

***COMPILER DESIGN***  
***LECTURE NOTES (Semester – II)***  
*for*  
***Master of Science in Computer Science***



***Department of Computer Science and Applications***

**School of Arts & Science**

**Vinayaka Mission's Research Foundation**

**A V Campus, Chennai – 603104.**

**Lecture Note Prepared by:**

**A.VIJAYA KUMAR, Asst. Professor**

## SYLLABUS

### DSC V: COMPILER DESIGN

**Objectives:** At the end of the course, the student should be able to do:

- Parsing techniques and different levels of translation.
- Apply the various optimization techniques.
- Use the different compiler construction tools.

#### UNIT I

Introduction on the phase of the compiler – Lexical Analysis, Regular Expression, Non deterministic Automata, Deterministic Automata equivalent to NFA's – Minimizing the states of DFA, Implementation of Lexical Analyzer.

#### UNIT II

Syntax Analysis – Top down Parsing Concepts, Recursive Descent Parsing, Predictive Parsers, Non recursive Predictive Parsing – Bottom Up Parsing, Handle pruning, Shift reduce parsing – Operator Precedence Parsing – Error recovery in Parsing, LR Parsers, Parser Generators – YACC.

#### UNIT III

Intermediate Code Generation: Syntax directed Definitions, Construction of Syntax trees – Top down Translation, Bottom up Evaluation of inherited Attributed, Recursive Evaluators, Assigning Space at Compiler Construction time – Type checking – Overloading of functions and operators Polymorphic function.

#### UNIT IV

Storage Organization : Storage Organization, Storage Allocation Strategies, Parameter Passing, Symbol tables, Dynamic Storage Allocation, Intermediate Languages – Representation of Declarations, Assignment Statement, Boolean Expression, Back patching, Procedure calls.

#### UNIT V

Code Generation and Optimization: Design of the code generators, Runtime storage Management, Basic blocks and flow graphs, Register Allocation and Assignment, DAG representation of Basic blocks, Peephole optimization, Code optimization – The principle sources of optimization, Optimization of basic blocks, Global data flow Analysis, Loop optimizations.

#### TEXT BOOKS:

- ❖ Alfred Aho, Ravi Sethi, Jeffrey D. Ullman, "Compilers – Principles, Techniques and Tools", 1986, Addison Wesley.
- ❖ Dick Grune, Henri E. Bal, Criel J. H. Jacobs, Koen G. Langondeon, "Modern Compiler Design", Wiley, Singapore, 2003

#### REFERENCES:

- ❖ Dhamdhare D.M., "Compiler Construction Principles and Practice", 1981, Macmillan India.
- ❖ Reinhard Wilhelm, Director Mauser, "Compiler Design", 1995, Addison Wesley.

## 1.1 INTRODUCTION OF LANGUAGE PROCESSING SYSTEM

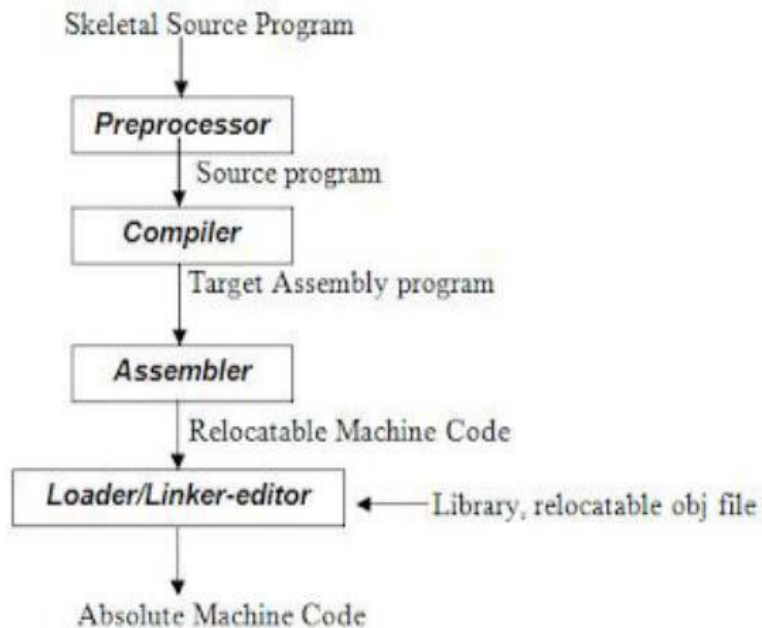


Fig 1.1: Language Processing System

### 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 translate it into an equivalent program in (MLL) the target program. As an important part of a compiler is error showing to the programmer.

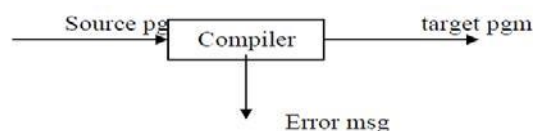


Fig 1.2: Structure of Compiler

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

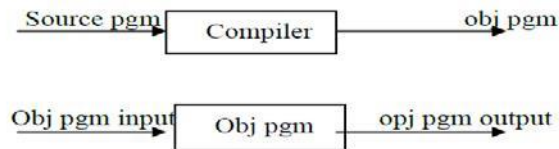


Fig 1.3: Execution process of source program in Compiler

### ASSEMBLER

Programmers found it difficult to write or read programs in machine language. They began 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 assemblers 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.

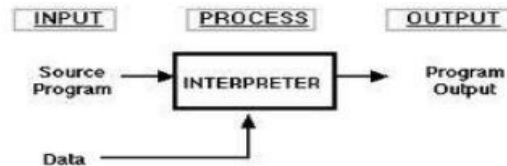


Fig1.4: Execution in Interpreter

Languages such as BASIC, SNOBOL, LISP can be translated using interpreters. JAVA also uses interpreter. The process of interpretation can be carried out in following phases.

1. Lexical analysis
2. Syntax analysis
3. Semantic analysis
4. Direct Execution

#### **Advantages:**

Modification of user program can be easily made and implemented as execution proceeds.  
 Type of object that denotes a 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.

## 1.2 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.

## 1.3 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. Common Lisp compilers
9. ECMAScript interpreters
10. Fortran compilers
11. Java compilers
12. Pascal compilers
13. PL/I compilers
14. Python compilers
15. Smalltalk compilers

## 1.4 STRUCTURE OF THE COMPILER DESIGN

**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.

- a. Analysis (Machine Independent/Language Dependent)
- b. Synthesis (Machine Dependent/Language independent)

Compilation process is partitioned into no-of-sub processes called '**phases**'.

### **Lexical Analysis:-**

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

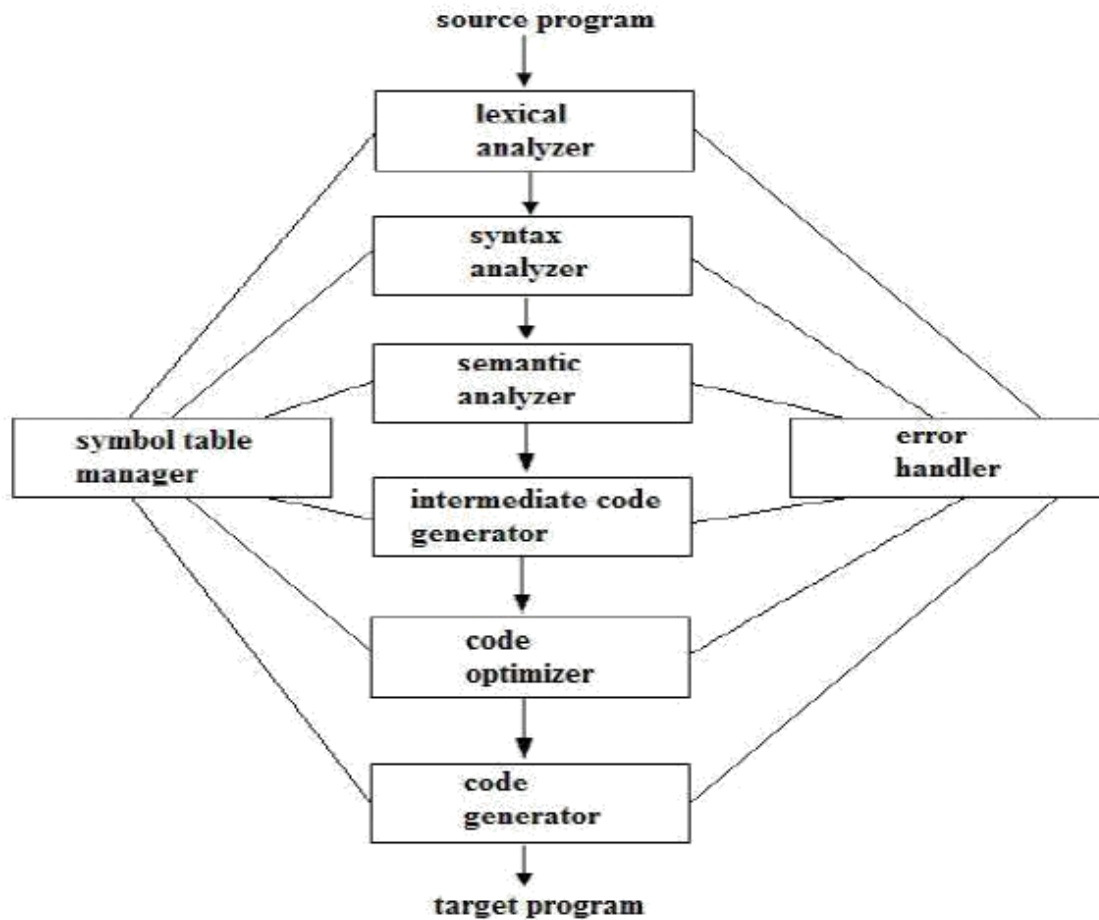


Fig 1.5: Phases of Compiler

**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.

**Intermediate Code Generations:-**

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

**Code Optimization :-**

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

**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.

**Table Management (or) Book-keeping:-** This is the portion to **keep the names** used by the about program and records essential information each. The data structure used to record this information called a ‘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 some job as the original, but in a way that saves time and / or spaces.

#### **a. 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  $B + C$ .

**b. 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 :-**

Code Generator produces the object code by deciding on the memory locations for data, selecting code to access each datum 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.

**Table Management OR Book-keeping :-**

A compiler needs to collect information about all the data objects that appear in the source program. The information about data objects is collected by the early phases of the compiler- lexical and syntactic analyzers. The data structure used to record this information is called as Symbol Table.

**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:

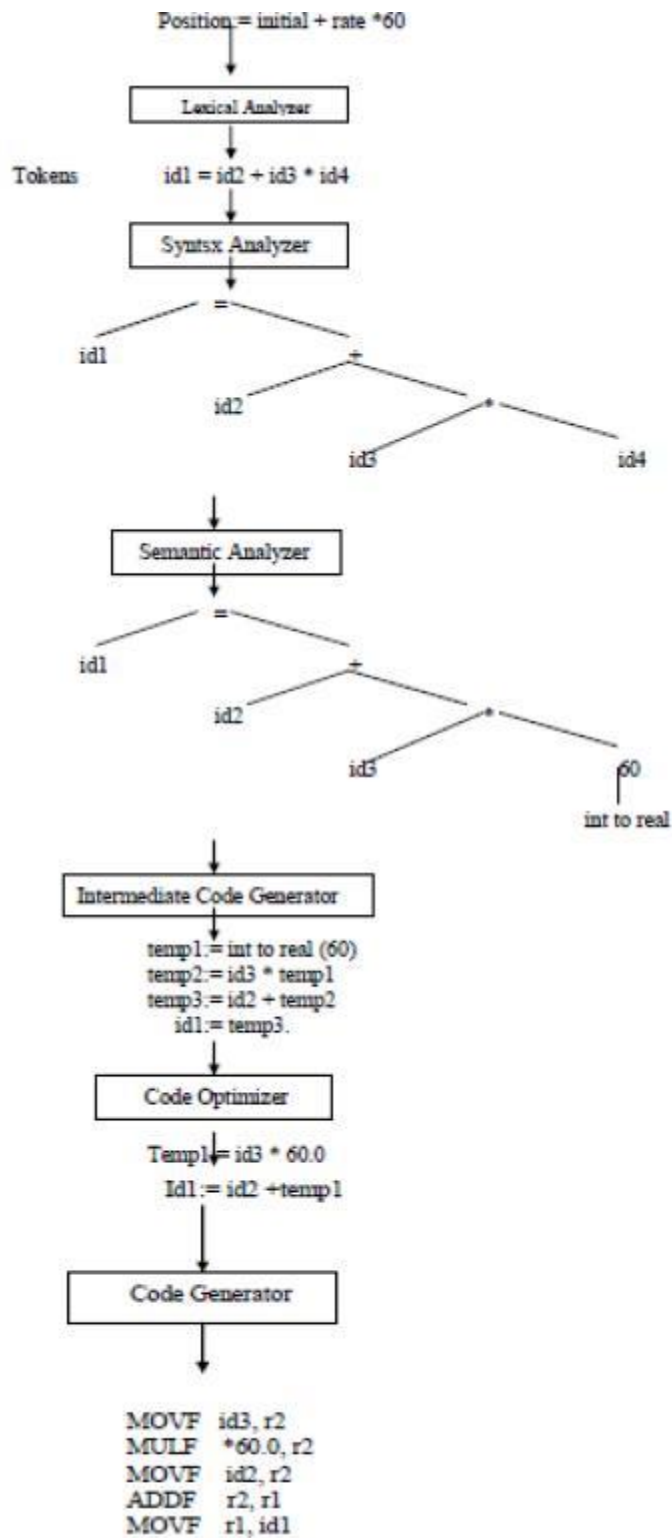


Fig 1.6: Compilation Process of a source code through phases

## 1.5 OVER VIEW OF LEXICAL ANALYSIS

- To identify the tokens we need some method of describing the possible tokens that can appear in the input stream. For this purpose we introduce regular expression, a notation that can be used to describe essentially all the tokens of programming language.
- Secondly , having decided what the tokens are, we need some mechanism to recognize these in the input stream. This is done by the token recognizers, which are designed using transition diagrams and finite automata.

## 1.6 ROLE OF LEXICAL ANALYZER

The LA is the first phase of a compiler. Its main task is to read the input character and produce as output a sequence of tokens that the parser uses for syntax analysis.

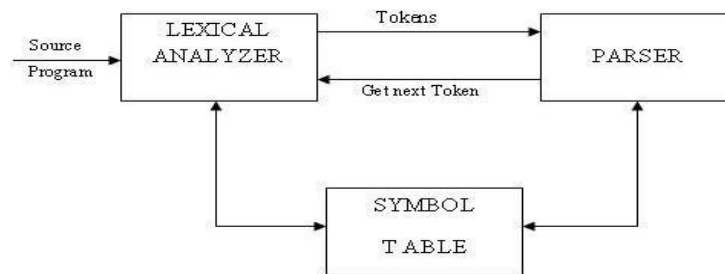


Fig. 1.7: Role of Lexical analyzer

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 returns 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.

## 1.7 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.

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
num	3.14	any character b/w "and "except"
literal	"core"	pattern

Fig. 1.8: Example of Token, Lexeme and Pattern

**1.8 LEXICAL ERRORS:**

Lexical errors are the errors thrown by your 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** recognised 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:

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

**1.9 REGULAR EXPRESSIONS**

Regular expression is a formula that describes a possible set of string. Component of regular expression..

- X** the character **x**
- .** any character, usually accept a new line
- [x y z]** any of the characters **x, y, z, .....**
- R?** a **R** or nothing (=optionally as **R**)
- R\*** zero or more occurrences.....
- R+** one or more occurrences .....
- R1R2** an **R1** followed by an **R2**
- R1|R2** either an **R1** or an **R2**.

A token is either a single string or one of a collection of strings of a certain type. If we view the set of strings in each token class as an language, we can use the regular-expression notation to describe tokens.

Consider an identifier, which is defined to be a letter followed by zero or more letters or digits.

In regular expression notation we would write.

Identifier = letter (letter | digit)\*

Here are the rules that define the regular expression over alphabet .

- $\epsilon$  is a regular expression denoting  $\{ \epsilon \}$ , that is, the language containing only the empty string.
- For each 'a' in  $\Sigma$ , 'a' is a regular expression denoting  $\{ a \}$ , the language with only one string consisting of the single symbol 'a' .
- If R and S are regular expressions, then

$(R) | (S)$  means  $L(r) \cup L(s)$

$R.S$  means  $L(r).L(s)$

$R^*$  denotes  $L(r^*)$

### 1.10. 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)\*

#### Recognition of tokens:

We learn how to express pattern using regular expressions. Now, we must study how to take the patterns for all the needed tokens and build a piece of code that examines the input string and finds a prefix that is a lexeme matching one of the patterns.

```

Stmt →if expr then stmt
      | If expr then else stmt
      | ε
Expr →term relop term
      | term
Term →id
      |number

```

For relop ,we use the comparison operations of languages like Pascal or SQL where = is “equals” and < > is “not equals” because it presents an interesting structure of lexemes.

