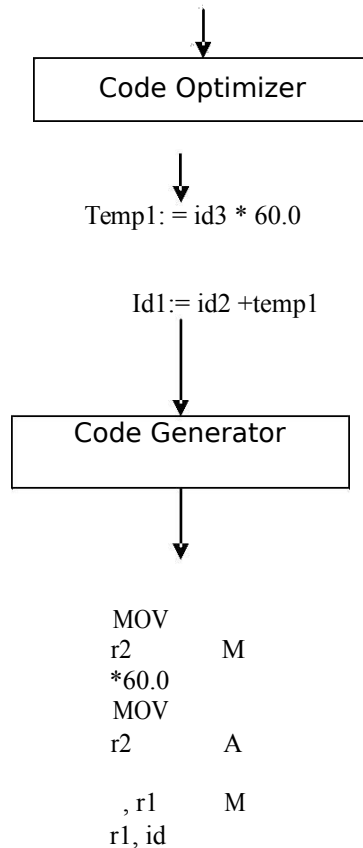
Error Handling :-

One of the most **important functions of a compiler is the detection and reporting of errors in the source program**. The error message should allow the programmer to determine exactly where the errors have occurred. Errors may occur in all or the phases of a compiler.

Whenever a phase of the compiler discovers an error, it must report the error to the error handler, which issues an appropriate diagnostic msg. Both of the table-management and error-handling routines interact with all phases of the compiler.

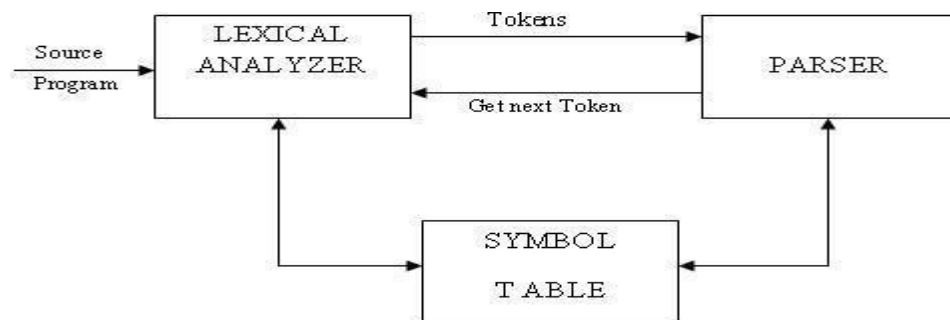
Example:





2.1 LEXICAL ANALYZER:

The LA is the first phase of a compiler. Lexical analysis is called as linear analysis or scanning. In this phase the stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of characters having a collective meaning.



Upon receiving a 'get next token' command from the parser, the lexical analyzer

reads the input character until it can identify the next token. The LA return to the parser representation for the token it has found. The representation will be an integer code, if the token is a simple construct such as parenthesis, comma or colon.

LA may also perform certain secondary tasks as the user interface. One such task is stripping out from the source program the commands and white spaces in the form of blank, tab and new line characters. Another is correlating error message from the compiler with the source program.

121

Token, Lexeme, Pattern:

Token: Token is a sequence of characters that can be treated as a single logical entity. Typical tokens are,

1) Identifiers 2) keywords 3) operators 4) special symbols 5) constants

Pattern: A set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.

Lexeme: A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

Example:

Description of token

Token	lexeme	pattern
const	const	const
if	if	If

relation	<, <=, =, >, >=, >	< or <= or = or > or >= or letter followed by letters & digit
i	pi	any numeric constant
nun	3.14	any character b/w "and "except"
literal	"core"	pattern

A pattern is a rule describing the set of lexemes that can represent a particular token in source program.

Lexical Errors:

Lexical errors are the errors thrown by the lexer when unable to continue. Which means that there's no way to recognise a lexeme as a valid token for you lexer? Syntax errors, on the other side, will be thrown by your scanner when a given set of **already** recognized valid tokens don't match any of the right sides of your grammar rules. Simple panic-mode error handling system requires that we return to a high-level parsing function when a parsing or lexical error is detected.

Error-recovery actions are:

- Delete one character from the remaining input.
- Insert a missing character in to the remaining input.
- Replace a character by another character.
- Transpose two adjacent characters.

- **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.
- 7. **Example of compiler:** Borland c compiler or Turbo C compiler compiles the programs written in C or C++.

4. REGULAR EXPRESSIONS:

: SPECIFICATION OF TOKENS

There are 3 specifications of tokens:

- 1) Strings
- 2) Language
- 3) Regular expression

Strings and Languages

An **alphabet** or character class is a finite set of symbols.

A **string** over an alphabet is a finite sequence of symbols drawn from that alphabet. A **language** is any countable set of strings over some fixed alphabet.

In language theory, the terms "sentence" and "word" are often used as synonyms for "string." The length of a string s , usually written $|s|$, is the number of occurrences of symbols in s . For example, banana is a string of length six. The empty string, denoted ϵ , is the string of length zero.

Operations on strings

The following string-related terms are commonly used:

1. A **prefix** of string s is any string obtained by removing zero or more symbols from the end of strings.
For example, ban is a prefix of banana.
 2. A **suffix** of string s is any string obtained by removing zero or more symbols from the beginning of s .
For example, nana is a suffix of banana.
 3. A **substring** of s is obtained by deleting any prefix and any suffix from s .
For example, nan is a substring of banana.
 4. The **proper prefixes, suffixes, and substrings** of a string s are those prefixes, suffixes, and substrings, respectively of s that are not ϵ or not equal to s itself.
 5. A **subsequence** of s is any string formed by deleting zero or more not necessarily consecutive positions of s
-

For example, baan is a subsequence of banana.

Operations on languages:

The following are the operations that can be applied to languages:

1. Union
2. Concatenation
3. Kleene closure
4. Positive closure

The following example shows the operations on strings:

Let $L = \{0,1\}$ and $S = \{a,b,c\}$

Union : $L \cup S = \{0,1,a,b,c\}$

Concatenation : $L.S = \{0a,1a,0b,1b,0c,1c\}$

Kleene closure : $L^* = \{\epsilon, 0, 1, 00, \dots\}$

Positive closure : $L^+ = \{0, 1, 00, \dots\}$

Regular Expressions:

Each regular expression r denotes a language $L(r)$.

Here are the rules that define the regular expressions over some alphabet Σ and the languages that those expressions denote:

1. ϵ is a regular expression, and $L(\epsilon)$ is $\{\epsilon\}$, that is, the language whose sole member is the empty string.
 2. If 'a' is a symbol in Σ , then 'a' is a regular expression, and $L(a) = \{a\}$, that is, the language with one string, of length one, with 'a' in its one position.
 3. Suppose r and s are regular expressions denoting the languages $L(r)$ and $L(s)$. Then,
 - $(r)|(s)$ is a regular expression denoting the language $L(r) \cup L(s)$.
 - $(r)(s)$ is a regular expression denoting the language $L(r)L(s)$.
 - $(r)^*$ is a regular expression denoting $(L(r))^*$.
 - (r) is a regular expression denoting $L(r)$.
 4. The unary operator $*$ has highest precedence and is left associative.
 5. Concatenation has second highest precedence and is left associative. has lowest precedence and is left associative.
-

REGULAR DEFINITIONS:

For notational convenience, we may wish to give names to regular expressions and to define regular expressions using these names as if they were symbols.

Identifiers are the set or string of letters and digits beginning with a letter. The following regular definition provides a precise specification for this class of string.

Example-1,

$Ab^*|cd^?$ Is equivalent to $(a(b^*)) | (c(d^?))$ Pascal identifier

Letter - $A | B | \dots | Z | a | b | \dots | z$ Digits - $0 | 1 | 2 | \dots | 9$

Id - letter (letter / digit)*

Shorthand's

Certain constructs occur so frequently in regular expressions that it is convenient to introduce notational shorthands for them.

1. One or more instances (+):

- The unary postfix operator $+$ means “one or more instances of”.
- If r is a regular expression that denotes the language $L(r)$, then $(r)^+$ is a regular expression that denotes the language $(L(r))^+$
- Thus the regular expression a^+ denotes the set of all strings of one or more a 's.
- The operator $^+$ has the same precedence and associativity as the operator * .

2. Zero or one instance (?):

- The unary postfix operator $?$ means “zero or one instance of”.
- The notation $r?$ is a shorthand for $r | \epsilon$.
- If ' r ' is a regular expression, then $(r)?$ is a regular expression that denotes the language $L(r) \cup \{\epsilon\}$.

3. Character Classes:

- The notation $[abc]$ where a , b and c are alphabet symbols denotes the regular expression $a | b | c$.
 - Character class such as $[a - z]$ denotes the regular expression $a | b | c | d | \dots | z$.
 - We can describe identifiers as being strings generated by the regular expression, $[A-Za-z][A-Za-z0-9]^*$
-

Non-regular Set

A language which cannot be described by any regular expression is a non-regular set. Example: The set of all strings of balanced parentheses and repeating strings cannot be described by a regular expression. This set can be specified by a context-free grammar.

RECOGNITION OF TOKENS:

Consider the following grammar fragment: $\text{stmt} \rightarrow \text{if expr then stmt}$
 $|\text{if expr then stmt else stmt} | \epsilon$

$\text{expr} \rightarrow \text{term relop term} | \text{term term} \rightarrow \text{id} | \text{num}$

where the terminals if, then, else, relop, id and num generate sets of strings given by the following regular definitions:

If $\rightarrow \text{if}$

then $\rightarrow \text{then}$

else $\rightarrow \text{else}$

relop $\rightarrow <|<=|<>|>|>=$

id $\rightarrow \text{letter}(\text{letter}|\text{digit})^*$

num $\rightarrow \text{digit}^+ (. \text{digit}^+)? (\text{E}(+|-)? \text{digit}^+)?$

For this language fragment the lexical analyzer will recognize the keywords if, then, else, as well as the lexemes denoted by relop, id, and num. To simplify matters, we assume keywords are reserved; that is, they cannot be used as identifiers.

Lexeme	Token Name	Attribute Value
Any ws		
if	if	
then	then	
else		
Any id	id	pointer to table entry
Any number	number	pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	ET
<>	relop	NE

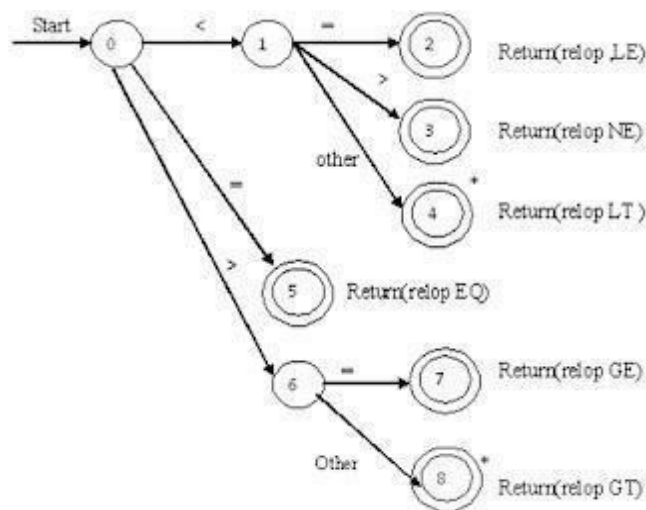
TRANSITION DIAGRAM:

Transition Diagram has a collection of nodes or circles, called states. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns. Edges are

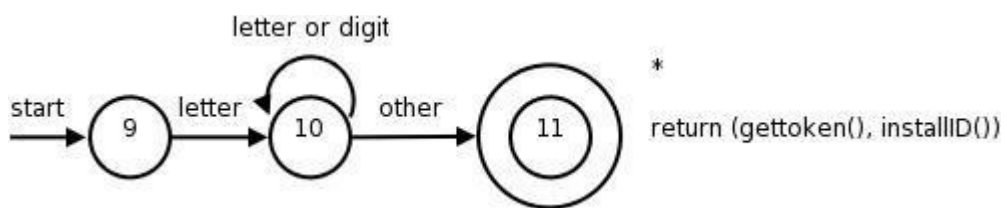
directed from one state of the transition diagram to another. each edge is labeled by a symbol or set of symbols. If we are in one state s , and the next input symbol is a , we look for an edge out of state s labeled by a . if we find such an edge, we advance the forward pointer and enter the state of the transition diagram to which that edge leads.

Some important conventions about transition diagrams are

1. Certain states are said to be accepting or final. These states indicate that a lexeme has been found, although the actual lexeme may not consist of all positions b/w the lexeme Begin and forward pointers. We always indicate an accepting state by a double circle.
2. In addition, if it is necessary to return the forward pointer one position, then we shall additionally place a $*$ near that accepting state.
3. One state is designed the state, or initial state, it is indicated by an edge labeled "start" entering from nowhere. the transition diagram always begins in the state before any input symbols have been used.



As an intermediate step in the construction of a LA, we first produce a stylized flowchart, called a transition diagram. Position in a transition diagram, are drawn as circles and are called as states.



The above TD for an identifier, defined to be a letter followed by any no of letters or digits. A sequence of transition diagram can be converted into program to look for the tokens specified by the diagrams. Each state gets a segment of code.

Automata:

Automation is defined as a system where information is transmitted and used for performing some functions without direct participation of man.

1. An automation in which the output depends only on the input is **called automation without memory.**
2. An automation in which the output depends on the input and state also is **called as automation with memory.**
3. An automation in which the output depends only on the state of the machine is **called a Moore machine.**
4. An automation in which the output depends on the state and input at any instant of time is **called a mealy machine.**

DESCRIPTION OF AUTOMATA

1. An automata has a mechanism to read input from input tape,
2. Any language is recognized by some automation, Hence these automation are basically language 'acceptors' or 'language recognizers'.

Types of Finite Automata

- Deterministic Automata
- Non-Deterministic Automata.

Deterministic Automata:

A deterministic finite automata has at most one transition from each state on any input. A DFA is a special case of a NFA in which:-

1. it has no transitions on input ϵ ,
2. Each input symbol has at most one transition from any state.

DFA formally defined by 5 tuple notation $M = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite 'set of states', which is non empty.

Σ is 'input alphabets', indicates input set.

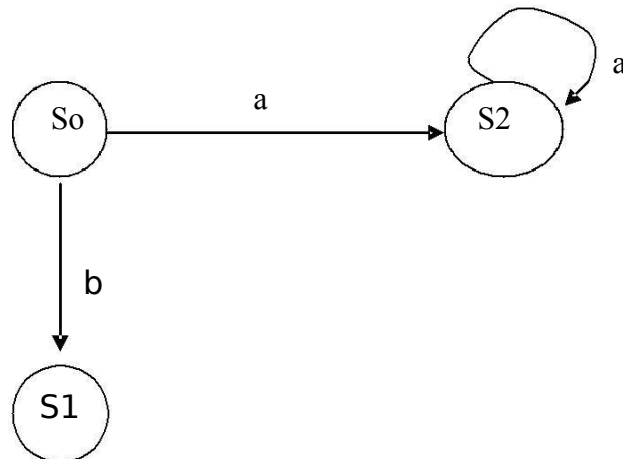
q_0 is an 'initial state' and q_0 is in Q ie, q_0, Σ, Q, F is a set of 'Final states',

δ is a 'transmission function' or mapping function, using this function the next state can be determined.

The regular expression is converted into minimized DFA by the following procedure:

Regular expression \rightarrow NFA \rightarrow DFA \rightarrow Minimized DFA

The Finite Automata is called DFA if there is only one path for a specific input from current state to next state.



From state S0 for input 'a' there is only one path going to S2. similarly from so there is only one path for input going to S1.

Nondeterministic Automata:

A NFA is a mathematical model consists of



A set of states S .



A set of input symbols Σ .



A transition is a move from one state to another.



A set of states F distinguished as accepting (or



A state s_0 that is distinguished as the start (or initial) state

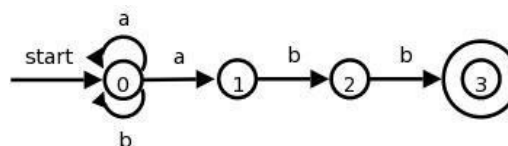


final) state. A number of transition to a single symbol.

A NFA can be diagrammatically represented by a labeled directed graph, called a transition graph, in which the nodes are the states and the labeled edges represent the transition function.

This graph looks like a transition diagram, but the same character can label two or more transitions out of one state and edges can be labeled by the special symbol ϵ as well as input symbols.

The transition graph for an NFA that recognizes the language $(a|b)^*abb$ is shown



CROSS COMPILER

--> A cross compiler is a type of compiler, that generates machine code targeted to run on a system different than the one generating it.

Eg: A compiler that runs in windows platform also generates a code that runs on linux platform is a cross compiler.

--> Translates program for different hardware/platform/machine other than the platform which it is running.

--> It is independent of System/machine and OS

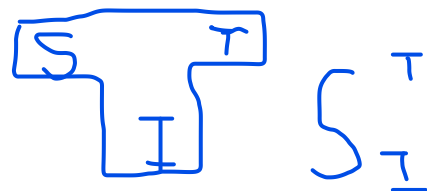
--> Keil is a cross compiler.
and GNU GCC is also an example of cross compiler

Bootstrapping

A compiler can be characterized by three Lang: 1.Source Language(S) 2.Target Lang(T) 3.Implementation Lang(I)

It is the process by which simple language is used to translate more complicated program, which in turn may handle an even more complicated program.

in bootstrapping this compiler is represented of as "T" diagram



5. Bootstrapping: It is used to produce self hosting compiler i.e, it can compile its own source code

When a computer is first turned on or restarted, a special type of absolute loader, called as bootstrap loader is executed. This bootstrap loads the first program to be run by the computer usually an operating system. The bootstrap itself begins at address 0 in the memory of the machine. It loads the operating system (or some other program) starting at address 80. After all of the object code from device has been loaded, the bootstrap program jumps to address 80, which begins the execution of the program that was loaded.

Such loaders can be used to run stand-alone programs independent of the operating system or the system loader. They can also be used to load the operating system or the loader itself into memory.

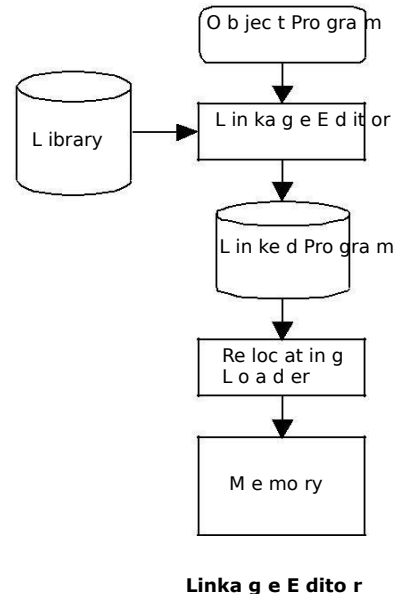
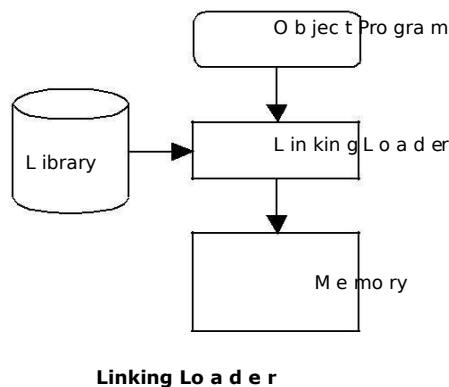
Loaders are of two types:

- Linking loader.
- Linkage editor.

Linkage loaders, perform all linking and relocation at load time.

Linkage editors, perform linking prior to load time and dynamic linking, in which the linking function is performed at execution time.

A linkage editor performs linking and some relocation; however, the linkaged program is written to a file or library instead of being immediately loaded into memory. This approach reduces the overhead when the program is executed. All that is required at load time is a very simple form of relocation.



6. Pass And Phases Of Translation:

Phases: (Phases are collected into a front end and back end)

Frontend:

The front end consists of those phases, or parts of phase, that depends primarily on the source language and is largely independent of the target machine. These normally include lexical and syntactic analysis, the creation of the symbol table, semantic analysis, and the generation of intermediate code.

A certain amount of code optimization can be done by front end as well. the front end also includes the error handling tha goes along with each of these phases.

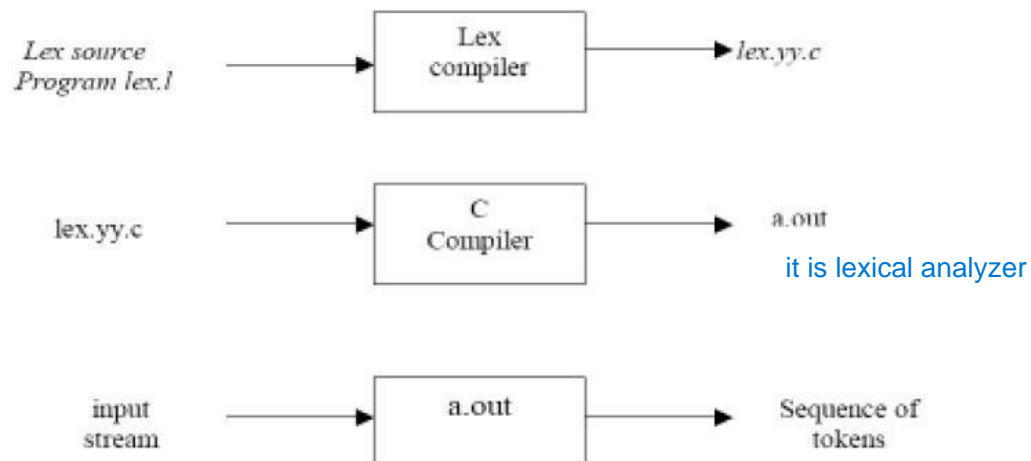
Back end:

The back end includes those portions of the compiler that depend on the target machine and generally, these portions do not depend on the source language .

7. Lexical Analyzer Generator:

Creating a lexical analyzer with Lex:

- First, a specification of a lexical analyzer is prepared by creating a program `lex.l` in the Lex language. Then, `lex.l` is run through the Lex compiler to produce a C program `lex.yy.c`.
- Finally, `lex.yy.c` is run through the C compiler to produce an object program `a.out`, which is the lexical analyzer that transforms an input stream into a sequence of tokens.



Lex Specification

Lex tool -
its main job is to break up an i/p stream into token
it is a tool for automatically generating a lexer

A Lex program consists of three parts:

```
{ definitions }  
%%  
{ rules }  
%%  
{ user subroutines }
```

```
%{  
    #include<stdio.h>  
    int vow=0;  
    int cons=0;  
}%  
%%  
{aeiou AEIOU} {vow++}  
{a-z A-Z} {cons++}  
%%  
{  
    print  
}
```

- **Definitions** include declarations of variables, constants, and regular definitions
- **Rules** are statements of the form $p_1 \{action_1\} p_2 \{action_2\} \dots p_n \{action\}$
- where p_i is regular expression and $action_i$ describes what action the lexical analyzer should take when pattern p_i matches a lexeme. Actions are written in C code.
- **User subroutines** are auxiliary procedures needed by the actions. These can be compiled separately and loaded with the lexical analyzer.

8. INPUT BUFFERING

The LA scans the characters of the source program one at a time to discover tokens. Because of large amount of time can be consumed scanning characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character.

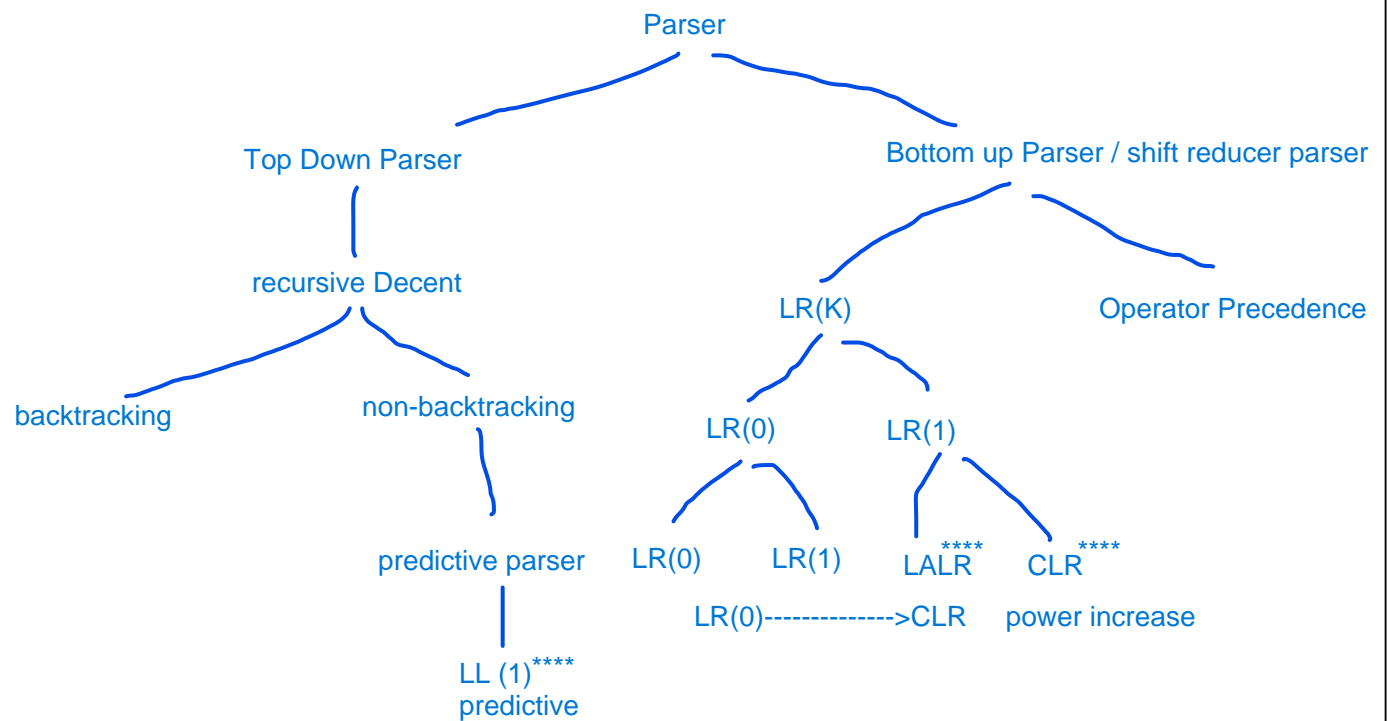
Buffering techniques: 1. Buffer pairs
2. Sentinels

The lexical analyzer scans the characters of the source program one at a time to discover tokens. Often, however, many characters beyond the next token may have to be examined before the next token itself can be determined. For this and other reasons, it is desirable for the lexical analyzer to read its input from an input buffer. Figure shows a buffer divided into two halves of, say 100 characters each. One pointer marks the beginning of the token being discovered. A look ahead pointer scans ahead of the beginning point, until the token is discovered. We view the position of each pointer as being between the character last read and the character next to be read. In practice each buffering scheme adopts one convention either a pointer is at the symbol last read or the symbol it is ready to read.

Token beginnings look ahead pointer, The distance which the look ahead pointer may have to travel past the actual token may be large.

For example, in a PL/I program we may see: DECALRE (ARG1, ARG2... ARG n) without knowing whether DECLARE is a keyword or an array name until we see the character that follows the right parenthesis.

UNIT - II



UNIT - II

TOPDOWN PARSING

1. Context-free Grammars: Definition:

Formally, a context-free grammar G is a 4-tuple $G = (V, T, P, S)$, where:

1. V is a finite set of variables (or nonterminals). These describe sets of “related” strings.
2. T is a finite set of terminals (i.e., tokens).
3. P is a finite set of productions, each of the form

$$A \rightarrow \alpha$$

where $A \in V$ is a variable, and $\alpha \in (V \cup T)^*$ is a sequence of terminals and nonterminals. $S \in V$ is the start symbol.

Example of CFG:

$$E \Rightarrow EAE \mid (E) \mid -E \mid \text{id} \quad A \Rightarrow + \mid - \mid * \mid /$$

Where E, A are the non-terminals while $\text{id}, +, *, -, /,(,)$ are the terminals. 2. Syntax analysis:

In syntax analysis phase the source program is analyzed to check whether it conforms to the source language's syntax, and to determine its phrase structure. This phase is often separated into two phases:

- Lexical analysis: which produces a stream of tokens?
- Parser: which determines the phrase structure of the program based on the context-free grammar for the language?

PARSING:

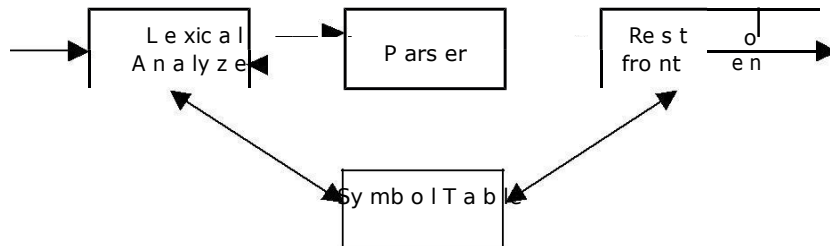
Parsing is the activity of checking whether a string of symbols is in the language of some grammar, where this string is usually the stream of tokens produced by the lexical analyzer. If the string is in the grammar, we want a parse tree, and if it is not, we hope for some kind of error message explaining why not.

There are two main kinds of parsers in use, named for the way they build the parse trees:

- **Top-down:** A top-down parser attempts to construct a tree from the root, applying productions forward to expand non-terminals into strings of symbols.
 - **Bottom-up:** A Bottom-up parser builds the tree starting with the leaves, using productions in reverse to identify strings of symbols that can be grouped together.
-

In both cases the construction of derivation is directed by scanning the input sequence from left to right, one symbol at a time.

Parse Tree:



A parse tree is the graphical representation of the structure of a sentence according to its grammar.

Example:

Let the production P is:

$$E \rightarrow T \mid E + T$$

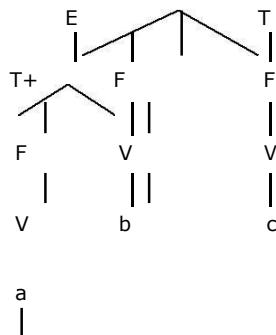
$$T \rightarrow F \mid T * F$$

$$F \rightarrow V \mid (E)$$

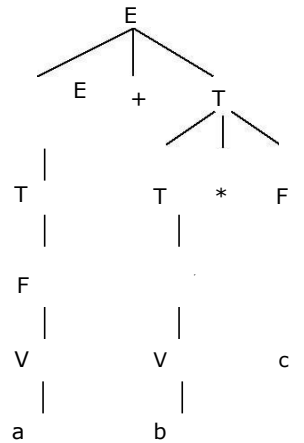
$$V \rightarrow a \mid b \mid c \mid d$$

The parse tree may be viewed as a representation for a derivation that filters out the choice regarding the order of replacement.

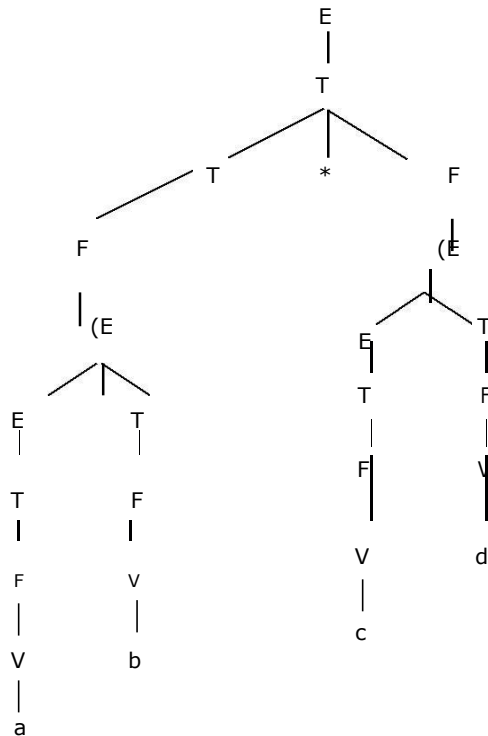
Parse tree for $a * b + c$



Parse tree for $a + b * c$ is:



Parse tree for $(a * b) * (c + d)$



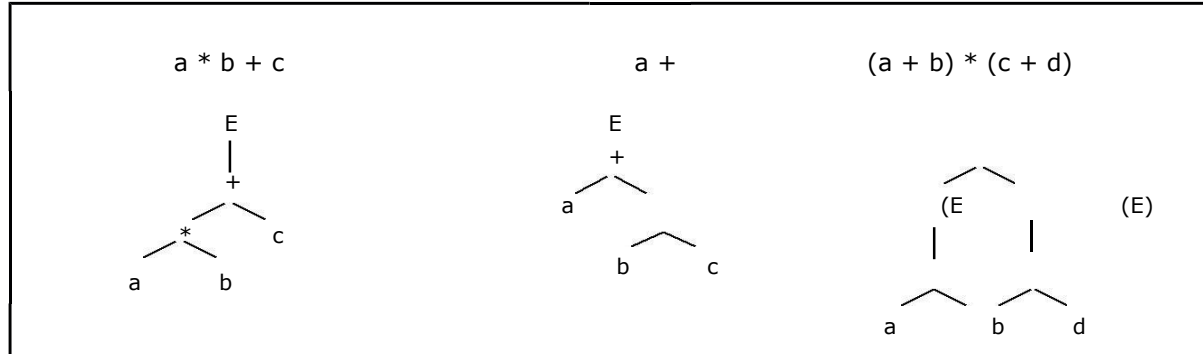
SYNTAX TREES:

Parse tree can be presented in a simplified form with only the relevant structure information by:

- Leaving out chains of derivations (whose sole purpose is to give operators difference precedence).

- Labeling the nodes with the operators in question rather than a non-terminal.

The simplified Parse tree is sometimes called as structural tree or syntax tree.



Syntax Trees

Syntax Error Handling:

If a compiler had to process only correct programs, its design & implementation would be greatly simplified. But programmers frequently write incorrect programs, and a good compiler should assist the programmer in identifying and locating errors. The programs contain errors at many different levels.