The terminal of grammar, which are if, then , else, relop ,id and numbers are the names of tokens as far as the lexical analyzer is concerned, the patterns for the tokens are described using regular definitions.

```

digit → [0,9]
digits →digit+
number →digit(.digit)?(e.[+-]?digits)?
letter → [A-Z,a-z]
id →letter(letter/digit)*
if → if
then →then

```

else →else  
 relop →< | > | <= | >= | == | <>

In addition, we assign the lexical analyzer the job stripping out white space, by recognizing the “token” we defined by:

WS → (blank/tab/newline)+

Here, blank, tab and newline are abstract symbols that we use to express the ASCII characters of the same names. Token ws is different from the other tokens in that ,when we recognize it, we do not return it to parser ,but rather restart the lexical analysis from the character that follows the white space . It is the following token that gets returned to the parser.

Lexeme	Token Name	Attribute Value
Any WS	-	-
if	if	-
then	then	-
else	else	-
Any id	Id	Pointer to table entry
Number	number	Pointer to table ent
<	relop	LT
<=	relop	LE
==	relop	EQ
<>	relop	NE

### 1.11. 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 indicates 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.

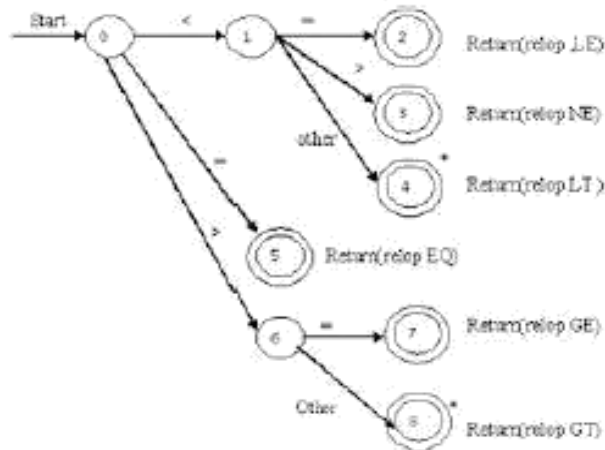


Fig. 1.9: Transition diagram of Relational operators

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.

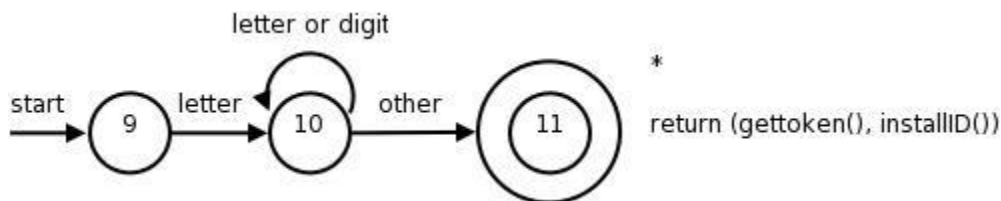


Fig. 1.10: Transition diagram of Identifier

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.

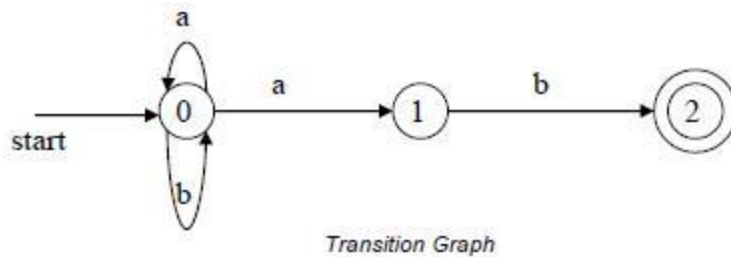
### 1.12. FINITE AUTOMATON

- A *recognizer* for a language is a program that takes a string x, and answers “yes” if x is a sentence of that language, and “no” otherwise.
- We call the recognizer of the tokens as a *finite automaton*.
- A finite automaton can be: *deterministic (DFA)* or *non-deterministic (NFA)*
- This means that we may use a deterministic or non-deterministic automaton as a lexical analyzer.
- Both deterministic and non-deterministic finite automaton recognize regular sets.
- Which one?
  - deterministic – faster recognizer, but it may take more space
  - non-deterministic – slower, but it may take less space
  - Deterministic automata are widely used lexical analyzers.
- First, we define regular expressions for tokens; Then we convert them into a DFA to get a lexical analyzer for our tokens.

**1.13. Non-Deterministic Finite Automaton (NFA)**

- A non-deterministic finite automaton (NFA) is a mathematical model that consists of:
  - o  $S$  - a set of states
  - o  $\Sigma$  - a set of input symbols (alphabet)
  - o move - a transition function move to map state-symbol pairs to sets of states.
  - o  $s_0$  - a start (initial) state
  - o  $F$  - a set of accepting states (final states)
- $\epsilon$ - transitions are allowed in NFAs. In other words, we can move from one state to another one without consuming any symbol.
- A NFA accepts a string  $x$ , if and only if there is a path from the starting state to one of accepting states such that edge labels along this path spell out  $x$ .

Example:



0 is the start state  $s_0$   
 {2} is the set of final states  $F$   
 $\Sigma = \{a,b\}$   
 $S = \{0,1,2\}$

Transition Function:

	a	b
0	{0,1}	{0}
1	$\emptyset$	{2}
2	$\emptyset$	$\emptyset$

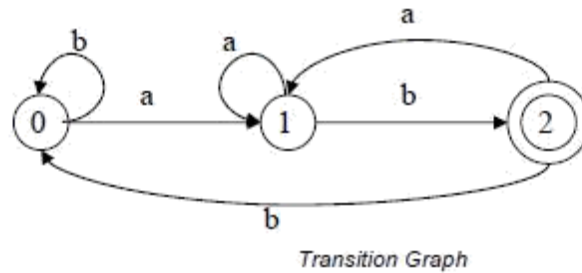
The language recognized by this NFA is  $(a|b)^*ab$

**1.14. Deterministic Finite Automaton (DFA)**

- A Deterministic Finite Automaton (DFA) is a special form of a NFA.
- No state has  $\epsilon$ - transition
- For each symbol  $a$  and state  $s$ , there is at most one labeled edge  $a$  leaving  $s$ . i.e. transition function is from pair of state-symbol to state (not set of states)

Example:

The DFA to recognize the language  $(a|b)^* ab$  is as follows.



0 is the start state  $s_0$   
 {2} is the set of final states  $F$   
 $\Sigma = \{a,b\}$   
 $S = \{0,1,2\}$

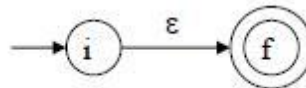
Transition Function:

	a	b
0	1	0
1	1	2
2	1	0

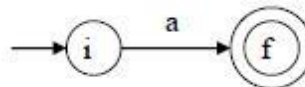
Note that the entries in this function are single value and not set of values (unlike NFA).

### 1.15. Converting RE to NFA

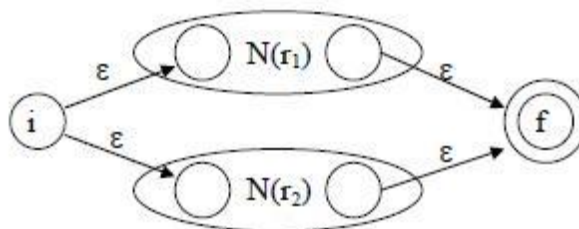
- This is one way to convert a regular expression into a NFA.
- There can be other ways (much efficient) for the conversion.
- Thomson's Construction is simple and systematic method.
- It guarantees that the resulting NFA will have exactly one final state, and one start state.
- Construction starts from simplest parts (alphabet symbols).
- To create a NFA for a complex regular expression, NFAs of its sub-expressions are combined to create its NFA.
- To recognize an empty string  $\epsilon$ :



- To recognize a symbol a in the alphabet  $\Sigma$ :



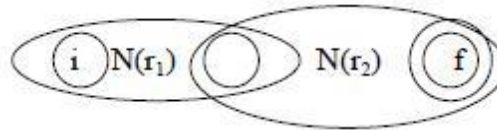
- For regular expression  $r_1 | r_2$ :



$N(r_1)$  and  $N(r_2)$  are NFAs for regular expressions  $r_1$  and  $r_2$ .

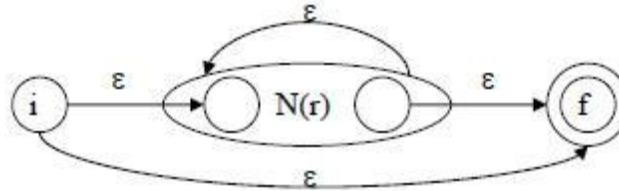


- For regular expression  $r_1 r_2$



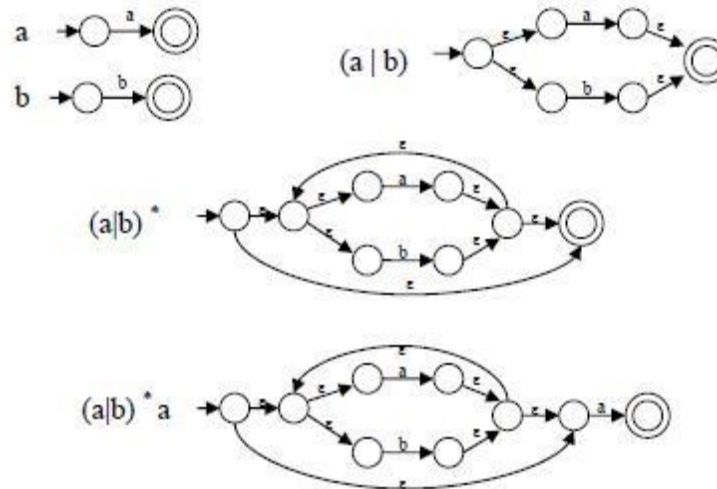
Here, final state of  $N(r_1)$  becomes the final state of  $N(r_1 r_2)$ .

- For regular expression  $r^*$



Example:

For a RE  $(a|b)^* a$ , the NFA construction is shown below.



### 1.16. Converting NFA to DFA (Subset Construction)

We merge together NFA states by looking at them from the point of view of the input characters:

- From the point of view of the input, any two states that are connected by an  $\epsilon$ -transition may as well be the same, since we can move from one to the other without consuming any character. Thus states which are connected by an  $\epsilon$ -transition will be represented by the same states in the DFA.
- If it is possible to have multiple transitions based on the same symbol, then we can regard a transition on a symbol as moving from a state to a set of states (ie. the union of all those states reachable by a transition on the current symbol). Thus these states will be combined into a single DFA state.

To perform this operation, let us define two functions:

- The **-closure** function takes a state and returns the set of states reachable from it based on (one or more)  $\epsilon$ -transitions. Note that this will always include the state itself. We should be able to get from a state to any state in its  $\epsilon$ -closure without consuming any input.
- The function **move** takes a state and a character, and returns the set of states reachable by one transition on this character.

We can generalise both these functions to apply to sets of states by taking the union of the application to individual states.

For Example, if A, B and C are states,  $move(\{A,B,C\},`a') = move(A,`a') \cup move(B,`a') \cup move(C,`a')$ .

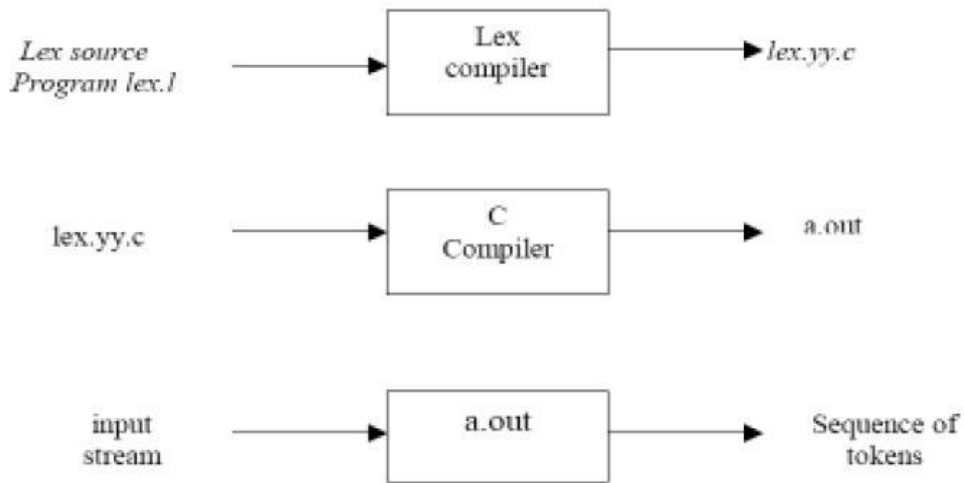
The Subset Construction Algorithm is as follows:

```

put  $\epsilon$ -closure( $\{s_0\}$ ) as an unmarked state into the set of DFA (DS)
while (there is one unmarked S1 in DS) do
  begin
    mark S1
    for each input symbol a do
      begin
         $S_2 \leftarrow \epsilon$ -closure(move(S1,a))
        if (S2 is not in DS) then
          add S2 into DS as an unmarked state
          transfunc[S1,a]  $\leftarrow$  S2
      end
    end
  end
end
    
```

- a state S in DS is an accepting state of DFA if a state in S is an accepting state of NFA
- the start state of DFA is  $\epsilon$ -closure( $\{s_0\}$ )

### 1.17. Lexical Analyzer Generator



overhead is limited.

## 2. SYNTAX ANALYSIS

### 2.1 ROLE OF THE PARSER :

Parser for any grammar is program that takes as input string  $w$  (obtain set of strings tokens from the lexical analyzer) and produces as output either a parse tree for  $w$ , if  $w$  is a valid sentences of grammar or error message indicating that  $w$  is not a valid sentences of given grammar. The goal of the parser is to determine the syntactic validity of a source string is valid, a tree is built for use by the subsequent phases of the computer. The tree reflects the sequence of derivations or reduction used during the parser. Hence, it is called parse tree. If string is invalid, the parse has to issue diagnostic message identifying the nature and cause of the errors in string. Every elementary subtree in the parse tree corresponds to a production of the grammar.

There are two ways of identifying an elementary subtree:

1. By deriving a string from a non-terminal or
2. By reducing a string of symbol to a non-terminal.

The two types of parsers employed are:

- a. Top down parser: which build parse trees from top(root) to bottom(leaves)
- b. Bottom up parser: which build parse trees from leaves and work up the root.

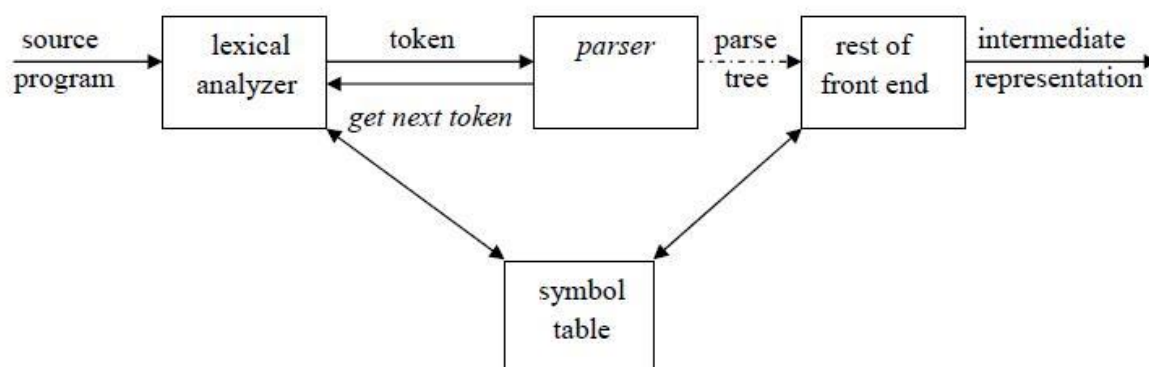


Fig . 2.1: position of parser in compiler model.

### 2.2 CONTEXT FREE GRAMMARS

Inherently recursive structures of a programming language are defined by a context-free Grammar. In a context-free grammar, we have four triples  $G( V,T,P,S)$ .

Here ,  $V$  is finite set of terminals (in our case, this will be the set of tokens)

$T$  is a finite set of non-terminals (syntactic-variables)

P is a finite set of productions rules in the following form

$A \rightarrow \alpha$  where A is a non-terminal and  $\alpha$  is a string of terminals and non-terminals (including the empty string)

S is a start symbol (one of the non-terminal symbol)

$L(G)$  is the language of G (the language generated by G) which is a set of sentences.

A sentence of  $L(G)$  is a string of terminal symbols of G. If S is the start symbol of G then

$\omega$  is a sentence of  $L(G)$  iff  $S \Rightarrow \omega$  where  $\omega$  is a string of terminals of G. If G is a context-free grammar,  $L(G)$  is a context-free language. Two grammar  $G_1$  and  $G_2$  are equivalent, if they produce same grammar.

Consider the production of the form  $S \Rightarrow \alpha$ , If  $\alpha$  contains non-terminals, it is called as a sentential form of G. If  $\alpha$  does not contain non-terminals, it is called as a sentence of G.

In general a derivation step is

$\alpha A \beta \Rightarrow \alpha \gamma \beta$  is sentential form and if there is a production rule  $A \rightarrow \gamma$  in our grammar. where  $\alpha$  and  $\beta$  are arbitrary strings of terminal and non-terminal symbols  $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n$  ( $\alpha_n$  derives from  $\alpha_1$  or  $\alpha_1$  derives  $\alpha_n$ ). There are two types of derivation

- 1 At each derivation step, we can choose any of the non-terminal in the sentential form of G for the replacement.
- 2 If we always choose the left-most non-terminal in each derivation step, this derivation is called as left-most derivation.