For example, errors can be:

- 1) Lexical – such as misspelling an identifier, keyword or operator
- 2) Syntactic – such as an arithmetic expression with un-balanced parentheses.
- 3) Semantic – such as an operator applied to an incompatible operand.
- 4) Logical – such as an infinitely recursive call.

Much of error detection and recovery in a compiler is centered around the syntax analysis phase. The goals of error handler in a parser are:

- It should report the presence of errors clearly and accurately.
- It should recover from each error quickly enough to be able to detect subsequent errors.
- It should not significantly slow down the processing of correct programs.

Ambiguity:

Several derivations will generate the same sentence, perhaps by applying the same productions in a different order. This alone is fine, but a problem arises if the same sentence has two distinct parse trees. A grammar is ambiguous if there is any sentence with more than one parse tree.

Any parser for an ambiguous grammar has to choose somehow which tree to return. There are a number of solutions to this; the parser could pick one arbitrarily, or we can provide

some hints about which to choose. Best of all is to rewrite the grammar so that it is not ambiguous.

There is no general method for removing ambiguity. Ambiguity is acceptable in spoken languages. Ambiguous programming languages are useless unless the ambiguity can be resolved.

Fixing some simple ambiguities in a grammar:

Ambiguous	language	unambiguous
(i) $A \rightarrow B \mid AA$	Lists of one or more B's $C \rightarrow A \mid E$	$A \rightarrow BC$
(ii) $A \rightarrow B \mid A;A$	Lists of one or more B's with punctuation $C \rightarrow ;A \mid E$	$A \rightarrow BC$
(iii) $A \rightarrow B \mid AA \mid E$	lists of zero or more B's	$A \rightarrow BA \mid E$

Any sentence with more than two variables, such as (arg, arg, arg) will have multiple parse trees.

Left Recursion:

If there is any non terminal A, such that there is a derivation $A \Rightarrow^+ A\alpha$ for some string α , then grammar is left recursive.

Algorithm for eliminating left Recursion:

- Group all the A productions together like this: $A \Rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$

Where,

A is the left recursive non-terminal,

α is any string of terminals and

β is any string of terminals and non terminals that does not begin with A.

- Replace the above A productions by the following: $A \Rightarrow \beta_1 A^I \mid \beta_2 A^I \mid \dots \mid \beta_n A^I$

$A^I \Rightarrow \alpha_1 A^I \mid \alpha_2 A^I \mid \dots \mid \alpha_m A^I \mid \epsilon$ Where, A^I is a new non terminal.

Top down parsers cannot handle left recursive grammars.

If our expression grammar is left recursive:

- This can lead to non termination in a top-down parser.
- for a top-down parser, any recursion must be right recursion.
- we would like to convert the left recursion to right recursion.

Example 1:

Remove the left recursion from the production: $A \rightarrow A \alpha \mid \beta$


Left Recursive.
Eliminate

Applying the transformation yields:

$$\begin{array}{l}
 A \rightarrow \beta A^I \\
 A^I \rightarrow \alpha A^I \mid \epsilon \\
 \uparrow \\
 \text{Remaining part after A.}
 \end{array}$$

Example 2:

Remove the left recursion from the productions:

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

Applying the transformation yields:

$$E \rightarrow T E^I$$

$$E^I \rightarrow T E^I \mid \epsilon$$

$$T \rightarrow F T^I$$

$$T^I \rightarrow * F T^I \mid \epsilon$$

Example 3:

Remove the left recursion from the productions:

$$E \rightarrow E + T \mid E - T \mid T$$

$$T \rightarrow T * F \mid T / F \mid F$$

Applying the transformation yields:

$$E \rightarrow T E^I$$

$$E^I \rightarrow + T E^I \mid - T E^I \mid \epsilon$$

$$T \rightarrow F T^I$$

$$T^I \rightarrow * F T^I \mid / F T^I \mid \epsilon$$

Example 4:

Remove the left recursion from the productions:

$$S \rightarrow A a \mid b$$

$$A \rightarrow A c \mid S d \mid \epsilon$$

1. The non terminal S is left recursive because $S \rightarrow A a \rightarrow S d a$ But it is not immediate left recursive.
2. Substitute S-productions in $A \rightarrow S d$ to obtain:

$$A \rightarrow A c \mid A a d \mid b d \mid \epsilon$$
3. Eliminating the immediate left recursion:

$$\begin{aligned}
 S &\rightarrow A a \mid b \\
 A &\rightarrow b d A^I \mid A^I \\
 A^I &\rightarrow c A^I \mid a d A^I \mid \epsilon
 \end{aligned}$$

Example 5:

Consider the following grammar and eliminate left recursion. $S \rightarrow A a \mid b$

$$A \rightarrow S c \mid d$$

The nonterminal S is left recursive in two steps: $S \rightarrow A a \rightarrow S c a \rightarrow A a c a \rightarrow S c a c a \dots$

Left recursion causes the parser to loop like this, so remove: Replace $A \rightarrow S c \mid d$ by $A \rightarrow A a c \mid b c \mid d$

and then by using Transformation rules: $A \rightarrow b c A^I \mid d A^I$

$$A^I \rightarrow a c A^I \mid \epsilon$$

Left Factoring:

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.

When it is not clear which of two alternative productions to use to expand a non-terminal A, we may be able to rewrite the productions to defer the decision until we have some enough of the input to make the right choice.

Algorithm:

For all $A \in$ non-terminal, find the longest prefix α that occurs in two or more right-hand sides of A.

If $\alpha \neq \epsilon$ then replace all of the A productions, $A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \mid \dots \mid \alpha \beta_n \mid r$

With

$$A \rightarrow \alpha A^I \mid r$$

$$A^I \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n \mid \epsilon$$

Where, A^I is a new element of non-terminal. Repeat until no common prefixes remain.

It is easy to remove common prefixes by left factoring, creating new non-terminal.

For example consider:

$$V \rightarrow \alpha \beta \mid \alpha r \text{ Change to:}$$

$$V \rightarrow \alpha V^I \quad V^I \rightarrow \beta \mid r$$

Example 1:

Eliminate Left factoring in the grammar: $S \rightarrow V := \text{int}$

$$V \rightarrow \text{alpha} \text{ '[' int ']' } \mid \text{alpha}$$

Becomes:

$$\begin{aligned} S &\rightarrow V := \text{int} \\ V &\rightarrow \text{alpha } V^I \\ V^I &\rightarrow '[\text{ int }] | \epsilon \end{aligned}$$

TOP DOWN PARSING:

Top down parsing is the construction of a **Parse tree by starting at start symbol** and “guessing” each derivation until we reach a string that matches input. That is, construct tree from root to leaves.

The advantage of top down parsing is that a parser can directly be written as a program. Table-driven top-down parsers are of minor practical relevance. Since bottom-up parsers are more powerful than top-down parsers, bottom-up parsing is practically relevant.

For example, let us consider the grammar to see how top-down parser works:

$$\begin{aligned} S &\rightarrow \text{if } E \text{ then } S \text{ else } S \mid \text{while } E \text{ do } S \mid \text{print} \\ E &\rightarrow \text{true} \mid \text{False} \mid \text{id} \end{aligned}$$

Note
TDP internally uses left most derivation
it constructed for the grammar if it is free from
ambiguity and left recursion

The input token string is: If id then while true do print else print.

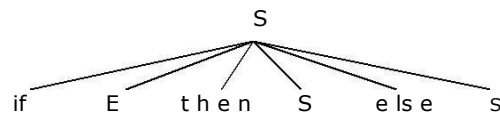
1. Tree:

S

Input: if id then while true do print else print.

Action: Guess for S.

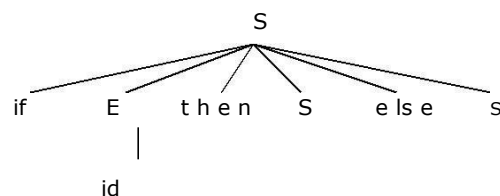
2. Tree:



Input: if id then while true do print else print.

Action: if matches; guess for E.

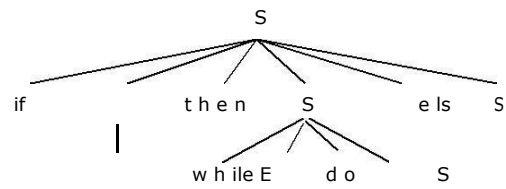
3. Tree:



Input: id then while true do print else print.

Action: id matches; then matches; guess for S.

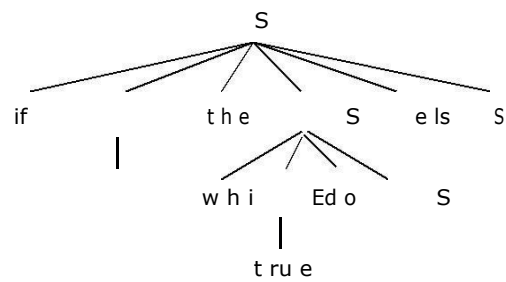
4. Tree:



Input: while true do print else print.

Action: while matches; guess for E.

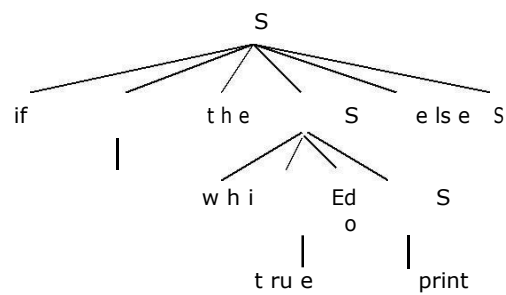
5. Tree:



Input: true do print else print

Action: true matches; do matches; guess S.

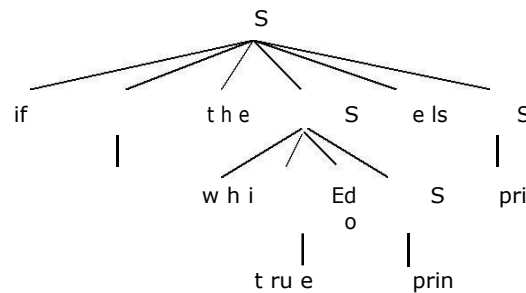
6. Tree:



Input: print else print.

Action: print matches; else matches; guess for S.

7. Tree:



Input: print.

Action: print matches; input exhausted; done.

Recursive Descent Parsing:

Top-down parsing can be viewed as an attempt to **find a left most derivation** for an input string. Equivalently, it can be viewed as an attempt to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder.

The special case of **recursive –descent parsing, called predictive parsing**, where no backtracking is required. The general form of top-down parsing, called recursive descent, that may involve backtracking, that is, making repeated scans of the input.

Recursive descent or predictive parsing works only on grammars where the first terminal symbol of each sub expression provides enough information to choose which production to use.

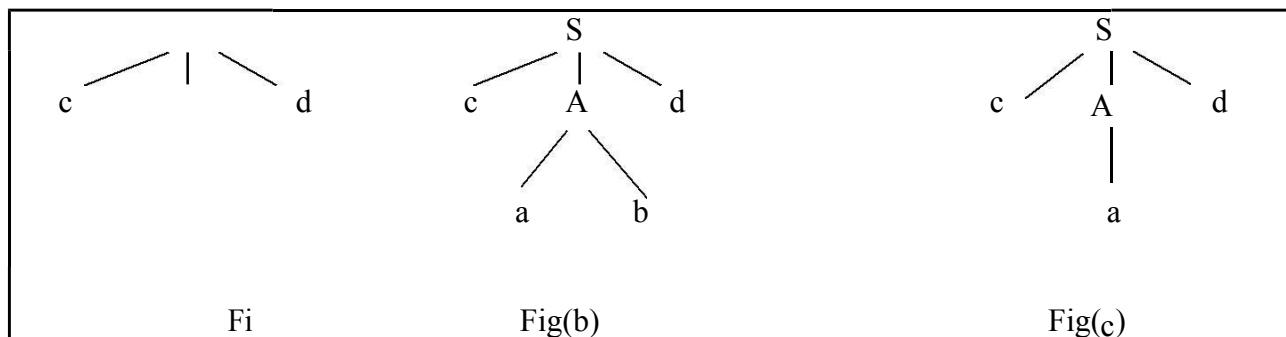
Recursive descent parser is a top down parser involving backtracking. It makes a repeated scans of the input. Backtracking parsers are not seen frequently, as backtracking is very needed to parse programming language constructs.

Example: consider the grammar

$S \rightarrow cAd$

$A \rightarrow ab|a$

And the input string $w=cad$. To construct a parse tree for this string top-down, we initially create a tree consisting of a single node labeled scan input pointer points to c , the first symbol of w . we then use the first production for S to expand tree and obtain the tree of Fig(a).



The left most leaf, labeled c, matches the first symbol of w, so we now advance the input pointer to a, the second symbol of w, and consider the next leaf, labeled A. We can then expand A using the first alternative for A to obtain the tree in Fig (b). we now have a match for the second input symbol so we advance the input pointer to d, the third, input symbol, and compare d against the next leaf, labeled b. since b does not match the d, we report failure and go back to A to see where there is any alternative for A c that we have not tried but that might produce a match.

In going back to A, we must reset the input pointer to position 2, we now try second alternative for A to obtain the tree of Fig(c). The leaf matches second symbol of w and the leaf d matches the third symbol.

The left recursive grammar can cause a recursive- descent parser, even one with backtracking, to go into an infinite loop. That is, when we try to expand A, we may eventually find ourselves again trying to expand A without having consumed any input.

Predictive Parsing:

Predictive parsing is **top-down parsing without backtracking** or look a head. For many languages, make perfect guesses (avoid backtracking) by using 1-symbol look-a-head. i.e., if:
 $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n.$

Choose correct α_i by looking at first symbol it derive. If ϵ is an alternative, choose it last.

This approach is **also called as predictive parsing**. There must be at most one production in order to avoid backtracking. If there is no such production then no parse tree exists and an error is returned.

The crucial property is that, the grammar must not be left-recursive.

Predictive parsing works well on those fragments of programming languages in which keywords occurs frequently.

For example:

$stmt \rightarrow \text{if exp then stmt else stmt} \mid \text{while expr do stmt}$
 $\mid \text{begin stmt-list end.}$

then the keywords if, while and begin tell, which alternative is the only one that could possibly succeed if we are to find a statement.

The model of predictive parser is as follows:

```

        pop X from the stack;
        push  $Y_k, Y_{k-1}, \dots, Y_1$  onto the stack, with  $Y_1$  on top; output the
        production  $X \rightarrow Y_1 Y_2 \dots Y_k$ 
    end
    else error()
until  $X = \$$  /*stack is empty*/

```

FIRST and FOLLOW:

The construction of a predictive parser is aided by two functions with a grammar G . these functions, FIRST and FOLLOW, allow us to fill in the entries of a predictive parsing table for G , whenever possible. Sets of tokens yielded by the FOLLOW function can also be used as synchronizing tokens during pannic-mode error recovery.

If α is any string of grammar symbols, let $FIRST(\alpha)$ be the set of terminals that begin the strings derived from α . If $\alpha \Rightarrow \epsilon$, then ϵ is also in $FIRST(\alpha)$.

Define FOLLOW (A), for nonterminals A , to be the set of terminals a that can appear immediately to the right of A in some sentential form, that is, the set of terminals a such that there exist a derivation of the form $S \Rightarrow \alpha A a \beta$ for some α and β . If A can be the rightmost symbol in some sentential form, then $\$$ is in FOLLOW(A).

Computation of FIRST ():

To compute **FIRST(X)** for all grammar symbols X , apply the following rules until no more terminals or ϵ can be added to any FIRST set.

- If X is terminal, then $FIRST(X)$ is $\{X\}$.
- If $X \rightarrow \epsilon$ is production, then add ϵ to $FIRST(X)$.
- If X is nonterminal and $X \rightarrow Y_1 Y_2 \dots Y_k$ is a production, then place a in $FIRST(X)$ if for some i , a is in $FIRST(Y_i)$, and ϵ is in all of $FIRST(Y_1), \dots, FIRST(Y_{i-1})$; that is $Y_1 \dots Y_{i-1} \Rightarrow \epsilon$. if ϵ is in $FIRST(Y_j)$, for all $j=1, 2, 3, \dots, k$, then add ϵ to $FIRST(X)$. for example, everything in $FIRST(Y_1)$ is surely in $FIRST(X)$. if Y_1 does not derive ϵ , then we add nothing more to $FIRST(X)$, but if $Y_1 \Rightarrow \epsilon$, then we add $FIRST(Y_2)$ and so on.

FIRST (A) = FIRST (α_1) U FIRST (α_2) U --- U FIRST (α_n) Where, $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$, are all the productions for A . **FIRST ($A\alpha$) = if $\epsilon \notin FIRST(A)$ then FIRST (A)**
else (FIRST (A) - { ϵ }) U FIRST (α)

Computation of FOLLOW ():

To compute **FOLLOW (A)** for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

- Place \$ in FOLLOW(s), where S is the start symbol and \$ is input right end marker .
- If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except for ϵ is placed in FOLLOW(B).
- If there is production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$ where FIRST (β) contains ϵ (i.e., $\beta \rightarrow \epsilon$), then everything in FOLLOW(A) is in FOLLOW(B).

Example:

Construct the FIRST and FOLLOW for the grammar:

$$A \rightarrow BC \mid EFGH \mid H$$

$$B \rightarrow b$$

$$C \rightarrow c \mid \epsilon$$

$$E \rightarrow e \mid \epsilon$$

$$F \rightarrow CE$$

$$G \rightarrow g$$

$$H \rightarrow h \mid \epsilon$$

Solution:

1. Finding first () set:

$$1. \quad \text{first}(H) = \text{first}(h) \cup \text{first}(\epsilon) = \{h, \epsilon\}$$

$$2. \quad \text{first}(G) = \text{first}(g) = \{g\}$$

$$3. \quad \text{first}(C) = \text{first}(c) \cup \text{first}(\epsilon) = c, \epsilon\}$$

$$4. \quad \text{first}(E) = \text{first}(e) \cup \text{first}(\epsilon) = \{e, \epsilon\}$$

$$5. \quad \text{first}(F) = \text{first}(CE) = (\text{first}(c) - \{\epsilon\}) \cup \text{first}(E)$$

$$= (c, \epsilon\} - \{\epsilon\}) \cup \{e, \epsilon\} = \{c, e, \epsilon\}$$

$$6. \quad \text{first}(B) = \text{first}(b) = \{b\}$$

$$7. \quad \text{first}(A) = \text{first}(BC) \cup \text{first}(EFGH) \cup \text{first}(H)$$

$$= \text{first}(B) \cup (\text{first}(E) - \{\epsilon\}) \cup \text{first}(FGH) \cup \{h, \epsilon\}$$

$$= \{b, h, \epsilon\} \cup \{e\} \cup (\text{first}(F) - \{\epsilon\}) \cup \text{first}(GH)$$

$$= \{b, e, h, \epsilon\} \cup \{C, e\} \cup \text{first}(G)$$

$$= \{b, c, e, h, \epsilon\} \cup \{g\} = \{b, c, e, g, h, \epsilon\}$$

2. Finding follow() sets:

1. $\text{follow}(A) = \{\$ \}$
2. $\text{follow}(B) = \text{first}(C) - \{\epsilon\} \cup \text{follow}(A) = \{C, \$ \}$
3. $\text{follow}(G) = \text{first}(H) - \{\epsilon\} \cup \text{follow}(A)$
 $= \{h, \epsilon\} - \{\epsilon\} \cup \{\$ \} = \{h, \$ \}$
4. $\text{follow}(H) = \text{follow}(A) = \{\$ \}$
5. $\text{follow}(F) = \text{first}(GH) - \{\epsilon\} = \{g\}$
6. $\text{follow}(E) = \text{first}(FGH) - \{\epsilon\} \cup \text{follow}(F)$
 $= ((\text{first}(F) - \{\epsilon\}) \cup \text{first}(GH)) - \{\epsilon\} \cup \text{follow}(F)$
 $= \{c, e\} \cup \{g\} \cup \{g\} = \{c, e, g\}$
7. $\text{follow}(C) = \text{follow}(A) \cup \text{first}(E) - \{\epsilon\} \cup \text{follow}(F)$
 $= \{\$ \} \cup \{e, \epsilon\} \cup \{g\} = \{e, g, \$ \}$

Example 1:

Construct a predictive parsing table for the given grammar or Check whether the given grammar is LL(1) or not.

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F F \rightarrow (E) \mid \text{id}$$

Step 1:

Suppose if the given grammar is left Recursive then convert the given grammar (and ϵ) into non-left Recursive grammar (as it goes to infinite loop).

$$E \rightarrow T E^I$$

$$E^I \rightarrow + T E^I \mid \epsilon \quad T^I \rightarrow F T^I$$

$$T^I \rightarrow * F T^I \mid \epsilon \quad F \rightarrow (E) \mid \text{id}$$

Step 2:

Find the FIRST(X) and FOLLOW(X) for all the variables.

The variables are: $\{E, E^I, T, T^I, F\}$

Terminals are: $\{+, *, (,), \text{id}\}$ and $\$$

Computation of FIRST() sets:

$\text{FIRST}(F) = \text{FIRST}(E) \cup \text{FIRST}(id) = \{(, id\}$
 $\text{FIRST}(T^I) = \text{FIRST}(*FT^I) \cup \text{FIRST}(\epsilon) = \{*, \epsilon\}$
 $\text{FIRST}(T) = \text{FIRST}(FT^I) = \text{FIRST}(F) = \{(, id\}$
 $\text{FIRST}(E^I) = \text{FIRST}(+TE^I) \cup \text{FIRST}(\epsilon) = \{+, \epsilon\}$
 $\text{FIRST}(E) = \text{FIRST}(TE^I) = \text{FIRST}(T) = \{(, id\}$
 Computation of FOLLOW () sets:

	Relevant production
$\text{FOLLOW}(E) = \{\$, \epsilon\} \cup \text{FIRST}(\epsilon) = \{\$, \epsilon\}$	$F \rightarrow (E)$
$\text{FOLLOW}(E^I) = \text{FOLLOW}(E) = \{\$, \epsilon\}$	$E \rightarrow TE^I$
$\text{FOLLOW}(T) = (\text{FIRST}(E^I) - \{\epsilon\}) \cup \text{FOLLOW}(E) \cup \text{FOLLOW}(E^I)$ $= \{+, \epsilon\}$	$E \rightarrow TE^I$ $E^I \rightarrow +TE^I$
$\text{FOLLOW}(T^I) = \text{FOLLOW}(T) = \{+, \epsilon, \$\}$	$T \rightarrow FT^I$
$\text{FOLLOW}(F) = (\text{FIRST}(T^I) - \{\epsilon\}) \cup \text{FOLLOW}(T) \cup \text{FOLLOW}(T^I)$ $= \{*, +, \epsilon\}$	$T \rightarrow T^I$

Step 3:

Construction of parsing table:

Terminals Variables	+		()	id	\$
E			$E \rightarrow TE$		$E \rightarrow TE^I$	
E^I	$E^I \rightarrow +TE^I$			$E^I \rightarrow \epsilon$		$E^I \rightarrow \epsilon$
T			$T \rightarrow FT$		$T \rightarrow FT^I$	
T^I	$T^I \rightarrow \epsilon$	$T^I \rightarrow *F$		$T^I \rightarrow \epsilon$		$T^I \rightarrow \epsilon$
F			$F \rightarrow (E)$		$F \rightarrow id$	

Table 3.1. Parsing Table

Fill the table with the production on the basis of the $\text{FIRST}(\alpha)$. If the input symbol is an ϵ in $\text{FIRST}(\alpha)$, then goto $\text{FOLLOW}(\alpha)$ and fill $\alpha \rightarrow \epsilon$, in all those input symbols.

Let us start with the non-terminal E, $\text{FIRST}(E) = \{(, id\}$. So, place the production $E \rightarrow TE^I$ at (and id.

For the non-terminal E^I , $\text{FIRST}(E^I) = \{+, \epsilon\}$.

So, place the production $E^I \rightarrow +TE^I$ at + and also as there is a ϵ in $\text{FIRST}(E^I)$, see $\text{FOLLOW}(E^I) = \{\$, \epsilon\}$. So write the production $E^I \rightarrow \epsilon$ at the place \$ and).

Similarly:

For the non-terminal T, $\text{FIRST}(T) = \{(, \text{id}\}$. So place the production $T \rightarrow FT^I$ at (and id.

For the non-terminal T^I , $\text{FIRST}(T^I) = \{*, \epsilon\}$

So place the production $T^I \rightarrow *FT^I$ at * and also as there is a ϵ in $\text{FIRST}(T^I)$, see $\text{FOLLOW}(T^I) = \{+, \$,)\}$, so write the production $T^I \rightarrow \epsilon$ at +, \$ and).

For the non-terminal F, $\text{FIRST}(F) = \{(, \text{id}\}$.

So place the production $F \rightarrow \text{id}$ at id location and $F \rightarrow (E)$ at (as it has two productions.

Finally, make all undefined entries as error.

As these were no multiple entries in the table, hence the given grammar is LL(1).

Step 4:

Moves made by predictive parser on the input id + id * id is:

STACK	INPUT	REMARKS
\$ E	id + id * id \$	E and id are not identical; so see E on id in parse table, the production is $E \rightarrow TE^I$; pop E, push E^I and T i.e., move in reverse order.
\$ E^I T	id + id * id \$	See T on id the production is $T \rightarrow FT^I$; Pop T, push T^I and F; Proceed until both are identical.
\$ E^I T^I F	id + id * id \$	$F \rightarrow \text{id}$
\$ E^I T^I id	id + id * id \$	Identical; pop id and remove id from input symbol.
\$ E^I T^I	+ id *	See T^I on +; $T^I \rightarrow \epsilon$ so, pop T^I
\$ E^I	+ id *	See E^I on +; $E^I \rightarrow +TE^I$; push E^I , + and T
\$ E^I T +	+ id *	Identical; pop + and remove + from input symbol.
\$ E^I T	id *	
\$ E^I T^I F	id *	$T \rightarrow FT^I$
\$ E^I T^I id	id *	$F \rightarrow \text{id}$
\$ E^I T^I	*	
\$ E^I T^I F *	*	$T^I \rightarrow *FT^I$
\$ E^I T^I F		
\$ E^I T^I id		$F \rightarrow \text{id}$

$\$ E^I T^I$		$T^I \rightarrow \epsilon$
$\$ E^I$		$E^I \rightarrow \epsilon$
$\$$		Accept.

Table 3.2 Moves made by the parser on input id + id * id

Predictive parser accepts the given input string. We can notice that \$ in input and stuck, i.e., both are empty, hence accepted.

CASE for LL parser

2.6.3 LL (1) Grammar:

1. for no terminal alpha and beta derive string beginning with a

2. at most one of alpha and beta can derive empty string

3. if $\beta \rightarrow \epsilon$ then 'a' does not derive any string with a terminal in FOLLOW(A)

The first L stands for “Left-to-right scan of input”. The second L stands for “Left-most derivation”. The ‘1’ stands for “1 token of look ahead”.

No LL (1) grammar can be ambiguous or left recursive.

If there were no multiple entries in the Recursive decent parser table, the given grammar is LL (1).

If the grammar G is ambiguous, left recursive then the recursive decent table will have atleast one multiply defined entry.

The weakness of LL(1) (Top-down, predictive) parsing is that, must predict which production to use.

Error Recovery in Predictive Parser:

Error recovery is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens appear. Its effectiveness depends on the choice of synchronizing set. The Usage of FOLLOW and FIRST symbols as synchronizing tokens works reasonably well when expressions are parsed.

For the constructed table., fill with **synch** for rest of the input symbols of FOLLOW set and then fill the rest of the columns with **error** term.

Terminal Variables	+	*	()	id	\$
E	error	error	$E \rightarrow TE$	synch	$E \rightarrow TE^I$	synch
E^I	$E^I \rightarrow +TE^I$	error	error	$E^I \rightarrow \epsilon$	error	$E^I \rightarrow \epsilon$
T	synch	error	$T \rightarrow FT$	synch	$T \rightarrow FT^I$	synch
T^I	$T^I \rightarrow \epsilon$	$T^I \rightarrow *F$	error	$T^I \rightarrow \epsilon$	error	$T^I \rightarrow \epsilon$
F	synch	synch	$F \rightarrow (E)$	synch	$F \rightarrow id$	synch

Table3.3 :Synchronizing tokens added to parsing table for table 3.1.

If the parser looks up entry in the table as synch, then the non terminal on top of the stack is popped in an attempt to resume parsing. If the token on top of the stack does not match the input symbol, then pop the token from the stack.

The moves of a parser and error recovery on the erroneous input) id*+id is as follows:

STACK	IN	REMARKS
\$ E) id * +	Error, skip)
\$ E	id * +	
\$ E ^I T	id * +	
\$ E ^I T ^I F	id * +	
\$ E ^I T ^I id	id * +	
\$ E ^I T ^I	* +	
\$ E ^I T ^I F *	* +	
\$ E ^I T ^I F	+	Error; F on + is synch; F has been popped.
\$ E ^I T ^I	+	
\$ E ^I	+	
\$ E ^I T +	+	
\$ E ^I T		
\$ E ^I T ^I F		
\$ E ^I T ^I id		
\$ E ^I T ^I		
\$ E ^I		
\$		Accept.

Example 2: Table 3.4. Parsing and error recovery moves made by predictive parser

Construct a predictive parsing table for the given grammar or Check whether the given grammar is LL(1) or not.

$$\begin{aligned}
 S &\rightarrow iEtSS^I \mid a \\
 S^I &\rightarrow eS \mid \epsilon \\
 E &\rightarrow b
 \end{aligned}$$

Solution:

1. Computation of First () set:

1. First (E) = first (b) = {b}
2. First (S^I) = first (eS) \cup first (ϵ) = {e, ϵ }
3. first (S) = first (iEtSS^I) \cup first (a) = {i, a}

2. Computation of follow() set:

1. follow (S) = {\$} \cup first (S^I) - { ϵ } \cup follow (S) \cup follow (S^I)
= {\$} \cup {e} = {e, \$}
2. follow (S^I) = follow (S) = {e, \$}
3. follow (E) = first (tSS^I) = {t}

3. The parsing table for this grammar is:

	a	b	e	i	t	
S	S \rightarrow a			S \rightarrow iEtSS ^I		
S ^I			S ^I \rightarrow ϵ S ^I \rightarrow e			S ^I \rightarrow ϵ
E		E \rightarrow b				

As the table multiply defined entry. The given grammar is not LL(1).

Example 3:

Construct the FIRST and FOLLOW and predictive parse table for the grammar:

$$S \rightarrow AC\$$$

$$C \rightarrow c \mid \epsilon$$

$$A \rightarrow aBCd \mid BQ \mid \epsilon$$

$$B \rightarrow bB \mid d$$

$$Q \rightarrow q$$

Solution:

1. Finding the first () sets: First (Q) = {q}
- First (B) = {b, d}

$$\text{First}(C) = \{c, \epsilon\}$$

$$\text{First}(A) = \text{First}(aBCd) \cup \text{First}(BQ) \cup \text{First}(\epsilon)$$

$$= \{a\} \cup \text{First}(B) \cup \text{First}(d) \cup \{\epsilon\}$$

$$= \{a\} \cup \text{First}(bB) \cup \text{First}(d) \cup \{\epsilon\}$$

$$= \{a\} \cup \{b\} \cup \{d\} \cup \{\epsilon\}$$

$$= \{a, b, d, \epsilon\} \quad \text{First}(S) = \text{First}(AC\$)$$

$$= (\text{First}(A) - \{\epsilon\}) \cup (\text{First}(C) - \{\epsilon\}) \cup \text{First}(\epsilon)$$

$$= (\{a, b, d, \epsilon\} - \{\epsilon\}) \cup (\{c, \epsilon\} - \{\epsilon\}) \cup \{\epsilon\}$$

$$= \{a, b, d, c, \epsilon\}$$

2. Finding Follow () sets: $\text{Follow}(S) = \{\#\}$

$$\text{Follow}(A) = (\text{First}(C) - \{\epsilon\}) \cup \text{First}(\$) = (\{c, \epsilon\} - \{\epsilon\}) \cup \{\$\}$$

$$\text{Follow}(B) = (\text{First}(C) - \{\epsilon\}) \cup \text{First}(d) \cup \text{First}(Q)$$

$$= \{c\} \cup \{d\} \cup \{q\} = \{c, d, q\} \quad \text{Follow}(C) = (\text{First}(\$) \cup \text{First}(d) = \{d, \$\})$$

$$\text{Follow}(Q) = (\text{First}(A) = \{c, \$\})$$

3. The parsing table for this grammar is:

	a	b	c	D	q	\$	
S	S AC\$	S AC	S AC	S AC		S AC	
A	A aBCd	A BQ	A ε	A BQ		A ε	
B		B bB		B d			
C			C c	C ε		C ε	
Q					Q q		

4. Moves made by predictive parser on the input abdcde\$ is:

Stack symbol	Input	Remarks
#S	abdcde\$#	S → AC\$
#\$CA	abdcde\$#	A → aBCd
#\$CdCBa	abdcde\$#	Pop a
#\$CdCB	bdcde\$#	B → bB
#\$CdCBb	bdcde\$#	Pop b
#\$CdCB	dcde\$#	B → d
#\$CdCd	dcde\$#	Pop d
#\$CdC	cde\$#	C → c
#\$Cde	cde\$#	Pop C
#\$Cd	de\$#	Pop d
#\$C	c\$#	C → c
#\$c	c\$#	Pop c
#\$	\$#	Pop \$
#	#	Accepted

BOTTOM UP PARSING

1. BOTTOM UP PARSING:

Bottom-up parser builds a derivation by working from the input sentence back towards the start symbol S. Right most derivation in reverse order is done in bottom-up parsing.

(The point of parsing is to construct a derivation. A derivation consists of a series of rewrite steps)

$$S \Rightarrow r_0 \Rightarrow r_1 \Rightarrow r_2 \Rightarrow \dots \Rightarrow r_{n-1} \Rightarrow r_n \Rightarrow \text{sentence}$$

Bottom-up

Assuming the production $A \rightarrow \beta$, to reduce $r_i r_{i-1}$ match some RHS β against r_i then replace β with its corresponding LHS, A .

In terms of the parse tree, this is working from leaves to root.

Example – 1:

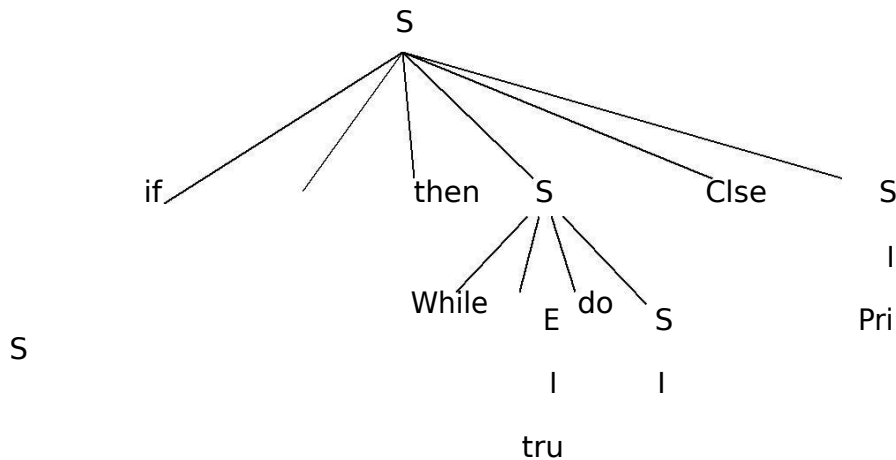
S→if E then S else S/while E do S/ print

E→ true/ False/id

Input: if id then while true do print else print.

Parse tree:

Basic idea: Given input string a , “reduce” it to the goal (start) symbol, by looking for substring that match production RHS.



```

⇒    if E then S else S
lm
⇒    if id then S else S
lm
⇒    if id then while E do S else S
lm
⇒    if id then while true do S else S
lm
⇒    if id then while true do print else S
lm
⇒    if id then while true do print else print
lm
⇐    if E then while true do print else print
rm
⇐    if E then while E do print else print
rm
⇐    if E then while E do S else print
rm
⇐    if E then S else print
rm
⇐    if E then S else S
rm
⇐    S
rm

```

Topdown Vs Bottom-up parsing:

Top-down	Bottom-up
1. Construct tree from root to leaves 2. “Guers” which RHS to substitute for nonterminal 3. Produces left-most derivation 4. Recursive descent, LL parsers 5. Recursive descent, LL parsers 6. Easy for humans	1. Construct tree from leaves to root 2. “Guers” which rule to “reduce” terminals 3. Produces reverse right-most derivation. 4. Shift-reduce, LR, LALR, etc. 5. “Harder” for humans.

→ Bottom-up can parse a larger set of languages than topdown.

→ Both work for most (but not all) features of most computer languages.

Example – 2:

Right-most derivation

$S \rightarrow aAcBe$

llp: abcde/

$S \rightarrow aAcBe$

$A \rightarrow Ab/b$

$\rightarrow aAcde$

$B \rightarrow d$

$\rightarrow aAbcde$

$\rightarrow abcde$

Bottom-up approach

“Right sentential form”	Reduction
abcde	
aAbcde	$A \rightarrow b$
Aacde	$A \rightarrow Ab$
AacBe	$B \rightarrow d$
S	$S \rightarrow aAcBe$

Steps correspond to a right-most derivation in reverse.

(must choose RHS wisely)

Example –

3: $S \rightarrow aABe$

$A \rightarrow Abc/b$

$B \rightarrow d$

1/p: abcde

Right most derivation:

aABe

aAde Since () $B \rightarrow d$

aAbcde Since () $A \rightarrow Abc$

abcde Since () $A \rightarrow b$

Parsing using Bottom-up approach:

Input	Production used
abbcde	
aAbcde	$A \rightarrow b$
AAde	$A \rightarrow Abc$
AABe	$B \rightarrow d$

S parsing is completed as we got a start symbol

Hence the 1/p string is acceptable.

Example – 4

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow id$

1/p: $id_1 + id_2 + id_3$

Right most derivation

$E \rightarrow E + E$

$\rightarrow E + E * E$

$\rightarrow E + E * id_3 \rightarrow E + id_2 * id_3$

$\rightarrow id_1 + id_2 * id_3$

Parsing using Bottom-up approach:

Go from left to right

$id_1 + id_2 * id_3$

$E + id_2 * id_3 \quad E \rightarrow id$

$E + E * id_3 \quad E \rightarrow id$

$E * id_3 \quad E \rightarrow E$

$E * \quad + E$

$E \quad E \rightarrow id$

E

= start symbol, Hence acceptable.

2. HANDLES:

Always making progress by replacing a substring with LHS of a matching production will not lead to the goal/start symbol.

For example:

abbcede

aAbcde $A \rightarrow b$

aAAcde $A \rightarrow b$

struck

Informally, A Handle of a string is a substring that matches the right side of a production, and whose reduction to the non-terminal on the left side of the production represents one step along the reverse of a right most derivation.

If the grammar is unambiguous, every right sentential form has exactly one handle.

More formally, A handle is a production $A \rightarrow \beta$ and a position in the current right-sentential form $\alpha\beta\omega$ such that:

$$S \Rightarrow \alpha A \omega \Rightarrow \alpha / \beta \omega$$

For example grammar, if current right-sentential form is

a/Abcde

Then the handle is $A \rightarrow Ab$ at the marked position. 'a' never contains non-terminals.

HANDLE PRUNING:

Keep removing handles, replacing them with corresponding LHS of production, until we reach S.

Example:

$$E \rightarrow E + E / E * E / (E) / id$$

Right-sentential form	Handle	Reducing production
a+b*c	a	$E \rightarrow id$
E+b*c	b	$E \rightarrow id$

$E + E * C$	C	$E \rightarrow id$
$E + E * E$	$E * E$	$E \rightarrow E * E$
$E + E$	$E + E$	$E \rightarrow E + E$
E		

The grammar is ambiguous, so there are actually two handles at next-to-last step. We can use parser-generators that compute the handles for us.

3. SHIFT- REDUCE PARSING: it is a process of reducing a string to the start symbol of a grammar

Shift Reduce Parsing uses a stack to hold grammar symbols and input buffer to hold string to be parsed, because handles always appear at the top of the stack i.e., there's no need to look deeper into the state.

A shift-reduce parser has just four actions:

1. **Shift**-next word is shifted onto the stack (input symbols) until a handle is formed.
2. **Reduce** – right end of handle is at top of stack, locate left end of handle within the stack. Pop handle off stack and push appropriate LHS. symbol will be replaced by the non-terminal
3. **Accept** – stop parsing on successful completion of parse and report success.
4. **Error** – call an error reporting/recovery routine.

Possible Conflicts:

Ambiguous grammars lead to parsing conflicts.

1. **Shift-reduce:** Both a shift action and a reduce action are possible in the same state (should we shift or reduce)

Example: dangling-else problem

2. **Reduce-reduce:** Two or more distinct reduce actions are possible in the same state. (Which production should we reduce with 2).

Example:

Stmt \rightarrow id (param) (a(i) is procedure call)

Param \rightarrow id

Expr \rightarrow id (expr) /id (a(i) is array subscript)

Stack	input buffer	action
\$...aa (i)\$	Reduce by ?	

Should we reduce to param or to expr? Need to know the type of a: is it an array or a function. This information must flow from declaration of a to this use, typically via a symbol table.

Shift – reduce parsing example: (Stack implementation)

Grammar: $E \rightarrow E + E / E * E / (E) / id$ Input: $id_1 + id_2 + id_3$

One Scheme to implement a handle-pruning, bottom-up parser is called a shift-reduce parser. Shift reduce parsers use stack and an input buffer.

The sequence of steps is as follows:

1. initialize stack with \$.
2. Repeat until the top of the stack is the goal symbol and the input token is “end of life”. **a. Find the handle**

If we don't have a handle on top of stack, shift an input symbol onto the stack.

b. Prune the handle

if we have a handle ($A \rightarrow \beta$) on the stack, reduce

- (i) pop β symbols off the stack
- (ii) push A onto the stack.

Stack	input	Action
\$	$id_1 + id_2 * id_3 \$$	Shift
\$ id_1	$+ id_2 * id_3 \$$	Reduce by $E \rightarrow id$
\$E	$+ id_2 * id_3 \$$	Shift
\$E+	$id_2 * id_3 \$$	Shift
\$E+ id_2	$* id_3 \$$	Reduce by $E \rightarrow id$

\$E+E	*id ₃ \$	Shift
\$E+E*	id ₃ \$	Shift
\$E+E* id ₃	\$	Reduce by E→id
\$E+E*E	\$	Reduce by E→E*E
\$E+E	\$	Reduce by E→E+E
\$E	\$	Accept

Example 2:

Goal \hookrightarrow Expr

Expr \hookrightarrow Expr + term | Expr – Term | Term

Term \hookrightarrow Tem & Factor | Term | factor | Factor

Factor \hookrightarrow number | id | (Expr)

The expression grammar : $x - z * y$

Stack	Input	Action
\$	Id - num * id	Shift
\$ id	- num * id	Reduce factor \hookrightarrow id
\$ Factor	- num * id	Reduce Term \hookrightarrow Factor
\$ Term	- num * id	Reduce Expr \hookrightarrow Term
\$ Expr	- num * id	Shift
\$ Expr -	num * id	Shift
\$ Expr – num	* id	Reduce Factor \hookrightarrow num
\$ Expr – Factor	* id	Reduce Term \hookrightarrow Factor
\$ Expr – Term	* id	Shift
\$ Expr – Term *	id	Shift

\$ Expr – Term * id		Reduce Factor id
\$ Expr – Term & Factor		Reduce Term Term * Factor
\$ Expr – Term		Reduce Expr Expr – Term
\$ Expr		Reduce Goal Expr
\$ Goal		Accept

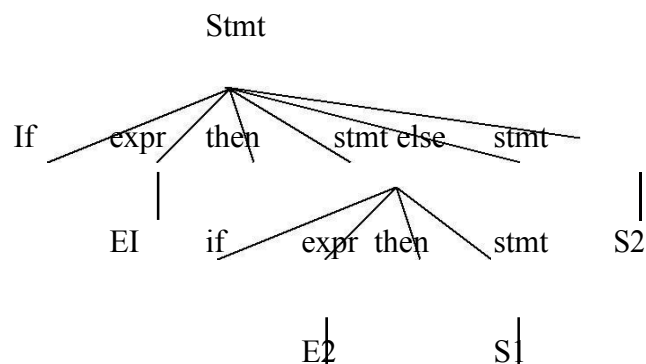
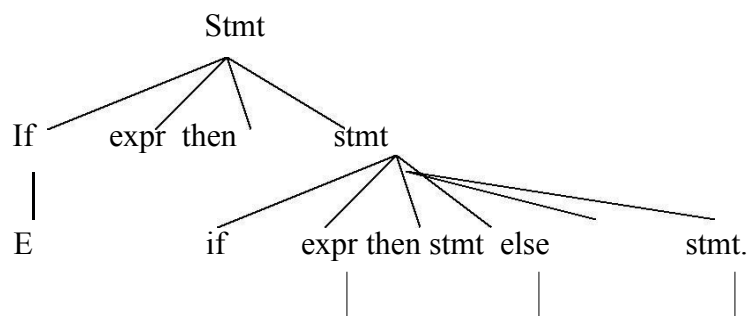
1. shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce.

Procedure:

1. Shift until top of stack is the right end of a handle.
2. Find the left end of the handle and reduce.

* Dangling-else problem:

stmt → if expr then stmt / if expr then stmt / other then example string is: if E₁ then if E₂ then S₁ else S₂ has two parse trees (ambiguity) and so this grammar is not of LR(k) type.



3. OPERATOR – PRECEDENCE PARSING:

Precedence/ Operator grammar: The grammars having the property:

1. No production right side is should contain ϵ .
2. No production sight side should contain two adjacent non-terminals.

Is called an **operator grammar**.

Operator – precedence parsing has three disjoint precedence relations, $<$, $=$ and $>$ between certain pairs of terminals. These precedence relations guide the selection of handles and have the following meanings:

RELATION	MEANING
$a < b$	'a' yields precedence to 'b'.
$a = b$	'a' has the same precedence 'b'
$a > b$	'a' takes precedence over 'b'.

Operator precedence parsing has a number of disadvantages:

1. It is hard to handle tokens like the minus sign, which has two different precedences.
2. Only a small class of grammars can be parsed.
3. The relationship between a grammar for the language being parsed and the operator-precedence parser itself is tenuous, one cannot always be sure the parser accepts exactly the desired language.

Disadvantages:

1. **$L(G) \neq L(\text{parser})$**
2. **error detection**
3. **usage is limited**
4. **They are easy to analyse manually Example:**

Eg Grammar: $E \rightarrow EAE | (E) | -E | id$

$A \rightarrow + | - | * | / | \uparrow$

Input string: $id + id * id$

The operator – precedence relations are:

	Id	+	*	\$
Id		.>	.>	.>
+	<.	.>	<.	.>
*	<.	.>	.>	.>
\$	<.	<.	<.	

RULES
id,a,b,c -> high
\$ -> low
id != id
\$ A \$

Solution: This is not operator grammar, so first reduce it to operator grammar form, by eliminating adjacent non-terminals.

Operator grammar is:

$$E \rightarrow E+E|E-E|E*E|E/E|E\uparrow E|(E)|-E|id$$

The input string with precedence relations interested is:

$$\$(id.> + <(id.> * <(id.> \$$$

Scan the string the from left end until first .> is encountered.

$$\$(id.> + <(id.> * <(id.< \$$$

This occurs between the first id and +.

Scan backwards (to the left) over any '='s until a '<'. Is encountered. We scan backwards to '\$'.

$$\$(id.> + <(id.> * <(id.> \$$$

↑ ↑

Everything to the left of the first .> and to the right of <. Is called handle. Here, the handle is the first id.

Then reduce id to E. At this point we have: E+id*id

By repeating the process and proceeding in the same way: \$+<(id.>*<(id.>\$

substitute $E \rightarrow id$,

After reducing the other id to E by the same process, we obtain the right-sentential form

$$E+E*E$$

Now, the 1/p string after detecting the non-terminals is:

$$\Rightarrow \$+*\$$$

Inserting the precedence relations, we get: $\$<.+<.*.>\$$

$\uparrow \uparrow$

The left end of the handle lies between + and * and the right end between * and \$. It indicates that, in the right sentential form $E+E*E$, the handle is $E*E$.

Reducing by $E \rightarrow E*E$, we get:

$E+E$

Now the input string is: $\$<.+ \$$

Again inserting the precedence relations, we get:

$\Rightarrow \$<.+>\$$

$\uparrow \uparrow$

reducing by $E \rightarrow E+E$, we get,

$\$ \$$

and finally we are left with:

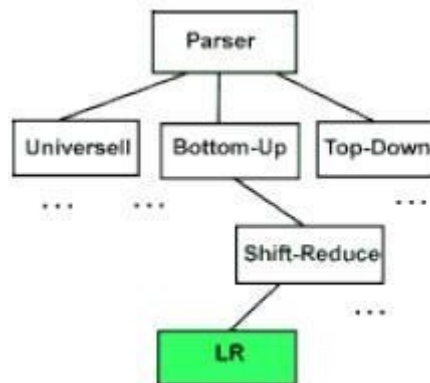
E

Hence accepted.

Input string	Precedence relations inserted	Action
id+id*id	$\$<.id.>+<.id.>*<.id.>\$$	
E+id*id	$\$+<.id.>*<.id.>\$$	$E \rightarrow id$
E+E*id	$\$+*<.id.>\$$	$E \rightarrow id$
E+E*E	$\$+*\$$	
E+E*E	$\$<.+<.*.>\$$	$E \rightarrow E*E$
E+E	$\$<.+ \$$	
E+E	$\$<.+>\$$	$E \rightarrow E+E$
E	$\$ \$$	Accepted

5. LR PARSING INTRODUCTION:

The "L" is for left-to-right scanning of the input and the "R" is for constructing a rightmost derivation in reverse.



WHY LR PARSING:

1. LR parsers can be constructed to recognize virtually all programming-language constructs for which context-free grammars can be written.
2. The LR parsing method is the most general non-backtracking shift-reduce parsing method known, yet it can be implemented as efficiently as other shift-reduce methods.
3. The class of grammars that can be parsed using LR methods is a proper subset of the class of grammars that can be parsed with predictive parsers.
4. An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.

The disadvantage is that it takes too much work to construct an LR parser by hand for a typical programming-language grammar. But there are lots of LR parser generators available to make this task easy.

LR PARSERS:

LR(k) parsers are most general non-backtracking shift-reduce parsers. Two cases of interest are $k=0$ and $k=1$. LR(1) is of practical relevance

'L' stands for "Left-to-right" scan of input.

'R' stands for "Rightmost derivation (in reverse)".

'K' stands for number of input symbols of look-a-head that are used in making parsing decisions.

When (K) is omitted, 'K' is assumed to be 1.

LR(1) parsers are table-driven, shift-reduce parsers that use a limited right context (1 token) for handle recognition.

LR(1) parsers recognize languages that have an LR(1) grammar. A grammar is LR(1) if, given a right-most derivation

$$S \Rightarrow r_0 \Rightarrow r_1 \Rightarrow r_2 \dots r_{n-1} \Rightarrow r_n \Rightarrow \text{sentence}.$$

We can isolate the handle of each right-sentential form r_i and determine the production by which to reduce, by scanning r_i from left-to-right, going atmost 1 symbol beyond the right end of the handle of r_i .

Parser accepts input when stack contains only the start symbol and no remaining input symbol are left.

LR(0) item: (no lookahead)

Grammar rule combined with a dot that indicates a position in its RHS.

Ex- 1: $S^I \rightarrow .S \mid S \rightarrow .x \mid S \rightarrow .(L)$

Ex-2: $A \rightarrow XYZ$ generates 4LR(0) items –

$A \rightarrow .XYZ$

$A \rightarrow X.YZ$

$A \rightarrow XY.Z$

$A \rightarrow XYZ.$

The '.' Indicates how much of an item we have seen at a given state in the parse.

$A \rightarrow .XYZ$ indicates that the parser is looking for a string that can be derived from XYZ.

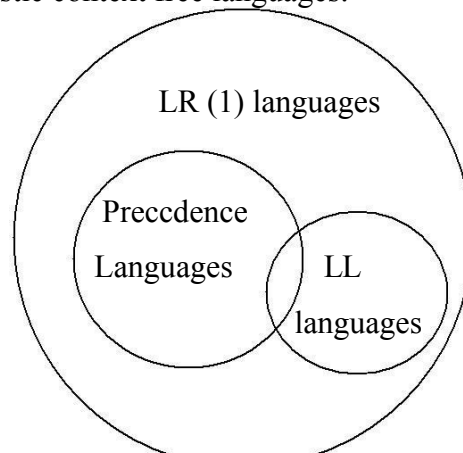
$A \rightarrow XYZ$ indicates that the parser has seen a string derived from XY and is looking for one derivable from Z .

→ LR(0) items play a key role in the SLR(1) table construction algorithm.

→ LR(1) items play a key role in the LR(1) and LALR(1) table construction algorithms. LR parsers have more information available than LL parsers when choosing a production:

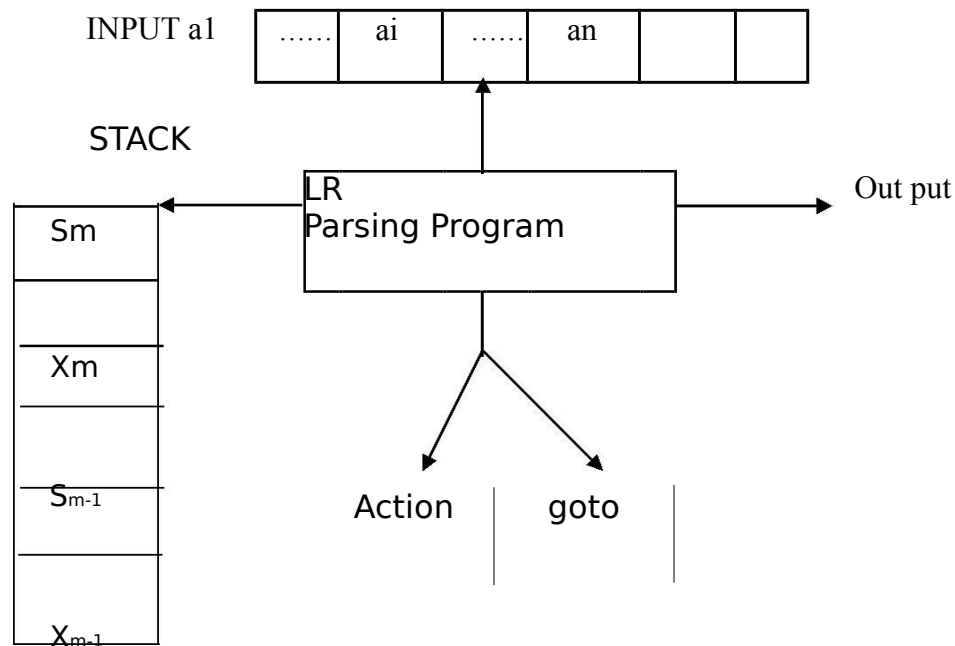
- * **LR knows everything derived from RHS plus 'K' lookahead symbols.**
- * **LL just knows 'K' lookahead symbols into what's derived from RHS.**

Deterministic context free languages:



LR PARSING ALGORITHM:

The schematic form of an LR parser is shown below:



It consists of an input, an output, a stack, a driver program, and a parsing table that has two parts: action and goto.

The LR parser program determines S_m , the current state on the top of the stack, and a_i , the current input symbol. It then consults action $[S_m, a_i]$, which can have one of four values:

1. **Shift S, where S is a state.**
2. **reduce by a grammar production $A \rightarrow \beta$**
3. **accept and**
4. **error**

The function goes to takes a state and grammar symbol as arguments and produces a state. The goto function of a parsing table constructed from a grammar G using the SLR, canonical LR or LALR method is the transition function of DFA that recognizes the viable prefixes of G. (Viable prefixes of G are those prefixes of right-sentential forms that can appear on the stack of a shift-reduce parser, because they do not extend past the right-most handle)

5.6 AUGMENTED GRAMMAR:

If G is a grammar with start symbol S, then G^I , the augmented grammar for G with a new start symbol S^I and production $S^I \rightarrow S$.

The purpose of this new start stating production is to indicate to the parser when it should stop parsing and announce acceptance of the input i.e., acceptance occurs when and only when the parser is about to reduce by $S^I \rightarrow S$.

CONSTRUCTION OF SLR PARSING TABLE:

Example:

The given grammar is:

1. $E \rightarrow E + T$
 2. $E \rightarrow T$
 3. $T \rightarrow T * F$
 4. $T \rightarrow F$
 5. $F \rightarrow (E)$
 6. $F \rightarrow \text{id}$
- Step I: The Augmented grammar is:**

Simple LR Parser (LR)
works on smallest class of grammar
few no. of states
simple and fast construction

|

$E \xrightarrow{I} E$ $E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

Step II: The collection of LR (0) items are:

$I_0: E \xrightarrow{I} \cdot E$

$E \rightarrow \cdot E+T$

$E \rightarrow \cdot T$

$T \rightarrow \cdot T * F$

$T \rightarrow \cdot F$

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot id$

Start with start symbol after since () there is E, start writing all productions of E.

Start writing 'T' productions

Start writing F productions

Goto (I_0, E):

States have successor states formed by advancing the marker over the symbol it precedes. For state 1 there are successor states reached by advancing the masks over

$E \xrightarrow{I} \cdot E$ - symbols E, T, F, C or id. Consider, first, the

$E \rightarrow E \cdot + T$

Goto (I_0, T):

$I_2: E \rightarrow T \cdot$ - reduced Item (RI)

$T \rightarrow T.*F$

Goto (I_0, F):

$I_2: \quad E \rightarrow T. \quad - \quad \text{reduced item (RI)}$

$T \rightarrow T.*F$

Goto (I_0, C):

$I_4: \quad F \rightarrow (.(E)$

$E \rightarrow .E+T$

$E \rightarrow .T$

$T \rightarrow .T*F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

If ‘.’ Precedes non-terminal start writing its corresponding production. Here first E then T after that F.

Start writing F productions.

Goto (I_0, id):

$I_5: \quad F \rightarrow id. \quad - \quad \text{reduced item.}$

E successor (I, state), it contains two items derived from state 1 and the closure operation adds no more (since neither marker precedes a non-terminal). The state I_2 is thus:

Goto ($I_1, +$):

$I_6: \quad E \rightarrow E+.T \quad \text{start writing T productions}$

$T \rightarrow .T*F$

$T \rightarrow .F \quad \text{start writing F productions}$

$F \rightarrow .(E)$

$F \rightarrow .id$

Goto (I₂,*):

I₇: $T \rightarrow T*.F$ start writing F productions

$F \rightarrow .(E)$

$F \rightarrow .id$

Goto (I₄,E):

$I_8: \quad F \rightarrow (E.)$

$E \rightarrow E.+T$

Goto (I₄,T):

I₂: $E \rightarrow T.$ these are same as I₂.

$T \rightarrow T.*F$

Goto (I₄,C):

$I_4: \quad F \rightarrow (.E)$

$E \rightarrow .E+T$

$E \rightarrow .T$

$T \rightarrow .T*F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

goto (I₄,id):

I₅: $F \rightarrow id.$ - reduced item

Goto (I₆,T):

I₉: $E \rightarrow E+T.$ - reduced item

$T \rightarrow T * F$

Goto (I₆,F):

I₃: $T \rightarrow F$. - reduced item Goto (I₆,C):

$I_4: \quad F \rightarrow (.E)$

$E \rightarrow .E + T$

$E \rightarrow .T$

$T \rightarrow .T * F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

Goto (I₆,id):

I₅: $F \rightarrow id$. reduced item.

Goto (I₇,F):

I₁₀: $T \rightarrow T * F$ reduced item

Goto (I₇,C):

$I_4: \quad F \rightarrow (.E)$

$E \rightarrow .E + T$

$E \rightarrow .T$

$T \rightarrow .T * F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

Goto (I₇,id):

I₅: $F \rightarrow id$. - reduced item

Goto (I₈,)):

I₁₁: F→(E). reduced item

Goto (I₈,+):

I₁₁: F→(E). reduced item

Goto (I₈,+):

I₆: E→E+.T

T→.T*F

T→.F

F→.(E)

F→.id

Goto (I₉,+):

I₇: T→T*.f

F→.(E)

F→.id

Step IV: Construction of Parse table:

Construction must proceed according to the algorithm 4.8

S→shift items

R→reduce items

Initially E^I→E. is in I₁ so, I = 1.

Set action [I, \$] to accept i.e., action [1, \$] to Acc

Action									Goto
State	Id	+	*	()	\$	E	T	F
I ₀	S ₅			S ₄			1	2	3
1		S ₆				Accept			
2		r ₂	S ₇		R ₂	R ₂			

3		R ₄	R ₄		R ₄	R ₄			
4	S ₅			S ₄					3
5		R ₆	R ₆		R ₆	R ₆			
6	S ₅			S ₄					3
7	S ₅			S ₄					10
8		S ₆			S ₁₁				
9		R ₁	S ₇		r ₁	r ₁			
10		R ₃	R ₃		R ₃	R ₃			
11		R ₅	R ₅		R ₅	R ₅			

As there are no multiply defined entries, the grammar is SLR®.

STEP – III Finding FOLLOW () set for all non-terminals.

	Relevant production
FOLLOW (E) = { \$ } U FIRST (+ T) U FIRST ())	$E \rightarrow E/B + T/B$
= { + ,) , \$ }	$F \rightarrow (E)$
	$B\beta$
FOLLOW (T) = FOLLOW (E) U	$E \rightarrow T$
FIRST (* F) U	$T \rightarrow T * F$
FOLLOW (E)	$E \rightarrow E + T$
	B
= { + , * ,) , \$ }	
FOLLOW (F) = FOLLOW (T)	
= { * , * ,) , \$ }	

Step – V:

1. Consider I₀:

1. The item $F \rightarrow .(E)$ gives rise to goto (I₀,C) = I₄, then action [0,C] = shift 4
2. The item $F \rightarrow .id$ gives rise goto (I₀,id) = I₄, then action [0,id] = shift 5

the other items in I₀ yield no actions. Goto (I₀,E) = I₁ then goto [0,E] = 1

Goto (I_0, T) = I_2 then goto [$0, T$] = 2

Goto (I_0, F) = I_3 then goto [$0, F$] = 3

2. Consider I_1 :

1. The item $E \xrightarrow{I} E$ is the reduced item, so $I = 1$ This gives rise to action [$1, \$$] to accept.

2. The item $E \rightarrow E.T$ gives rise to goto ($I_1, +$) = I_6 , then action [$1, +$] = shift 6.

3. Consider I_2 :

1. The item $E \rightarrow T$ is the reduced item, so take FOLLOW (E),

FOLLOW (E) = $\{+,), \$\}$

The first item $+$, makes action [$Z, +$] = reduce $E \rightarrow T$. $E \rightarrow T$ is production rule no.2. So action [$Z, +$] = reduce 2.

The second item, makes action [$Z,)$] = reduce 2 The third item $\$$, makes action [$Z, \$$] = reduce 2

2. The item $T \rightarrow T.*F$ gives rise to

goto [$I_2, *$] = I_7 , then action [$Z, *$] = shift 7.

4. Consider I_3 :

1. $T \rightarrow F$ is the reduced item, so take FOLLOW (T).

FOLLOW (T) = $\{+, *,), \$\}$

So, make action [$3, +$] = reduce 4

Action [$3, *$] = reduce 4

Action [$3,)$] = reduce 4

Action [3,\$] = reduce 4

In forming item sets a closure operation must be performed to ensure that whenever the marker in an item of a set precedes a non-terminal, say E, then initial items must be included in the set for all productions with E on the left hand side.

The first item set is formed by taking initial item for the start state and then performing the closure operation, giving the item set;

We construct the action and goto as follows:

1. **If there is a transition from state I to state J under the terminal symbol K, then set action [I,k] to S_J.**
2. **If there is a transition under a non-terminal symbol a, say from state 'i' to state 'J', set goto [I,A] to S_J.**
3. **If state I contains a transition under \$ set action [I,\$] to accept.**
4. **If there is a reduce transition #p from state I, set action [I,k] to reduce #p for all terminals k belonging to FOLLOW (A) where A is the subject to production #P.**

If any entry is multiply defined then the grammar is not SLR(1). Blank entries are represented by dash (-).

5. Consider I₄ items:

The item $F \rightarrow id$ gives rise to $goto [I_4, id] = I_5$ so,

Action (4,id) \rightarrow shift 5

The item $F \rightarrow .E$ action (4,c) \rightarrow shift 4

The item $goto (I_4, F) \rightarrow I_3$, so $goto [4, F] = 3$

The item $goto (I_4, T) \rightarrow I_2$, so $goto [4, F] = 2$

The item $goto (I_4, E) \rightarrow I_8$, so $goto [4, F] = 8$

6. Consider I₅ items:

$F \rightarrow id$. Is the reduced item, so take FOLLOW (F).

FOLLOW (F) = {+, *, }, \$

$F \rightarrow id$ is rule no.6 so reduce 6

Action (5,+) = reduce 6

Action (5,*) = reduce 6

Action (5,)) = reduce 6

Action (5,)) = reduce 6

Action (5,\$) = reduce 6

7. Consider I_6 items:

goto (I_6, T) = I_9 , then goto [$6, T$] = 9 goto (I_6, F) = I_3 , then
goto [$6, F$] = 3 goto (I_6, C) = I_4 , then goto [$6, C$] = 4 goto
(I_6, id) = I_5 , then goto [$6, id$] = 5

8. Consider I_7 items:

1. goto (I_7, F) = I_{10} , then goto [$7, F$] = 10
2. goto (I_7, C) = I_4 , then action [$7, C$] = shift 4
3. goto (I_7, id) = I_5 , then goto [$7, id$] = shift 5

9. Consider I_8 items:

1. goto ($I_8,))$ = I_{11} , then action [$8,))$] = shift 11
2. goto ($I_8, +$) = I_6 , then action [$8, +$] = shift 6

10. Consider I_9 items:

1. $E \rightarrow E+T$. is the reduced item, so take FOLLOW (E).

FOLLOW (E) = {+,), \$}

$E \rightarrow E+T$ is the production no.1., so

Action [$9, +$] = reduce 1

Action [$9,))$] = reduce 1

Action [$9, \$$] = reduce 1

2. goto [$I_5, *$] = I_7 , then action [$9, *$] = shift 7.
-

11. Consider I_{10} items:

1. $T \rightarrow T * F$. is the reduced item, so take

$FOLLOW(T) = \{+, *,), \$\}$

$T \rightarrow T * F$ is production no.3., so

Action $[10, +] = \text{reduce } 3$

Action $[10, *] = \text{reduce } 3$

Action $[10,)] = \text{reduce } 3$

Action $[10, \$] = \text{reduce } 3$

12. Consider I_{11} items:

1. $F \rightarrow (E)$. is the reduced item, so take

$FOLLOW(F) = \{+, *,), \$\}$

$F \rightarrow (E)$ is production no.5., so

Action $[11, +] = \text{reduce } 5$

Action $[11, *] = \text{reduce } 5$

Action $[11,)] = \text{reduce } 5$

Action $[11, \$] = \text{reduce } 5$

VI MOVES OF LR PARSER ON $\text{id} * \text{id} + \text{id}$:

	STACK	INPUT	ACTION
1.	0	$\text{id} * \text{id} + \text{id} \$$	shift by S5
2.	0id5	$* \text{id} + \text{id} \$$	sec 5 on * reduce by $F \rightarrow \text{id}$ If $A \rightarrow \beta$ Pop $2 * \beta $ symbols. $= 2 * 1 = 2$ symbols. Pop 2 symbols off the stack State 0 is then exposed on F.