Example:

$E \rightarrow E+E \mid E-E \mid E * E \mid E/E \mid -E \quad E \rightarrow (E)$

$E \rightarrow id$

Leftmost derivation :

$E \rightarrow E+E$

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

The string is derive from the grammar  $w = id * id+id$ , which is consists of all terminal symbols

Rightmost derivation

$E \rightarrow E+E$

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

Given grammar  $G : E \rightarrow E+E \mid E * E \mid ( E ) \mid - E \mid$

id Sentence to be derived :  $-(id+id)$

LEFTMOST DERIVATION

RIGHTMOST DERIVATION

$E \rightarrow -E$

$E \rightarrow -E$

$E \rightarrow -(E)$

$E \rightarrow -(E)$

$E \rightarrow -(E+E)$

$E \rightarrow -(E+E)$

$E \rightarrow -(id+E)$

$E \rightarrow -(E+id)$

$E \rightarrow -(id+id)$

$E \rightarrow -(id+id)$

- String that appear in leftmost derivation are called left sentinel forms.
- String that appear in rightmost derivation are called right sentinel forms.

Sentinels:

- Given a grammar G with start symbol S, if  $S \rightarrow \alpha$ , where  $\alpha$  may contain non-terminals or terminals, then  $\alpha$  is called the sentinel form of G.

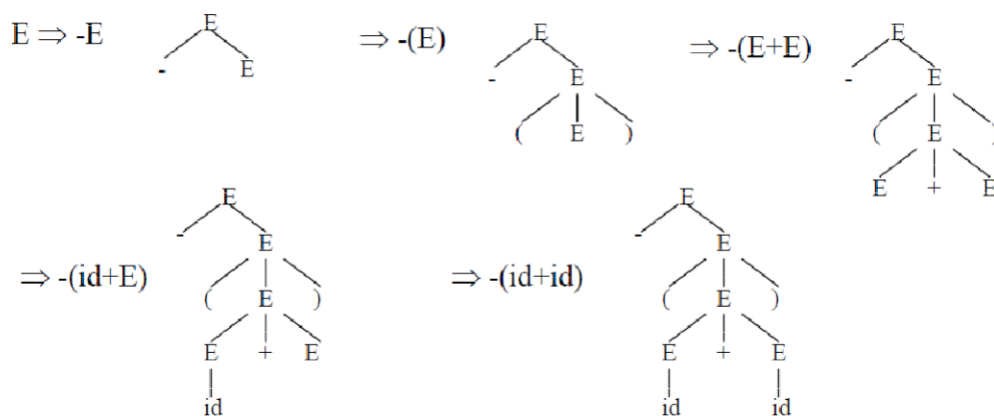
Yield or frontier of tree:

- Each interior node of a parse tree is a non-terminal. The children of node can be a terminal or non-terminal of the sentinel forms that are read from left to right. The sentinel form in the parse tree is called yield or frontier of the tree.

2.3. PARSE TREE

- Inner nodes of a parse tree are non-terminal symbols.
- The leaves of a parse tree are terminal symbols.
- A parse tree can be seen as a graphical representation of a derivation.

Example:



Ambiguity:

A grammar that produces more than one parse for some sentence is said to be ambiguous grammar.

Example : Given grammar  $G : E \rightarrow E+E \mid E * E \mid ( E ) \mid - E \mid id$

The sentence  $id+id*id$  has the following two distinct leftmost derivations:

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

The two corresponding parse trees are :



Example:

To disambiguate the grammar  $E \rightarrow E+E \mid E * E \mid E^E \mid id \mid (E)$ , we can use precedence of operators as follows:

- $\wedge$  (right to left)
- $/, * (left to right) -$
- $, + (left to right)$

We get the following unambiguous grammar:

$E \rightarrow E+T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow G^F \mid G$

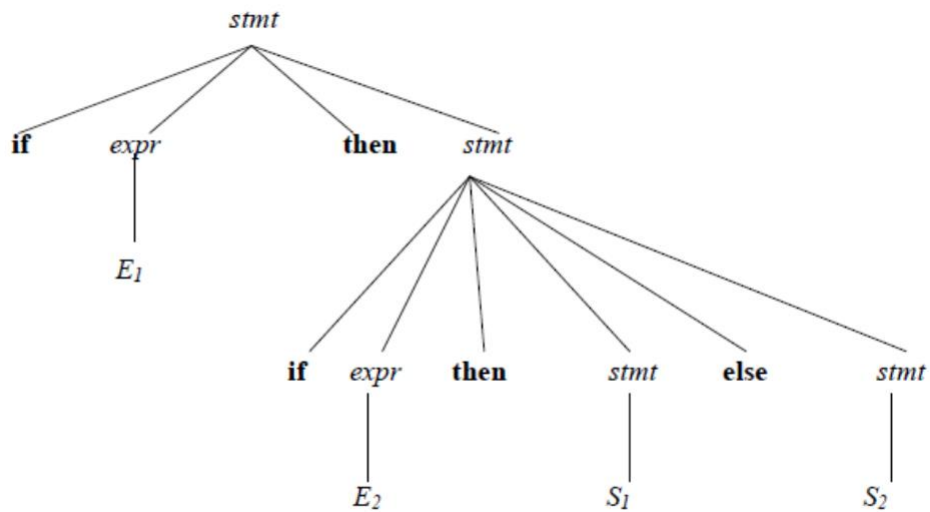
$G \rightarrow id \mid (E)$

Consider this example,  $G: stmt \rightarrow if\ expr\ then\ stmt \mid if\ expr\ then\ stmt\ else\ stmt \mid other$

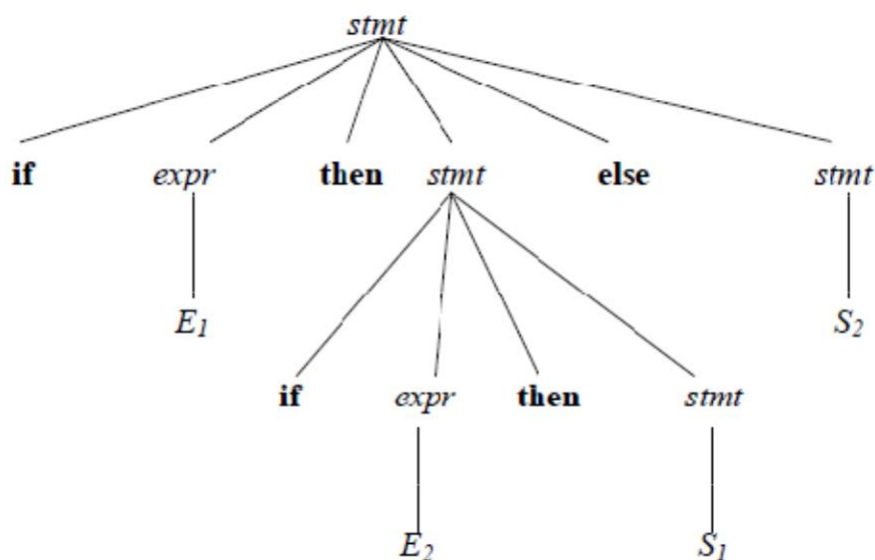
This grammar is ambiguous since the string  $if\ E1\ then\ if\ E2\ then\ S1\ else\ S2$  has the following

Two parse trees for leftmost derivation :

1.



2.



To eliminate ambiguity, the following grammar may be used:

$stmt \rightarrow matched\_stmt \mid unmatched\_stmt$

$matched\_stmt \rightarrow if\ expr\ then\ matched\_stmt\ else\ matched\_stmt \mid other$

$unmatched\_stmt \rightarrow if\ expr\ then\ stmt \mid if\ expr\ then\ matched\_stmt\ else\ unmatched\_stmt$

Eliminating Left Recursion:

A grammar is said to be left recursive if it has a non-terminal A such that there is a derivation

$A \Rightarrow A\alpha$  for some string  $\alpha$ . Top-down parsing methods cannot handle left-recursive grammars.

Hence, left recursion can be eliminated as follows:

If there is a production  $A \rightarrow A\alpha \mid \beta$  it can be replaced with a sequence of two productions

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \epsilon \end{aligned}$$

Without changing the set of strings derivable from A.

Example : Consider the following grammar for arithmetic expressions:

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

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

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

First eliminate the left recursion for E as

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \epsilon$$

Then eliminate for T as

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \epsilon$$

Thus the obtained grammar after eliminating left recursion

$$\text{is } E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \epsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \epsilon$$

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

Algorithm to eliminate left recursion:

1. Arrange the non-terminals in some order  $A_1, A_2 \dots A_n$ .
2. for  $i := 1$  to  $n$  do begin
  - for  $j := 1$  to  $i-1$  do begin
    - replace each production of the form  $A_i \rightarrow A_j \gamma$
    - by the productions  $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$
    - where  $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$  are all the current  $A_j$ -productions;
  - end
  - eliminate the immediate left recursion among the  $A_i$ -productions
- end



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 can rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice.

If there is any production  $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ , it can be rewritten as

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 \mid \beta_2$$

Consider the grammar,  $G : S \rightarrow iEtS \mid iEtSeS \mid a$

$$E \rightarrow b$$

Left factored, this grammar becomes

$$S \rightarrow iEtSS' \mid a$$

$$S' \rightarrow eS \mid \epsilon$$

$$E \rightarrow b$$

## 2.4 TOP-DOWN PARSING

It can be viewed as an attempt to find a left-most derivation for an input string or an attempt to construct a parse tree for the input starting from the root to the leaves.

Types of top-down parsing :

1. Recursive descent parsing
2. Predictive parsing

### 2.4.1. RECURSIVE DESCENT PARSING

- ◆ Recursive descent parsing is one of the top-down parsing techniques that uses a set of recursive procedures to scan its input.
- ◆ This parsing method may involve backtracking, that is, making repeated scans of the input.

Example for backtracking :

Consider the grammar  $G : S \rightarrow cAd$

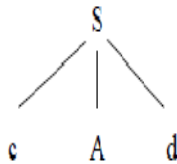
$$A \rightarrow ab \mid a$$

and the input string  $w=cad$ .

The parse tree can be constructed using the following top-down approach :

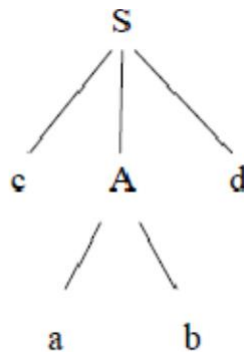
Step1:

Initially create a tree with single node labeled S. An input pointer points to 'c', the first symbol of w. Expand the tree with the production of S.



Step2:

The leftmost leaf 'c' matches the first symbol of w, so advance the input pointer to the second symbol of w 'a' and consider the next leaf 'A'. Expand A using the first alternative.



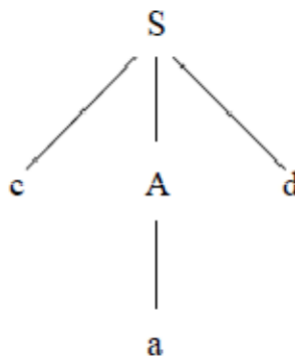
Step3:

The second symbol 'a' of w also matches with second leaf of tree. So advance the input pointer to third symbol of w 'd'. But the third leaf of tree is b which does not match with the input symbol d.

Hence discard the chosen production and reset the pointer to second position. This is called backtracking.

Step4:

Now try the second alternative for A.



Now we can halt and announce the successful completion of parsing.

Example for recursive decent parsing:

A left-recursive grammar can cause a recursive-descent parser to go into an infinite loop.

Hence, elimination of left-recursion must be done before parsing.

Consider the grammar for arithmetic expressions

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

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

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

After eliminating the left-recursion the grammar

becomes,  $E \rightarrow TE'$

$$E' \rightarrow +TE' \mid \epsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \epsilon$$

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

Now we can write the procedure for grammar as follows:

Recursive procedure:

Procedure E()

begin

T();

EPRIME();

End

Procedure EPRIME( )

begin

If input\_symbol='+' then

ADVANCE();

T();

EPRIME( );

end

Procedure T( )

begin

F();

TPRIME( );

End

Procedure TPRIME()

```

Begin
    If input_symbol='*' then
        ADVANCE();
        F();
        TPRIME();
    End

```

Procedure F()

```

Begin
    If input-symbol='id' then
        ADVANCE();
    else if input-symbol='(' then
        ADVANCE();
        E();
    else if input-symbol=')' then
        ADVANCE();
    End
    else ERROR();

```

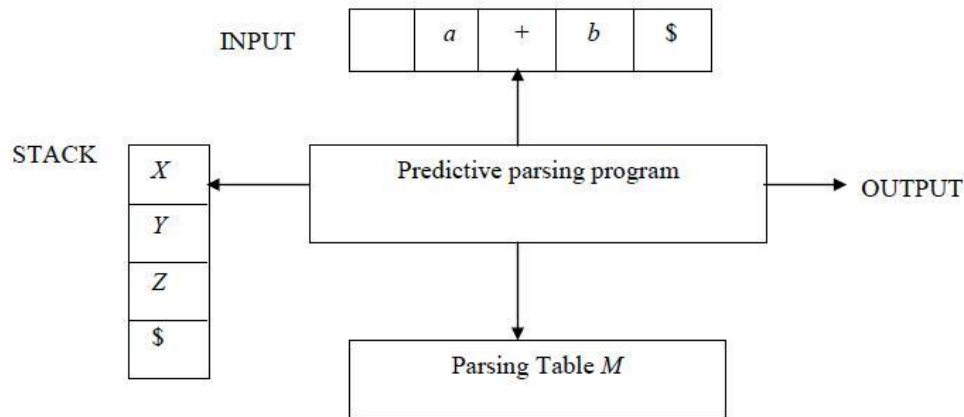
Stack implementation:

PROCEDURE	INPUT STRING
E()	<u>id</u> +id*id
T()	<u>id</u> +id*id
F()	<u>id</u> +id*id
ADVANCE()	id+ <u>id</u> *id
TPRIME()	id+ <u>id</u> *id
EPRIME()	id+ <u>id</u> *id
ADVANCE()	id+id* <u>id</u>
T()	id+id* <u>id</u>
F()	id+id* <u>id</u>
ADVANCE()	id+id* <u>id</u>
TPRIME()	id+id* <u>id</u>
ADVANCE()	id+id* <u>id</u>
F()	id+id* <u>id</u>
ADVANCE()	id+id* <u>id</u>
TPRIME()	id+id* <u>id</u>

### 2.4.2. PREDICTIVE PARSING

- ◆ Predictive parsing is a special case of recursive descent parsing where no backtracking is required.
- ◆ The key problem of predictive parsing is to determine the production to be applied for a non-terminal in case of alternatives.

Non-recursive predictive parser



The table-driven predictive parser has an input buffer, stack, a parsing table and an output stream.

Input buffer:

It consists of strings to be parsed, followed by \$ to indicate the end of the input string.

Stack:

It contains a sequence of grammar symbols preceded by \$ to indicate the bottom of the stack.

Initially, the stack contains the start symbol on top of \$.

Parsing table:

It is a two-dimensional array  $M[A, a]$ , where 'A' is a non-terminal and 'a' is a terminal.

Predictive parsing program:

The parser is controlled by a program that considers X, the symbol on top of stack, and a, the current input symbol. These two symbols determine the parser action. There are three possibilities:

1. If  $X = a = \$$ , the parser halts and announces successful completion of parsing.
2. If  $X = a \neq \$$ , the parser pops X off the stack and advances the input pointer to the next input symbol.
3. If X is a non-terminal, the program consults entry  $M[X, a]$  of the parsing table M.

This entry will either be an X-production of the grammar or an error entry.

If  $M[X, a] = \{X \rightarrow UVW\}$ , the parser replaces  $X$  on top of the stack by  $UVW$ . If  $M[X, a] = \text{error}$ , the parser calls an error recovery routine.

Algorithm for nonrecursive predictive parsing:

Input : A string  $w$  and a parsing table  $M$  for grammar  $G$ .

Output : If  $w$  is in  $L(G)$ , a leftmost derivation of  $w$ ; otherwise, an error indication.

Method : Initially, the parser has  $\$S$  on the stack with  $S$ , the start symbol of  $G$  on top, and  $w\$$  in the input buffer. The program that utilizes the predictive parsing table  $M$  to produce a parse for the input is as follows:

set  $ip$  to point to the first symbol of  $w\$$ ;

repeat

    let  $X$  be the top stack symbol and  $a$  the symbol pointed to by

$ip$ ; if  $X$  is a terminal or  $\$$  then

        if  $X = a$  then

            pop  $X$  from the stack and advance  $ip$

        else error()

    else                                    /\*  $X$  is a non-terminal \*/

        if  $M[X, a] = X \rightarrow Y_1 Y_2 \dots Y_k$  then begin

            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  $X = \$$

Predictive parsing table construction:

The construction of a predictive parser is aided by two functions associated with a grammar

$G$  :

1. FIRST
2. FOLLOW

Rules for first():

1. If  $X$  is terminal, then  $\text{FIRST}(X)$  is  $\{X\}$ .
2. If  $X \rightarrow \epsilon$  is a production, then add  $\epsilon$  to  $\text{FIRST}(X)$ .
3. If  $X$  is non-terminal and  $X \rightarrow a\alpha$  is a production then add  $a$  to  $\text{FIRST}(X)$ .