			Since goto of state 0 on F is 3, F and 3 are pushed onto the stack
3.	0F3	*id+id\$	reduce by $T \rightarrow F$ pop 2 symbols push T. Since goto of state 0 on T is 2, T and 2, T and 2 are pushed onto the stack.
4.	0T2	*id+id\$	shift by S7
5.	0T2*7	id+id\$	shift by S5
6.	0T2*7id5	+id\$	reduce by r6 i.e. $F \rightarrow id$ Pop 2 symbols, Append F, Secn 7 on F, it is 10
7.	0T2*7F10	+id\$	reduce by r3, i.e., $T \rightarrow T * F$ Pop 6 symbols, push T Sec 0 on T, it is 2 Push 2 on stack.
8.	0T2	+id\$	reduce by r2, i.e., $E \rightarrow T$ Pop two symbols, Push E See 0 on E. It 10 1 Push 1 on stack
9.	0E1	+id\$	shift by S6.
10.	0E1+6	id\$	shift by S5
11.	0E1+6id5	\$	reduce by r6 i.e.,

		F →id
		Pop 2 symbols, push F, see 6
	on F	
0E1+6F3	\$	It is 3, push 3
		reduce by r4, i.e.,
		T →F
		Pop2 symbols,
		Push T, see 6 on T
		It is 9, push 9.
0E1+6T9	\$	reduce by r1, i.e.,
		E →E+T
		Pop 6 symbols, push E
		See 0 on E, it is 1
		Push 1.
0E1	\$	Accept

Procedure for Step-V

The parsing algorithm used for all LR methods uses a stack that contains alternatively state numbers and symbols from the grammar and a list of input terminal symbols terminated by \$. For example:

AAbBcCdDeEf/uvwxyz\$

Where, a.....f are state numbers

A . . . E are grammar symbols (either terminal or non-terminals) u.....z are the terminal symbols of the text still to be parsed. The parsing algorithm starts in state I₀ with the configuration –



0 / whole program upto \$.

Repeatedly apply the following rules until either a syntactic error is found or the parse is complete.

(i) **If action $[f,4] = S_i$ then transform** aAbBcCdDeEf / uvwxyz\$

to aAbBcCdDeEfui / vwxyz\$ This is called a SHIFT transition

(ii) **If action $[f,4] = \#P$ and production # P is of length 3, say, then it will be of the form $P \rightarrow CDE$ where CDE exactly matches the top three symbols on the stack, and P is some non-terminal, then assuming $\text{goto}[C,P] = g$**

aAbBcCdDeEfui / vwxyz\$ will transform to

aAbBcPg / vwxyz\$

The symbols in the stack corresponding to the right hand side of the production have been replaced by the subject of the production and a new state chosen using the goto table. This is called a REDUCE transition.

(iii) **If action $[f,u] = \text{accept}$. Parsing is completed**

(iv) **If action $[f,u] = -$ then the text parsed is syntactically in-correct.**

Canonical LR(O) collection for a grammar can be constructed by augmented grammar and two functions, closure and goto.

The closure operation:

If I is the set of items for a grammar G, then closure (I) is the set of items constructed from I by the two rules:

i) **initially, every item in I is added to closure (I).**

5. CANONICAL LR PARSING:

Example:

$$S \rightarrow CC$$

$$C \rightarrow CC/d.$$

1. Number the grammar productions:

1. $S \rightarrow CC$
2. $C \rightarrow CC$
3. $C \rightarrow d$

2. The Augmented grammar is:

$$S^I \rightarrow S$$

$$S \rightarrow CC$$

$$C \rightarrow CC$$

$$C \rightarrow d.$$

Constructing the sets of LR(1) items:

We begin with:

$$S^I \rightarrow .S, \$ \text{ begin with look-a-head (LAH) as } \$.$$

We match the item $[S^I \rightarrow .S, \$]$ with the term $[A \rightarrow \alpha.B\beta, a]$

In the procedure closure, i.e.,

$$A = S^I \quad \alpha = \epsilon$$

$$B = S$$

$$\beta = \epsilon \quad a = \$$$

Function closure tells us to add $[B \rightarrow .r, b]$ for each production $B \rightarrow r$ and terminal b in FIRST (βa).

Now $\beta \rightarrow r$ must be $S \rightarrow CC$, and since β is ϵ and a is $\$, b$ may only be $\$$. Thus,

$S \rightarrow .CC, \$$

We continue to compute the closure by adding all items $[C \rightarrow .r, b]$ for b in $\text{FIRST}[C\$]$ i.e., matching $[S \rightarrow .CC, \$]$ against $[A \rightarrow \alpha.B\beta, a]$ we have, $A=S$, $\alpha=\epsilon$, $B=C$ and $a=\$$. $\text{FIRST}(C\$) = \text{FIRST } \odot$

$\text{FIRST } \odot = \{c, d\}$ We add items:

$C \rightarrow .cC, C$

$C \rightarrow cC, d$

$C \rightarrow .d, c$

$C \rightarrow .d, d$

None of the new items have a non-terminal immediately to the right of the dot, so we have completed our first set of LR(1) items. The initial I_0 items are:

$I_0 : S \xrightarrow{I} .S, \$ \quad S \rightarrow .CC, \$ \quad C \rightarrow .CC, c/d \quad C \rightarrow .d, c/d$

Now we start computing $\text{goto}(I_0, X)$ for various non-terminals i.e., $\text{Goto}(I_0, S)$:

$I_1 : S \xrightarrow{I} S., \$ \rightarrow \text{reduced item.}$

$\text{Goto}(I_0, C$

$I_2 : S \rightarrow C.C, \$$

$C \rightarrow .cC, \$$

$C \rightarrow .d, \$$

$\text{Goto}(I_0, C :$

$I_2 : C \rightarrow c.C, c/d$

$C \rightarrow .cC, c/d$

$C \rightarrow .d, c/d$

Goto (I ₀ ,d)	
I ₄	C→d., c/d→ reduced item.
Goto (I ₂ ,C)	I ₅
	S→CC.,\$ → reduced item.
Goto (I ₂ ,C)	I ₆
	C→c.C,\$
	C→.cC,\$
	C→.d,\$
Goto (I ₂ ,d)	I ₇
	C→d.,\$ → reduced item.
Goto (I ₃ ,C)	I ₈
	C→cC.,c/d → reduced item.
Goto (I ₃ ,C)	I ₃
	C→c.C, c/d
	C→.cC,c/d
	C→.d,c/d
Goto (I ₃ ,d)	I ₄
	C→d.,c/d. → reduced item.
Goto (I ₆ ,C)	I ₉
	C→cC.,\$ → reduced item.
Goto (I ₆ ,C)	I ₆
	C→c.C,\$
	C→.cC,\$
	C→.d,\$
Goto (I ₆ ,d)	I ₇
	C→d.,\$ → reduced item.

All are completely reduced. So now we construct the canonical LR(1) parsing table –

Here there is no need to find FOLLOW () set, as we have already taken look-a-head for each set of productions while constructing the states.

Constructing LR(1) Parsing table:

State	Action			goto	
	C	D	\$	S	C
I 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		

1. Consider I₀ items:

The item $S \rightarrow .S.\$$ gives rise to goto $[I_0, S] = I_1$ so goto $[0, s] = 1$.

The item $S \rightarrow .CC, \$$ gives rise to goto $[I_0, C] = I_2$ so goto $[0, C] = 2$.

The item $C \rightarrow .cC, c/d$ gives rise to goto $[I_0, C] = I_3$ so goto $[0, C] = \text{shift } 3$

The item $C \rightarrow .d, c/d$ gives rise to goto $[I_0, d] = I_4$ so goto $[0, d] = \text{shift } 4$

2. Consider I₁ items:

The item $S^I \rightarrow S., \$$ is in I_1 , then set action $[1, \$] = \text{accept}$

3. Consider I₂ items:

The item $S \rightarrow C.C, \$$ gives rise to goto $[I_2, C] = I_5$. so goto $[2, C] = 5$

The item $C \rightarrow .cC, \$$ gives rise to goto $[I_2, C] = I_6$. so action $[0, C] = \text{shift}$ The item $C \rightarrow .d, \$$ gives rise to goto $[I_2, d] = I_7$. so action $[2, d] = \text{shift } 7$

4. Consider I₃ items:

The item $C \rightarrow .cC, c/d$ gives rise to goto $[I_3, C] = I_8$. so goto $[3, C] = 8$

The item $C \rightarrow .cC, c/d$ gives rise to goto $[I_3, C] = I_3$. so action $[3, C] = \text{shift } 3$. The item $C \rightarrow .d, c/d$ gives rise to goto $[I_3, d] = I_4$. so action $[3, d] = \text{shift } 4$.

5. Consider I₄ items:

The item $C \rightarrow .d$, c/d is the reduced item, it is in I_4 so set action $[4, c/d]$ to reduce $c \rightarrow d$. (production rule no.3)

6. Consider I₅ items:

The item $S \rightarrow CC. \$$ is the reduced item, it is in I_5 so set action $[5, \$]$ to $S \rightarrow CC$ (production rule no.1)

7. Consider I₆ items:

The item $C \rightarrow c.C, \$$ gives rise to goto $[I_6, C] = I_9$. so goto $[6, C] = 9$

The item $C \rightarrow .cC, \$$ gives rise to goto $[I_6, C] = I_6$. so action $[6, C] = \text{shift } 6$

The item $C \rightarrow .d, \$$ gives rise to goto $[I_6, d] = I_7$. so action $[6, d] = \text{shift } 7$

8. Consider I₇ items:

The item $C \rightarrow d. \$$ is the reduced item, it is in I_7 .

So set action $[7, \$]$ to reduce $C \rightarrow d$ (production no.3)

9. Consider I₈ items:

The item $C \rightarrow CC.c/d$ in the reduced item, It is in I_8 , so set action $[8, c/d]$ to reduce $C \rightarrow cd$ (production rule no .2)

10. Consider I₉ items:

The item $C \rightarrow cC, \$$ is the reduced item, It is in I_9 , so set action $[9, \$]$ to reduce $C \rightarrow cC$ (Production rule no.2)

If the Parsing action table has no multiply –defined entries, then the given grammar is called as LR(1) grammar

LALR PARSING:

Example:

1. Construct $C = \{I_0, I_1, \dots, I_n\}$ The collection of sets of LR(1) items

2. For each core present among the set of LR (1) items, find all sets having that core, and replace there sets by their Union# (clus them into a single term)

$I_0 \rightarrow$ same as previous

$I_1 \rightarrow$ “

$I_2 \rightarrow$ “

I_{36} – Clubbing item I_3 and I_6 into one I_{36} item.

$C \rightarrow cC, c/d/\$$

$C \rightarrow cC, c/d/\$$

$C \rightarrow d, c/d/\$$

$I_5 \rightarrow$ some as previous

$I_{47} \rightarrow C \rightarrow d, c/d/\$$

$I_{89} \rightarrow C \rightarrow cC, c/d/\$$

LALR Parsing table construction:

State	Action			Goto	
	c	d			C
I_0	S_{36}	S_{47}			2
1			Accept		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r3	r3			
5			r1		
89	r2	r2	r2		



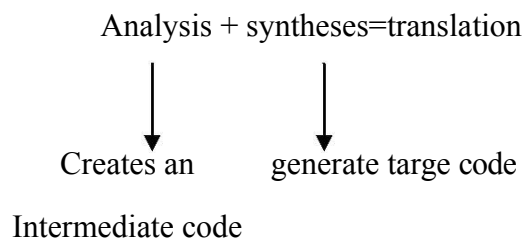
UNIT - III

UNIT-III

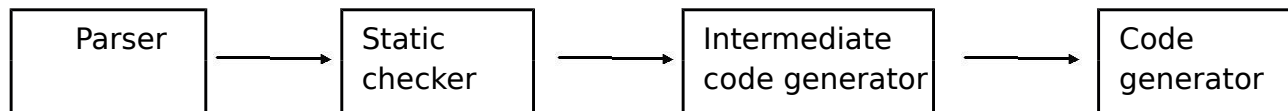
INTERMEDIATE CODE GENERATION

1. **Intermediate code forms:**

An intermediate code form of source program is an internal form of a program created by the compiler while translating the program created by the compiler while translating the program from a high –level language to assembly code(or)object code(machine code).an intermediate source form represents a more attractive form of target code than does assembly. An optimizing Compiler performs optimizations on the intermediate source form and produces an object module.



In the analysis –synthesis model of a compiler, the front-end translates a source program into an intermediate representation from which the back-end generates target code, in many compilers the source code is translated into a language which is intermediate in complexity between a HLL and machine code .the usual intermediate code introduces symbols to stand for various temporary quantities.



position of intermediate code generator

We assume that the source program has already been parsed and statically checked.. the various intermediate code forms are:

- a) Polish notation
 - b) Abstract syntax trees(or)syntax trees
 - c) Quadruples
 - d) Triples
 - e) Indirect triples
 - f) Abstract machine code(or)pseudocode
- } three address code
- a. postfix notation:

The ordinary (infix) way of writing the sum of a and b is with the operator in the middle: $a+b$. the postfix (or postfix polish) notation for the same expression places the operator at the right end, as $ab+$.

In general, if e_1 and e_2 are any postfix expressions, and \emptyset to the values denoted by e_1 and e_2 is indicated in postfix notation by $e_1e_2\emptyset$. no parentheses are needed in postfix notation because the position and priority (number of arguments) of the operators permits only one way to decode a postfix expression.

Example:

1. $(a+b)*c$ in postfix notation is $ab+c*$, since $ab+$ represents the infix expression $(a+b)$.
2. $a*(b+c)$ is $abc+*$ in postfix.
3. $(a+b)*(c+d)$ is $ab+cd+*$ in postfix.

Postfix notation can be generalized to k -ary operators for any $k \geq 1$. if k -ary operator \emptyset is applied to postfix expression e_1, e_2, \dots, e_k , then the result is denoted by $e_1e_2\dots e_k\emptyset$. if we know the priority of each operator then we can uniquely decipher any postfix expression by scanning it from either end.

Example:

Consider the postfix string $ab+c*$.

The right hand $*$ says that there are two arguments to its left. since the next –to-rightmost symbol is c , simple operand, we know c must be the second operand of $*$. continuing to the left, we encounter the operator $+$. we know the sub expression ending in $+$ makes up the first operand of $*$. continuing in this way, we deduce that $ab+c*$ is “parsed” as $((a,b+),c)*$.

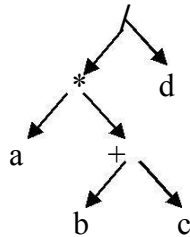
b. syntax tree:

The parse tree itself is a useful intermediate-language representation for a source program, especially in optimizing compilers where the intermediate code needs to extensively restructure.

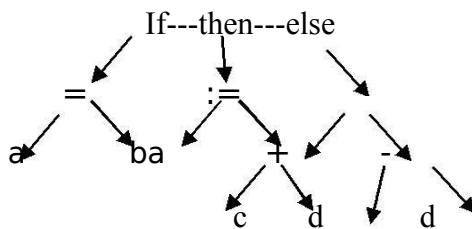
A parse tree, however, often contains redundant information which can be eliminated, Thus producing a more economical representation of the source program. One such variant of a parse tree is what is called an (abstract) syntax tree, a tree in which each leaf represents an operand and each interior node an operator.

Exmples:

1) Syntax tree for the expression $a*(b+c)/d$



2) syntax tree for **if a=b then a:=c+d else b:=c-d**



Three-Address Code:

- In three-address code, there is at most one operator on the right side of an instruction; that is, no built-up arithmetic expressions are permitted.

$x+y*z$ $t_1 = y * z$ $t_2 = x + t_1$

• Example

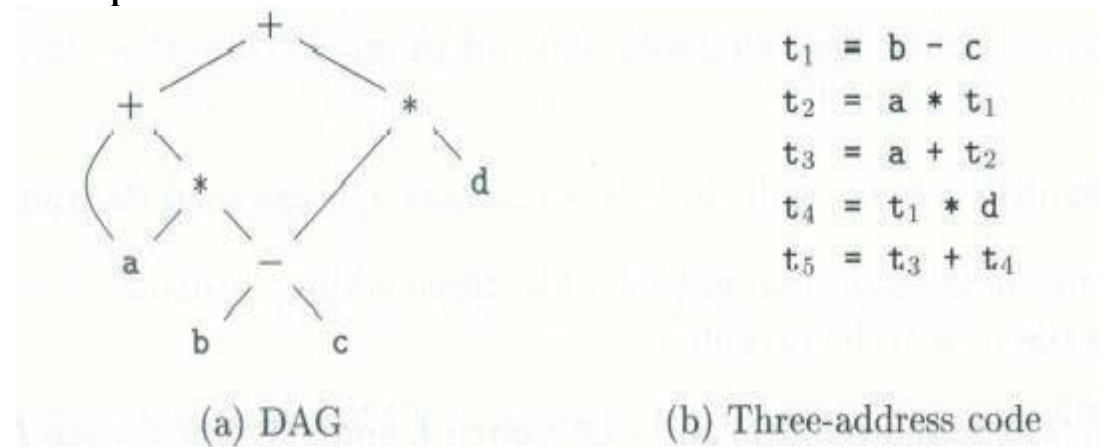


Figure 6.8: A DAG and its corresponding three-address code

Problems:

Write the 3-address code for the following expression

1. $\text{if}(x + y * z > x * y + z) \text{ a} = 0;$
2. $(2 + a * (b - c / d)) / e$
3. $A := b * -c + b * -c$

Address and Instructions

-
- **Example** Three-address code is built from two concepts: addresses and instructions.
- An address can be one of the following:
 - A name: A source name is replaced by a pointer to its symbol table entry.
- **A name:** For convenience, allow source-program names to Appear as addresses in three-address code. In an Implementation, a source name is replaced by a pointer to its symbol-table entry, where all information about the name is kept.
 - A constant
- **A constant:** In practice, a compiler must deal with many different types of constants and variables
 - A compiler-generated temporary
- **A compiler-generated temporary.** It is useful, especially in optimizing compilers, to create a distinct name each time a temporary is needed. These temporaries can be combined, if possible, when registers are allocated to variables.

A list of common three-address instruction forms: Assignment statements

- $x = y \text{ op } z$, where op is a binary operation
- $x = \text{op } y$, where op is a unary operation
- Copy statement: $x = y$
- Indexed assignments: $x = y[i]$ and $x[i] = y$
 - Pointer assignments: $x = \&y$, $*x = y$ and $x = *y$

Control flow statements

- Unconditional jump: `goto L`
 - Conditional jump: `if x relop y goto L ; if x goto L; if False x goto L`
 - Procedure calls: `call procedure p with n parameters and return y`, is Optional
- param x1 param x2
...
param xn call p, n

- **do** $i = i + 1$; **while** $(a[i] < v)$;
-

```

L:  t1 = i + 1
    i = t1
    t2 = i * 8
    t3 = a [ t2 ]
    if t3 < v goto L

```

(a) Symbolic labels.

```

100: t1 = i + 1
101: i = t1
102: t2 = i * 8
103: t3 = a [ t2 ]
104: if t3 < v goto 100

```

(b) Position numbers.

The multiplication $i * 8$ is appropriate for an array of elements that each take 8 units of space.

C. quadruples:

- Three-address instructions can be implemented as objects or as record with fields for the operator and operands.
- Three such representations
 - Quadruple, triples, and indirect triples
- A quadruple (or quad) has four fields: op, arg1, arg2, and result.

Example D. Triples

- A triple has only three fields: op, arg1, and arg2
- Using triples, we refer to the result of an operation $x \text{ op } y$ by its position, rather by an explicit temporary name.

Example

```

t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
a = t5

```

(a) Three-address code

	op	arg ₁	arg ₂	result
0	minus	c		t ₁
1	*	b	t ₁	t ₂
2	minus	c		t ₃
3	*	b	t ₃	t ₄
4	+	t ₂	t ₄	t ₅
5	=	t ₅		a
		...		

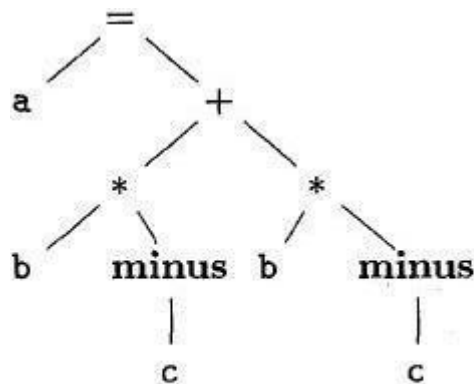
(b) Quadruples

d. Triples:

- A triple has only three fields: op, arg1, and arg2

- Using triples, we refer to the result of an operation $x \text{ op } y$ by its position, rather by an explicit temporary name.

Example



(a) Syntax tree

	<i>op</i>	<i>arg₁</i>	<i>arg₂</i>
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

(b) Triples

Fig: Representations of $a = b * - c + b * - c$

instruction

35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	...

op *arg₁* *arg₂*

0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

Fig: Indirect triples representation of 3-address code

-> The benefit of **Quadruples** over **Triples** can be seen in an optimizing compiler, where instructions are often moved around.

-> With **quadruples**, if we move an instruction that computes a temporary **t**, then the instructions that use **t** require no change. With **triples**, the result of an operation is referred to by its position, so moving an instruction may require changing all references to that result. **This problem does not occur with indirect triples.**

Single-Assignment Static Form

Static single assignment form (SSA) is an intermediate representation that facilitates certain code optimization.

- Two distinct aspects distinguish SSA from three-address code.

– All assignments in SSA are to variables with distinct names; hence the term static single-assignment.

```
p = a + b
q = p - c
p = q * d
p = e - p
q = p + q
```

```
p1 = a + b
q1 = p1 - c
p2 = q1 * d
p3 = e - p2
q2 = p3 + q1
```

(a) Three-address code. (b) Static single-assignment form.

Figure 6.13: Intermediate program in three-address code and SSA

```
if (flag) x = -1; else x = 1;
y = x * a
if (flag) x1 = -1; else x2 = 1;
x3 = φ(x1, x2)
```

2. Type Checking:

•A compiler has to do semantic checks in addition to syntactic checks. •Semantic Checks

–Static –done during compilation

–Dynamic –done during run-time

•**Type checking** is one of these static checking operations.

–we may not do all type checking at compile-time.

–Some systems also use dynamic type checking too.

•A **type system** is a collection of rules for assigning type expressions to the parts of a program.

•A **type checker** implements a type system.

•A **sound type** system eliminates run-time type checking for type errors.

- In practice, some of type checking operations is done at run-time (so, most of the programming languages are not strongly typed).

$$S \rightarrow d = E \quad \{ \text{if } (id.type = E.type \text{ then } S.type = \text{void} \\ \text{else } S.type = \text{type-error}) \}$$

S->while E do S1 { if (E.type=boolean then
S.type=S1.type else S.type=type-error }

```
E->E1( E2)          { ██████████  
                      else E.type=type-error }
```

f:

```
graph BT
    arg[argument] --> x
    types[types] --> x
    x[double x] --> y
    y[char->int] --> ret[return type]
```

- How do we know that two type expressions are equal?
- As long as type expressions are built from basic types (no type names), we may use structural equivalence between two type expressions

```
else if (s = s1 [REDACTED]) else return false
```

Names for Type Expressions:

- In some programming languages, we give a name to a type expression, and we use that name as a type expression afterwards.

type link = ↑cell; ? p,q,r,s have same types ? var p,q : link;

var r,s : ↑cell

- How do we treat type names?

– Get equivalent type expression for a type name (then use structural equivalence), or

– Treat a type name as a basic type

3. Syntax Directed Translation: grammar + semantic rules = SDT

- A formalist called as syntax directed definition is used for specifying translations for programming language constructs.
- A syntax directed definition is a generalization of a context free grammar in which each grammar symbol has associated set of attributes and each and each productions is associated with a set of semantic rules

Definition of (syntax Directed definition) SDD :

SDD is a generalization of CFG in which each grammar productions $X \rightarrow \alpha$ is associated with it a set of semantic rules of the form

$a = f(b_1, b_2, \dots, b_k)$

Where a is an attributes obtained from the function f .

- A syntax-directed definition is a generalization of a context-free grammar in which:
 - Each grammar symbol is associated with a set of attributes.
 - This set of attributes for a grammar symbol is partitioned into two subsets called **synthesized** and **inherited** attributes of that grammar symbol.
 - Each production rule is associated with a set of semantic rules.
 - Semantic rules set up dependencies between attributes which can be represented by a dependency graph.
-

- This dependency graph determines the evaluation order of these semantic rules.
- Evaluation of a semantic rule defines the value of an attribute. But a semantic rule may also have some side effects such as printing a value.

The two attributes for non terminal are :

1) Synthesized attribute (S-attribute) : (\uparrow)

An attribute is said to be **synthesized attribute** if its value at a parse tree node is determined from attribute **values at the children of the node**

2) Inherited attribute: (\uparrow, \rightarrow)

An inherited attribute is one whose **value at parse tree node is determined in terms of attributes at the parent and | or siblings** of that node.

- The attribute can be string, a number, a type, a, memory location or anything else.
- The parse tree showing the value of attributes at each node is called an annotated parse tree.

The process of computing the attribute values at the node is called annotating or decorating the parse tree. **Terminals** can have synthesized attributes, but not inherited attributes.

Annotated Parse Tree

- A parse tree showing the values of attributes at each node is called an **Annotated parse tree**.
- The process of computing the attributes values at the nodes is called **annotating (or decorating)** of the parse tree.
- Of course, the order of these computations depends on the dependency graph induced by the semantic rules.

Ex1:1) Synthesized Attributes : Ex: Consider the CFG :

$S \rightarrow EN$ $E \rightarrow E+T$ $E \rightarrow E-T$ $E \rightarrow T$ $T \rightarrow T * F$ $T \rightarrow T / F$ $T \rightarrow F$ $F \rightarrow (E)$ $F \rightarrow \text{digit}$ $N \rightarrow ;$

Solution: The syntax directed definition can be written for the above grammar by using semantic actions for each production.

Production rule	Semantic actions
$S \rightarrow EN$	$S.val = E.val$
$E \rightarrow E1 + T$	$E.val = E1.val + T.val$
$E \rightarrow E1 - T$	$E.val = E1.val - T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T * F$	$T.val = T.val * F.val$
$T \rightarrow T \mid F$	$T.val = T.val \mid F.val$
$F \rightarrow (E)$	$F.val = E.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow \text{digit}$	$F.val = \text{digit.lexval}$
$N \rightarrow ;$	can be ignored by lexical Analyzer as; I is terminating symbol

For the Non-terminals E,T and F the values can be obtained using the attribute “Val”.

The taken digit has synthesized attribute “lexval”.

In $S \rightarrow EN$, symbol S is the start symbol. This rule is to print the final answer of expressed.

Following steps are followed to Compute S attributed definition

1. Write the SDD 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 node is supposed to be final output.

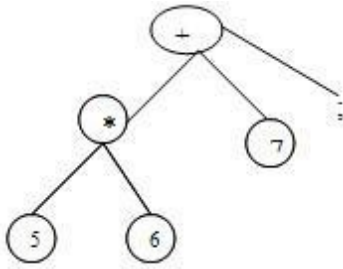
PROBLEM 1:

Consider the string $5*6+7$; Construct Syntax tree, parse tree and annotated tree.

Solution:

The corresponding annotated parse tree is shown below for the string $5*6+7$;

Syntax tree:



Annotated parse tree :

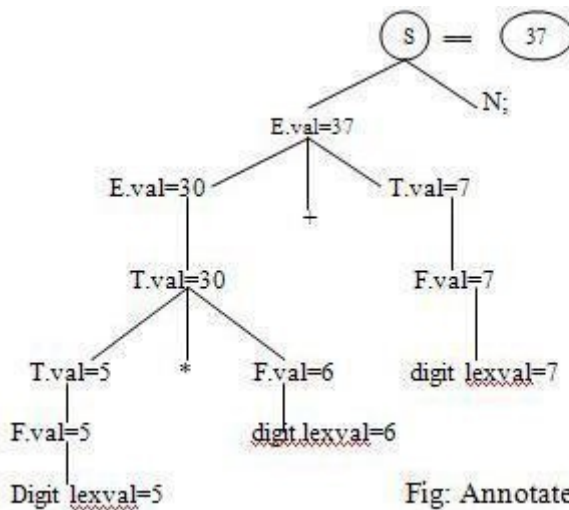


Fig: Annotated parse tree

Advantages: SDDs are more readable and hence useful for specifications

Disadvantages: not very efficient.

Ex2:

PROBLEM : Consider the grammar that is used for Simple desk calculator. Obtain the Semantic action and also the annotated parse tree for the string

3*5+4n. $L \rightarrow En$ $E \rightarrow E1 + T$

$E \rightarrow T$

$T \rightarrow T_1 * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow \text{digit}$

Solution :

Production rule

Semantic actions

$L \rightarrow E n$

$L.val = E.val$

$E \rightarrow E_1 + T$

$E.val = E_1.val + T.val$

$E \rightarrow T$

$E.val = T.val$

$T \rightarrow T_1 * F$

$T.val = T_1.val * F.val$

$T \rightarrow F$

$T.val = F.val$

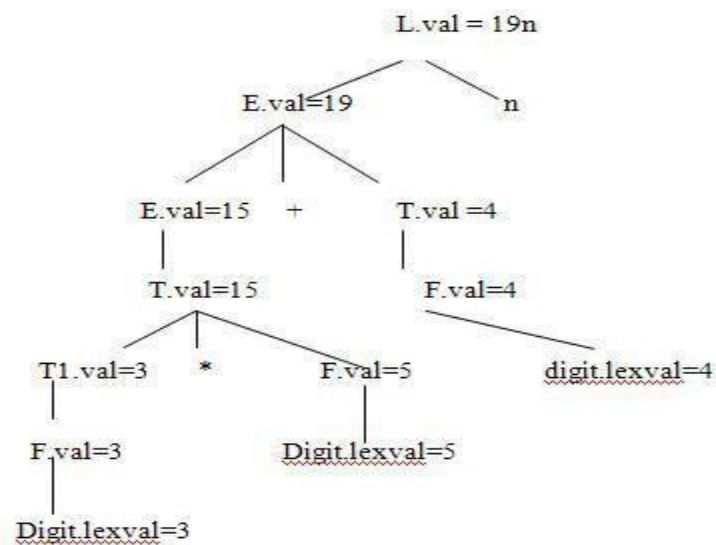
$F \rightarrow (E)$

$F.val = E.val$

$F \rightarrow \text{digit}$

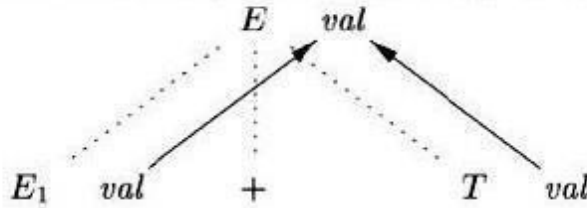
$F.val = \text{digit.lexval}$

The corresponding annotated parse tree U shown below, for the string $3*5+4n$.



Dependency Graphs:

Figure 5.6. $E.val$ is synthesized from $E_1.val$ and $E_2.val$



Dependency graph and topological sort:

- For each parse-tree node, say a node labeled by grammar symbol X , the dependency graph has a node for each attribute associated with X .
- If a semantic rule associated with a production p defines the value of synthesized attribute $A.b$ in terms of the value of $X.c$. Then the dependency graph has an edge from $X.c$ to $A.b$.
- If a semantic rule associated with a production p defines the value of inherited attribute $B.c$ in terms of the value $X.a$. Then, the dependency graph has an edge from $X.a$ to $B.c$.

Applications of Syntax-Directed Translation

- Construction of syntax Trees
 - The nodes of the syntax tree are represented by objects with a suitable number of fields.
 - Each object will have an *op* field that is the label of the node.
 - The objects will have additional fields as follows
- If the node is a leaf, an additional field holds the lexical value for the leaf. A constructor function **Leaf**(*op*, *val*) creates a leaf object.
- If nodes are viewed as records, the Leaf returns a pointer to a new record for a leaf.
- If the node is an interior node, there are as many additional fields as the node has children in the syntax tree. A constructor function **Node** takes two or more arguments:
Node(*op*, *c1*, *c2*, ..., *ck*) creates an object with first field *op* and *k* additional fields for the *k* children *c1*, *c2*, ..., *ck*.

Syntax-Directed Translation Schemes

A SDT scheme is a context-free grammar with program fragments embedded within production bodies. The program fragments are called semantic actions and can appear at any position within the production body.

Any SDT can be implemented by first building a parse tree and then pre-forming the actions in a left-to-right depth first order. i.e during preorder traversal.

The use of SDT's to implement two important classes of SDD's

1. If the grammar is LR parsable, then SDD is S-attributed.
2. If the grammar is LL parsable, then SDD is L-attributed.

Postfix Translation Schemes

The postfix SDT implements the desk calculator SDD with one change: the action for the first production prints the value. As the grammar is LR, and the SDD is S-attributed.