4. If  $X$  is non-terminal and  $X \rightarrow Y_1 Y_2 \dots Y_k$  is a production, then place  $a$  in  $\text{FIRST}(X)$  if for some  $i$ ,  $a$  is in  $\text{FIRST}(Y_i)$ , and  $\epsilon$  is in all of  $\text{FIRST}(Y_1), \dots, \text{FIRST}(Y_{i-1})$ ; that is,  $Y_1, \dots, Y_{i-1} \Rightarrow \epsilon$ . If  $\epsilon$  is in  $\text{FIRST}(Y_j)$  for all  $j=1, 2, \dots, k$ , then add  $\epsilon$  to  $\text{FIRST}(X)$ .

Rules for  $\text{follow}()$ :

1. If  $S$  is a start symbol, then  $\text{FOLLOW}(S)$  contains  $\$$ .
2. If there is a production  $A \rightarrow \alpha B \beta$ , then everything in  $\text{FIRST}(\beta)$  except  $\epsilon$  is placed in  $\text{follow}(B)$ .
3. If there is a production  $A \rightarrow \alpha B$ , or a production  $A \rightarrow \alpha B \beta$  where  $\text{FIRST}(\beta)$  contains  $\epsilon$ , then everything in  $\text{FOLLOW}(A)$  is in  $\text{FOLLOW}(B)$ .

Algorithm for construction of predictive parsing table:

Input : Grammar  $G$

1. For each production  $A \rightarrow \alpha$  of the grammar, do steps 2 and 3.
2. For each terminal  $a$  in  $\text{FIRST}(\alpha)$ , add  $A \rightarrow \alpha$  to  $M[A, a]$ .
3. If  $\epsilon$  is in  $\text{FIRST}(\alpha)$ , add  $A \rightarrow \alpha$  to  $M[A, b]$  for each terminal  $b$  in  $\text{FOLLOW}(A)$ . If  $\epsilon$  is in  $\text{FIRST}(\alpha)$  and  $\$$  is in  $\text{FOLLOW}(A)$ , add  $A \rightarrow \alpha$  to  $M[A, \$]$ .
4. Make each undefined entry of  $M$  be error.

Example:

Consider the following grammar :

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

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

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

After eliminating left-recursion the grammar is

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \epsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \epsilon$$

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

$\text{First}()$  :

$$\text{FIRST}(E) = \{ (, \text{id} \}$$

$$\text{FIRST}(E') = \{ +, \epsilon \}$$

$$\text{FIRST}(T) = \{ (, \text{id} \}$$

$$\text{FIRST}(T') = \{ *, \epsilon \}$$

$$\text{FIRST}(F) = \{ (, \text{id} \}$$

$\text{Follow}()$ :

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

$$\text{FOLLOW}(E') = \{ \$, ) \}$$

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

$\text{FOLLOW}(T') = \{ +, \$, ) \}$

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

**Predictive parsing table :**

NON-TERMINAL	id	+	*	(	)	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

**Stack implementation:**

stack	Input	Output
\$E	id+id*id \$	
\$E'T	id+id*id \$	$E \rightarrow TE'$
\$E'T'F	id+id*id \$	$T \rightarrow FT'$
\$E'T'id	id+id*id \$	$F \rightarrow id$
\$E'T'	+id*id \$	
\$E'	+id*id \$	$T' \rightarrow \epsilon$
\$E'T+	+id*id \$	$E' \rightarrow +TE'$
\$E'T	id*id \$	
\$E'T'F	id*id \$	$T \rightarrow FT'$
\$E'T'id	id*id \$	$F \rightarrow id$
\$E'T'	*id \$	
\$E'T'F*	*id \$	$T' \rightarrow *FT'$
\$E'T'F	id \$	
\$E'T'id	id \$	$F \rightarrow id$
\$E'T'	\$	
\$E'	\$	$T' \rightarrow \epsilon$
\$	\$	$E' \rightarrow \epsilon$

LL(1) grammar:

The parsing table entries are single entries. So each location has not more than one entry.

This type of grammar is called LL(1) grammar.

Consider this following grammar:

$S \rightarrow iEtS \mid iEtSeS \mid a$

$E \rightarrow b$



After eliminating left factoring, we have

$$S \rightarrow iEtSS' \mid a$$

$$S' \rightarrow eS \mid \epsilon$$

$$E \rightarrow b$$

To construct a parsing table, we need FIRST() and FOLLOW() for all the non-terminals.  $FIRST(S) = \{ i, a \}$

$$FIRST(S') = \{ e, \epsilon \}$$

$$FIRST(E) = \{ b \}$$

$$FOLLOW(S) = \{ \$, e \}$$

$$FOLLOW(S') = \{ \$, e \}$$

$$FOLLOW(E) = \{ t \}$$

**Parsing table:**

NON-TERMINAL	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow eS$ $S' \rightarrow \epsilon$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				

Since there are more than one production, the grammar is not LL(1) grammar.

Actions performed in predictive parsing:

1. Shift
2. Reduce
3. Accept
4. Error

1. Elimination of left recursion, left factoring and ambiguous grammar.
2. Construct FIRST() and FOLLOW() for all non-terminals.
3. Construct predictive parsing table.
4. Parse the given input string using stack and parsing table.

**2.5. BOTTOM-UP PARSING**

Constructing a parse tree for an input string beginning at the leaves and going towards the root is called bottom-up parsing.

A general type of bottom-up parser is a shift-reduce parser.

**2.5.1. SHIFT-REDUCE PARSING**

Shift-reduce parsing is a type of bottom -up parsing that attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

Example:

Consider the grammar:

$$S \rightarrow aABe$$

$$A \rightarrow Abc \mid b$$

$$B \rightarrow d$$

The sentence to be recognized is abbcde.

### 3.1 Intermediate Code Generation

- Grammar symbols are associated with **attributes** to associate information with the programming language constructs that they represent.
- Values of these attributes are evaluated by the **semantic rules** associated with the production rules.
- Evaluation of these semantic rules:
  - may generate intermediate codes
  - may put information into the symbol table
  - may perform type checking
  - may issue error messages
  - may perform some other activities
  - In fact, they may perform almost any activities.
- An attribute may hold almost any thing.
  - A string, a number, a memory location, a complex record.
- 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.

Example:

<u>Production</u>	<u>Semantic Rule</u>	<u>Program Fragment</u>
$L \rightarrow E \text{ return}$	$\text{print}(E.val)$	$\text{print}(val[top-1])$
$E \rightarrow E^1 + T$	$E.val = E^1.val + T.val$	$val[ntop] = val[top-2] + val[top]$
$E \rightarrow T$	$E.val = T.val$	
$T \rightarrow T^1 * F$	$T.val = T^1.val * F.val$	$val[ntop] = val[top-2] * val[top]$
$T \rightarrow F$	$T.val = F.val$	
$F \rightarrow ( E )$	$F.val = E.val$	$val[ntop] = val[top-1]$
$F \rightarrow \text{digit}$	$F.val = \text{digit.lexval}$	$val[top] = \text{digit.lexval}$

- Symbols E, T, and F are associated with an attribute *val*.
- The token **digit** has an attribute *lexval* (it is assumed that it is evaluated by the lexical analyzer).
- The *Program Fragment* above represents the implementation of the semantic rule for a bottom-up parser.
- At each shift of **digit**, we also push **digit.lexval** into *val-stack*.
- At all other shifts, we do not put anything into *val-stack* because other terminals do not have attributes (but we increment the stack pointer for *val-stack*).
- The above model is suited for a desk calculator where the purpose is to evaluate and to generate code.

#### Intermediate Code Generation

- *Intermediate codes* are machine independent codes, but they are close to machine instructions.
- The given program in a source language is converted to an equivalent program in an intermediate language by the intermediate code generator.
- Intermediate language can be many different languages, and the designer of the compiler decides this intermediate language.
  - syntax trees can be used as an intermediate language.
  - postfix notation can be used as an intermediate language.
  - three-address code (Quadraples) can be used as an intermediate language
    - we will use quadraples to discuss intermediate code generation
    - quadraples are close to machine instructions, but they are not actual machine instructions.

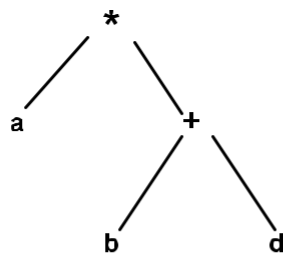
### Syntax Tree

Syntax Tree is a variant of the Parse tree, where each leaf represents an operand and each interior node an operator.

Example:

<u>Production</u>	<u>Semantic Rule</u>
$E \rightarrow E1 \text{ op } E2$	$E.val = \text{NODE}(\text{op}, E1.val, E2.val)$
$E \rightarrow (E1)$	$E.val = E1.val$
$E \rightarrow - E1$	$E.val = \text{UNARY}(-, E1.val)$
$E \rightarrow \text{id}$	$E.val = \text{LEAF}(\text{id})$

A sentence  $a*(b+d)$  would have the following syntax tree:



### Postfix Notation

Postfix Notation is another useful form of intermediate code if the language is mostly expressions.

Example:

<u>Production</u>	<u>Semantic Rule</u>	<u>Program Fragment</u>
$E \rightarrow E1 \text{ op } E2$	$E.code = E1.code \parallel E2.code \parallel \text{op}$	print op
$E \rightarrow (E1)$	$E.code = E1.code$	
$E \rightarrow \text{id}$	$E.code = \text{id}$	print id

### Three Address Code

- We use the term "three-address code" because each statement usually contains three addresses (two for operands, one for the result).
- The most general kind of three-address code is:

$$x := y \text{ op } z$$

where x, y and z are names, constants or compiler-generated temporaries; **op** is any operator.

- But we may also the following notation for quadraples (much better notation because it looks like a machine code instruction)

$$\text{op } y,z,x$$

apply operator op to y and z, and store the result in x.

**Representation of three-address codes**

Three-address code can be represented in various forms viz. Quadruples, Triples and Indirect Triples. These forms are demonstrated by way of an example below.

Example:

$A = -B * (C + D)$   
 Three-Address code is as follows:  
 $T1 = -B$   
 $T2 = C + D$   
 $T3 = T1 * T2$   
 $A = T3$

Quadruple:

	<i>Operator</i>	<i>Operand 1</i>	<i>Operand 2</i>	<i>Result</i>
(1)	-	B		T1
(2)	+	C	D	T2
(3)	*	T1	T2	T3
(4)	=	A	T3	

Triple:

	<i>Operator</i>	<i>Operand 1</i>	<i>Operand 2</i>
(1)	-	B	
(2)	+	C	D
(3)	*	(1)	(2)
(4)	=	A	(3)

Indirect Triple:

	<i>Statement</i>
(0)	(56)
(1)	(57)
(2)	(58)
(3)	(59)

	<i>Operator</i>	<i>Operand 1</i>	<i>Operand 2</i>
(56)	-	B	
(57)	+	C	D
(58)	*	(56)	(57)
(59)	=	A	(58)

### Translation of Assignment Statements

A statement  $A := - B * (C + D)$  has the following three-address translation:

```
T1 := - B
T2 := C+D
T3 := T1* T2
A := T3
```

<u>Production</u>	<u>Semantic Action</u>
$S \rightarrow id := E$	$S.code = E.code \parallel gen( id.place = E.place )$
$E \rightarrow E1 + E2$	$E.place = newtemp();$ $E.code = E1.code \parallel E2.code \parallel gen( E.place = E1.place + E2.place )$
$E \rightarrow E1 * E2$	$E.place = newtemp();$ $E.code = E1.code \parallel E2.code \parallel gen( E.place = E1.place * E2.place )$
$E \rightarrow - E1$	$E.place = newtemp();$ $E.code = E1.code \parallel gen( E.place = - E1.place )$
$E \rightarrow ( E1 )$	$E.place = E1.place;$ $E.code = E1.code$
$E \rightarrow id$	$E.place = id.place;$ $E.code = null$

### Translation of Boolean Expressions

Grammar for Boolean Expressions is:

```
E → E or E
E → E and E
E → not E
E → ( E )
E → id
E → id relop id
```

There are two representations viz. Numerical and Control-Flow.

#### Numerical Representation of Boolean

TRUE is denoted by 1 and FALSE by 0.

- Expressions are evaluated from left to right, in a manner similar to arithmetic expressions.

Example:

The translation for **A or B and C** is the three-address sequence:

```
T1 := B and C
T2 := A or T1
```

Also, the translation of a relational expression such as  $A < B$  is the three-address sequence:

- (1) if A < B goto (4)
- (2) T := 0
- (3) goto (5)
- (4) T := 1
- (5)

Therefore, a Boolean expression A < B or C can be translated as:

- (1) if A < B goto (4)
- (2) T1 := 0
- (3) goto (5)
- (4) T1 := 1
- (5) T2 := T1 or C

<u>Production</u>	<u>Semantic Action</u>
E → E1 or E2	T = newtemp (); E.place = T; Gen (T = E1.place or E2.place)
E → E1 and E2	T = newtemp (); E.place = T; Gen (T = E1.place and E2.place)
E → not E1	T = newtemp (); E.place = T; Gen (T = not E1.place)
E → ( E1 )	E.place = E1.place; E.code = E1.code
E → id	E.place = id.place; E.code = null
·	T = newtemp (); E.place = T; Gen (if id1.place relop id2.place goto NEXTQUAD+3) Gen (T = 0) Gen (goto NEXTQUAD+2) Gen (T = 1)

- Quadruples are being generated and NEXTQUAD indicates the next available entry in the quadruple array.

### Control-Flow Representation of Boolean Expressions

If we evaluate Boolean expressions by program position, we may be able to avoid evaluating the entire expressions.

- In A or B, if we determine A to be true, we need not evaluate B and can declare the entire expression to be true.
- In A and B, if we determine A to be false, we need not evaluate B and can declare the entire expression to be false.
- A better code can thus be generated using the above properties.

Example:

The statement **if (A<B || C<D) x = y + z;** can be translated as

- (1) if A<B goto (4)
- (2) if C<D goto (4)
- (3) goto (6)
- (4) T = y + z
- (5) X = T
- (6)

Here (4) is a true exit and (6) is a false exit of the Boolean expressions.

### Generating 3-address code for Numerical Representation of Boolean expressions

- Consider a production **E → E1 or E2** that represents the OR Boolean expression. If E1 is true, we know that E is true so we make the location TRUE for E1 be the same as TRUE for E. If E1 is false, then we must evaluate E2, so we make FALSE for E1 be the first statement in the code for E2. The TRUE and FALSE exits can be made the same as the TRUE and FALSE exits of E, respectively.
- Consider a production **E → E1 and E2** that represents the AND Boolean expression. If E1 is false, we know that E is false so we make the location FALSE for E1 be the same as FALSE for E. If E1 is true, then we must evaluate E2, so we make TRUE for E1 be the first statement in the code for E2. The TRUE and FALSE exits can be made the same as the TRUE and FALSE exits of E, respectively.
- Consider the production **E → not E** that represents the NOT Boolean expression. We may simply interchange the TRUE and FALSE exits of E1 to get the TRUE and FALSE exits of E.
- To generate quadruples in the manner suggested above, we use three functions- MakeList, Merge and Backpatch that shall work on the list of quadruples as suggested by their name.
- If we need to proceed to E2 after evaluating E1, we have an efficient way of doing this by modifying our grammar as follows:

```

E → E or M E
E → E and M E
E → not E
E → ( E )
E → id
E → id relop id
M → ε
    
```

- The translation scheme for this grammar would as follows:

<u>Production</u>	<u>Semantic Action</u>
E → E1 or M E2	BACKPATCH (E1.FALSE, M.QUAD); E.TRUE = MERGE (E1.TRUE, E2.TRUE); E.FALSE = E2.FALSE;
E → E1 and M E2	BACKPATCH (E1.TRUE, M.QUAD); E.TRUE = E2.TRUE; E.FALSE = MERGE (E1.FALSE, E2.FALSE);
E → not E1	E.TRUE = E1.FALSE; E.FALSE = E1.TRUE;

$E \rightarrow ( E1 )$	$E.TRUE = E1.TRUE;$ $E.FALSE = E1.FALSE;$
$E \rightarrow id$	$E.TRUE = MAKELIST (NEXTQUAD);$ $E.FALSE = MAKELIST (NEXTQUAD + 1);$ $GEN (if id.PLACE goto \_);$ $GEN (goto \_);$
$E \rightarrow id1 relop id2$	$E.TRUE = MAKELIST (NEXTQUAD);$ $E.FALSE = MAKELIST (NEXTQUAD + 1);$ $GEN ( if id1.PLACE relop id2.PLACE goto \_ );$ $GEN (goto \_);$
$M \rightarrow \epsilon$	$M.QUAD = NEXTQUAD;$

Example:

For the expression  $P < Q$  or  $R < S$  and  $T$ , the parsing steps and corresponding semantic actions are shown below. We assume that NEXTQUAD has an initial value of 100.