$L \rightarrow E \text{ n } \{ \text{print}(E.\text{val}); \}$

$E \rightarrow E_1 + T \{ E.\text{val} = E_1.\text{val} + T.\text{val} \}$

$E \rightarrow E_1 - T \{ E.\text{val} = E_1.\text{val} - T.\text{val} \}$

$E \rightarrow T \{ E.\text{val} = T.\text{val} \}$

$T \rightarrow T_1 * F \{ T.\text{val} = T_1.\text{val} * F.\text{val} \} \quad T \rightarrow F \{ T.\text{val} = F.\text{val} \}$

$F \rightarrow (E) \{ F.\text{val} = E.\text{val} \}$

$F \rightarrow \text{digit} \{ F.\text{val} = \text{digit}.\text{lexval} \}$

PRODUCTION	SEMANTIC RULES
1) $E \rightarrow E_1 + T$	$E.\text{node} = \text{new Node}('+', E_1.\text{node}, T.\text{node})$
2) $E \rightarrow E_1 - T$	$E.\text{node} = \text{new Node}('-', E_1.\text{node}, T.\text{node})$
3) $E \rightarrow T$	$E.\text{node} = T.\text{node}$
4) $T \rightarrow (E)$	$T.\text{node} = E.\text{node}$
5) $T \rightarrow \text{id}$	$T.\text{node} = \text{new Leaf}(\text{id}, \text{id}.\text{entry})$
6) $T \rightarrow \text{num}$	$T.\text{node} = \text{new Leaf}(\text{num}, \text{num}.\text{val})$

UNIT - IV

UNIT - IV

RUN TIME ENVIRONMENT

Symbol table:

A **symbol table** is a major data structure used in a compiler:

- Associates **attributes** with identifiers used in a program.
- For instance, a **type attribute** is usually associated with each identifier.



A symbol table is a necessary component.



Definition (declaration) of identifiers appears once in a program Use of identifiers may appear in many places of the program text Identifiers and attributes are entered by the analysis phases



When processing a definition (declaration) of an identifier



In simple languages with only global variables and implicit declarations:



In block-



The scanner can enter an identifier into a symbol table if it is not already there structured languages with scopes and explicit declarations:



The parser and/or semantic analyzer enter identifiers and corresponding attributes

- **Symbol table information is used by the analysis and synthesis phases**
- To verify that used identifiers have been defined (declared)
- To verify that expressions and assignments are semantically correct – **type checking**
- To generate intermediate or target code

Symbol Table Interface:

The basic operations defined on a symbol table include:



allocate – to allocate a new empty symbol table

- **free** – to remove all entries and free the storage of a symbol table
- **insert** – to insert a name in a symbol table and return a pointer to its entry
- **lookup** – to search for a name and return a pointer to its entry
- **set_attribute** – to associate an attribute with a given entry
- **get_attribute** – to get an attribute associated with a given entry
- Other operations can be added depending on requirement

For example, a **delete** operation removes a name previously inserted Some identifiers become invisible (out of scope) after exiting a block

- This interface provides an abstract view of a symbol table.
- Supports the simultaneous existence of multiple tables
- Implementation can vary without modifying the interface

Basic Implementation Techniques:

First consideration is how to **insert** and **lookup** names

Variety of implementation techniques

Unordered List

Simplest to implement

Implemented as an array or a linked list

Linked list can grow dynamically – alleviates problem of a fixed size array

Insertion is fast $O(1)$, but lookup is slow for large tables – $O(n)$ on average

Ordered List

If an array is sorted, it can be searched using binary search – $O(\log_2 n)$

Insertion into a sorted array is expensive – $O(n)$ on average

Useful when set of names is known in advance – table of reserved words

Binary Search Tree

Can grow dynamically

Insertion and lookup are $O(\log_2 n)$ on average

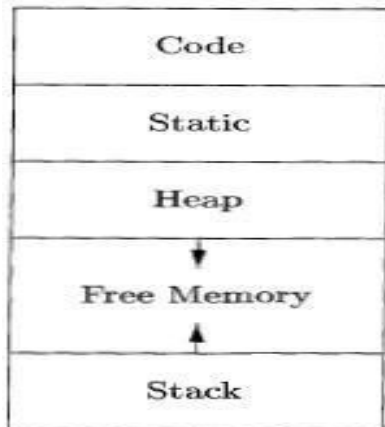
Hash Tables and Hash Functions:

- A **hash table** is an array with index range: 0 to $TableSize - 1$
 - Most commonly used data structure to implement symbol tables
 - Insertion and lookup can be made very fast – $O(1)$
 - A **hash function** maps an identifier name into a table index
 - ☞ A hash function, $h(name)$, should depend solely on name
 - ☞ $h(name)$ should be computed quickly
 - ☞ h should be **uniform** and **randomizing** in distributing names
 - ☞ All table indices should be mapped with equal probability
 - ☞ Similar names should not cluster to the same table index.
-

Storage Allocation:

- Compiler must do the storage allocation and provide access to variables and data
- Memory management
 - ☞ Stack allocation
 - ☞ Heap management
 - ☞ Garbage collection

Storage Organization:



- Assumes a logical address space
 - Operating system will later map it to physical addresses, decide how to use cache memory, etc.
- Memory typically divided into areas for
 - ☞ Program code
 - Other static data storage, including global constants and compiler-generated data
 - Stack to support call/return policy for procedures
 - Heap to store data that can outlive a call to a procedure

Static vs. Dynamic Allocation:

- Static: Compile time, Dynamic: Runtime allocation
 - Many compilers use some combination of following
 - ☞ Stack storage: for local variables, parameters and so on
 - ☞ Heap storage: Data that may outlive the call to the procedure that created it
-



Stack allocation is a valid allocation for procedures since procedure calls are nest

Static storage allocation

- > In static allocation, names are bound to storage locations.
- > If memory is created at compile time then the memory will be created in static area and only once.
- > Static allocation supports the dynamic data structure that means memory is created only at compile time and deallocated after program completion.
- > The drawback with static storage allocation is that the size and position of data objects should be known at compile time.
- > Heap storage allocation does not supports the recursion process.

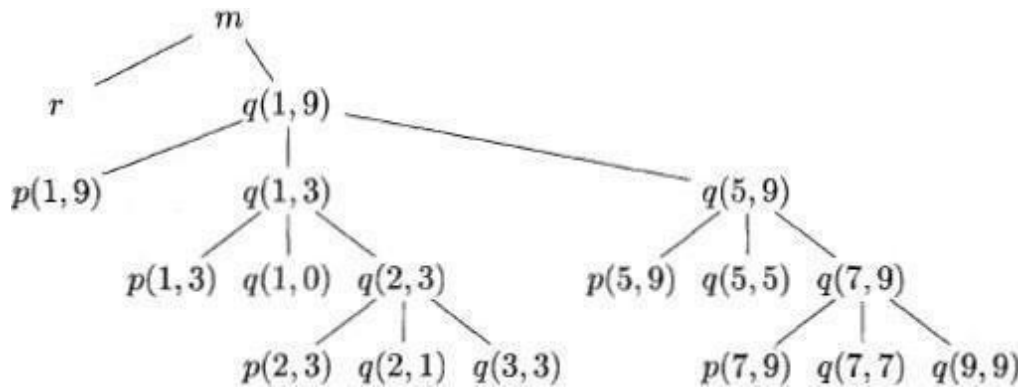
Stack Storage Allocation

- > In static storage allocation, storage is organized as a stack.
- > An activation record is pushed into the stack when activation begins and it is popped when the activation end.
- > Activation record contains the locals so that they are bound to fresh storage in each activation record. The value of locals is deleted when the activation ends.
- > It works on the basis of last-in-first-out (LIFO) and this allocation supports the recursion process.

Heap Storage Allocation

- > Heap allocation is the most flexible allocation scheme.
 - > Allocation and deallocation of memory can be done at any time and at any place depending upon the user's requirement.
 - > Heap allocation is used to allocate memory to the variables dynamically and when the variables are no more used then claim it back.
 - > Heap storage allocation supports the recursion process.
-

Activation tree representing calls during an execution of quicksort:



Activation records



Procedure calls and returns are usually managed by a run-time stack called the control stack.



Each live activation has an activation record (sometimes called a frame)



The root of activation tree is at the bottom of the stack



The current execution path specifies the content of the stack with the last



Activation has record in the top of the stack.

A General Activation Record

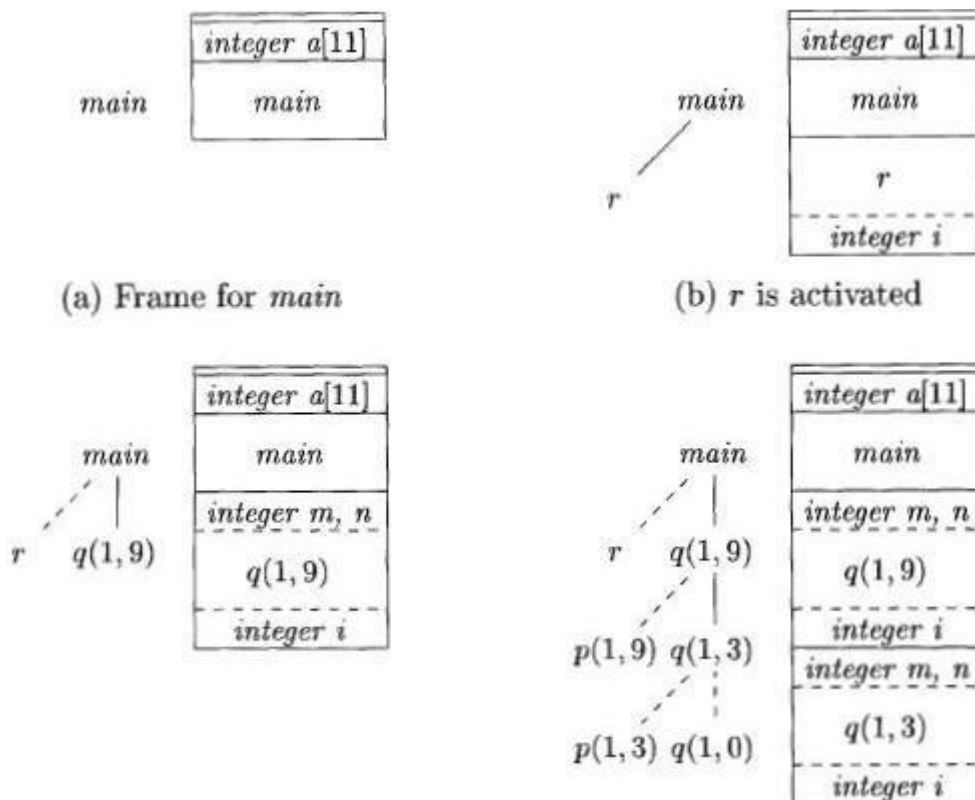
Actual parameters
Returned values
Control link
Access link
Saved machine status
Local data
Temporaries

Activation Record

- Temporary values
 - Local data
 - A saved machine status
 - An “access link”
 - A control link
-

- Space for the return value of the called function
 - The actual parameters used by the calling procedure
-
- Elements in the activation record:
 - ⌘ Temporary values that could not fit into registers.
 - ⌘ Local variables of the procedure.
 - ⌘ Saved machine status for point at which this procedure called. Includes return address and contents of registers to be restored.
 - ⌘ Access link to activation record of previous block or procedure in lexical scope chain.
 - ⌘ Control link pointing to the activation record of the caller.
 - ⌘ Space for the return value of the function, if any.
 - ⌘ actual parameters (or they may be placed in registers, if possible)

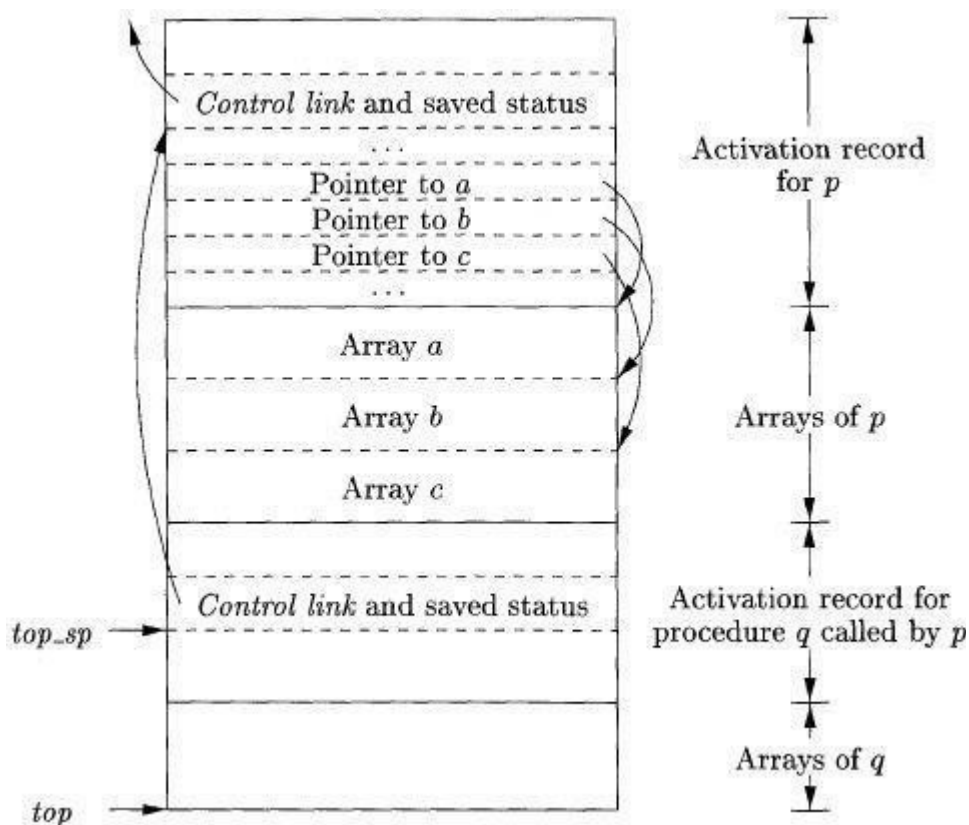
Downward-growing stack of activation records:



Designing Calling Sequences:

- Values communicated between caller and callee are generally placed at the beginning of callee's activation record
- Fixed-length items: are generally placed at the middle
- Items whose size may not be known early enough: are placed at the end of activation record
- We must locate the top-of-stack pointer judiciously: a common approach is to have it point to the end of fixed length fields

Access to dynamically allocated arrays:



ML:

- ML is a functional language
- Variables are defined, and have their unchangeable values initialized, by a statement of the form:
val (name) = (expression)
- Functions are defined using the syntax:

fun (name) ((arguments)) = (body)

- For function bodies we shall use let-statements of the form: let (list of definitions) in (statements) end

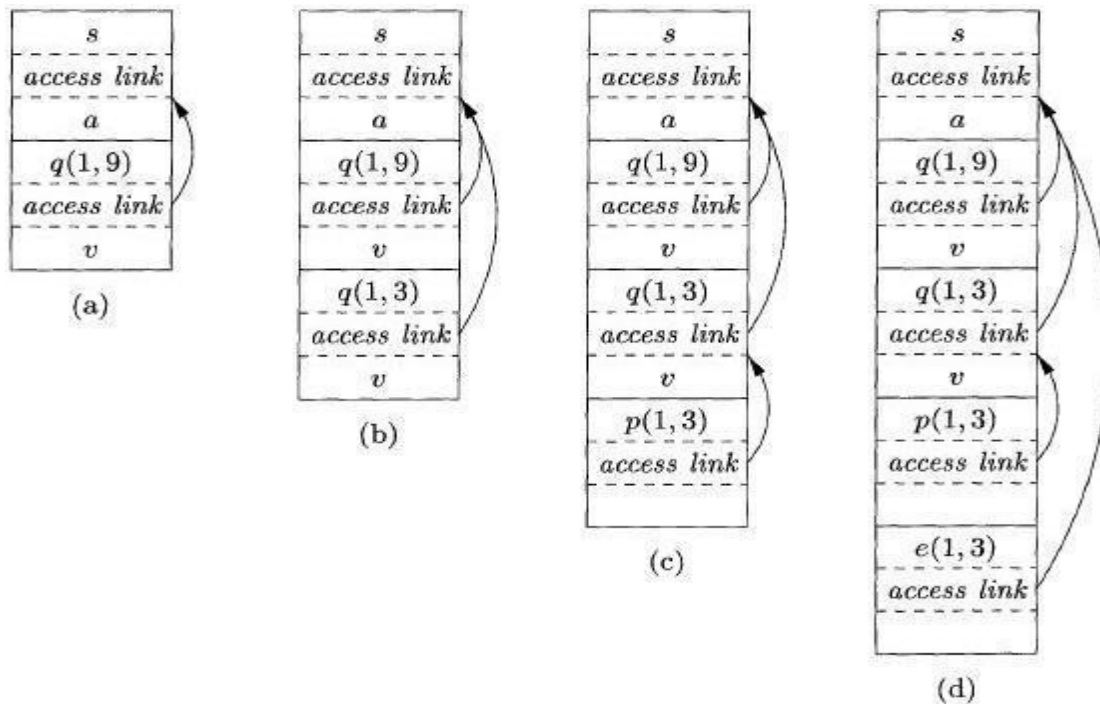
A version of quick sort, in ML style, using nested functions:

```

1) fun sort(inputFile, outputFile) =
    let
2)       val a = array(11,0);
3)       fun readArray(inputFile) = ... ;
4)       ... a ... ;
5)       fun exchange(i,j) =
6)         ... a ... ;
7)       fun quicksort(m,n) =
          let
8)         val v = ... ;
9)         fun partition(y,z) =
10)        ... a ... v ... exchange ...
          in
11)        ... a ... v ... partition ... quicksort
          end
    in
12)    ... a ... readArray ... quicksort ...
    end;

```

Access links for finding nonlocal data:



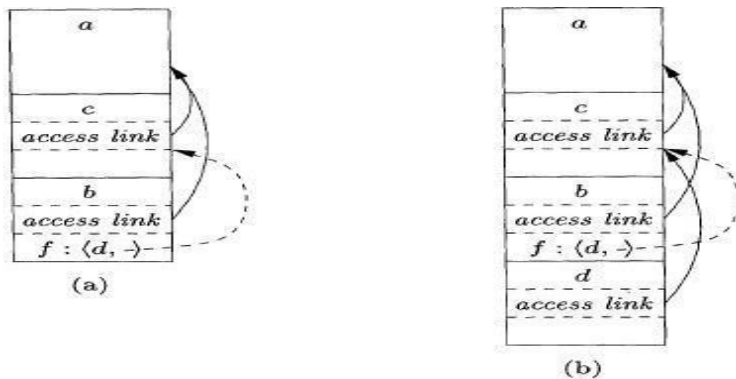
Sketch of ML program that uses function-parameters:

```

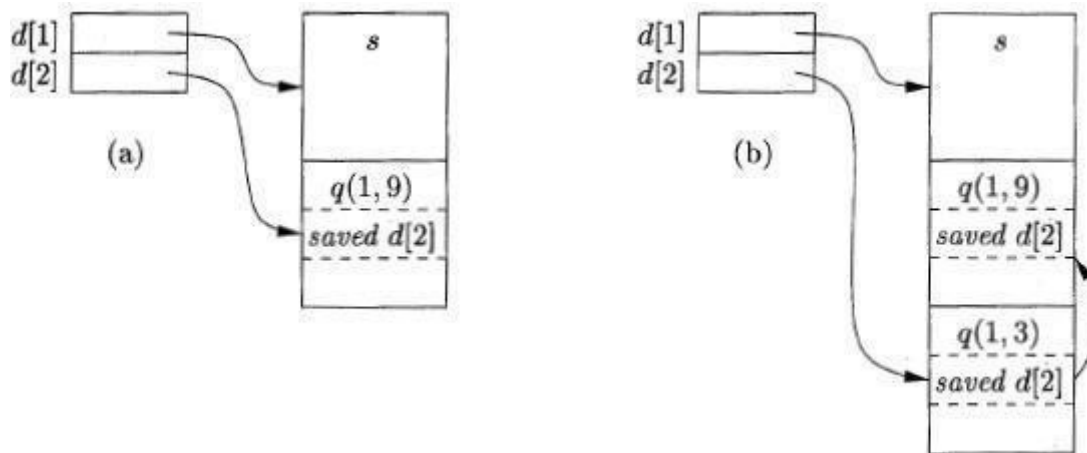
fun a(x) =
  let
    fun b(f) =
      ... f ... ;
    fun c(y) =
      let
        fun d(z) = ...
      in
        ... b(d) ...
      end
  in
    ... c(1) ...
  end;

```

Actual parameters carry their access link with them:



Maintaining the Display:



Memory Manager:

- Two basic functions:
 - ☞ Allocation
 - ☞ Deallocation
- Properties of memory managers:
 - ☞ Space efficiency
 - ☞ Program efficiency
 - ☞ Low overhead

Typical Memory Hierarchy Configurations:

Typical Sizes		Typical Access Times
> 2GB	Virtual Memory (Disk)	3 - 15 ms
	↕	
256MB - 2GB	Physical Memory	100 - 150 ns
	↕	
128KB - 4MB	2nd-Level Cache	40 - 60 ns
	↕	
16 - 64KB	1st-Level Cache	5 - 10 ns
	↕	
32 Words	Registers (Processor)	1 ns

Locality in Programs:

The conventional wisdom is that programs spend 90% of their time executing 10% of the code:

- ☞ Programs often contain many instructions that are never executed.
 - ☞ Only a small fraction of the code that could be invoked is actually executed in atypical run of the program.
 - ☞ The typical program spends most of its time executing innermost loops and tight recursive cycles in a program.
-

Symbol Table

Symbol table organization is **important for improving the efficiency of the compiler**. It is important to understand the different forms of symbol table and how it effects the performance while retrieving the variable information

Symbol table is an important data structure which stores the information of all the variables and procedures in the program. This table is created during first phase and is used by all the phases for inserting the information or retrieving the information. Organizing the elements in the symbol table shows the impact on the performance of the compiler. This chapter focuses on what type of details are stored in the symbol table, how they are organized and how the arrangement of these elements effect the retrieval time. The focus in mainly on simple array structure, linked list, trees and hash tables for both block and non-block structured languages.

Introduction

So far we have discussed the different phases of the compiler, each performing some specific task. The main objective of any compiler is to generate a target code that corresponds to a given source code in terms of task execution, correctness, and meaningfulness. These objectives are achieved with the support of a special data structure called the symbol table. It is a data structure that stores information about the name, type, and scope of the variables for performing the tasks defined in all the phases of a compiler.

The symbol table is created during the lexical analysis phase and is maintained till the last phase is completed (**Table 9.1**). It is referred to in every phase of the compiler for some purpose or the other.

Symbol table

S.N o.	Phase	Usage
1.	Lexical	Creates new entries for each new identifiers
2.	Syntax	Adds information regarding attributes like type, scope, and dimension, line of reference, and line of use.
3.	Semantic	Uses the available information to check for semantics
4.	Intermediate	Helps to add temporary variables information.
5.	Optimization	Helps in machine-dependent optimization by considering

S.N o.	Phase	Usage
		address and aliased variables information.
6.	Code Generation	Generates the code by using the address information of the identifiers.

It is clear that every time the compiler finds a new identifier in the source code during lexical analysis phase, it needs to check if this identifier is already in the table, and if not it needs to store it there. Every insertion operation is always preceded with search operation during lexical phase. During other phases it searches in the symbol table to access the attribute information of the entries.

The basic two operations that are often performed with the symbol table are insert—to add new entries—and lookup—to find existing entries. The performance of the table depends on how these elements are arranged in the table. The different mechanisms that govern the performance of the table are linear list, hierarchical list, and hash-based lists. These mechanisms are evaluated based on the number of insertions(n) and the number of lookup(e) operations. A linear list is the simplest to implement, but its performance is poor when n and e are large. Hierarchical list gives performance proportional to $n(\log(n)) + e(\log(n))$, where n is the number of insertions and e is the number of lookup operations. Hashing schemes provide better performance with greater programming effort and space overhead.

Apart from organizing the elements, the size of the table is also an important factor. The size can be fixed when the compiler is written. The fixed size has a limitation—if chosen small, it cannot store more variables; if chosen large, a lot of space is wasted. It is important for the symbol table to be dynamic to allow the increase in size at compile time.

The entries in the symbol table are made during the lexical phase as soon as the name is seen in the declaration statement of the input program. The information on the attributes is added when available. In some cases, the attribute information is added along with entry, on the first appearance of the variable.

For example, the C declarations

```
char NAME[20];

int AGE;

char ADDRESS[30];

int PHONENO[10];
```

For block structured languages, the entries are made when the syntactic role played by the name is discovered. The attributes are entered as action corresponding to identifying a LEXEME for a token, which is an identifier in declaration statements. This action is performed for every colon encountered in the sequence of the variable list.

Symbol Table Entries

Each entry in the symbol table is associated with attributes that support the compiler in different phases. These attributes may not be important for all compilers, but each should be considered for a particular implementation.

- Name
- Size
- Dimension
- Type
- Line of declaration
- Line of usage
- Address

There is a distinction between the token id for an identifier or name, the lexeme consisting of the character string forming the name, and the attributes of the name. Lexeme is used mainly to distinguish the attributes of one variable from the other categorized as the same token type. During the lexical analysis, when the lexeme that corresponds to an identifier is found, it first looks up in the symbol table and if it does not appear then the entry is made. A common representation of a name is a pointer to a symbol table entry for it.

All these attributes are not of a fixed size. Their space requirement in symbol table is not always constant. For instance, the size of the variable's name is language dependent. If there is an upper limit on the length, then the characters in the name can be stored in the symbol table entry as shown in Figure 9.1.

Attributes

Name	Type	Size	Dim	Line of decle	Line of usage	address
N A M E	Char	20	1
A G E	Int	1	0
A D D R E S S	Char	30	1
P H O N E N O	Int	20	1
.....

FigureE: Symbol table

In fixed size space within a record.

In language like BASIC and FORTRAN, the name is of a limited size of two or six. In such languages it is better if the name is stored in the symbol table.

If the limit on the length is not fixed, then it is not feasible to fix the size in the symbol table. The solution in such a case is to use the indirect scheme. Instead of storing the variable name, it is good to store the address of the location where the variable is stored. The indirect scheme permits the size of the name field of the symbol table entry itself to remain a constant.

The complete lexeme constituting a name must be stored in a separate location to ensure that all uses of the same name can be associated with the symbol table record. The following table shows the entries in the symbol table for the above example, which is with fixed size space requirement.

In languages like C, C++, Java, etc., the variable name can vary from one character to 31 characters. In such cases the indirect scheme provides a better way of storing the information.

Name		Type	Size	Dim	Line of decl	Line of usage	address
0	4	Char	20	1
5	3	Int	1	0
9	8	Char	30	1
17	7	Int	20	1
	

N	A	M	E	#	A	G	E	#	A	D	D	R	E	S	S	#	P	H	O	N	E	N	O	#		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29

The name attribute in the symbol table has two values, the starting address of the name and the size of the name. In this approach, good space is saved but the disadvantage is when the compiler has to look for the attribute information. It first looks in the symbol table for the first name location; then it checks if that is the required variable; if that is not the required variable, it has to again make the symbol table reference until it gets the required variable information. A lot of time is wasted in searching for the required information.

The operations performed on the symbol table are dependent on whether the language is block structured or non-block structured. In case of non-block structured languages, there is only one instance of the variable declaration and its scope is throughout the program. In block structured languages, the variables may be re-declared and its scope is within the block. To handle the variable information, some more operations are required to keep track of the scope of the variable.

- Insert
- Lookup

- Insert
- Lookup
- Set
- Reset

Insert (s,t) performs the insertion of string s and token t and returns the index of this new entry.

Lookup(s) finds for string S, if found, it returns the index, to get attributes; otherwise, it returns 0.

Set operation is performed on every block entry to mark the beginning of a new scope of variables declared in that block. Every insert and look up operation depends on this information entered.

Reset is performed at every block exit, to remove all declarations inside this scope and the details of the block.

Set and reset operations are performed in block structured languages to keep the scope information of the variables. Since, scope behave like stacks the best way to implement these functions is to use a stack. When we begin a new scope, we push a special marker to the stack to indicate the beginning of the block. The list of new declarations is inserted after this special marker is verified. At the exit of block the scope the element ends and hence all the elements inserted are popped off the stack.

Symbol Table Organization

We can use any data structure for the symbol table. The performance of any compiler depends on the organization of the symbol table and the type of declaration supported. Some programming languages support implicit declaration. In such languages, the variable can be declared or can be directly used without declaration. For example, in FORTRAN, the variables can be used without declaration. It considers the variable as int if it starts with I, J, K, L, M, or N; otherwise, it is real. In some languages, all declarations are done at the start of the program and are used later. All insert operations are performed first followed by lookup operations in case of explicit languages. In implicit languages, the insert and lookup operations may be performed at any time. The various ways a symbol table can be stored are as follows and each has its own advantages and disadvantages.

- Linear list
 - Array
 - Ordered
 - Unordered
 - Linked list
 - Ordered
 - Unordered
 - Hierarchical
 - Binary search tree
 - Height balanced tree
- Hash table.

Non-block Structured Language

Linear List in Non-block Structured Language

It is the simplest form of organizing the symbol table to add or retrieve attributes. The list can be ordered or unordered. Consider the following example in Ada language shown in [Figure 9.3](#).

```
void Number(int Mike, int Alice, int John_Smith, float F=1.0)
{
printf( "Enter value of Mike \n" );
scanf("%d",&Mike);
printf( "Enter value of Alice\n");
scanf("%d",&Alice);
John_Smith = 3*Mike + 2*Alice + 2;
printf("%d\n",3*Mike + 2*Alice + 11);
printf("%d\n",John_Smith);
John_Smith=Mike + Alice+1000000;
printf("A million more than Mike and Alice %d\n", John_Smith);
F=F+float(Mike) + 3.1415265;
printf("F as an Integer %d\n".F);
}
```

Figure : Program in Ada

In the above example, the variables are Mike, Alice, and John_Smith declared as integers and F declared as float, and value assigned as 1.0.

Ordered List

In an ordered symbol table, the entries in the table are lexically ordered on the variable names. Every insertion operation must be preceded with a lookup operation to find the position of the new entry. Two search techniques can be applied, that is, linear or binary search. The following [Figure 9.4](#) shows

Name	Type	Size	Value
Mike	Integer	1

(a) After inserting Mike

Name	Type	Size	Value
Alice	Integer	1
Mike	Integer	1

(b) After inserting Alice

Name	Type	Size	Value
Alice	Integer	1
John_Smith	Integer	1
Mike	Integer	1

(c) After inserting John_smith
the order in which these variables are inserted.

Name	Type	Size	Value
Alice	Integer	1
F	Float	1	1.0
John_Smith	Integer	1
Mike	Integer	1

(d) After inserting F

Figure 9.4 Insert operation in ordered list

Performance:

Insert operation has the overhead of moving the elements to find the place to insert the new element. On the average, the lookup time would be $(n + 1)/2$ if linear search is used. This can be improved if binary search is used and it would be proportional to $(\log n)$.

Unordered List

As shown in [Figure 9.5](#), it is easy to insert the elements in an unordered list, since it inserts at the end. The look up time increases as it has to search the entire list to fetch the information of variables.

Name	Type	Size	Value
Mike	Integer	1

(a) After inserting Mike

Name	Type	Size	Value
Mike	Integer	1
Alice	Integer	1

(b) After inserting Alice

Name	Type	Size	Value
Mike	Integer	1
Alice	Integer	1
John_Smith	Integer	1

(c) After inserting John_smith

Name	Type	Size	Value
Mike	Integer	1
Alice	Integer	1
John_Smith	Integer	1
F	Float	1	1.0

(d) After inserting F

Figure unordered list – Insert

Performance:

If the language supports explicit declaration then there is no need to perform a lookup operation for every insertion. But to avoid duplication, complete table checkup may be needed after all insertions. The lookup operation requires, on an average, a search length of $(n + 1)/2$ assuming there are n records in the table. This value is derived based on the position of the element while comparisons are made as follows:

$$\frac{1}{n} \sum_{i=1}^n i = \frac{1}{n} (n + 1) * (n)/2 = (n + 1)/2$$

If the element is not found, it indicates an error condition and the error handler should handle it.

If variables are declared implicitly, attributes are added according to the order in which they are encountered during compilation. Every insert operation must be preceded by a look up operation. If the lookup operation returns 0, it indicates the absence of a variable and it must be inserted. In worst case scenario, it requires n elements to be checked and then inserted.

If only lookup operation is performed, then on an average it requires $(n + 1)/2$ comparisons. Let the ratio of first variable reference to total variable references be denoted by x ; then the lookup and insertion process requires on an average

$$x * n + (1 - x) * (n + 1) / 2$$

This unordered table organization should be used only if the number of variables are less or the table size is small, since the average time for lookup and insertion is directly proportional to the table size.

These tables are not suitable where the insert and look up operations are always applied. They are good to be used to store the reserved words in a language. The main drawback with arrays is the overhead and fixed size. This is overcome by the linked list.

Linked List or Self-organizing Tables

Ordered list

The variables information is inserted as shown in Figure 9.6 in the sequence encountered. The node requires an extra field to store the pointer to the next node and the last node in the list has NULL.

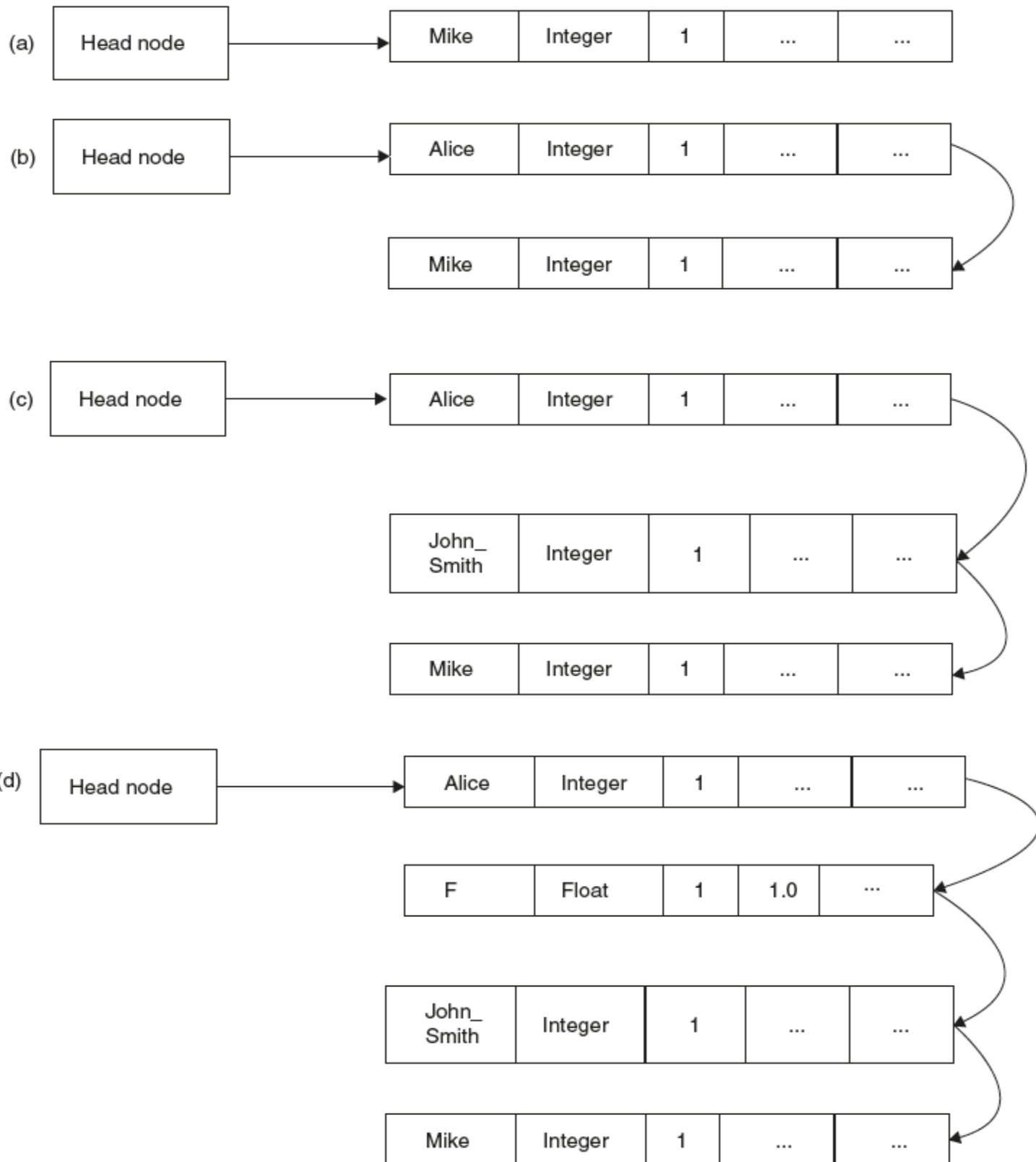


Figure Insertion sequence

For each insertion operation, it should check if the element is there in the list, if not present, then it creates a new node and places in the order changing only two link pointers. This makes the insertion to overcome the overhead of moving the elements. The time to insert the element is $O(n) + O(1)$. The lookup operation is always sequential with worst case $O(n)$.

Unordered Linked List

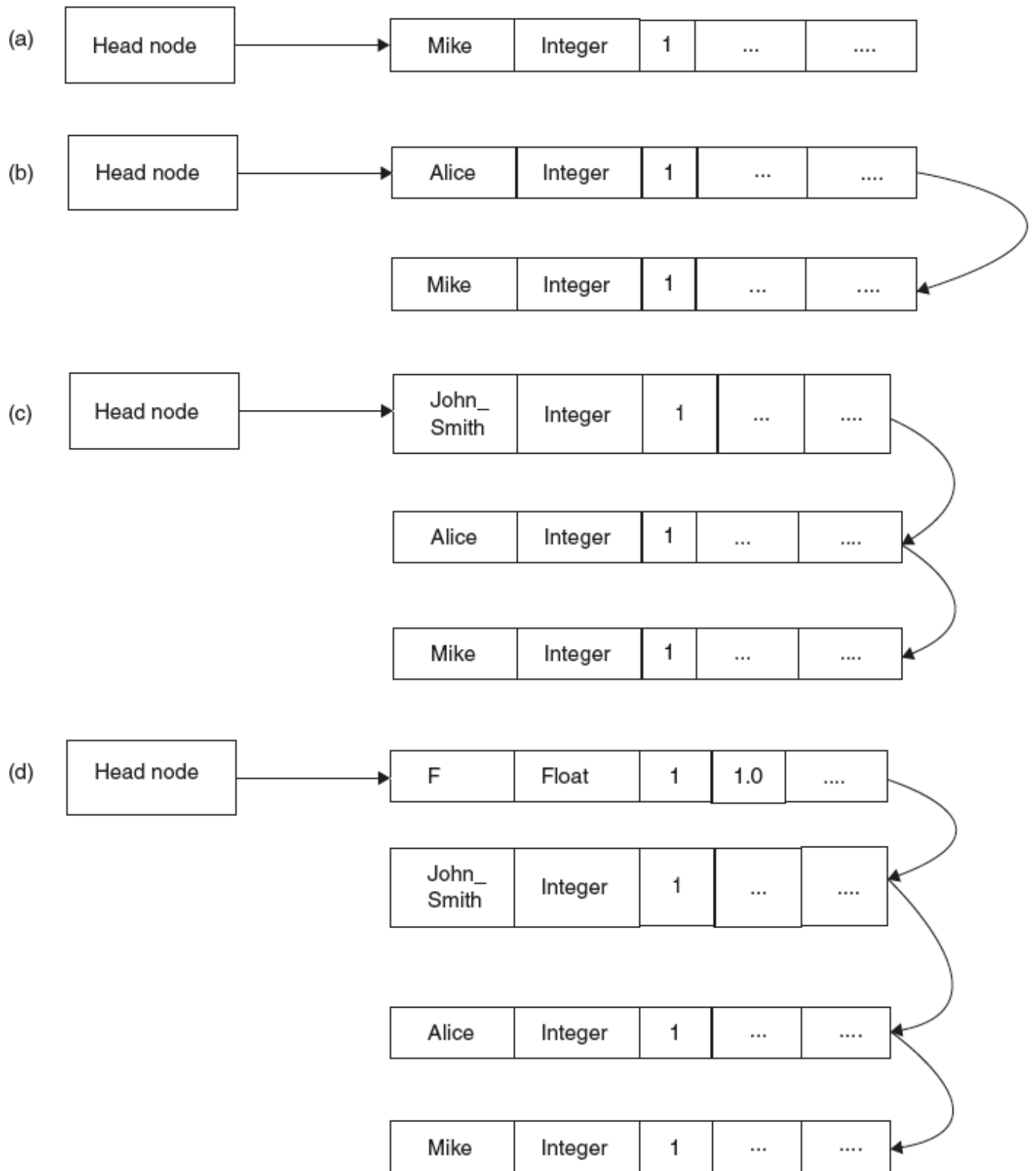


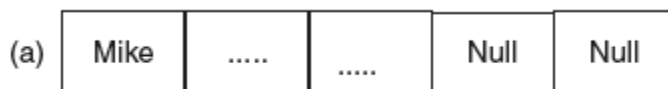
Figure Insertion in unordered list

In case of an unordered list, the time for insertion is $O(1)$ as the insertion is done at head node as shown in **Figure 9.7**. Each lookup operation always has the worst case time since it has to search the entire list.

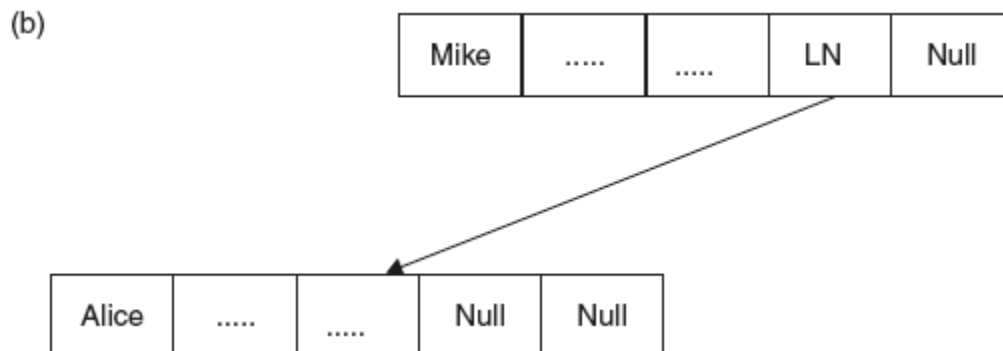
Hierarchical List

Binary search tree is a more efficient approach to symbol table organization. We add two links, left and right, in each record, and these links point to the record in the search tree. Whenever a new name is found decision is made either to add it or ignore it. First the name is searched in the tree if it exists, then it is ignored. If it does not exist, then a new record is created for the new name and is inserted as a leaf node. This organization follows a lexicographical order, that is, all the names accessible from name_i with value less than name_i are found by following a left link. Similarly, all the names accessible from name_i that follow name_i in alphabetical order are found by following the right link. The expected time needed to enter n names and to make m queries is proportional to $(m + n) \log_2 n$. So for a greater number of records (higher n), this method has advantages over linear list organization. For the example program the tree structure is shown below in **Figure 9.8**.

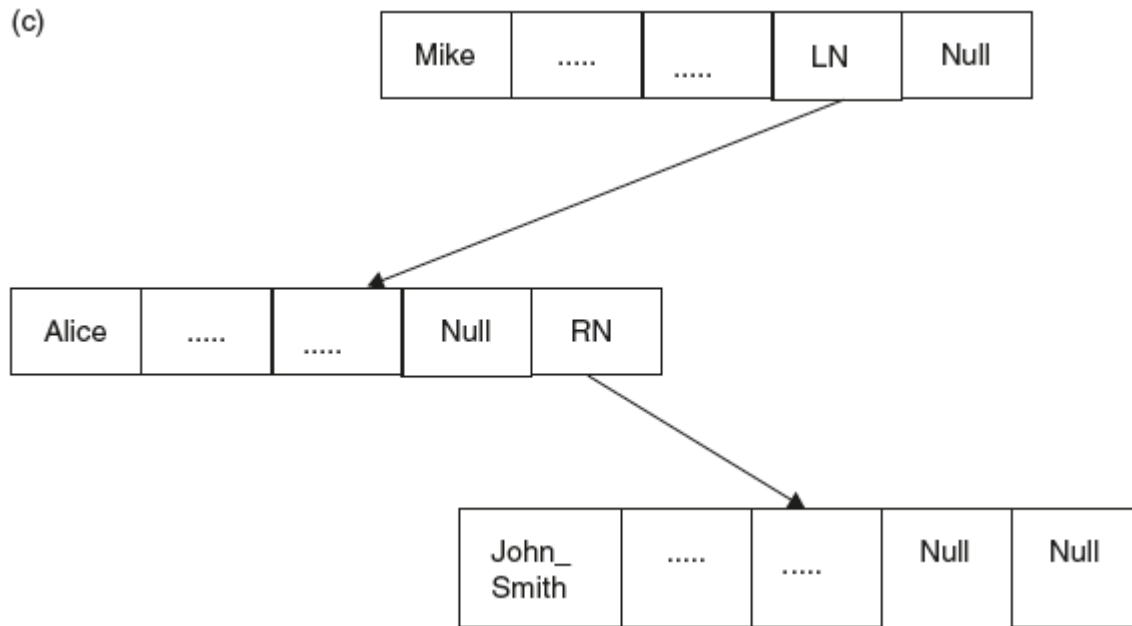
1. On inserting first variable Mike.



2. On inserting the second variable Alice.



3. On inserting third element John_smith.



4. On inserting the last element F the tree is as shown below.

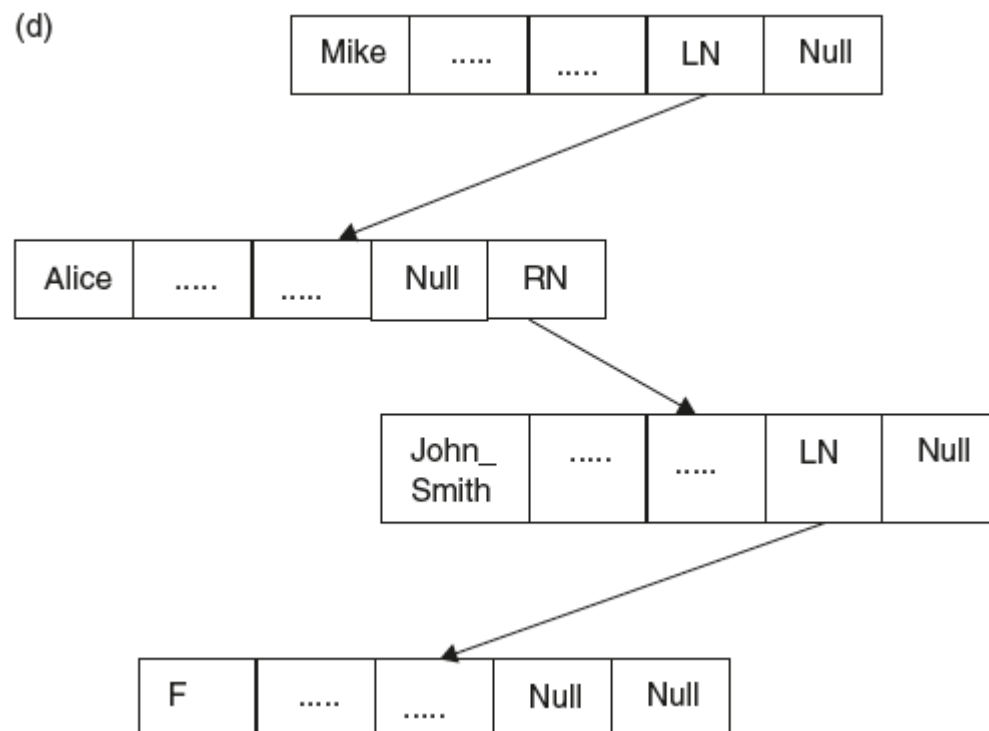


Figure Insertion in hierarchical list

From the above example, it is clear that the tree structure may not always give better performance. If the tree formed is a skewed tree (each level has only one node), then the time complexity is again dependent on the number of elements in the program. This can be overcome by using height balanced trees like AVL trees or 2 – 3 trees.

AVL Trees

An AVL tree is similar to a binary search tree but involves balancing operations after insertions and deletions when it leaves the tree unbalanced. These operations are called rotations, which help to restore the height balance of the sub-trees.

Lookup

Lookup in an AVL tree is performed exactly as in an unbalanced binary search tree. The only difference lies in the time taken to execute the lookup operation. It is proportional to the height of the tree which is $O(\log n)$. The time for search is maintained as $O(\log n)$. The efficiency can be further increased by adding index number to each node and the elements are retrieved based on index.

The parent or child node can be explored in amortized constant time after a node is searched in the tree. The traversing requires maintaining at most $2 \times \log(n)$ links. If it is required to explore n nodes it may require at most approximately $2 \times (n - 1)/n$ links, which is just 2.

Insertion

Every node in an AVL tree is associated with a balance factor bf . This bf is the difference of height of left and right sub-trees.

$$bf = H_l - H_r$$

The permissible value can be -1 , 0 , or $+1$ if the value is 0 then the node is balanced. If the value is 1 , then we say the node is left heavy as the left sub-tree has height one more than the right sub-tree. If it is -1 , then we say it is right heavy as the height of the right sub-tree is one more than the left sub-tree. However, if the balance factor becomes ± 2 , then the sub-tree rooted at this node is unbalanced and rotation is applied to balance the sub-tree.

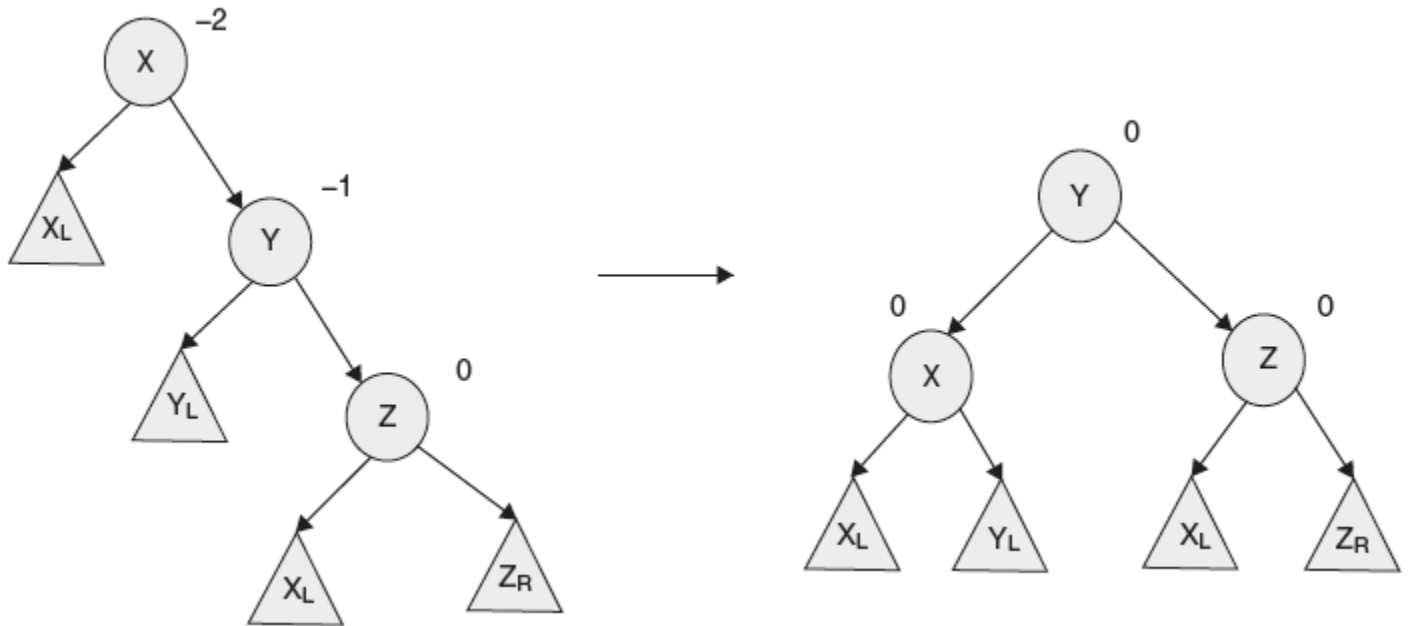
After inserting a node, the balance factor is adjusted for all the nodes that lie in the path from the point of insertion and the root. If there is any node whose value is ± 2 , then depending on the condition, the balancing rules are applied to balance it and are shown in [Figure 9.9](#) to [9.12](#).

The following figures explain how rotations can rebalance the tree, proceeding toward the root and updating the balance factor of the nodes that lie in the path. There are four types of rotations where two are symmetric to the other two in opposite directions.

Right–Right (RR)

This case occurs when X is the root sub-tree with Y as the right child of X . Let Z be the right child of Y . Let the height of X_L , Y_L , Z_L , Z_R be H . Then the bf of Z is 0 , Y is -1 and for X is -2 . Hence, node X

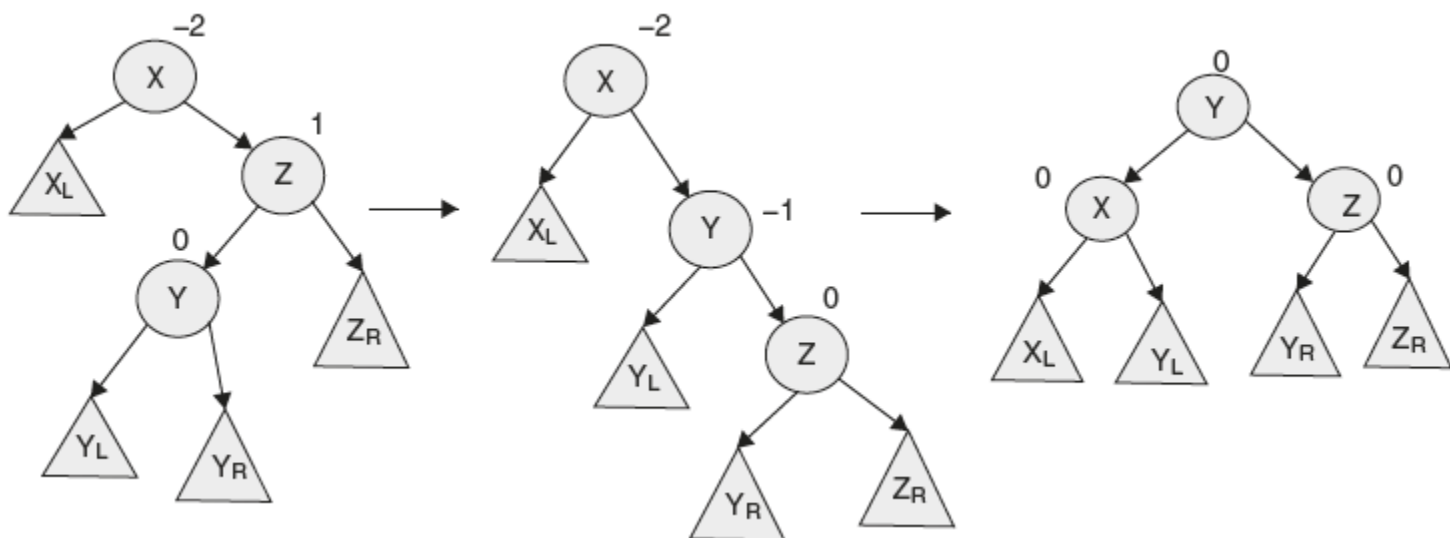
is critical. Since the node is critical to the right and the right child is right heavy, we apply Right–Right rotation (single). The resultant tree has the node Y as root of the tree and X as left child and Z as right child. The sub-trees left to Y are adjusted as shown in the [Figure 9.9](#).



Figure

Right–Left (RL)

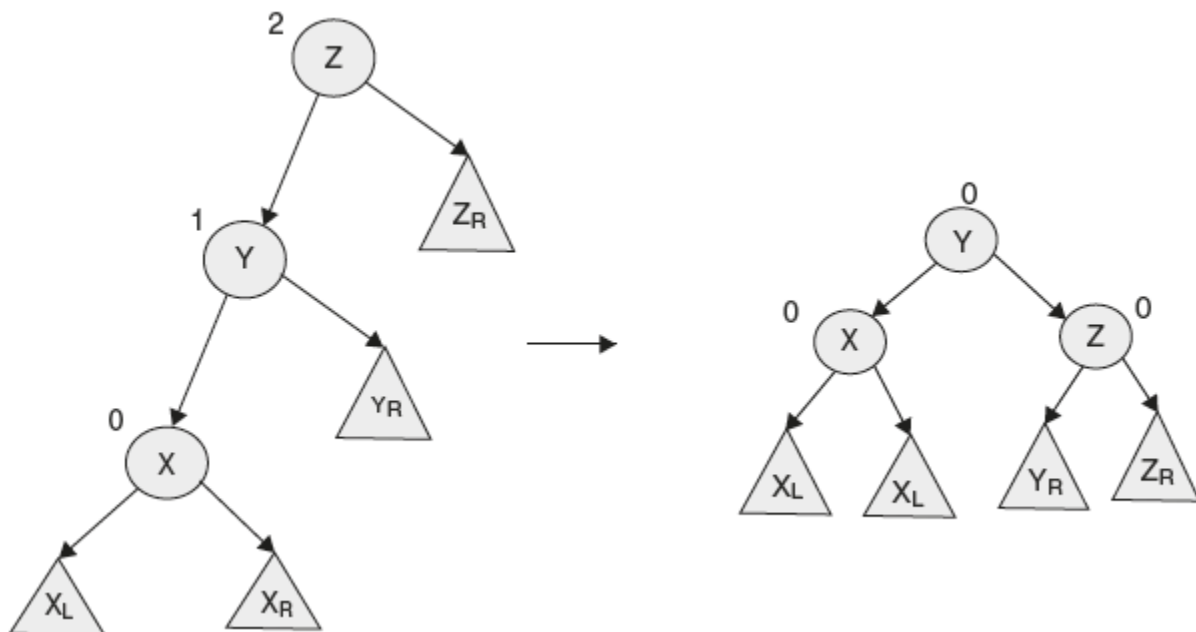
This case occurs when X is the root sub-tree with Z as the right child of X. Let Y is the left child of Z. Let the height if X_L, Y_L, Y_R, Z_R be H. then the bf Y is 0, Z is 1 and for X is -2 . Hence node X is critical. Since the node is critical to the right and the right child is left heavy, we apply Right–Left rotation (double). The resultant tree has the node Y as the root of the tree and X as the left child and Z as the right child. The sub-trees left to Y are adjusted as shown in the [Figure 9.10](#).



Figure

Left-Left (LL)

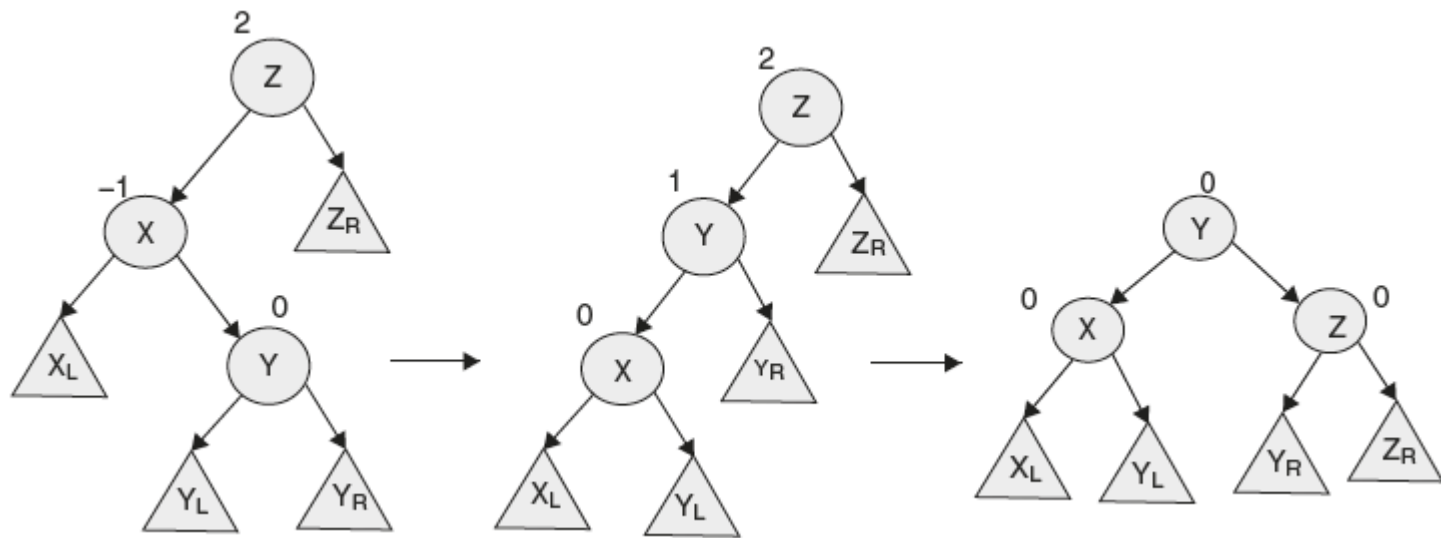
This case occurs when Z is the root sub-tree with Y as the left child of Z. Let X be the left child of Y. Let the height of X_L , X_R , Y_R , Z_R be H. Then the *bf* of X is 0, Y is 1, and for Z is 2. Hence, node Z is critical. Since the node is critical to the left and the left child is left heavy, we apply Left-Left rotation(single). The resultant tree has the node Y as the root of the tree and X as left child and Z as right child. The sub-trees right to Y are adjusted as shown in the [Figure 9.11](#).



Figure

Left-Right (LR)

This case occurs when Z is the root sub-tree with X as the left child of Z. Let Y be the right child of X. Let the height of X_L , Y_L , Y_R , Z_R be H. Then the *bf* of Y is 0, X is -1 and for Z is 2. Hence node Z is critical. Since the node is critical to the left and the left child is right heavy, we apply Left-Right rotation (double). The resultant tree has the node Y as the root of the tree and X as the left child and Z as the right child. The sub-trees left to Y are adjusted as shown in the [Figure 9.12](#)



Figure

Deletion

Deletion of the node is similar to the binary tree. After deletion, the balance factor of the nodes is adjusted till it encounters the root node. If the balance factor for the tree is $+2$ / -2 and that of right/left sub-tree is 0 , then right/left rotation is performed at the root of that sub-tree.

While retracing, if the balance factor of any node has a value between -2 and $+2$, based upon the balance factor, do any one of the following.

- If the balance factor is either -1 or $+1$, then the tree remains unchanged so stop adjusting.
- If the balance factor is 0 , it indicates that the height of sub-tree is decreased by one, hence, it continues to retrace towards root.
- If balance factor is either -2 or $+2$ it indicates that the node is critical; hence, rotation is applied. If the balance factor of any node is 0 , then retrace toward the root.

The time required for the deletion operation is $O(\log n)$ as the time required for lookup and adjusting nodes backwards is $O(\log n) + O(\log n)$.

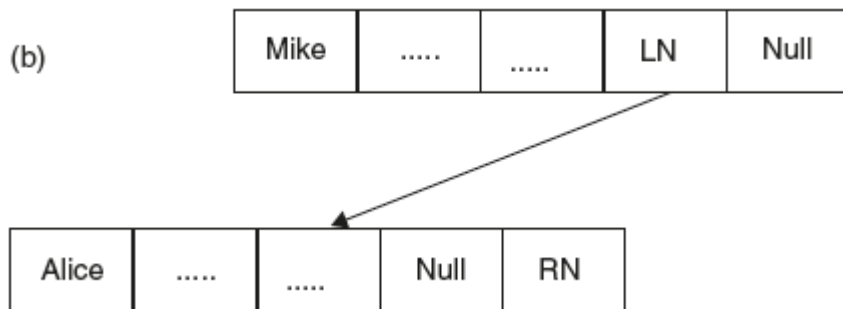
For the previous example, the AVL tree would be constructed as follows.

1. On inserting first variable Mike.

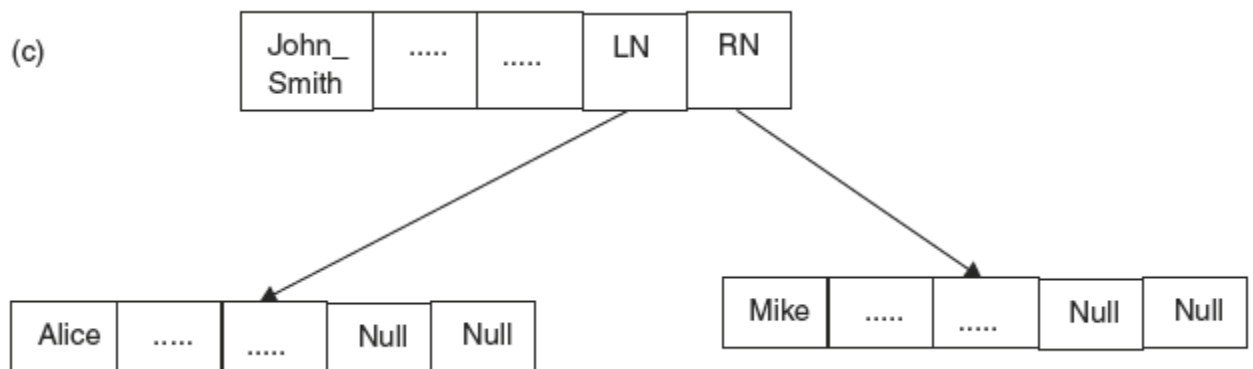
(a)

Mike	Null	Null
------	-------	-------	------	------

2. On inserting the second variable Alice.



3. On inserting third element John_smith it requires to apply rotation to balance the tree structure.



4. On inserting the last element F the tree is as shown below in [Figure 9.13](#).

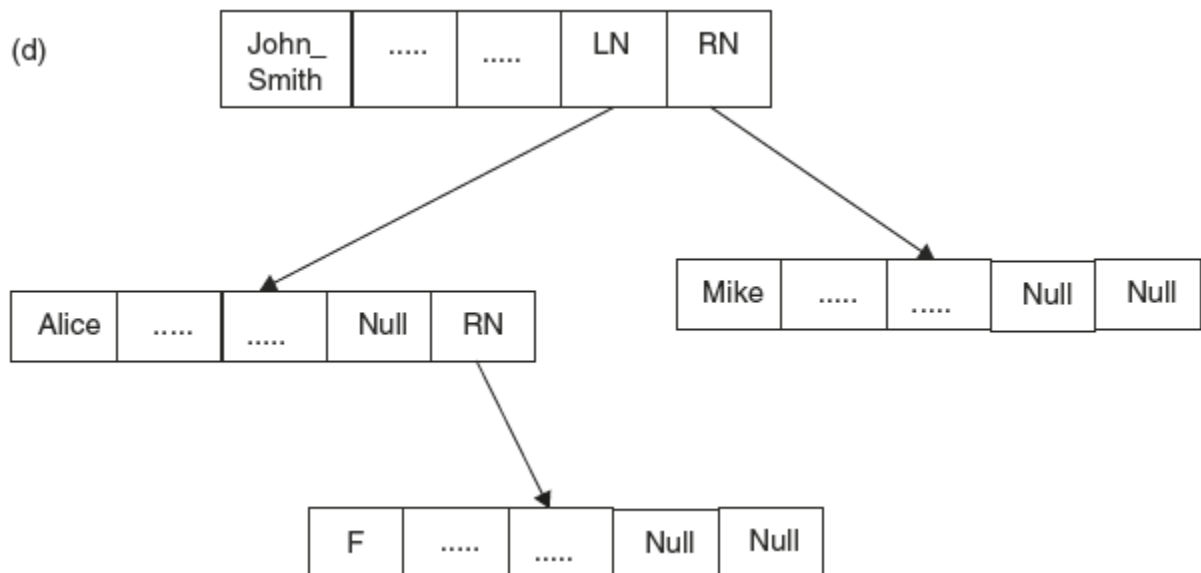


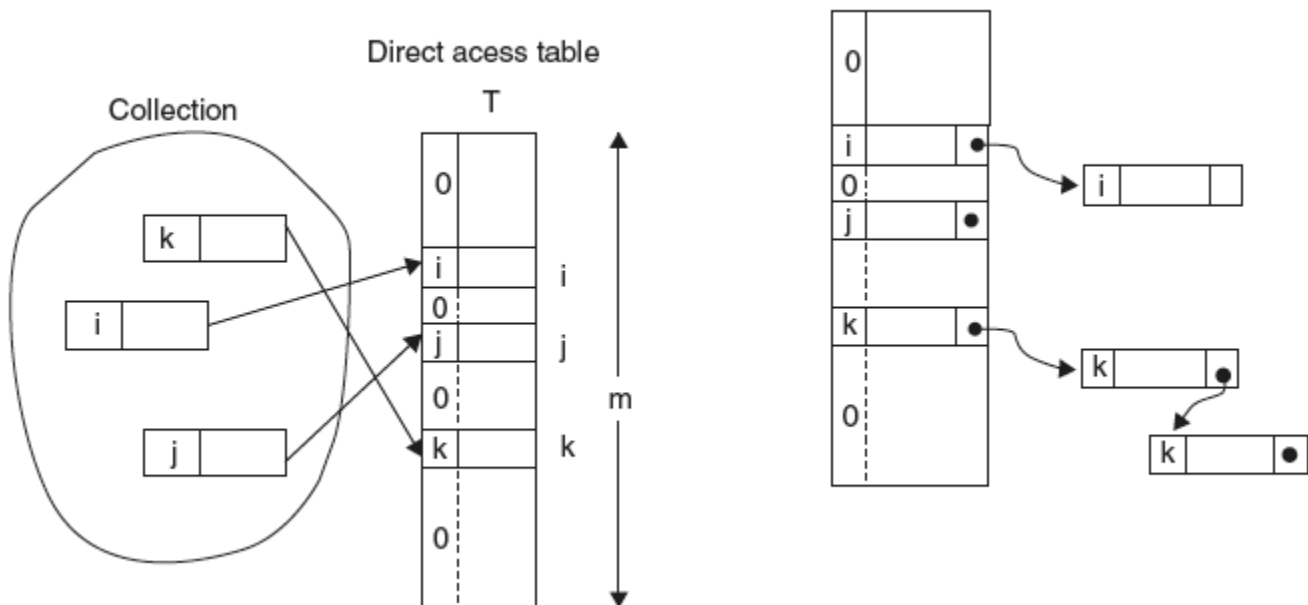
Figure 9.13

The data structures discussed so far have the time complexity that is expressed as a function of n . As the value of n becomes large, the time requirement also increases. There is a special structure where the operation time is independent of n .