Step 1:  $P < Q$  gets reduced to  $E$  by  $E \rightarrow id relop id$ . The grammatical form is  $E1$  or  $R < S$  and  $T$ .

We have the following code generated (Makelist).

```
100: if P<Q goto _
101: goto _
```

$E1$  is true if goto of 100 is reached and false if goto of 101 is reached.

Step 2:  $R < S$  gets reduced to  $E$  by  $E \rightarrow id relop id$ . The grammatical form is  $E1$  or  $E2$  and  $T$ .

We have the following code generated (Makelist).

```
102: if R<S goto _
103: goto _
```

$E2$  is true if goto of 102 is reached and false if goto of 103 is reached.

Step 3:  $T$  gets reduced to  $E$  by  $E \rightarrow id$ . The grammatical form is  $E1$  or  $E2$  and  $E3$ .

We have the following code generated (Makelist).

```
104: if T goto _
105: goto _
```

$E3$  is true if goto of 104 is reached and false if goto of 105 is reached.

Step 4:  $E2$  and  $E3$  gets reduced to  $E$  by  $E \rightarrow E$  and  $E$ . The grammatical form is  $E1$  or  $E4$ .

We have no new code generated but changes are made in the already generated code (Backpatch).



```

100: if P<Q goto _
101: goto _
102: if R<S goto 104
103: goto _
104: if T goto _
105: goto _

```

E4 is true only if E3.TRUE (goto of 104) is reached. E4 is false if E2.FALSE (goto of 103) or E3.FALSE (goto of 105) is reached (Merge).

Step 5: E1 or E4 gets reduced to E by  $E \rightarrow E \text{ or } E$ . The grammatical form is E.

We have no new code generated but changes are made in the already generated code (Backpatch).

```

100: if P<Q goto _
101: goto 102
102: if R<S goto 104
103: goto _
104: if T goto _
105: goto _

```

E is true only if E1.TRUE (goto of 100) or E2.TRUE (goto of 104) is reached (Merge). E is false if E4.FALSE (goto of 103 or 105) is reached.

### Mixed Mode Expressions

- Boolean expressions may in practice contain arithmetic sub expressions e.g.  $(A+B)>C$ .
- We can accommodate such sub-expressions by adding the production  $E \rightarrow E \text{ op } E$  to our grammar.
- We will also add a new field MODE for E. If E has been achieved after reduction using the above (arithmetic) production, we make  $E.MODE = \text{arith}$ , otherwise make  $E.MODE = \text{bool}$ .
- If  $E.MODE = \text{arith}$ , we treat it arithmetically and use  $E.PLACE$ . If  $E.MODE = \text{bool}$ , we treat it as Boolean and use  $E.FALSE$  and  $E.TRUE$ .

### Statements that Alter Flow of Control

- In order to implement goto statements, we need to define a LABEL for a statement. A production can be added for this purpose:

$$S \rightarrow \text{LABEL} : S$$

$$\text{LABEL} \rightarrow \text{id}$$

- The semantic action attached with this production is to record the LABEL and its value (NEXTQUAD) in the symbol table. It will also Backpatch any previous references to this LABEL with its current value.
- Following grammar can be used to incorporate structured Flow-of-control constructs:

- (1)  $S \rightarrow \text{if } E \text{ then } S$
- (2)  $S \rightarrow \text{if } E \text{ then } S \text{ else } S$
- (3)  $S \rightarrow \text{while } E \text{ do } S$
- (4)  $S \rightarrow \text{begin } L \text{ end}$
- (5)  $S \rightarrow A$

- (6)  $L \rightarrow L ; S$
- (7)  $L \rightarrow S$

Here, S denotes a statement, L a statement-list, A an assignment statement and E a Boolean-valued expression.

**Translation Scheme for statements that alter flow of control**

- We introduce a new field NEXT for S and L like TRUE and FALSE for E. S.NEXT and L.NEXT are respectively the pointers to a list of all conditional and unconditional jumps to the quadruple following statement S and statement-list L in execution order.
- We also introduce the marker non-terminal M as in the case of grammar for Boolean expressions. This is put before statement in if-then, before both statements in if-then-else and the statement in while-do as we may need to proceed to them after evaluating E. In case of while-do, we also need to put M before E as we may need to come back to it after executing S.
- In case of if-then-else, if we evaluate E to be true, first S will be executed. After this we should ensure that instead of second S, the code after this if-then-else statement be executed. We thus place another non-terminal marker N after first S i.e. before else.
- The grammar now is as follows:

- (1)  $S \rightarrow \text{if } E \text{ then } M S$
- (2)  $S \rightarrow \text{if } E \text{ then } M S N \text{ else } M S$
- (3)  $S \rightarrow \text{while } M E \text{ do } M S$
- (4)  $S \rightarrow \text{begin } L \text{ end}$
- (5)  $S \rightarrow A$
- (6)  $L \rightarrow L ; M S$
- (7)  $L \rightarrow S$
- (8)  $M \rightarrow \epsilon$
- (9)  $N \rightarrow \epsilon$

- The translation scheme for this grammar would as follows:

<u>Production</u>	<u>Semantic Action</u>
$S \rightarrow \text{if } E \text{ then } M S1$	BACKPATCH (E.TRUE, M.QUAD) S.NEXT = MERGE (E.FALSE, S1.NEXT)
$S \rightarrow \text{if } E \text{ then } M1 S1 N \text{ else } M2 S2$	BACKPATCH (E.TRUE, M1.QUAD) BACKPATCH (E.FALSE, M2.QUAD) S.NEXT = MERGE (S1.NEXT, N.NEXT, S2.NEXT)
$S \rightarrow \text{while } M1 E \text{ do } M2 S1$	BACKPATCH (S1.NEXT, M1.QUAD) BACKPATCH (E.TRUE, M2.QUAD) S.NEXT = E.FALSE GEN (goto M1.QUAD)
$S \rightarrow \text{begin } L \text{ end}$	S.NEXT = L.NEXT
$S \rightarrow A$	S.NEXT = MAKELIST ( )
$L \rightarrow L1 ; M S$	BACKPATCH (L1.NEXT, M.QUAD) L.NEXT = S.NEXT
$L \rightarrow S$	L.NEXT = S.NEXT

$M \rightarrow \epsilon$	$M\_QUAD = NEXTQUAD$
$N \rightarrow \epsilon$	$N\_NEXT = MAKELIST (NEXTQUAD)$ $GEN (goto \_)$

**Postfix Translations**

In an production  $A \rightarrow \alpha$ , the translation rule of A.CODE consists of the concatenation of the CODE translations of the non-terminals in  $\alpha$  in the same order as the non-terminals appear in  $\alpha$ .

- Productions can be factored to achieve Postfix form.

**Postfix translation of while statement**

The production

$S \rightarrow \text{while } M1 \text{ E do } M2 \text{ S1}$

can be factored as

$S \rightarrow C \text{ S1}$   
 $C \rightarrow W \text{ E do}$   
 $W \rightarrow \text{while}$

A suitable translation scheme would be

<u>Production</u>	<u>Semantic Action</u>
$W \rightarrow \text{while}$	$W\_QUAD = NEXTQUAD$
$C \rightarrow W \text{ E do}$	$C\_QUAD = W\_QUAD$ $BACKPATCH (E.TRUE, NEXTQUAD)$ $C.FALSE = E.FALSE$
$S \rightarrow C \text{ S1}$	$BACKPATCH (S1.NEXT, C\_QUAD)$ $S\_NEXT = C.FALSE$ $GEN (goto C\_QUAD)$

**Postfix translation of for statement**

Consider the following production which stands for the for-statement

$S \rightarrow \text{for } L = E1 \text{ step } E2 \text{ to } E3 \text{ do } S1$

Here L is any expression with l-value, usually a variable, called the index. E1, E2 and E3 are expressions called the initial value, increment and limit, respectively. Semantically, the for-statement is equivalent to the following program.

```
begin
  INDEX = addr ( L );
  *INDEX = E1;
  INCR = E2;
  LIMIT = E3;
  while *INDEX <= LIMIT do
    begin
```

```

        code for statement S1;
        *|INDEX = *|INDEX + |INCR;
    end
end

```

The non-terminals L, E1, E2, E3 and S appear in the same order as in the production. The production can be factored as

- (1)  $F \rightarrow \text{for } L$
- (2)  $T \rightarrow F = E1 \text{ by } E2 \text{ to } E3 \text{ do}$
- (3)  $S \rightarrow T S1$

A suitable translation scheme would be

<u>Production</u>	<u>Semantic Action</u>
$F \rightarrow \text{for } L$	$F.INDEX = L.INDEX$
$T \rightarrow F = E1 \text{ by } E2 \text{ to } E3 \text{ do}$	$GEN (*F.INDEX = E1.PLACE)$ $INCR = NEWTEMP ( )$ $LIMIT = NEWTEMP ( )$ $GEN (INCR = E2.PLACE)$ $GEN (LIMIT = E3.PLACE)$ $T.QUAD = NEXTQUAD$ $T.NEXT = MAKELIST (NEXTQUAD)$ $GEN (IF *F.INDEX > LIMIT \text{ goto } \_)$ $T.INDEX = F.INDEX$ $T.INCR = INCR$
$S \rightarrow T S1$	$BACKPATCH (S1,NEXT, NEXTQUAD)$ $GEN (*T.INDEX = *T.INDEX + T.INCR)$ $GEN (\text{goto } T.QUAD)$ $S.NEXT = T.NEXT$

### Translation with a Top-Down Parser

- Any translation done by top-down parser can be done in a bottom-up parser also.
- But in certain situations, translation with a top-down parser is advantageous as tricks such as placing a marker non-terminal can be avoided.
- Semantic routines can be called in the middle of productions in top-down parser.

### Array references in arithmetic expressions

- Elements of arrays can be accessed quickly if the elements are stored in a block of consecutive locations.
- For a one-dimensional array A:

**Base (A)** is the address of the first location of the array A,  
**width** is the width of each array element,  
**low** is the index of the first array element

location of  $A[i] = \text{baseA} + (i - \text{low}) * \text{width}$

$\text{baseA} + (i - \text{low}) * \text{width}$

can be re-written as

$$i * \text{width} + (\text{baseA} - \text{low} * \text{width})$$

↙
↘

should be computed at run-time
can be computed at compile-time

- So, the location of A[i] can be computed at the run-time by evaluating the formula  $i * \text{width} + c$  where  $c$  is  $(\text{baseA} - \text{low} * \text{width})$  which is evaluated at compile-time.
- Intermediate code generator should produce the code to evaluate this formula  $i * \text{width} + c$  (one multiplication and one addition operation).
- A two-dimensional array can be stored in either row-major (row-by-row) or column-major (column-by-column).
- Most of the programming languages use row-major method.
- The location of  $A[i_1, i_2]$  is  $\text{baseA} + ((i_1 - \text{low}_1) * n_2 + i_2 - \text{low}_2) * \text{width}$

**baseA** is the location of the array A.  
**low1** is the index of the first row  
**low2** is the index of the first column  
**n2** is the number of elements in each row  
**width** is the width of each array element

Again, this formula can be re-written as

$$((i_1 * n_2) + i_2) * \text{width} + (\text{baseA} - ((\text{low}_1 * n_1) + \text{low}_2) * \text{width})$$

↙
↘

should be computed at run-time
can be computed at compile-time

- Arrays of any dimension can be dealt in a similar but general manner.

In general, the location of  $A[i_1, i_2, \dots, i_k]$  is

$$(( \dots ((i_1 * n_2) + i_2) \dots) * n_k + i_k) * \text{width} + (\text{baseA} - ((\dots((\text{low}_1 * n_1) + \text{low}_2) \dots) * n_k + \text{low}_k) * \text{width})$$

So, the intermediate code generator should produce the codes to evaluate the following formula (to find the location of  $A[i_1, i_2, \dots, i_k]$ ) :

$$(( \dots ((i_1 * n_2) + i_2) \dots) * n_k + i_k) * \text{width} + c$$

To evaluate the  $(( \dots ((i_1 * n_2) + i_2) \dots) * n_k + i_k)$  portion of this formula, we can use the recurrence equation:

$$e_1 = i_1$$

$$e_m = e_{m-1} * n_m + i_m$$

### Grammar and Translation Scheme

The grammar and suitable translation scheme for arithmetic expressions with array references is as given below:

<u>Production</u>	<u>Semantic Action</u>
$S \rightarrow L = E$	if (L.OFFSET = NULL) then GEN (L.PLACE = E.PLACE) else GEN(L.PLACE [ L.OFFSET ] = E.PLACE)
$E \rightarrow E_1 + E_2$	E.PLACE = NEWTEMP ( )

	GEN (E.PLACE = E1.PLACE + E2.PLACE)
E → ( E1 )	E.PLACE = E1.PLACE
E → L	if (L.OFFSET = NULL) then E.PLACE = L.PLACE else {E.PLACE = NEWTEMP ( ); GEN (E.PLACE = L.PLACE[L.OFFSET])}
L → id	L.PLACE = id.PLACE L.OFFSET = NULL
L → E[LIST ]	L.PLACE = NEWTEMP ( ) L.OFFSET = NEWTEMP ( ) GEN (L.PLACE = E[LIST.ARRAY - C) GEN (L.OFFSET = E[LIST.PLACE * WIDTH (E[LIST.ARRAY))
E[LIST → E[LIST1 , E	E[LIST.ARRAY = E[LIST1.ARRAY E[LIST.PLACE = NEWTEMP ( ) E[LIST.NDIM = E[LIST1.NDIM + 1 GEN (E[LIST.PLACE = E[LIST1.PLACE * LIMIT (E[LIST.ARRAY, E[LIST.NDIM)) GEN (E[LIST.PLACE = E.PLACE + E[LIST.PLACE)
E[LIST → id [ E	E[LIST.ARRAY = id.PLACE E[LIST.PLACE = E.PLACE E[LIST.NDIM = 1

Here, NDIM denotes the number of dimensions, LIMIT (ARRAY, i) function returns the upper limit along the ith dimension of ARRAY i.e. ni, WIDTH (ARRAY) returns the number of bytes for one element of ARRAY.

### Declarations

Following is the grammar and a suitable translation scheme for declaration statements:

<u>Production</u>	<u>Semantic Action</u>
D → integer, id	ENTER (id.PLACE, integer) D.ATTR = integer
D → real, id	ENTER (id.PLACE, real) D.ATTR = real
D → D1, id	ENTER (id.PLACE, D1.ATTR) D.ATTR = D1.ATTR

Here, ENTER makes the entry into symbol table while ATTR is used to trace the data type.

### Procedure Calls

Following is the grammar and a suitable translation scheme for Procedure Calls:

<u>Production</u>	<u>Semantic Action</u>
S → call id (E[LIST)	for each item p on QUEUE do GEN (param p) GEN (call id.PLACE)

ELIST → ELIST, E                    append E.PLACE to the end of QUEUE  
 ELIST → E                            initialize QUEUE to contain only E.PLACE

QUEUE is used to store the list of parameters in the procedure call.

### Case Statements

The case statement has following syntax:

```

switch E
begin
    case V1: S1
    case V2: S2
    .
    .
    .
    case Vn-1: Sn-1
    default: Sn
end
    
```

The translation scheme for this shown below:

```

                                code to evaluate E into T
                                goto TEST
L1:   code for S1
                                goto NEXT
L2:   code for S2
                                goto NEXT
    .
    .
    .
Ln-1: code for Sn-1
                                goto NEXT
Ln:   code for Sn
                                goto NEXT
TEST: if T = V1 goto L1
                                if T = V2 goto L2
    .
    .
    .
                                if T = Vn-1 goto Ln-1
                                goto Ln
NEXT:
    
```

## 3.2 TYPE CHECKING

### TYPE CHECKING

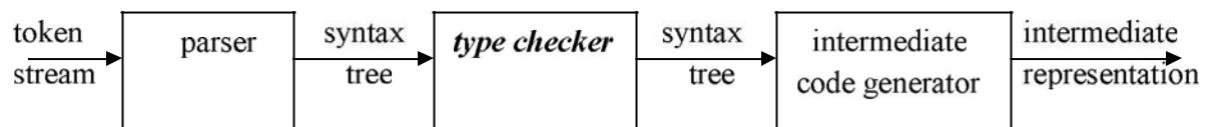
A compiler must check that the source program follows both syntactic and semantic conventions of the source language.

This checking, called *static checking*, detects and reports programming errors.

Some examples of static checks:

1. **Type checks** – A compiler should report an error if an operator is applied to an incompatible operand. Example: If an array variable and function variable are added together.
2. **Flow-of-control checks** – Statements that cause flow of control to leave a construct must have some place to which to transfer the flow of control. Example: An error occurs when an enclosing statement, such as `break`, does not exist in switch statement.

#### Position of type checker



- A *type checker* verifies that the type of a construct matches that expected by its context. For example : arithmetic operator *mod* in Pascal requires integer operands, so a type checker verifies that the operands of *mod* have type integer.
- Type information gathered by a type checker may be needed when code is generated.

### TYPE SYSTEMS

The design of a type checker for a language is based on information about the syntactic constructs in the language, the notion of types, and the rules for assigning types to language constructs.

For example : “ if both operands of the arithmetic operators of +,- and \* are of type integer, then the result is of type integer ”

#### Type Expressions

- The type of a language construct will be denoted by a “type expression.”
- A type expression is either a basic type or is formed by applying an operator called a *type constructor* to other type expressions.
- The sets of basic types and constructors depend on the language to be checked.

The following are the definitions of type expressions:

1. Basic types such as *boolean*, *char*, *integer*, *real* are type expressions.

A special basic type, *type\_error*, will signal an error during type checking; *void* denoting “the absence of a value” allows statements to be checked.

2. Since type expressions may be named, a type name is a type expression.

3. A type constructor applied to type expressions is a type expression.

Constructors include:

**Arrays** : If T is a type expression then *array* (I,T) is a type expression denoting the type of an array with elements of type T and index set I.

**Products** : If T<sub>1</sub> and T<sub>2</sub> are type expressions, then their Cartesian product T<sub>1</sub> X T<sub>2</sub> is a type expression.



**Records** : The difference between a record and a product is that the fields of a record have names. The *record* type constructor will be applied to a tuple formed from field names and field types.

For example:

```

type row = record
    address: integer;
    lexeme: array[1..15] of char
end;
var table: array[1..101] of row;
    
```

declares the type name *row* representing the type expression *record((address X integer) X (lexeme X array(1..15,char)))* and the variable *table* to be an array of records of this type.

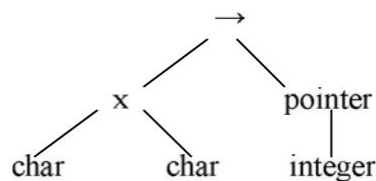
**Pointers** : If T is a type expression, then *pointer*(T) is a type expression denoting the type “pointer to an object of type T”.

For example, *var p: ↑ row* declares variable p to have type *pointer*(row).

**Functions** : A function in programming languages maps a *domain type D* to a *range type R*. The type of such function is denoted by the type expression  $D \rightarrow R$

4. Type expressions may contain variables whose values are type expressions.

**Tree representation for  $\text{char} \times \text{char} \rightarrow \text{pointer}(\text{integer})$**



**Type systems**

- A *type system* is a collection of rules for assigning type expressions to the various parts of a program.
- A type checker implements a type system. It is specified in a syntax-directed manner.
- Different type systems may be used by different compilers or processors of the same language.

**Static and Dynamic Checking of Types**

- Checking done by a compiler is said to be static, while checking done when the target program runs is termed dynamic.
- Any check can be done dynamically, if the target code carries the type of an element along with the value of that element.

### Sound type system

A *sound* type system eliminates the need for dynamic checking for type errors because it allows us to determine statically that these errors cannot occur when the target program runs. That is, if a sound type system assigns a type other than *type\_error* to a program part, then type errors cannot occur when the target code for the program part is run.

### Strongly typed language

A language is strongly typed if its compiler can guarantee that the programs it accepts will execute without type errors.

### Error Recovery

- Since type checking has the potential for catching errors in program, it is desirable for type checker to recover from errors, so it can check the rest of the input.
- Error handling has to be designed into the type system right from the start; the type checking rules must be prepared to cope with errors.

### SPECIFICATION OF A SIMPLE TYPE CHECKER

Here, we specify a type checker for a simple language in which the type of each identifier must be declared before the identifier is used. The type checker is a translation scheme that synthesizes the type of each expression from the types of its subexpressions. The type checker can handle arrays, pointers, statements and functions.

#### A Simple Language

Consider the following grammar:

$$\begin{aligned} P &\rightarrow D ; E \\ D &\rightarrow D ; D \mid id : T \\ T &\rightarrow char \mid integer \mid array [ num ] \text{ of } T \mid \uparrow T \\ E &\rightarrow literal \mid num \mid id \mid E \text{ mod } E \mid E [ E ] \mid E \uparrow \end{aligned}$$

#### Translation scheme:

$$\begin{aligned} P &\rightarrow D ; E \\ D &\rightarrow D ; D \\ D &\rightarrow id : T \quad \{ addtype (id.entry, T.type) \} \\ T &\rightarrow char \quad \{ T.type := char \} \\ T &\rightarrow integer \quad \{ T.type := integer \} \\ T &\rightarrow \uparrow T_1 \quad \{ T.type := pointer(T_1.type) \} \\ T &\rightarrow array [ num ] \text{ of } T_1 \quad \{ T.type := array ( 1 \dots num.val, T_1.type) \} \end{aligned}$$

In the above language,

- There are two basic types : char and integer ;
- *type\_error* is used to signal errors;
- the prefix operator  $\uparrow$  builds a pointer type. Example ,  $\uparrow$  **integer** leads to the type expression **pointer ( integer )**.

**Type checking of expressions**

In the following rules, the attribute *type* for E gives the type expression assigned to the expression generated by E.

1.  $E \rightarrow \mathbf{literal}$        $\{ E.type := char \}$   
 $E \rightarrow \mathbf{num}$        $\{ E.type := integer \}$   
 Here, constants represented by the tokens **literal** and **num** have type *char* and *integer*.

2.  $E \rightarrow \mathbf{id}$        $\{ E.type := lookup ( id.entry ) \}$   
 $lookup ( e )$  is used to fetch the type saved in the symbol table entry pointed to by e.

3.  $E \rightarrow E_1 \mathbf{mod} E_2$      $\{ E.type := \mathbf{if} E_1.type = integer \mathbf{and}$   
 $E_2.type = integer \mathbf{then} integer$   
 $\mathbf{else} type\_error \}$

The expression formed by applying the mod operator to two subexpressions of type integer has type integer; otherwise, its type is *type\_error*.

4.  $E \rightarrow E_1 [ E_2 ]$        $\{ E.type := \mathbf{if} E_2.type = integer \mathbf{and}$   
 $E_1.type = array(s,t) \mathbf{then} t$   
 $\mathbf{else} type\_error \}$

In an array reference  $E_1 [ E_2 ]$ , the index expression  $E_2$  must have type integer. The result is the element type *t* obtained from the type  $array(s,t)$  of  $E_1$ .

5.  $E \rightarrow E_1 \uparrow$        $\{ E.type := \mathbf{if} E_1.type = pointer (t) \mathbf{then} t$   
 $\mathbf{else} type\_error \}$

The postfix operator  $\uparrow$  yields the object pointed to by its operand. The type of  $E \uparrow$  is the type *t* of the object pointed to by the pointer E.

**Type checking of statements**

Statements do not have values; hence the basic type *void* can be assigned to them. If an error is detected within a statement, then *type\_error* is assigned.

**Translation scheme for checking the type of statements:****1. Assignment statement:**

$$S \rightarrow \mathbf{id} := E \quad \{ S.type := \mathbf{if} id.type = E.type \mathbf{then} void$$

$$\mathbf{else} type\_error \}$$
**2. Conditional statement:**

$$S \rightarrow \mathbf{if} E \mathbf{then} S_1 \quad \{ S.type := \mathbf{if} E.type = boolean \mathbf{then} S_1.type$$

$$\mathbf{else} type\_error \}$$
**3. While statement:**

$$S \rightarrow \mathbf{while} E \mathbf{do} S_1 \quad \{ S.type := \mathbf{if} E.type = boolean \mathbf{then} S_1.type$$

$$\mathbf{else} type\_error \}$$

**4. Sequence of statements:**
$$S \rightarrow S_1 ; S_2 \quad \{ S.type := \mathbf{if} S_1.type = \mathit{void} \text{ and} \\ S_1.type = \mathit{void} \mathbf{then} \mathit{void} \\ \mathbf{else} \mathit{type\_error} \}$$
**Type checking of functions**

The rule for checking the type of a function application is :

$$E \rightarrow E_1 ( E_2 ) \quad \{ E.type := \mathbf{if} E_2.type = s \mathbf{and} \\ E_1.type = s \rightarrow t \mathbf{then} t \\ \mathbf{else} \mathit{type\_error} \}$$

## 4.1 STORAGE ORGANIZATION

### SOURCE LANGUAGE ISSUES

#### Procedures:

A *procedure definition* is a declaration that associates an identifier with a statement. The identifier is the *procedure name*, and the statement is the *procedure body*.

For example, the following is the definition of procedure named *readarray* :

```
procedure readarray;  
  var i : integer;  
  
  begin  
    for i := 1 to 9 do read(a[i])  
  end;
```

When a procedure name appears within an executable statement, the procedure is said to be *called* at that point.

#### Activation trees:

An *activation tree* is used to depict the way control enters and leaves activations. In an activation tree,

1. Each node represents an activation of a procedure.
2. The root represents the activation of the main program.
3. The node for *a* is the parent of the node for *b* if and only if control flows from activation *a* to *b*.
4. The node for *a* is to the left of the node for *b* if and only if the lifetime of *a* occurs before the lifetime of *b*.

#### Control stack:

- A *control stack* is used to keep track of live procedure activations. The idea is to push the node for an activation onto the control stack as the activation begins and to pop the node when the activation ends.
- The contents of the control stack are related to paths to the root of the activation tree. When node *n* is at the top of control stack, the stack contains the nodes along the path from *n* to the root.

**The Scope of a Declaration:**

A declaration is a syntactic construct that associates information with a name.

Declarations may be explicit, such as:

```
var i : integer ;
```

or they may be implicit. Example, any variable name starting with I is assumed to denote an integer.

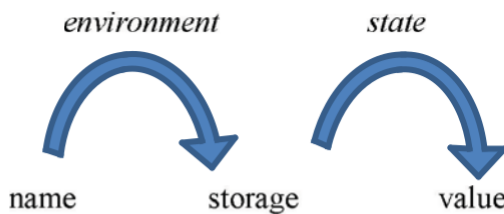
The portion of the program to which a declaration applies is called the *scope* of that declaration.

**Binding of names:**

Even if each name is declared once in a program, the same name may denote different data objects at run time. “Data object” corresponds to a storage location that holds values.

The term *environment* refers to a function that maps a name to a storage location.

The term *state* refers to a function that maps a storage location to the value held there.

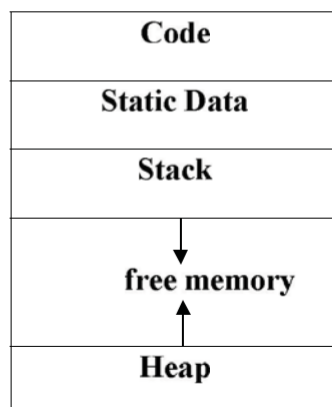


When an *environment* associates storage location *s* with a name *x*, we say that *x* is *bound* to *s*. This association is referred to as a *binding* of *x*.

**STORAGE ORGANISATION**

- The executing target program runs in its own logical address space in which each program value has a location.
- The management and organization of this logical address space is shared between the compiler, operating system and target machine. The operating system maps the logical address into physical addresses, which are usually spread throughout memory.

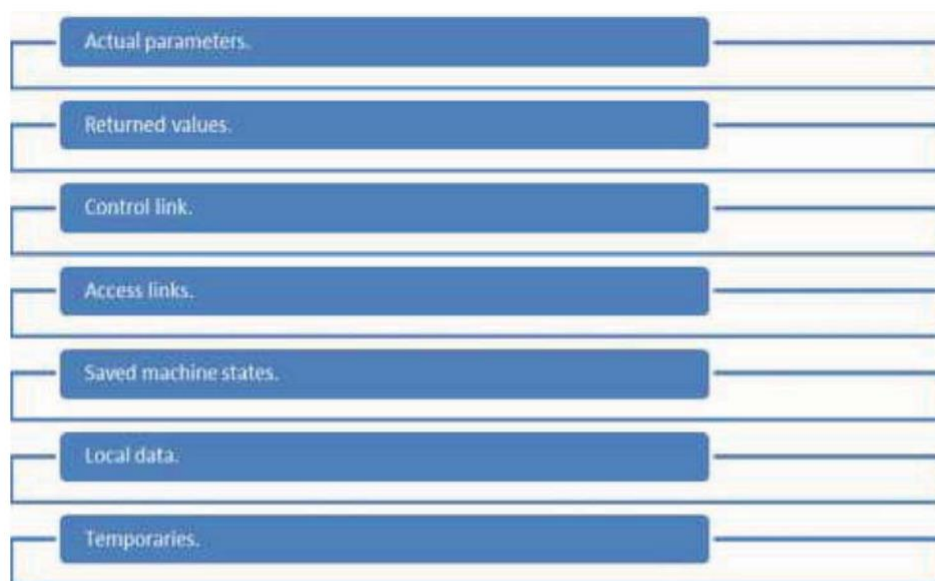
**Typical subdivision of run-time memory:**



- Run-time storage comes in blocks, where a byte is the smallest unit of addressable memory. Four bytes form a machine word. Multibyte objects are stored in consecutive bytes and given the address of first byte.
- The storage layout for data objects is strongly influenced by the addressing constraints of the target machine.
- A character array of length 10 needs only enough bytes to hold 10 characters, a compiler may allocate 12 bytes to get alignment, leaving 2 bytes unused.
- This unused space due to alignment considerations is referred to as padding.
- The size of some program objects may be known at run time and may be placed in an area called static.
- The dynamic areas used to maximize the utilization of space at run time are stack and heap.

### 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 on the control stack, with the root of the activation tree at the bottom, the latter activation has its record at the top of the stack.
- The contents of the activation record vary with the language being implemented. The diagram below shows the contents of activation record.



- Temporary values such as those arising from the evaluation of expressions.
- Local data belonging to the procedure whose activation record this is.
- A saved machine status, with information about the state of the machine just before the call to procedures.
- An access link may be needed to locate data needed by the called procedure but found elsewhere.
- A control link pointing to the activation record of the caller.

- Space for the return value of the called functions, if any. Again, not all called procedures return a value, and if one does, we may prefer to place that value in a register for efficiency.
- The actual parameters used by the calling procedure. These are not placed in activation record but rather in registers, when possible, for greater efficiency.

### **STORAGE ALLOCATION STRATEGIES**

The different storage allocation strategies are :

1. **Static allocation** – lays out storage for all data objects at compile time
2. **Stack allocation** – manages the run-time storage as a stack.
3. **Heap allocation** – allocates and deallocates storage as needed at run time from a data area known as heap.

### **STATIC ALLOCATION**

- In static allocation, names are bound to storage as the program is compiled, so there is no need for a run-time support package.
- Since the bindings do not change at run-time, everytime a procedure is activated, its names are bound to the same storage locations.
- Therefore values of local names are *retained* across activations of a procedure. That is, when control returns to a procedure the values of the locals are the same as they were when control left the last time.
- From the type of a name, the compiler decides the amount of storage for the name and decides where the activation records go. At compile time, we can fill in the addresses at which the target code can find the data it operates on.

### **STACK ALLOCATION OF SPACE**

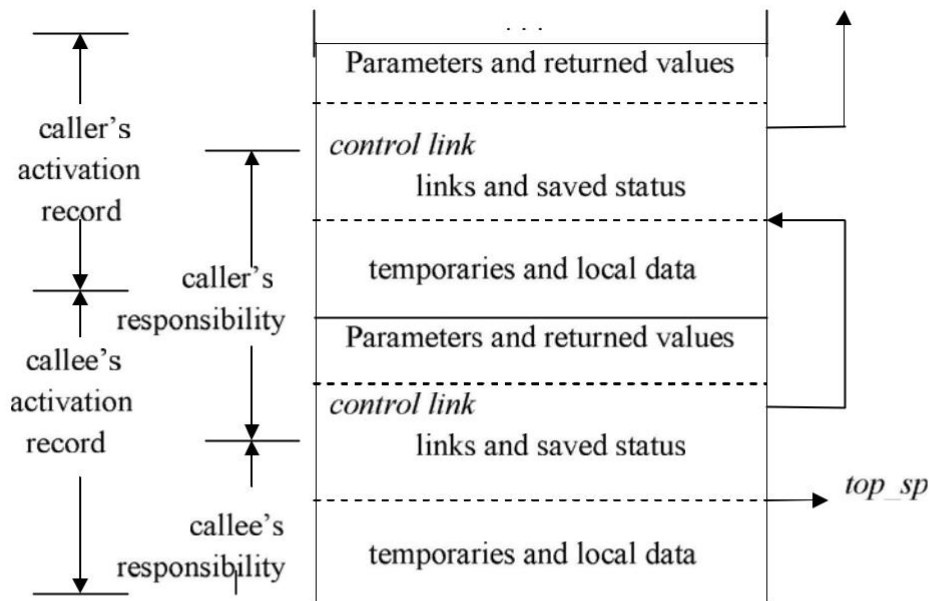
- All compilers for languages that use procedures, functions or methods as units of user-defined actions manage at least part of their run-time memory as a stack.
- Each time a procedure is called , space for its local variables is pushed onto a stack, and when the procedure terminates, that space is popped off the stack.

### **Calling sequences:**

- Procedures called are implemented in what is called as calling sequence, which consists of code that allocates an activation record on the stack and enters information into its fields.
- A return sequence is similar to code to restore the state of machine so the calling procedure can continue its execution after the call.
- The code in calling sequence is often divided between the calling procedure (caller) and the procedure it calls (callee).
- When designing calling sequences and the layout of activation records, the following principles are helpful:
  - Values communicated between caller and callee are generally placed at the beginning of the callee's activation record, so they are as close as possible to the caller's activation record.