Let there be m locations and n elements whose keys are unique. If $m \geq n$, then each element is stored in the table $T[m]$, so that the hash function applied on the key K results in the location T_i . If the location T_i is empty then the element is inserted; otherwise, if it contains an element it applies the second strategy to insert element. When searching for a key element K , in location T_i it would return the element if found, otherwise returns NULL. Sample table is shown in [Figure 9.14](#).

To use the hash table technique, it requires the keys to be unique. Also the range of keys must be bounded with the addresses of the locations.

Note: If keys are not unique, then there are various mechanisms that can be adopted. A simple method is to construct a set of m lists that store the heads of these lists in the direct address table. If the elements in the collection have at least one distinguishing feature other than the key, then the search for the specific element in that collection depends on the maximum number of duplicates.



Figure

Mapping Function

The direct address approach requires a hash function, $h(K)$ to map the key K to one of the location that is in the range $(1, m)$ where $1, m$ indicates the range of memory address. It is said to be a perfect mapping function if it is one-to-one as it results in search time that is $O(1)$.

It is easy to define such a perfect hash function theoretically but practically it is always not possible. For example, consider $(1, m)$ to be $(0, 100)$. To map elements with keys 12, 112, 512, if the

hash function is $K\%100$, then all the keys are mapped to the 12th location. When more than one key is mapped to the same location, we call the condition, as collusion. When there are collusions, then more than one element has to be stored in the same location; this is not possible. In such a condition, we apply collusion-resolving techniques.

Handling the Collisions

Techniques have to be applied for resolving collisions for easy insertion and search. The following is the list of collusion-resolving techniques.

1. chaining,
2. re-hashing,
 1. linear probing(using neighboring slots),
 2. quadratic probing,
 3. random probing.
- . overflow areas

Chaining

It is the simplest technique to chain all the collisions in the list attached to the appropriate slot. This method doesn't require a priori knowledge of how many elements are contained in the collection. This would exhibit poor performance if all the elements are mapped to the same location as it would create a linked list whose time is again proportional to $O(n)$.

Re-Hashing

This technique uses a second hash function when there is a collusion. This is repeated until it finds an empty space in the table. In general, the second function could be the same function with varying parameters constituting a new function. This technique requires applying the same hash function in the same order for searching of insertion. It also involves overhead in finding the elements and requires more hash functions.

a. Linear probing

Simple rehash function that can be chosen is $+1$ or -1 , that is, looking in the consecutive locations until a free slot is found as shown in Figure 9.15. It is easy to implement.

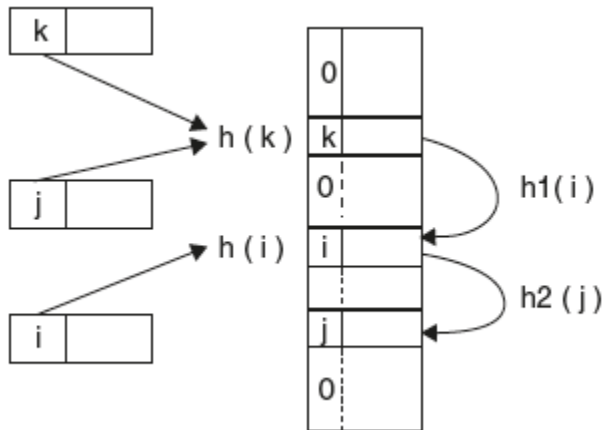


Figure 9.15

b. Random probing

This technique uses a random function to find the next location to insert the element. If the location has an element, then the random function is applied until a free slot is found. This ensures the proper utilization of the complete table as the random function generates the index which ranges uniformly. The problem is, the time taken by the random function.

c. Quadratic probing

In quadratic probing when a collision occurs, secondary hash function is used to map the key to address. The advantage of this technique is that the address obtained by secondary function is distributed quadratically.

Address = $h(\text{key}) + c \cdot i^2$ on i^{th} re-hash.

(A more complex function of i may also be used to have better performance.)

This re-hashing scheme uses the space within the allocated table avoiding the overhead of maintaining the linked list. To apply the approach it is required to know the items that are to be stored in advance. At the same time it adds up a drawback of creating collisions for other valid keys as the table space is pre-occupied by the elements that caused the collision earlier.

Overflow area

The table is divided into two sections in this technique. One is the primary area to which the keys are mapped and overflow area to take care of collisions as shown in [Figure 9.16](#).

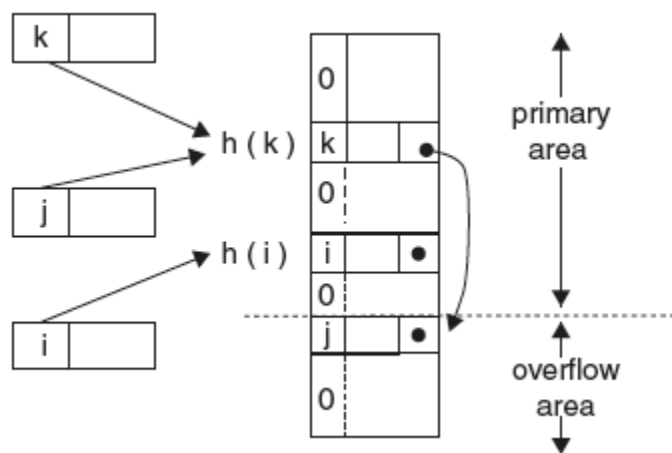


Figure 9.16

Whenever collision occurs, the space in overflow area is used for the new entry. This location is linked to the primary table space. This appears to be similar to chaining but there is slight difference in the allocation of table space. Here the extra space is allocated along the actual table; hence, it provides faster access. It is essential to know the size of elements before allocation of space for table

It is possible to design the system with multiple overflow tables, or with a mechanism for handling overflow out of the overflow area which provides flexibility without losing the advantage of the overflow scheme.

The following table gives the summary of the hash table organization:

Organizatio n	Advantages	Disadvantages
Chaining	Unlimited number of elements Unlimited number of collisions	Overhead of multiple linked lists
Re-hashing	Fast re-hashing Fast access through use of main table space	Maximum number of elements must be known Multiple collisions may become probable
Overflow area	Fast access Collisions don't use primary table space	Two parameters which govern performance need to be estimated

Example: Let the variables names be

Mike, Alice: Integer;

John_Smith: Integer;

F: Float := 1.0;

Let the hash function be chosen as the sum of ASCII representation of each alphabet and let the table size be 10. We use linear probing to overcome the collusion.

Solution:

The result of hash function on each variable is shown in the table below.

Variable Name	Sum of ASCII values	Total value(TV)	Hash function (TV)/10	Mapping location
Mike	77+105+107+101	390	390/10	0
Alice	65+108+105+99+101	478	478/10	8
John_Smith	74+111+104+110+95+83 +109+105+116+104	1011	1011/10	1
F	70	70	70/10	0

The insertions are done as shown in Figure

Insert Mike

0	Mike
1	
2	
3	
4	
5	
6	
7	
8	
9	

Insert Alice

0	Mike
1	
2	
3	
4	
5	
6	
7	
8	Alice
9	

(a) Insert John_Smith

0	Mike
1	John_Smith
2	
3	
4	
5	
6	
7	
8	Alice
9	

(b)

(c) Insert F

0	Mike
1	John_Smith
2	F
3	
4	
5	
6	
7	
8	Alice
9	

(d)

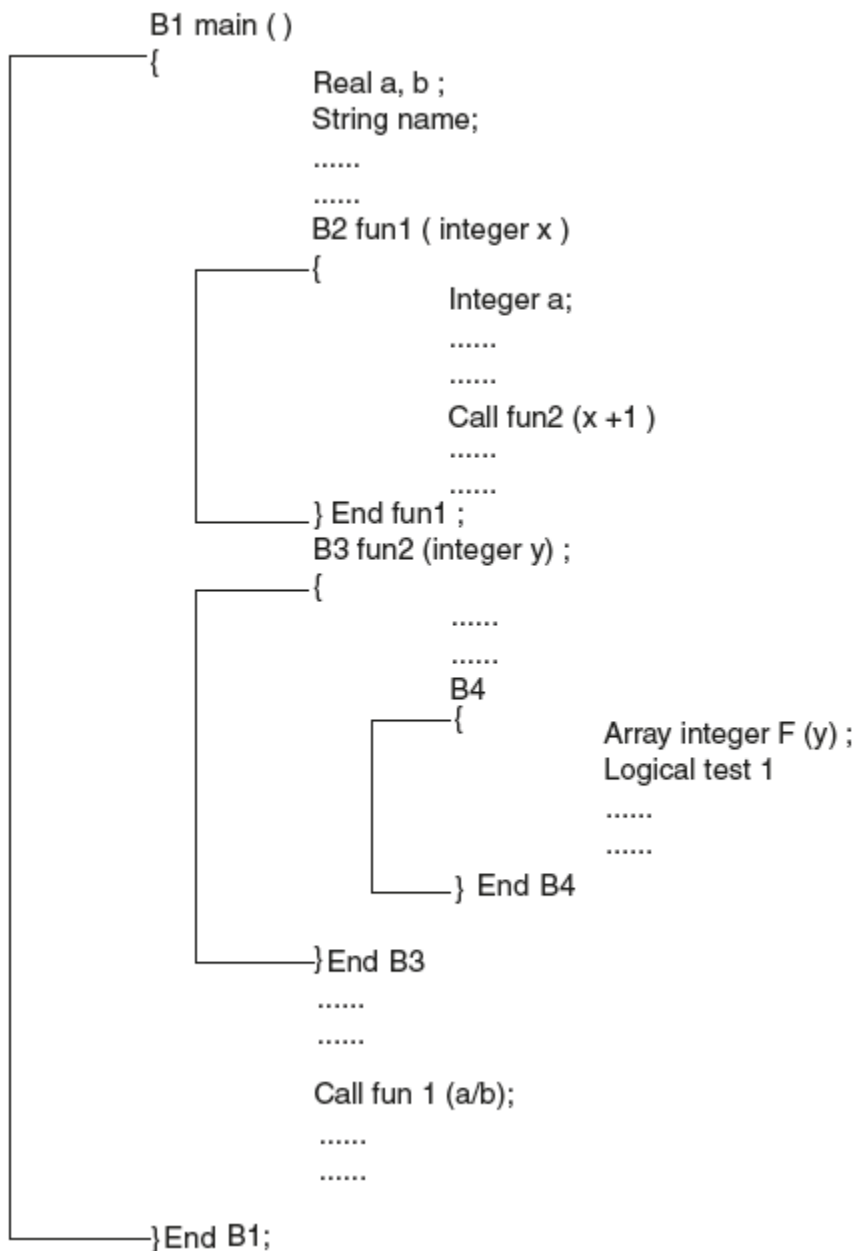
Figure 9.17

Block Structured Language

Block structured languages comprise a class of high-level languages in which a program is made up of *blocks*, which may include *nested blocks* as components, such nesting being repeated to any depth. A block is a group of statements that are preceded by declarations of variables that are visible throughout the block and the nested blocks. These declarations are invisible if the inner blocks have the same variables declared. Once the scope of the inner block is completed, the variables of the outer block become effective. Variables are said to have *nested scopes*. The concept of block structure was first introduced in the Algol family of languages. The symbol table organization in block structured languages is complex, compared to the structured languages. It

requires the additional information to be stored at every block entry and exit. Let us consider the following example shown in Figure 9.18.

This program contains four blocks. **B1** is main that has two inner blocks **B2** and **B3**. Block **B3** has another inner block **B4**. On execution first the **main** is called, which invokes the function **fun1**; **fun1** in turn calls **fun2**, which includes **B4**. On completion of **B4**, it executes the remaining part of **fun2()**. On completion **fun2()** it returns to next statement of **Call fun2()** in **fun1()**. On completion of **fun1()** it executes the next statement of **call fun1()** in **main**.



Figure

During execution, these two blocks behave differently, but during compilation both types of blocks require similar types of processing. During the compilation, at the block entry, a sub-table should be created for new variables using the set operation. For the variables declared with the same name in both inner block and outer block, care should be taken, so that the variables of the outer block are inactive and the variables in the inner block are active. At the block exit, these entries should be deleted using the reset operation. The following is the trace of compilation with set and reset operations at every block.

B1 entry: Set operation is performed and no variable is either active or inactive.

B2 entry: Set operation is performed and at this moment variables *a*, *b*, *name* and *fun1* are active. No variable is inactive.

B2 exit: Reset operation is performed, *a*, *b*, *name*, *fun1*, *x* and *a* are active. No variable is inactive.

B3 entry: Set operation is performed, *a*, *b*, *name*, *fun1* and *fun2* are active. *x* and *a* are inactive.

B4 entry: Set operation is performed, *a*, *b*, *name*, *fun1*, *fun2* and *y* are active. *x* and *a* are inactive.

B4 exit: Reset operation is performed, *a*, *b*, *name*, *fun1*, *fun2*, *y*, *F* and *test1* are active. *x* and *a* are inactive.

B3 exit: Reset operation is performed, *a*, *b*, *name*, *fun1*, *fun2* and *y* are active. *x*, *a*, *F* and *test1* are inactive.

B1 exit: Reset operation is performed, *a*, *b*, *name*, *fun1* and *fun2* are active. *x*, *a*, *F*, *test1* and *y* are inactive.

End of compilation: All variables *a*, *b*, *name*, *fun1*, *fun2*, *x*, *a*, *F*, *test1* and *y* are inactive.

Note: In block B2 both the instances of **a** are active but the lookup operation should return the attributes of the recent instance of **a**. Hence, the best suitable data structure that can be used is a stack.

Stack Symbol Tables

In this organization, records containing the attributes of the variables are stacked as they are encountered. At the block exit these records are deleted since they are not required outside the

block. This organization contains two stacks—one that holds the records and the other that holds the block index details. The four operations performed on the table are explained below.

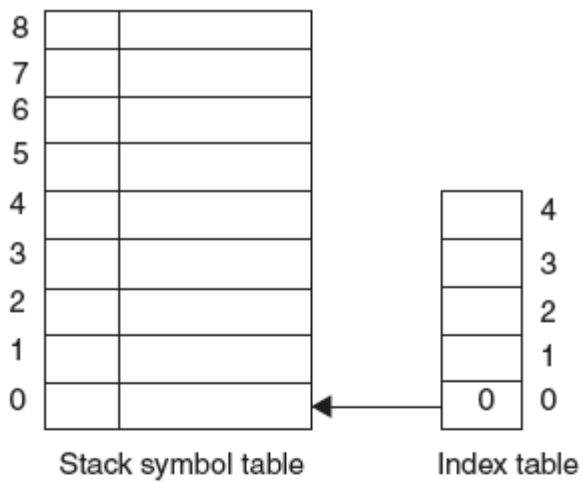
Set: This operation generates a new block index entry at the top of block index table, which corresponds to the top of the symbol stack. This entry marks the start of the variable in the new block.

Reset: This operation removes all the records corresponding to the current completed block. This corresponds to setting the top in symbol stack to the value pointed by the block index and popping the current top in block index.

Insert: This operation is simple and involves in adding the new record on top of symbol stack. This operation requires examining that no duplicates exist in the same block.

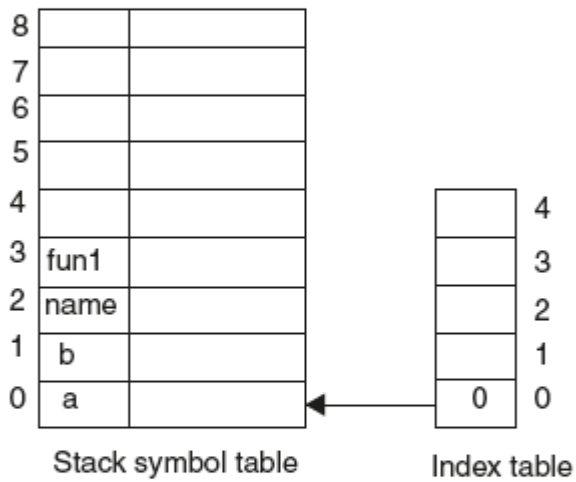
Look up: This operation is similar to the linear search of the table from the top to bottom. It searches for the variable that is the latest declaration.

For the above example, the following Figure 9.19 shows how the variable information is added or removed after the entry/exit of every block.



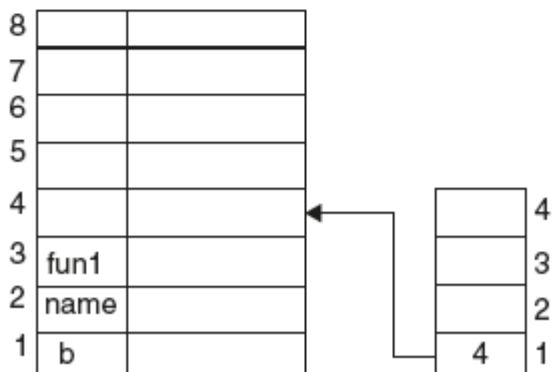
(a) Set at B1 entry

Insert a , b , name , fun1

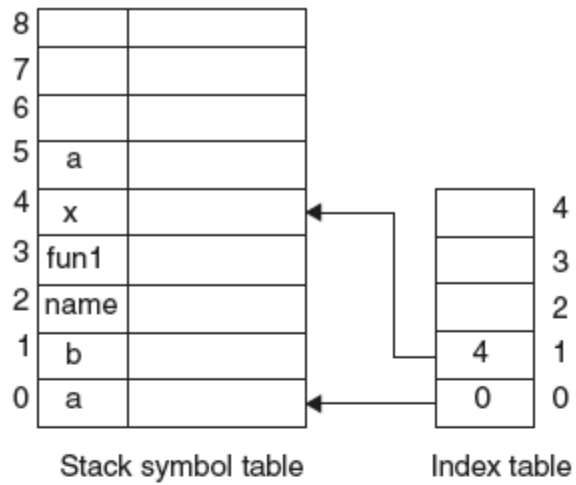


(b) Insert a, b, name ,fun1

Set at B2 entry

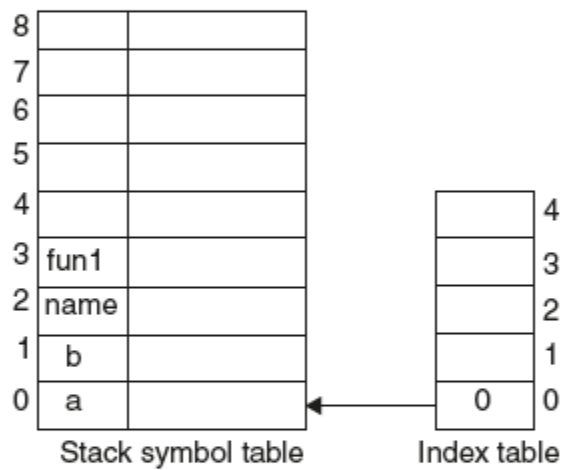


Insert x , a



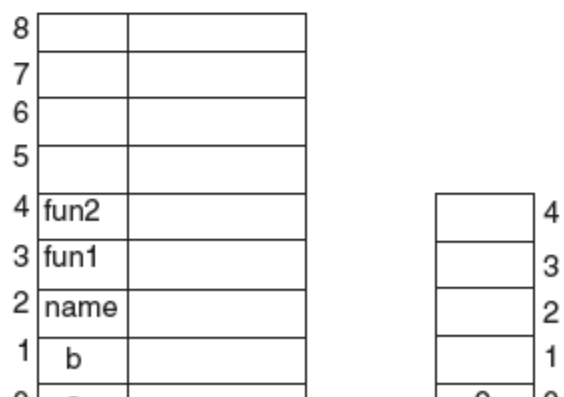
(d) Insert x , a

Resert at B2 exit & Delete x , a

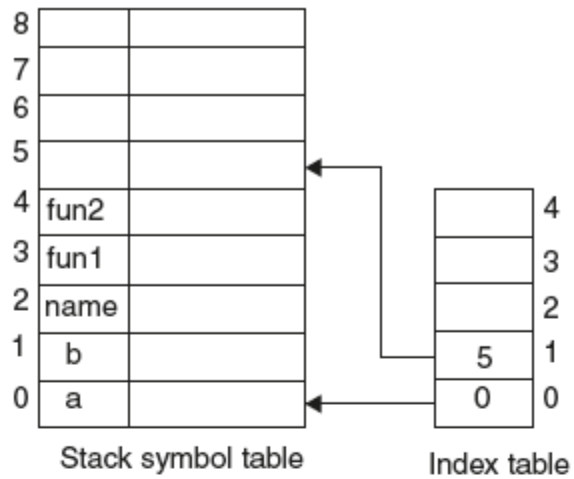


(e) Delete ex,a & Resert at B2 exi

Insert fun2

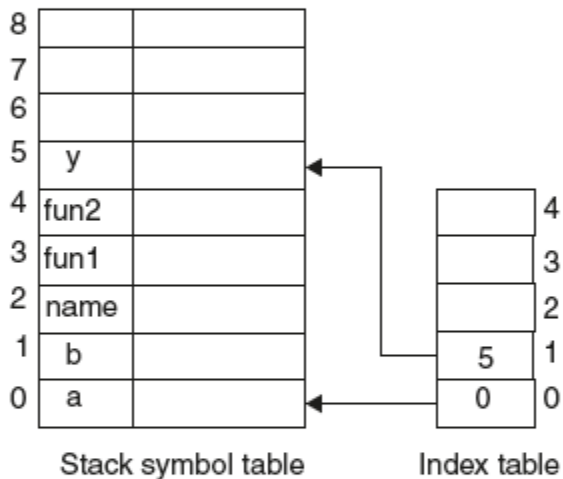


Set at B3 entry



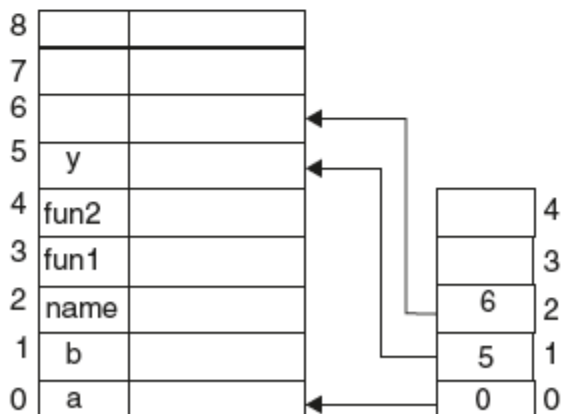
(g) Set at B3 entry

Insert y

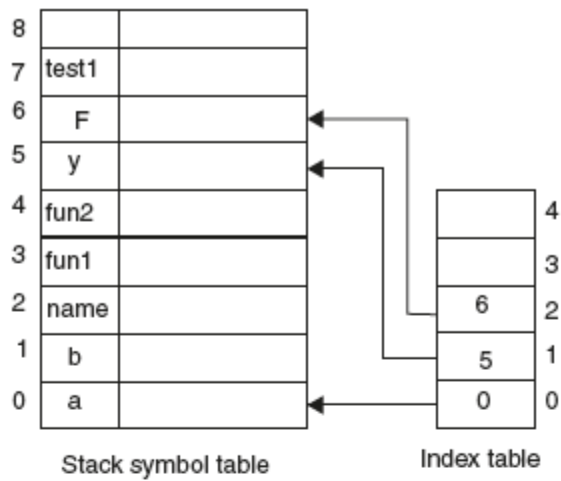


(h) Set at B3 entry

Set at B4 entry

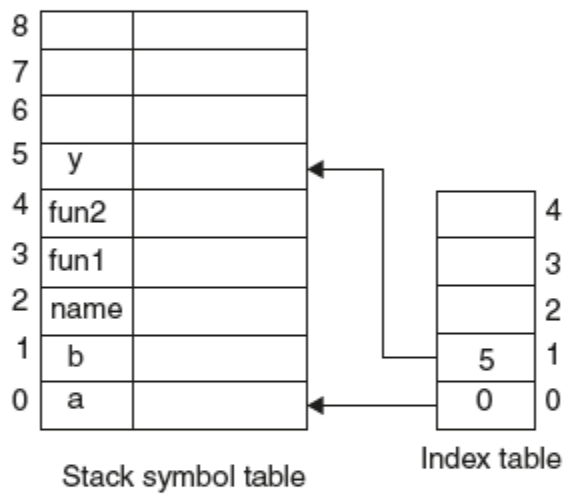


Insert F , test1



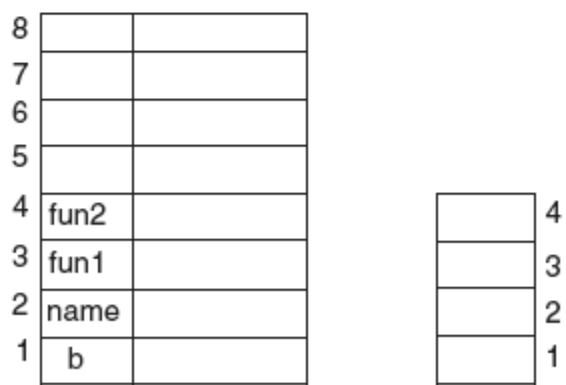
(j) Insert F and test1

Reset B4 exit and Delete test1, F



(k) Reset at B4 exit

Reset at B3 exit and Delete y



Reset at B1 exit and Delete a , b , name, fun1, fun2

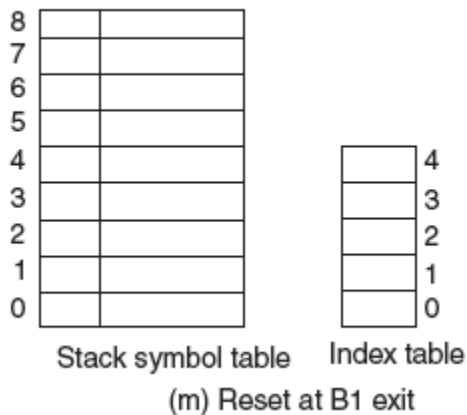


Figure 9.19

Stack-Implemented Tree-structured Symbol Tables

Tree structured symbol table organization can be done in two different forms for block structured languages. The first approach is using a single tree for all the variables. This involves removal of records for a block, when the compilation of the block is completed. Since the records of all the blocks are merged as one tree, it requires a complex procedure to address the required records while applying the operations on the tree.

Insertions are always done as the leaf node; care should be taken while performing a lookup operation to ensure that the reference is for the current block. Every deletion operation involves the following steps.

1. Locate the position of the record in the tree.
2. Remove the record from the tree and adjust the links to bypass the node.
3. Rebalance the tree if the deletion causes the tree unbalanced.

Note: Single tree structure may not be suited to compile the nested languages.

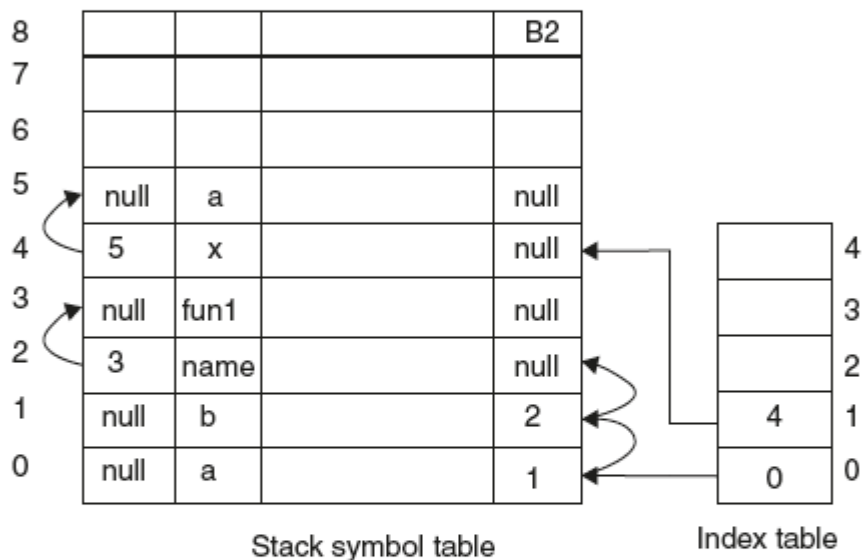
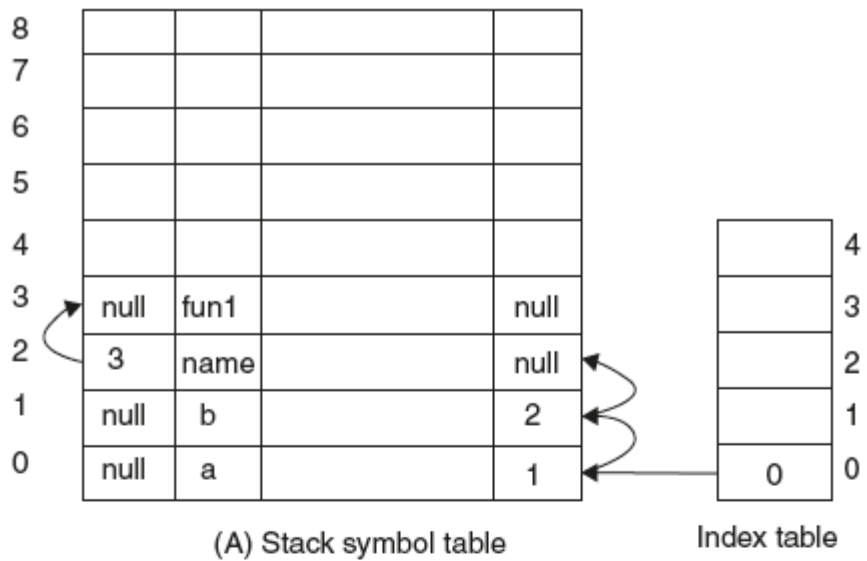
The second approach is to construct a forest of trees where each block has allocated its own tree—structured table. When the compilation of the block is complete, the entire tree for that block is deleted.

In this organization, the node for each record is associated with two special pointers along with all the attributes—one left pointer to point the left node and right pointer to point to right node. The symbol table is maintained as a stack. When the block is entered during compilation, the value of top of stack table is stored at the top of the block index table. As decelerations are encountered, records are inserted at the top of the symbol table and rebalanced if the insertions make it unbalanced.