- Fixed length items are generally placed in the middle. Such items typically include the control link, the access link, and the machine status fields.
- Items whose size may not be known early enough are placed at the end of the activation record. The most common example is dynamically sized array, where the value of one of the callee's parameters determines the length of the array.
- We must locate the top-of-stack pointer judiciously. A common approach is to have it point to the end of fixed-length fields in the activation record. Fixed-length data can then be accessed by fixed offsets, known to the intermediate-code generator, relative to the top-of-stack pointer.

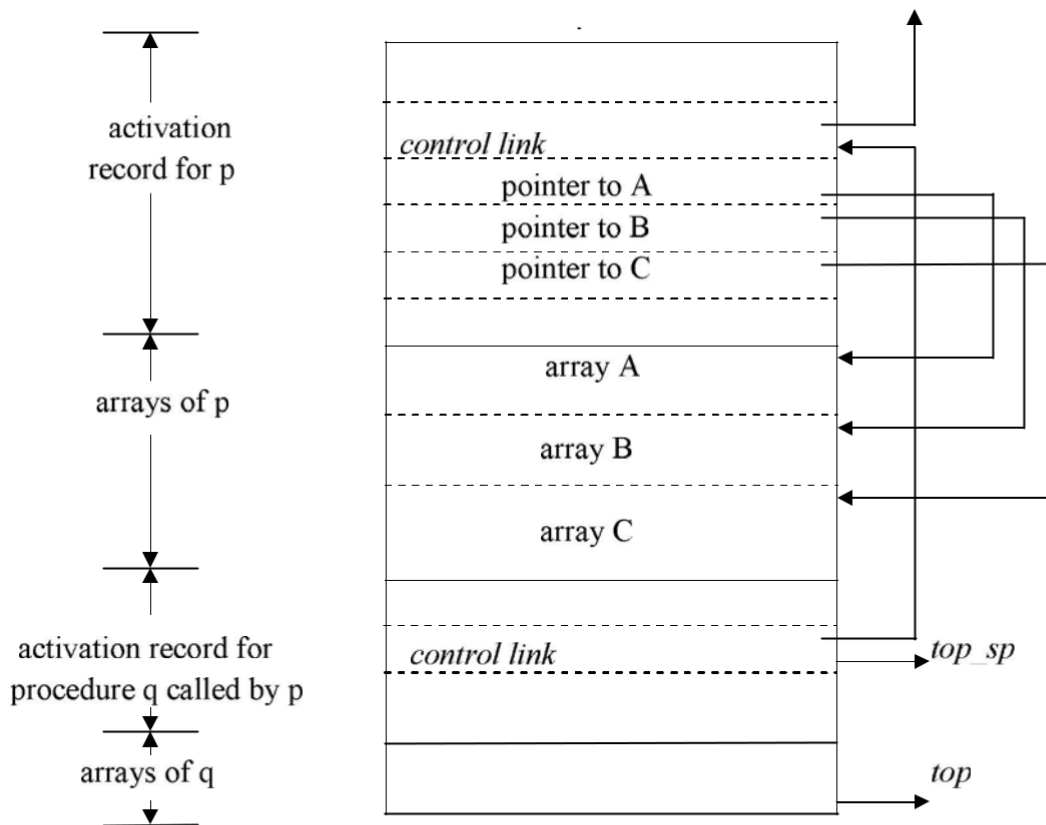


**Division of tasks between caller and callee**

- The calling sequence and its division between caller and callee are as follows.
  - The caller evaluates the actual parameters.
  - The caller stores a return address and the old value of *top\_sp* into the callee's activation record. The caller then increments the *top\_sp* to the respective positions.
  - The callee saves the register values and other status information.
  - The callee initializes its local data and begins execution.
- A suitable, corresponding return sequence is:
  - The callee places the return value next to the parameters.
  - Using the information in the machine-status field, the callee restores *top\_sp* and other registers, and then branches to the return address that the caller placed in the status field.
  - Although *top\_sp* has been decremented, the caller knows where the return value is, relative to the current value of *top\_sp*; the caller therefore may use that value.

**Variable length data on stack:**

- The run-time memory management system must deal frequently with the allocation of space for objects, the sizes of which are not known at the compile time, but which are local to a procedure and thus may be allocated on the stack.
- The reason to prefer placing objects on the stack is that we avoid the expense of garbage collecting their space.
- The same scheme works for objects of any type if they are local to the procedure called and have a size that depends on the parameters of the call.



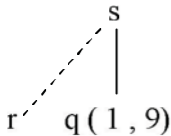
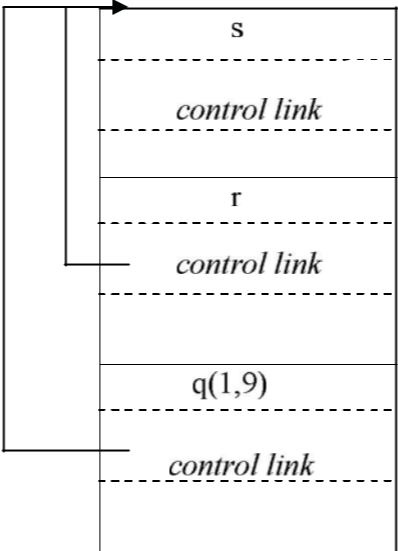
**Access to dynamically allocated arrays**

- Procedure p has three local arrays, whose sizes cannot be determined at compile time. The storage for these arrays is not part of the activation record for p.
- Access to the data is through two pointers, top and top-sp. Here the top marks the actual top of stack; it points the position at which the next activation record will begin.
- The second top-sp is used to find local, fixed-length fields of the top activation record.
- The code to reposition top and top-sp can be generated at compile time, in terms of sizes that will become known at run time.

**HEAP ALLOCATION**

Stack allocation strategy cannot be used if either of the following is possible :

1. The values of local names must be retained when an activation ends.
  2. A called activation outlives the caller.
- Heap allocation parcels out pieces of contiguous storage, as needed for activation records or other objects.
  - Pieces may be deallocated in any order, so over the time the heap will consist of alternate areas that are free and in use.

Position in the activation tree	Activation records in the heap	Remarks
		<p>Retained activation record for r</p>

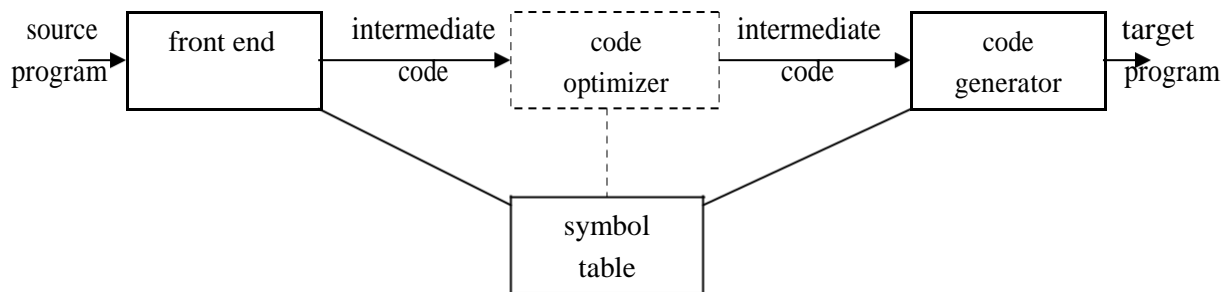
- The record for an activation of procedure r is retained when the activation ends.
- Therefore, the record for the new activation q(1 , 9) cannot follow that for s physically.
- If the retained activation record for r is deallocated, there will be free space in the heap between the activation records for s and q.



## 5.1 CODE GENERATION

The final phase in compiler model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program. The code generation techniques presented below can be used whether or not an optimizing phase occurs before code generation.

Position of code generator



### ISSUES IN THE DESIGN OF A CODE GENERATOR

The following issues arise during the code generation phase :

1. Input to code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation
6. Evaluation order

#### 1. Input to code generator:

The input to the code generation consists of the intermediate representation of the source program produced by front end , together with information in the symbol table to determine run-time addresses of the data objects denoted by the names in the intermediate representation.

Intermediate representation can be :

- a. Linear representation such as postfix notation
- b. Three address representation such as quadruples
- c. Virtual machine representation such as stack machine code
- d. Graphical representations such as syntax trees and dags.

Prior to code generation, the front end must be scanned, parsed and translated into intermediate representation along with necessary type checking. Therefore, input to code generation is assumed to be error-free.

#### 2. Target program:

The output of the code generator is the target program. The output may be :

- a. Absolute machine language
  - It can be placed in a fixed memory location and can be executed immediately.

## Compiler Design

- b. Relocatable machine language
  - It allows subprograms to be compiled separately.
- c. Assembly language
  - Code generation is made easier.

### 3. Memory management:

Names in the source program are mapped to addresses of data objects in run-time memory by the front end and code generator.

It makes use of symbol table, that is, a name in a three-address statement refers to a symbol-table entry for the name.

Labels in three-address statements have to be converted to addresses of instructions. For example,

$j$  : goto  $i$  generates jump instruction as follows :  
 if  $i < j$ , a backward jump instruction with target address equal to location of code for quadruple  $i$  is generated.  
 if  $i > j$ , the jump is forward. We must store on a list for quadruple  $i$  the location of the first machine instruction generated for quadruple  $j$ . When  $i$  is processed, the machine locations for all instructions that forward jumps to  $i$  are filled.

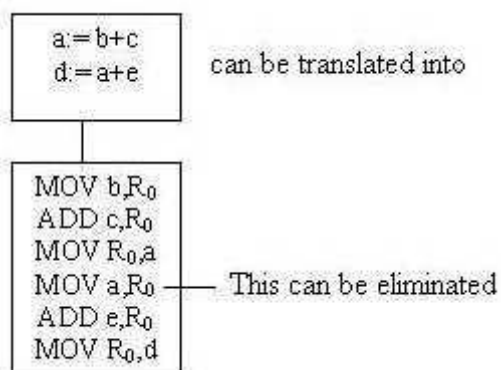
### 4. Instruction selection:

The instructions of target machine should be complete and uniform.

Instruction speeds and machine idioms are important factors when efficiency of target program is considered.

The quality of the generated code is determined by its speed and size.

The former statement can be translated into the latter statement as shown below:



### 5. Register allocation

Instructions involving register operands are shorter and faster than those involving operands in memory.

The use of registers is subdivided into two subproblems :

Register allocation – the set of variables that will reside in registers at a point in the program is selected.

Register assignment – the specific register that a variable will reside in is picked.

Certain machine requires even-odd register pairs for some operands and results.  
For example, consider the division instruction of the form :

D x, y

where, x – dividend even register in even/odd register pair

y – divisor

even register holds the remainder

odd register holds the quotient

## 6. Evaluation order

The order in which the computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others.

## TARGET MACHINE

Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator.

The target computer is a byte-addressable machine with 4 bytes to a word. It has n general-purpose registers,  $R_0, R_1, \dots, R_{n-1}$ .

It has two-address instructions of the form:

op source, destination

where, op is an op-code, and source and destination are data fields.

It has the following op-codes :

MOV (move source to destination)

ADD (add source to destination)

SUB (subtract source from destination)

The source and destination of an instruction are specified by combining registers and memory locations with address modes.

Address modes with their assembly-language forms

MODE	FORM	ADDRESS	ADDED COST
absolute	M	M	1
register	R	R	0
indexed	c(R)	c+contents(R)	1
indirect register	*R	contents (R)	0
indirect indexed	*c(R)	contents(c+ contents(R))	1
literal	#c	c	1

## Compiler Design

For example : MOV R0, M stores contents of Register R0 into memory location M ; MOV 4(R0), M stores the value contents(4+contents(R0)) into M.

### Instruction costs :

Instruction cost = 1+cost for source and destination address modes. This cost corresponds to the length of the instruction.

Address modes involving registers have cost zero.

Address modes involving memory location or literal have cost one.

Instruction length should be minimized if space is important. Doing so also minimizes the time taken to fetch and perform the instruction.

For example : MOV R0, R1 copies the contents of register R0 into R1. It has cost one, since it occupies only one word of memory.

The three-address statement  $a := b + c$  can be implemented by many different instruction sequences :

i) MOV b, R0

ADD c, R0                      cost = 6

MOV R0, a

ii) MOV b, a

ADD c, a                      cost = 6

iii) Assuming R0, R1 and R2 contain the addresses of a, b, and c :

MOV \*R1, \*R0

ADD \*R2, \*R0                cost = 2

In order to generate good code for target machine, we must utilize its addressing capabilities efficiently.

## RUN-TIME STORAGE MANAGEMENT

Information needed during an execution of a procedure is kept in a block of storage called an activation record, which includes storage for names local to the procedure.

The two standard storage allocation strategies are:

1. Static allocation
2. Stack allocation

In static allocation, the position of an activation record in memory is fixed at compile time.

In stack allocation, a new activation record is pushed onto the stack for each execution of a procedure. The record is popped when the activation ends.

The following three-address statements are associated with the run-time allocation and deallocation of activation records:

1. Call,
2. Return,
3. Halt, and
4. Action, a placeholder for other statements.

We assume that the run-time memory is divided into areas for:

1. Code
2. Static data
3. Stack



## Compiler Design

### Static allocation

Implementation of call statement:

The codes needed to implement static allocation are as follows:

```
MOV #here + 20, callee.static_area      /*It saves return address*/
GOTO callee.code_area                  /*It transfers control to the target code for the called procedure */
```

where,

callee.static\_area – Address of the activation record

callee.code\_area – Address of the first instruction for called procedure

#here + 20 – Literal return address which is the address of the instruction following GOTO.

Implementation of return statement:

A return from procedure callee is implemented by :

```
GOTO *callee.static_area
```

This transfers control to the address saved at the beginning of the activation record.

Implementation of action statement:

The instruction ACTION is used to implement action statement.

Implementation of halt statement:

The statement HALT is the final instruction that returns control to the operating system. Stack allocation

Static allocation can become stack allocation by using relative addresses for storage in activation records. In stack allocation, the position of activation record is stored in register so words in activation records can be accessed as offsets from the value in this register.

The codes needed to implement stack allocation are as follows:

Initialization of stack:

```
MOV #stackstart , SP                /* initializes stack */
```

Code for the first procedure

```
HALT                                /* terminate execution */
```

Implementation of Call statement:

```
ADD #caller.recordsize, SP          /* increment stack pointer */
```

```
MOV #here + 16, *SP                /*Save return address */
```

```
GOTO callee.code_area
```

Compiler Design

where,

caller.recordsize – size of the activation record

#here + 16 – address of the instruction following the GOTO

Implementation of Return statement:

```
GOTO *0 ( SP )      /*return to the caller */
```

```
SUB #caller.recordsize, SP  /* decrement SP and restore to previous value */
```

## BASIC BLOCKS AND FLOW GRAPHS

### Basic Blocks

A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without any halt or possibility of branching except at the end.

The following sequence of three-address statements forms a basic

block:  $t_1 := a * a$

$t_2 := a * b$

$t_3 := 2 * t_2$

$t_4 := t_1 + t_3$

$t_5 := b * b$

$t_6 := t_4 + t_5$

Basic Block Construction:

Algorithm: Partition into basic blocks

Input: A sequence of three-address statements

Output: A list of basic blocks with each three-address statement in exactly one block

Method:

1. We first determine the set of leaders, the first statements of basic blocks. The rules we use are of the following:
  - a. The first statement is a leader.
  - b. Any statement that is the target of a conditional or unconditional goto is a leader.
  - c. Any statement that immediately follows a goto or conditional goto statement is a leader.
2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

Consider the following source code for dot product of two vectors a and b of length 20

```

begin
    prod :=0;
    i:=1;
    do begin
        prod :=prod+ a[i] * b[i];
        i :=i+1;
    end
    while i <= 20
end

```

The three-address code for the above source program is given as :

```

(1)   prod := 0
(2)   i := 1
(3)   t1 := 4* i
(4)   t2 := a[t1]    /*compute a[i] */
(5)   t3 := 4* i
(6)   t4 := b[t3]    /*compute b[i] */
(7)   t5 := t2*t4
(8)   t6 := prod+t5
(9)   prod := t6
(10)  t7 := i+1
(11)  i := t7
(12)  if i<=20 goto (3)

```

Basic block 1: Statement (1) to (2)

Basic block 2: Statement (3) to (12)

## Transformations on Basic Blocks:

A number of transformations can be applied to a basic block without changing the set of expressions computed by the block. Two important classes of transformation are :

Structure-preserving transformations

Algebraic transformations

## 1. Structure preserving transformations:

## a) Common subexpression elimination:

$a := b + c$	$a := b + c$
$b := a - d$	$b := a - d$
$c := b + c$	$c := b + c$
$d := a - d$	$d := b$

Since the second and fourth expressions compute the same expression, the basic block can be transformed as above.

## b) Dead-code elimination:

Suppose  $x$  is dead, that is, never subsequently used, at the point where the statement  $x := y + z$  appears in a basic block. Then this statement may be safely removed without changing the value of the basic block.