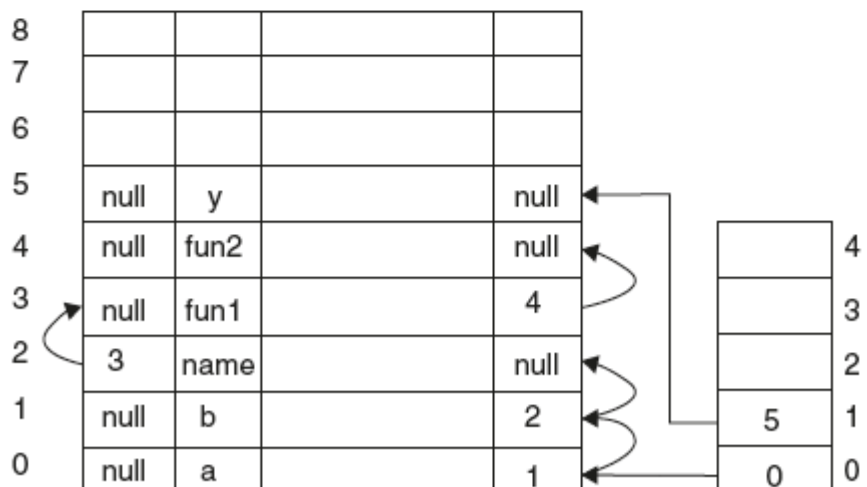
A lookup operation must ensure that the latest occurrence is located. The search must begin at the tree structure for the last block to be entered and proceed down the block index table till it points the root of the tree for the first block entered. For example, to lookup for variable “a”

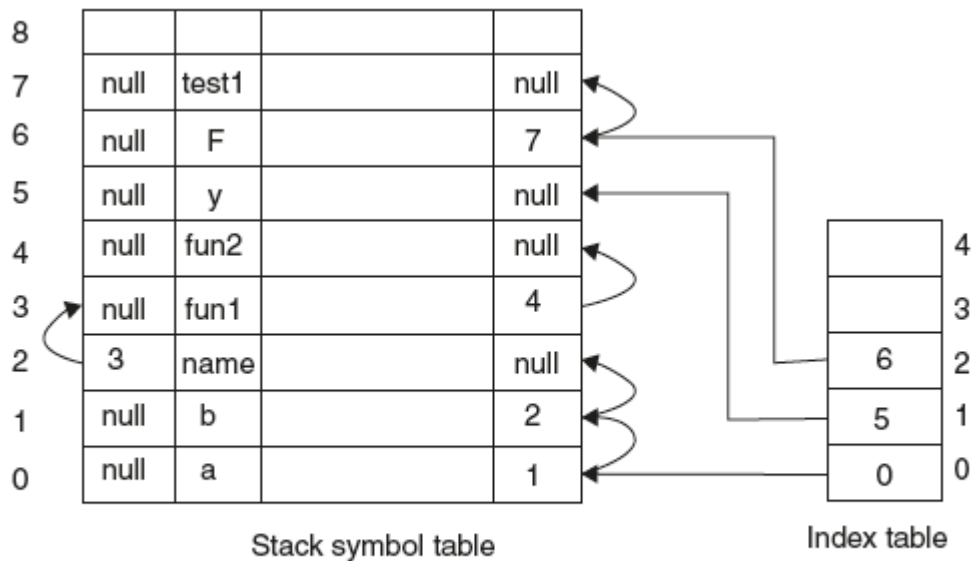
in Figure 9.20B, it first starts the search at the root pointed by the top of the block index. The top points to location 4, compares with it, and searches in the left sub-tree until it is found and returns the index as 5. When the search is for b, the sub-tree pointed by the top of block index returns null; hence a search is made in the sub-tree pointed by the (top-1) and returns the location is at 1.

The following figure shows the operations on every entry and exit of the block.



(B) Before call to fun2 in B2





(D) In the Block B4

Figure 9.20

9.6.3 Stack-Implemented Hash-Structured Symbol Table

Applying hashing technique is complex for block structured languages, as it requires some techniques to preserve the information regarding the variables of same block. This is achieved by using an intermediate hash table. The hash table stores the link to the location in the stack symbol table where the variable is stored. The stack symbol table stores the information of the variable along with the link, to the location of the variable which maps to the same location in hash table. The block index table stores the starting location of the variables on the current block. The operations performed are also complex.

Set: On every block entry, the current top of the stack symbol table is stored in block index table.

Insert: First the hash function is applied on the key

- If it maps to a location with no collusion, then the variable information is stored in the current top of stack symbol table and that index is stored in hash table.
- If it maps to a location with collusion, then the index stored in the hash table is stored along with the variable information in the current top of the stack symbol table and this index is updated in the hash table. This enables to store the variable information without losing the information of the variable previously inserted.

Lookup: First the hash function is applied on the key,

- If the hash table has null, then return null.
- If it points to the some location in the stack symbol table,
 - If it is the required variable, return the index if the index is greater than the current top of the block index.
 - Otherwise, use the link pointer to go to next location and perform step a.

Reset: Delete all the variables from the stack symbol table whose index is greater than or equal to the current top in block index. While deleting the variable, it requires the following modifications.

- If the link field of the variable is null, then delete the variable and set the index in hash table to null which points it.
- If the link field is not null, then store this index in the hash table and delete the variable information.

Example: The symbol table organization for block structured languages using hashing technique is shown below.

Let the hash function be chosen as the sum of ASCII representation of each alphabet and let the table size be 10. We use linear probing to overcome the collusion.

Solution:

The result of hash function on each variable is shown in the table below. The [Figure 9.21](#) shows the content of hash table after set and insert operation in each block entry.

Variable Name	Sum of ASCII values	Total value (TV)	Mapping location (TV/10)
A	97	97	7
B	98	98	8
Name	110+97+109+101	417	7
fun1	102+117+110+49	378	8
X	120	120	0
fun2	102+117+110+50	379	9
Y	121	121	1
F	70	70	0
test1	116+101+115+116+49	497	7

Block B1: Four variables are inserted a , b , $name$, and $fun1$ where a and $name$ map to the 7th index and b and $fun1$ map to the 8th index in the hash table.

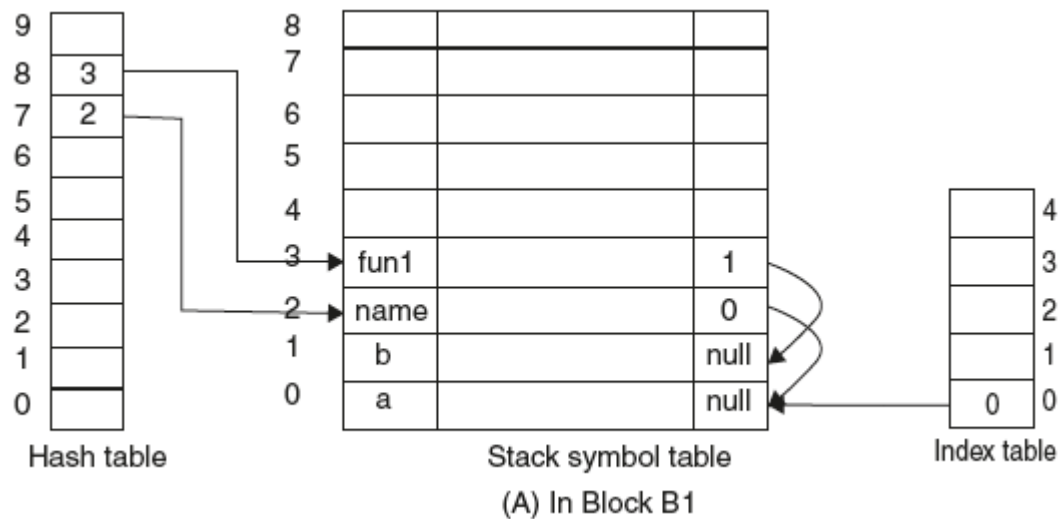
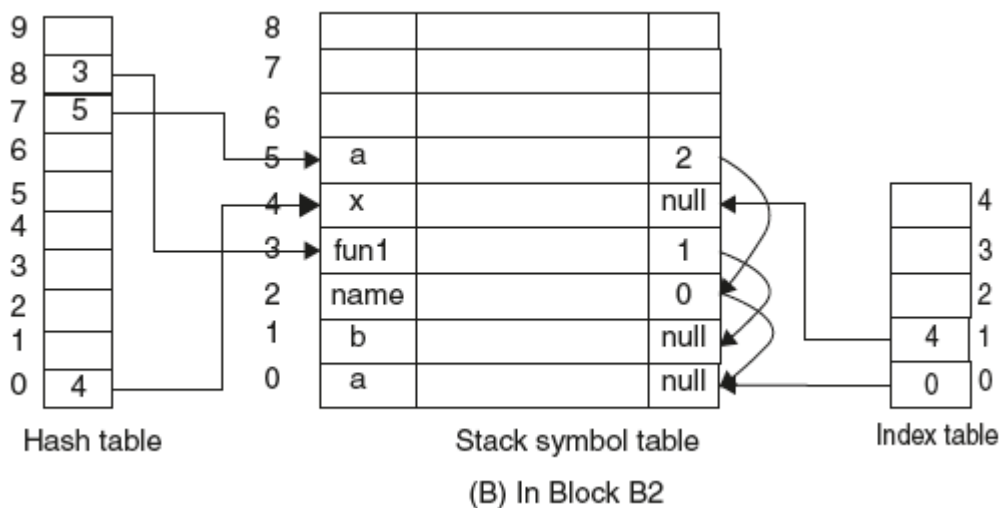
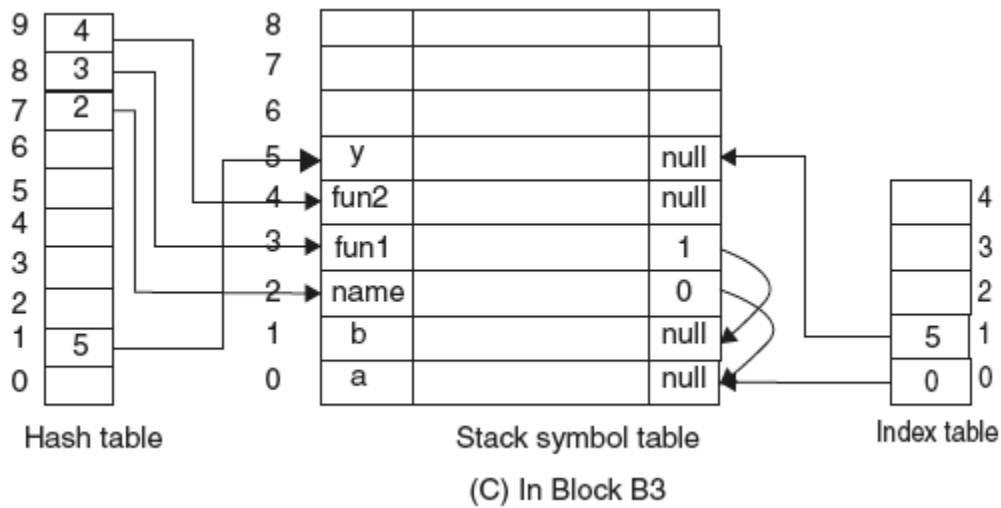


Figure 9.21

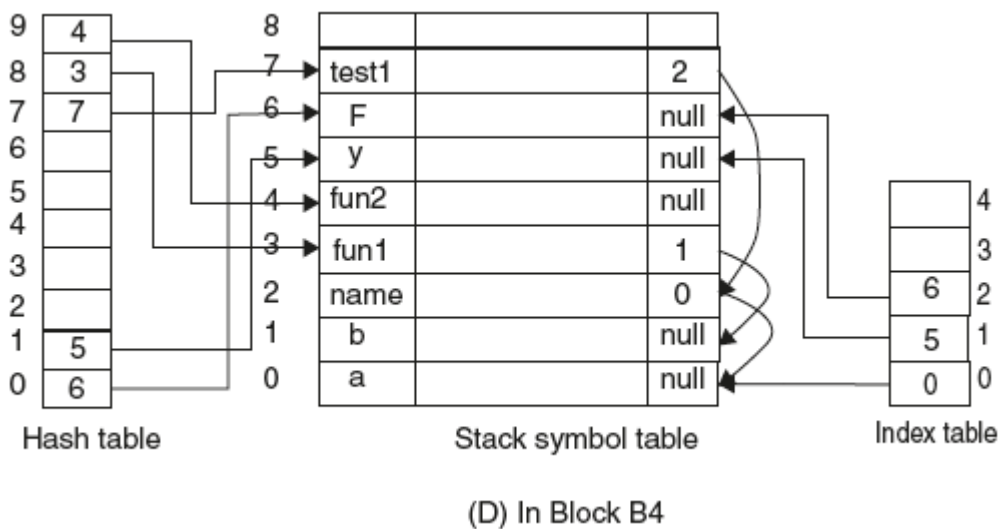
Block B2: Two variables x and a are inserted where a is mapped to the 7th index and x is mapped to the 0th index in the hash table.



Block B3: Two variables $fun2$ and y are inserted where $fun2$ is mapped to the 9th index and y is mapped to the 1st index in the hash table.



Block B4: Two variables F and test are inserted where F is mapped to the 9th index and test1 is mapped to the 1st index in the hash table.



Figure

UNIT-V

CODE OPTIMIZATION

1. INTRODUCTION

- The code produced by the straight forward compiling algorithms can often be made to run faster or take less space, or both. This improvement is achieved by program transformations that are traditionally called optimizations. Compilers that apply code-improving transformations are called optimizing compilers.
- Optimizations are classified into two categories. They are
- Machine independent optimizations:
- Machine dependant optimizations:

Machine independent optimizations:

Machine independent optimizations are program transformations that improve the target code without taking into consideration any properties of the target machine.

Machine dependant optimizations:

Machine dependant optimizations are based on register allocation and utilization of special machine- instruction sequences.

The criteria for code improvement transformations:

- Simply stated, the best program transformations are those that yield the most benefit for the least effort.
- The transformation must preserve the meaning of programs. That is, the optimization must not change the output produced by a program for a given input, or cause an error such as division by zero, that was not present in the original source program. At all times we take the “safe” approach of missing an opportunity to apply a transformation rather than risk changing what the program does.
- A transformation must, on the average, speed up programs by a measurable amount. We are also interested in reducing the size of the compiled code although the size of the code has less importance than it once had. Not every transformation succeeds in improving every program, occasionally an “optimization” may slow down a program slightly.
- The transformation must be worth the effort. It does not make sense for a compiler writer

to expend the intellectual effort to implement a code improving transformation and to have the compiler expend the additional time compiling source programs if this effort is not repaid when the target programs are executed. “Peephole” transformations of this kind are simple enough and beneficial enough to be included in any compiler.

- Flow analysis is a fundamental prerequisite for many important types of code improvement.
- Generally control flow analysis precedes data flow analysis.
- Control flow analysis (CFA) represents flow of control usually in form of graphs, CFA constructs such as
- A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global.
- Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

Function-Preserving Transformations

- There are a number of ways in which a compiler can improve a program without changing the function it computes.
 - The transformations
 - Common sub expression elimination, ○ Copy propagation,
 - Dead-code elimination, and
 - Constant folding, are common examples of such function-preserving transformations. The other transformations come up primarily when global optimizations are performed.
 - Frequently, a program will include several calculations of the same value, such as an offset in an array. Some of the duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language.
-

Common Sub expressions elimination:

- An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation. We can avoid recomputing the expression if we can use the previously computed value.
- For example
t1: =4*i t2: =a [t1] t3: =4*j t4:=4*i t5: =n
t 6: =b [t 4] +t 5

The above code can be optimized using the common sub-expression elimination as t1: =4*i
t2: =a [t1] t3: =4*j t5: =n
t6: =b [t1] +t5

The common sub expression t 4: =4*i is eliminated as its computation is already in t1. And value of i is not been changed from definition to use.

Copy Propagation:

Assignments of the form $f := g$ called copy statements, or copies for short. The idea behind the copy-propagation transformation is to use g for f, whenever possible after the copy statement $f := g$. Copy propagation means use of one variable instead of another. This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate x.

For example: $x = Pi$;

.....
 $A = x * r * r$;

The optimization using copy propagation can be done as follows:

$A = Pi * r * r$;

Here the variable x is eliminated

Dead-Code Eliminations:

A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. A related idea is dead or useless code, statements that compute values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations. An optimization can be done by eliminating dead code.

Example:

```
i=0;  
if(i=1)  
{  
  a=b+5;  
}
```

Here, 'if' statement is dead code because this condition will never get satisfied.

Constant folding:

- We can eliminate both the test and printing from the object code. More generally, deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.
- One advantage of copy propagation is that it often turns the copy statement into dead code.

For example,

a=3.14157/2 can be replaced by
a=1.570 there by eliminating a division operation.

Loop Optimizations:

- We now give a brief introduction to a very important place for optimizations, namely loops, especially the inner loops where programs tend to spend the bulk of their time. The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop.
 - Three techniques are important for loop optimization:
 - ☞ code motion, which moves code outside a loop;
 - ☞ Induction -variable elimination, which we apply to replace variables from inner loop.
 - ☞ Reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.
-

Code Motion:

- An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop. Note that the notion “before the loop” assumes the existence of an entry for the loop. For example, evaluation of limit-2 is a loop-invariant computation in the following while-statement:

```
while (i <= limit-2) /* statement does not change Limit*/ Code motion will result in
the equivalent of
t= limit-2;
while (i<=t) /* statement does not change limit or t */
```

Induction Variables :

- Loops are usually processed inside out. For example consider the loop around B3.
- Note that the values of j and t4 remain in lock-step; every time the value of j decreases by 1, that of t4 decreases by 4 because $4*j$ is assigned to t4. Such identifiers are called induction variables.
- When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination. For the inner loop around B3 in Fig. we cannot get rid of either j or t4 completely; t4 is used in B3 and j in B4.
- However, we can illustrate reduction in strength and illustrate a part of the process of induction-variable elimination. Eventually j will be eliminated when the outer loop of B2 - B5 is considered.

Example:

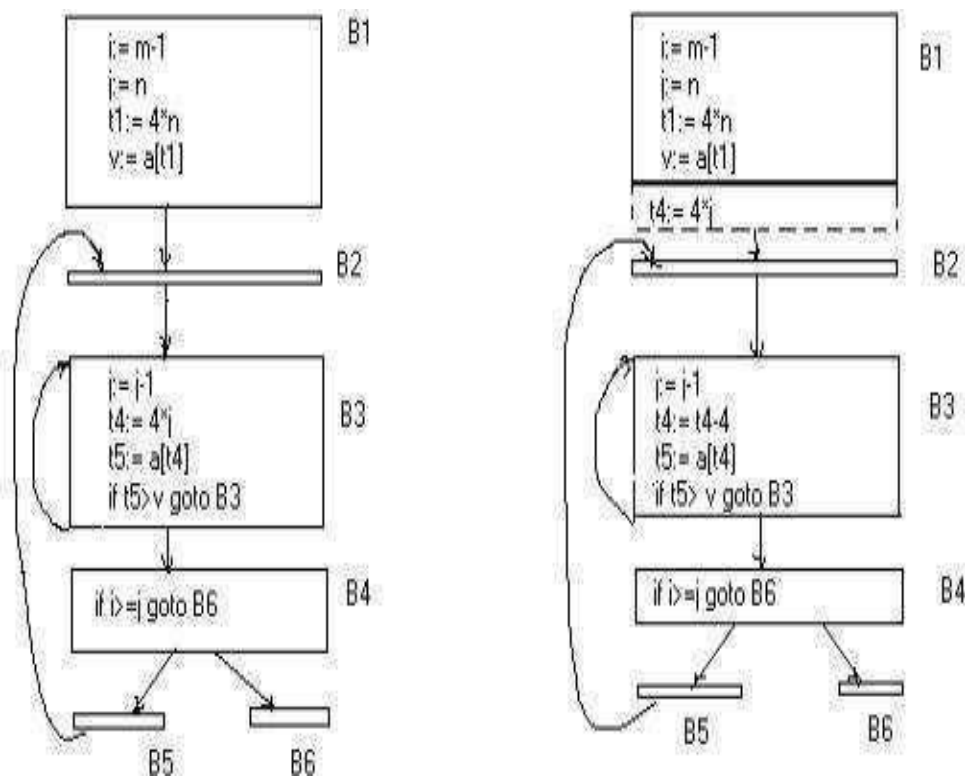
As the relationship $t4:=4*j$ surely holds after such an assignment to t4 in Fig. and t4 is not changed elsewhere in the inner loop around B3, it follows that just after the statement $j:=j-1$ the relationship $t4:=4*j-4$ must hold. We may therefore replace the assignment $t4:=4*j$ by $t4:=t4-4$. The only problem is that t4 does not have a value when we enter block B3 for the first time. Since we must maintain the relationship $t4=4*j$ on entry to the block B3, we place an initialization of t4 at the end of the block where j itself is initialized, shown by the dashed addition to block B1 in second Fig.

The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.

Reduction in Strength:

- Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.
- For example, x^2 is invariably cheaper to implement as $x*x$ than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

3. OPTIMIZATION OF BASIC BLOCKS



There are two types of basic block optimizations. They are :

Structure -Preserving Transformations
Algebraic Transformations

Structure- Preserving Transformations:

The primary Structure-Preserving Transformation on basic blocks are:

- Common sub-expression elimination
- Dead code elimination
- Renaming of temporary variables
- Interchange of two independent adjacent statements.

Common sub-expression elimination:

Common sub expressions need not be computed over and over again. Instead they can be computed once and kept in store from where it's referenced when encountered again – of course providing the variable values in the expression still remain constant.

Example:

```
a: =b+c  
b: =a-d  
c: =b+c  
d: =a-d
```

The 2nd and 4th statements compute the same expression: b+c and a-d

Basic block can be transformed to

```
a: =b+c  
b: =a-d  
c: =a  
d: =b
```

Dead code elimination:

It's possible that a large amount of dead (useless) code may exist in the program. This might be especially caused when introducing variables and procedures as part of construction or error - correction of a program – once declared and defined, one forgets to remove them in case they serve no purpose. Eliminating these will definitely optimize the code.

Renaming of temporary variables:

- A statement $t := b + c$ where t is a temporary name can be changed to $u := b + c$ where u is another temporary name, and change all uses of t to u .
 - In this we can transform a basic block to its equivalent block called normal-form block.
-

Interchange of two independent adjacent statements:

Two statements

$t1 := b + c$

$t2 := x + y$

can be interchanged or reordered in its computation in the basic block when value of $t1$ does not affect the value of $t2$.

Algebraic Transformations:

- Algebraic identities represent another important class of optimizations on basic blocks. This includes simplifying expressions or replacing expensive operation by cheaper ones i.e. reduction in strength.
- Another class of related optimizations is constant folding. Here we evaluate constant expressions at compile time and replace the constant expressions by their values. Thus the expression $2 * 3.14$ would be replaced by 6.28 .
- The relational operators \leq , \geq , $<$, $>$, $+$ and $=$ sometimes generate unexpected common sub expressions.
- Associative laws may also be applied to expose common sub expressions. For example, if the source code has the assignments

$a := b + c$ $e := c + d + b$

the following intermediate code may be generated:

$a := b + c$ $t := c + d$

$e := t + b$

Example:

$x := x + 0$ can be removed

$x := y ** 2$ can be replaced by a cheaper statement $x := y * y$

- The compiler writer should examine the language carefully to determine rearrangements of computations are permitted; since computer arithmetic does always obey the algebraic identities of mathematics. Thus, a compiler may evaluate $x*y-x*z$ as $x*(y-z)$ but it may not evaluate $a+(b-c)$ as $(a+b)-c$.

4. LOOPS IN FLOW GRAPH

A graph representation of three-address statements, called a **flow graph**, is useful for understanding code-generation algorithms, even if the graph is not explicitly constructed by a code-generation algorithm. Nodes in the flow graph represent computations, and the edges represent the flow of control.

Dominators:

In a flow graph, a node d dominates node n , if every path from initial node of the flow graph to n goes through d . This will be denoted by $d \text{ dom } n$. Every initial node dominates all the remaining nodes in the flow graph and the entry of a loop dominates all nodes in the loop. Similarly every node dominates itself.

Example:

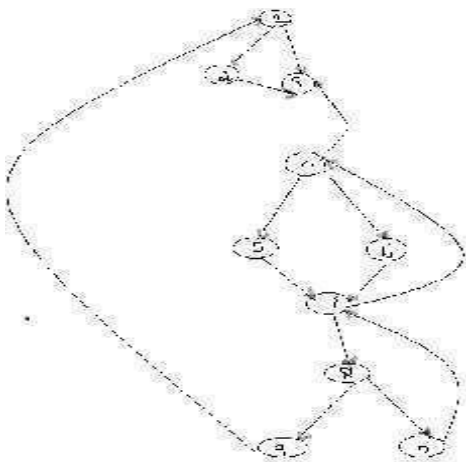
*In the flow graph below,

*Initial node, node 1 dominates every node. *node 2 dominates itself *node 3 dominates all but 1 and 2. *node 4 dominates all but 1, 2 and 3.

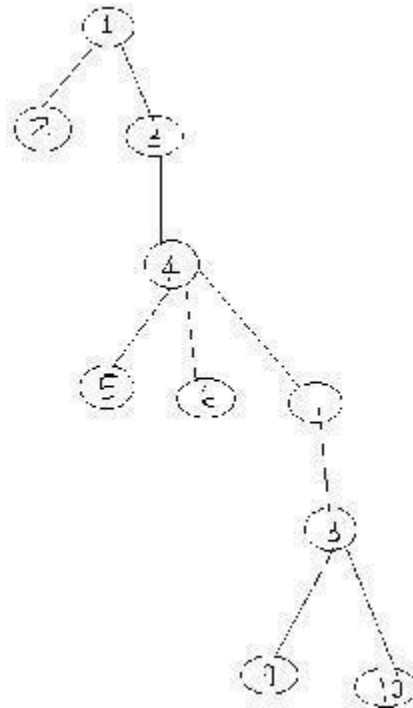
*node 5 and 6 dominates only themselves, since flow of control can skip around either by going through the other.

*node 7 dominates 7, 8, 9 and 10. *node 8 dominates 8, 9 and 10.

*node 9 and 10 dominates only themselves.



- The way of presenting dominator information is in a tree, called the dominator tree in which the initial node is the root.
- The parent of each other node is its immediate dominator.
- Each node d dominates only its descendants in the tree.
- The existence of dominator tree follows from a property of dominators; each node has a unique immediate dominator in that is the last dominator of n on any path from the initial node to n .
- In terms of the dom relation, the immediate dominator m has the property is $d \neq n$ and $d \text{ dom } n$, then $d \text{ dom } m$.



$D(1) = \{1\}$

$D(2) = \{1, 2\}$

$D(3) = \{1, 3\}$

$D(4) = \{1, 3, 4\}$

$D(5) = \{1, 3, 4, 5\}$

$D(6)=\{1,3,4,6\}$

$D(7)=\{1,3,4,7\}$

$D(8)=\{1,3,4,7,8\}$

$D(9)=\{1,3,4,7,8,9\}$

$D(10)=\{1,3,4,7,8,10\}$

Natural Loop:

- One application of dominator information is in determining the loops of a flow graph suitable for improvement.
- The properties of loops are
 - A loop must have a single entry point, called the header. This entry point dominates all nodes in the loop, or it would not be the sole entry to the loop.
 - There must be at least one way to iterate the loop (i.e.) at least one path back to the header.
- One way to find all the loops in a flow graph is to search for edges in the flow graph whose heads dominate their tails. If $a \rightarrow b$ is an edge, b is the head and a is the tail. These types of edges are called as back edges.

Example:

In the above graph,

$\rightarrow 4$ 4 DOM 7

$\rightarrow 7$ 7 DOM 10

$\rightarrow 3$

$\rightarrow 3$

$9 \rightarrow 1$

- The above edges will form loop in flow graph.
- Given a back edge $n \rightarrow d$, we define the natural loop of the edge to be d plus the set of nodes that can reach n without going through d . Node d is the header of the loop.

Algorithm: Constructing the natural loop of a back edge.

Input: A flow graph G and a back edge $n \rightarrow d$

Output: The set loop consisting of all nodes in the natural loop $n \rightarrow d$.

Method: Beginning with node n , we consider each node $m \neq d$ that we know is in loop, to make sure that m 's predecessors are also placed in loop. Each node in loop, except for d , is placed once on stack, so its predecessors will be examined. Note that because d is put in the loop initially, we never examine its predecessors, and thus find only those nodes that reach n without going through d .

Procedure insert(m);

if m is not in *loop* **then begin** $loop := loop \cup \{m\}$; push m onto *stack*

end;

$stack := \text{empty}$; $loop := \{d\}$; insert(n);

while *stack* is not empty **do begin**

 pop m , the first element of *stack*, off *stack*; **for** each predecessor p of m **do** insert(p)

end Inner

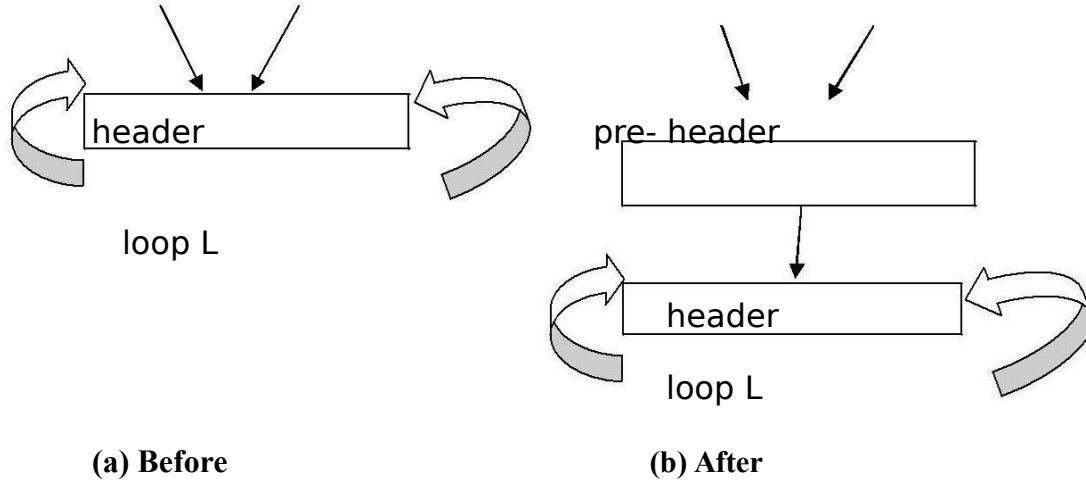
5.LOOP:

- If we use the natural loops as “the loops”, then we have the useful property that unless two loops have the same header, they are either disjointed or one is entirely contained in the other. Thus, neglecting loops with the same header for the moment, we have a natural notion of inner loop: one that contains no other loop.
- When two natural loops have the same header, but neither is nested within the other, they are combined and treated as a single loop.

Pre-Headers:

- Several transformations require us to move statements “before the header”. Therefore begin treatment of a loop L by creating a new block, called the preheader.
 - The pre -header has only the header as successor, and all edges which formerly entered the header of L from outside L instead enter the pre-header.
 - Edges from inside loop L to the header are not changed.
-

- Initially the pre-header is empty, but transformations on L may place statements in it.



Reducible flow graphs:

- Reducible flow graphs are special flow graphs, for which several code optimization transformations are especially easy to perform, loops are unambiguously defined, dominators can be easily calculated, data flow analysis problems can also be solved efficiently.
 - Exclusive use of structured flow-of-control statements such as if-then-else, while-do, continue, and break statements produces programs whose flow graphs are always reducible. The most important properties of reducible flow graphs are that there are no jumps into the middle of loops from outside; the only entry to a loop is through its header.
 - Definition:**
 - A flow graph G is reducible if and only if we can partition the edges into two disjoint groups, *forward* edges and *back* edges, with the following properties.
 - The forward edges from an acyclic graph in which every node can be reached from initial node of G.
 - The back edges consist only of edges where heads dominate their tails.
 - Example: The above flow graph is reducible.
 - If we know the relation DOM for a flow graph, we can find and remove all the back edges.
-

- The remaining edges are forward edges.
- If the forward edges form an acyclic graph, then we can say the flow graph reducible.
- In the above example remove the five back edges $4 \rightarrow 3$, $7 \rightarrow 4$, $8 \rightarrow 3$, $9 \rightarrow 1$ and $10 \rightarrow 7$ whose heads dominate their tails, the remaining graph is acyclic.
- The key property of reducible flow graphs for loop analysis is that in such flow graphs every set of nodes that we would informally regard as a loop must contain a back edge.



UNIT – V

CODE GENERATION

PEEPHOLE OPTIMIZATION

- A statement-by-statement code-generations strategy often produce target code that contains redundant instructions and suboptimal constructs .The quality of such target code can be improved by applying “optimizing” transformations to the target program.
- A simple but effective technique for improving the target code is peephole optimization, a method for trying to improving the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence, whenever possible.
- The peephole is a small, moving window on the target program. The code in the peephole need not contiguous, although some implementations do require this.it is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.
- We shall give the following examples of program transformations that are characteristic of peephole optimizations:



Redundant-instructions elimination

- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms
- Unreachable Code

Redundant Loads And Stores:

If we see the instructions sequence

- (1) MOV R0,a
- (2) MOV a,R0

we can delete instructions (2) because whenever (2) is executed. (1) will ensure that the value of **a** is already in register R0.If (2) had a label we could not be sure that (1) was always executed immediately before (2) and so we could not remove (2).

Unreachable Code:

- Another opportunity for peephole optimizations is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable **debug** is 1. In C, the source code might look like:

```
#define debug
0 ....
If ( debug ) {

    Print debugging information

}
```

In the intermediate representations the if-statement may be translated as:

```
debug =1 goto L2

goto L2

L1: print debugging information

L2:.....(a)
```

- One obvious peephole optimization is to eliminate jumps over jumps .Thus no matter what the value of **debug**; (a) can be replaced by:

```
If debug ≠1 goto L2

Print debugging information

L2:.....(b)
```

- As the argument of the statement of (b) evaluates to a constant **true** it can be replaced by
If debug ≠0 goto L2

```
Print debugging information

L2:.....(c)
```

- As the argument of the first statement of (c) evaluates to a constant true, it can be replaced by goto L2. Then all the statement that print debugging aids are manifestly

unreachable and can be eliminated one at a time.

Flows-Of-Control Optimizations:

- The unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the jump sequence

```
goto L1
```

```
....
```

```
L1: goto L2 by the sequence
```

```
goto L2
```

```
....
```

```
L1: goto L2
```

- If there are now no jumps to L1, then it may be possible to eliminate the statement L1:goto L2 provided it is preceded by an unconditional jump. Similarly, the sequence

```
if a < b goto L1
```

```
....
```

```
L1: goto L2
```

```
can be replaced by if a < b goto L2
```

```
....
```

```
L1: goto L2
```

- Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto. Then the sequence

```
goto L1
```

```
.....
```

```
L1: if a < b goto L2
```

```
L3:.....(1)
```

- Maybe replaced by if a < b goto L2

```
goto L3
```

```
.....
```