## c) Renaming temporary variables:

A statement  $t := b + c$  ( $t$  is a temporary) can be changed to  $u := b + c$  ( $u$  is a new temporary) and all uses of this instance of  $t$  can be changed to  $u$  without changing the value of the basic block.

Such a block is called a normal-form block.

## d) Interchange of statements:

Suppose a block has the following two adjacent statements:

$t1 := b + c$
$t2 := x + y$

We can interchange the two statements without affecting the value of the block if and only if neither  $x$  nor  $y$  is  $t1$  and neither  $b$  nor  $c$  is  $t2$ .

## 2. Algebraic transformations:

Algebraic transformations can be used to change the set of expressions computed by a basic block into an algebraically equivalent set.

Examples:

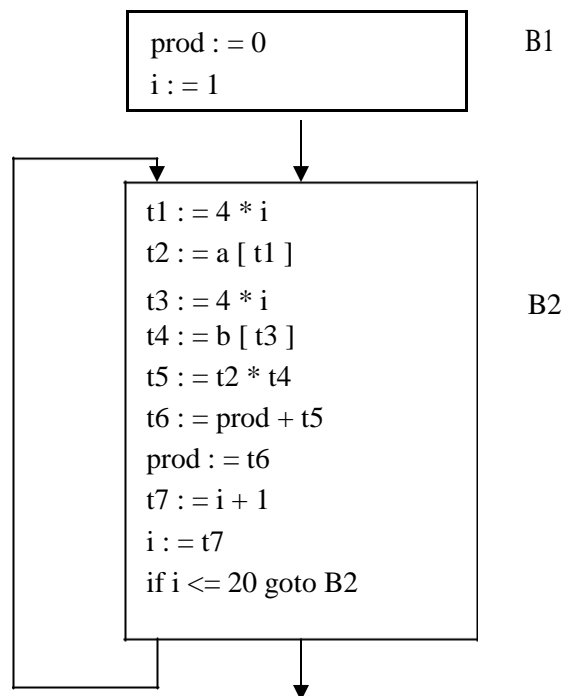
- i)  $x := x + 0$  or  $x := x * 1$  can be eliminated from a basic block without changing the set of expressions it computes.
- ii) The exponential statement  $x := y * * 2$  can be replaced by  $x := y * y$ .

Flow Graphs

Flow graph is a directed graph containing the flow-of-control information for the set of basic blocks making up a program.

The nodes of the flow graph are basic blocks. It has a distinguished initial node.

E.g.: Flow graph for the vector dot product is given as follows:



B<sub>1</sub> is the initial node. B<sub>2</sub> immediately follows B<sub>1</sub>, so there is an edge from B<sub>1</sub> to B<sub>2</sub>. The target of jump from last statement of B<sub>1</sub> is the first statement B<sub>2</sub>, so there is an edge from B<sub>1</sub> (last statement) to B<sub>2</sub> (first statement).

B<sub>1</sub> is the predecessor of B<sub>2</sub>, and B<sub>2</sub> is a successor of B<sub>1</sub>.

Loops

A loop is a collection of nodes in a flow graph such that

1. All nodes in the collection are strongly connected.
2. The collection of nodes has a unique entry.

A loop that contains no other loops is called an inner loop.

## NEXT-USE INFORMATION

If the name in a register is no longer needed, then we remove the name from the register and the register can be used to store some other names.

**Input:** Basic block B of three-address statements

**Output:** At each statement  $i: x = y \text{ op } z$ , we attach to  $i$  the liveness and next-uses of  $x$ ,  $y$  and  $z$ .

**Method:** We start at the last statement of B and scan backwards.

1. Attach to statement  $i$  the information currently found in the symbol table regarding the next-use and liveness of  $x$ ,  $y$  and  $z$ .
2. In the symbol table, set  $x$  to “not live” and “no next use”.
3. In the symbol table, set  $y$  and  $z$  to “live”, and next-uses of  $y$  and  $z$  to  $i$ .

Symbol Table:

Names	Liveness	Next-use
x	not live	no next-use
y	Live	i
z	Live	i

### A SIMPLE CODE GENERATOR

A code generator generates target code for a sequence of three- address statements and effectively uses registers to store operands of the statements.

For example: consider the three-address statement  $a := b+c$   
 It can have the following sequence of codes:

```

ADD Rj, Ri           Cost = 1    // if Ri contains b and Rj contains c
                        (or)
ADD c, Ri           Cost = 2    // if c is in a memory location
                        (or)
MOV c, Rj           Cost = 3    // move c from memory to Rj and add
ADD Rj, Ri
    
```

Register and Address Descriptors:

A register descriptor is used to keep track of what is currently in each registers. The register descriptors show that initially all the registers are empty.

An address descriptor stores the location where the current value of the name can be found at run time.

## Compiler Design

A code-generation algorithm:

The algorithm takes as input a sequence of three-address statements constituting a basic block. For each three-address statement of the form  $x := y \text{ op } z$ , perform the following actions:

1. Invoke a function `getreg` to determine the location  $L$  where the result of the computation  $y \text{ op } z$  should be stored.
2. Consult the address descriptor for  $y$  to determine  $y'$ , the current location of  $y$ . Prefer the register for  $y'$  if the value of  $y$  is currently both in memory and a register. If the value of  $y$  is not already in  $L$ , generate the instruction `MOV  $y'$ , L` to place a copy of  $y$  in  $L$ .
3. Generate the instruction `OP  $z'$ , L` where  $z'$  is a current location of  $z$ . Prefer a register to a memory location if  $z$  is in both. Update the address descriptor of  $x$  to indicate that  $x$  is in location  $L$ . If  $x$  is in  $L$ , update its descriptor and remove  $x$  from all other descriptors.
4. If the current values of  $y$  or  $z$  have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of  $x := y \text{ op } z$ , those registers will no longer contain  $y$  or  $z$ .

Generating Code for Assignment Statements:

The assignment  $d := (a-b) + (a-c) + (a-c)$  might be translated into the following three-address code sequence:

```
t := a - b
u := a - c
v := t + u
d := v + u
```

with  $d$  live at the end.

Code sequence for the example is:

Statements	Code Generated	Register descriptor	Address descriptor
		Register empty	
$t := a - b$	MOV a, R0 SUB b, R0	R0 contains t	t in R0
$u := a - c$	MOV a, R1 SUB c, R1	R0 contains t R1 contains u	t in R0 u in R1
$v := t + u$	ADD R1, R0	R0 contains v R1 contains u	u in R1 v in R0
$d := v + u$	ADD R1, R0 MOV R0, d	R0 contains d	d in R0 d in R0 and memory

## Compiler Design

### Generating Code for Indexed Assignments

The table shows the code sequences generated for the indexed assignment statements

$a := b[i]$  and  $a[i] := b$

Statements	Code Generated	Cost
$a := b[i]$	MOV b(R <sub>i</sub> ), R	2
$a[i] := b$	MOV b, a(R <sub>i</sub> )	3

### Generating Code for Pointer Assignments

The table shows the code sequences generated for the pointer assignments

$a := *p$  and  $*p := a$

Statements	Code Generated	Cost
$a := *p$	MOV *R <sub>p</sub> , a	2
$*p := a$	MOV a, *R <sub>p</sub>	2

### Generating Code for Conditional Statements

Statement	Code
if $x < y$ goto z	CMP x, y CJ< z      /* jump to z if condition code is negative */
$x := y + z$ if $x < 0$ goto z	MOV y, R <sub>0</sub> ADD z, R <sub>0</sub> MOV R <sub>0</sub> , x CJ< z

## THE DAG REPRESENTATION FOR BASIC BLOCKS

A DAG for a basic block is a directed acyclic graph with the following labels on nodes:

1. Leaves are labeled by unique identifiers, either variable names or constants.
2. Interior nodes are labeled by an operator symbol.
3. Nodes are also optionally given a sequence of identifiers for labels to store the computed values.

DAGs are useful data structures for implementing transformations on basic blocks.

It gives a picture of how the value computed by a statement is used in subsequent statements.

It provides a good way of determining common sub - expressions.



## Algorithm for construction of DAG

Input: A basic block

Output: A DAG for the basic block containing the following information:

1. A label for each node. For leaves, the label is an identifier. For interior nodes, an operator symbol.
2. For each node a list of attached identifiers to hold the computed values.

Case (i)  $x := y \text{ OP } z$

Case (ii)  $x := \text{OP } y$

Case (iii)  $x := y$

Method:

Step 1: If  $y$  is undefined then create node( $y$ ).

If  $z$  is undefined, create node( $z$ ) for case(i).

Step 2: For the case(i), create a node(OP) whose left child is node( $y$ ) and right child is

node( $z$ ). ( Checking for common sub expression). Let  $n$  be this node.

For case(ii), determine whether there is node(OP) with one child node( $y$ ). If not create such a node.

For case(iii), node  $n$  will be node( $y$ ).

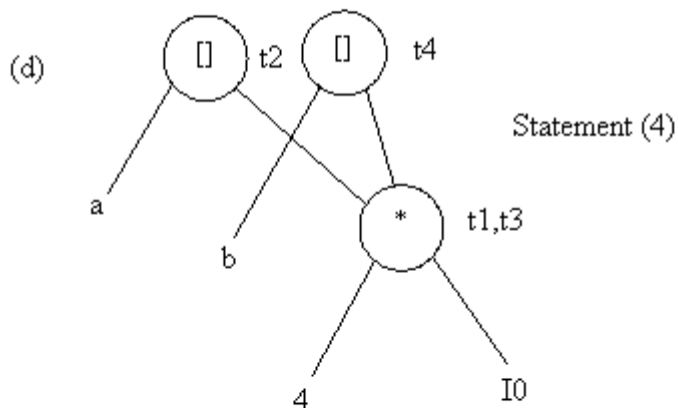
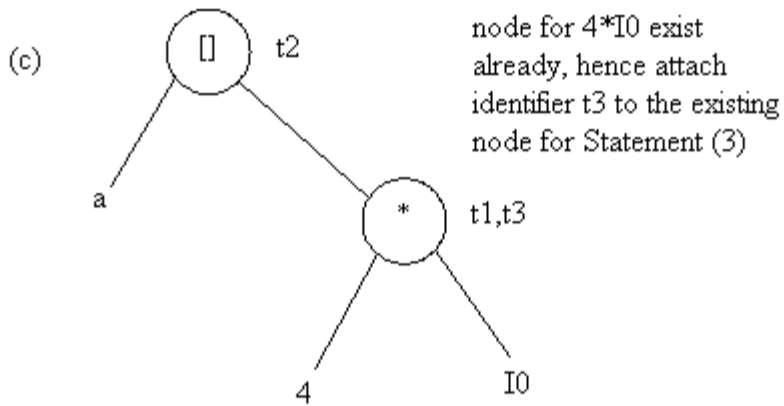
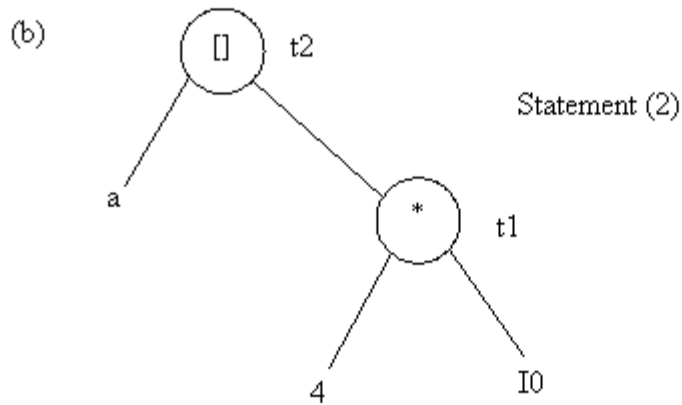
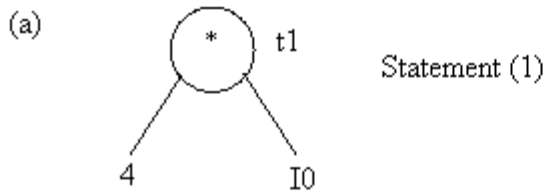
Step 3: Delete  $x$  from the list of identifiers for node( $x$ ). Append  $x$  to the list of attached

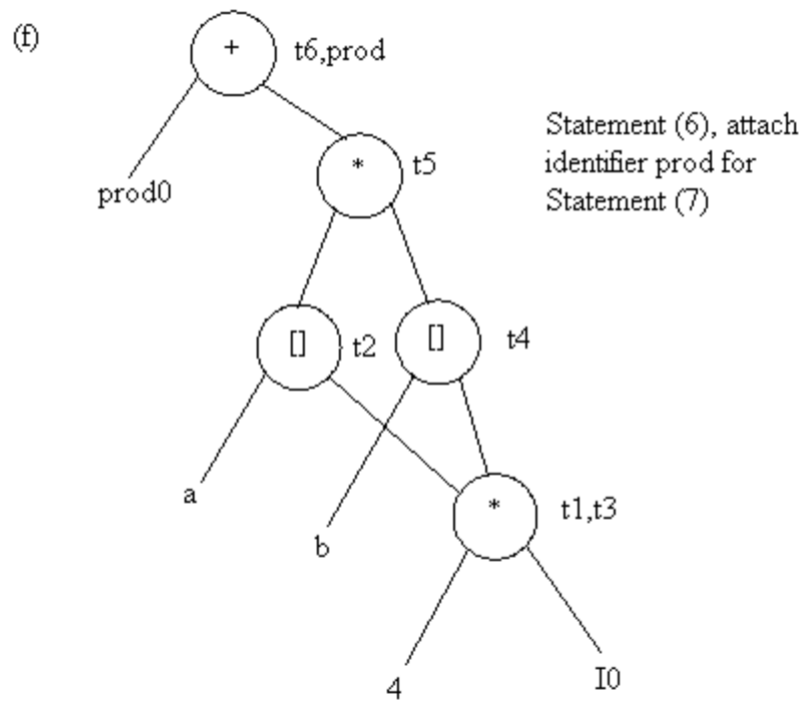
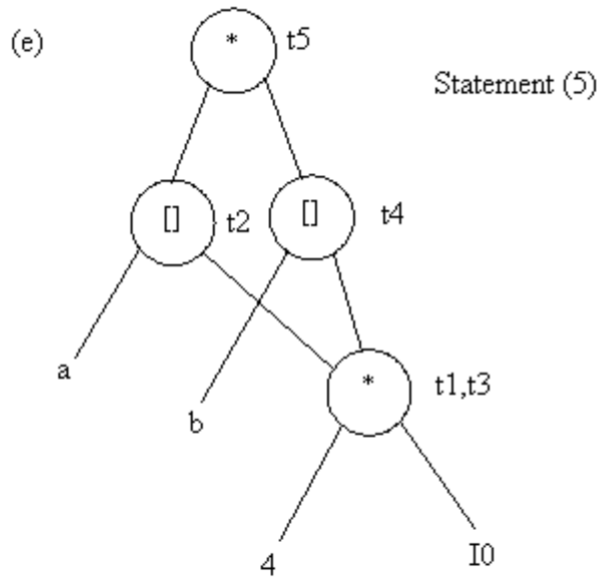
identifiers for the node  $n$  found in step 2 and set node( $x$ ) to  $n$ .

Example: Consider the block of three- address statements:

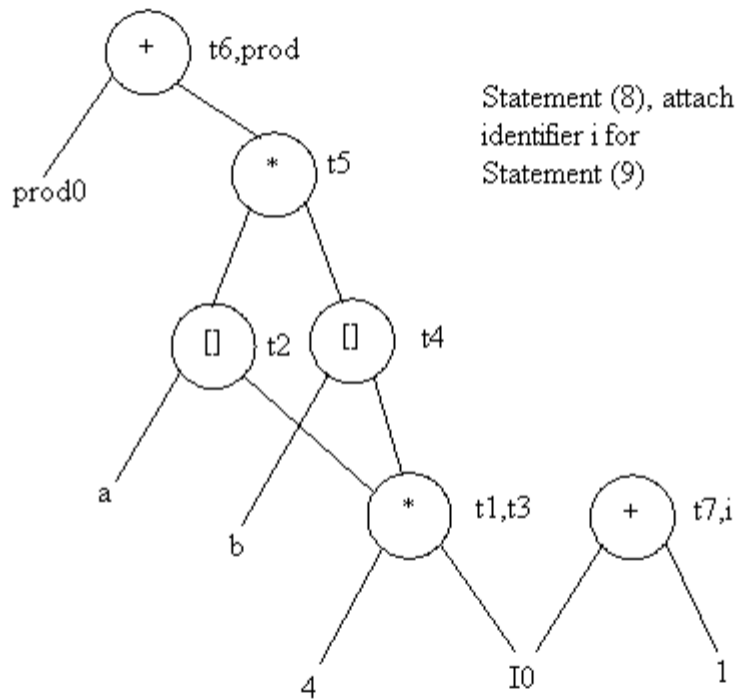
1.  $t_1 := 4 * i$
2.  $t_2 := a[t_1]$
3.  $t_3 := 4 * i$
4.  $t_4 := b[t_3]$
5.  $t_5 := t_2 * t_4$
6.  $t_6 := \text{prod} + t_5$
7.  $\text{prod} := t_6$
8.  $t_7 := i + 1$
9.  $i := t_7$
10. if  $i \leq 20$  goto (1)

Stages in DAG Construction

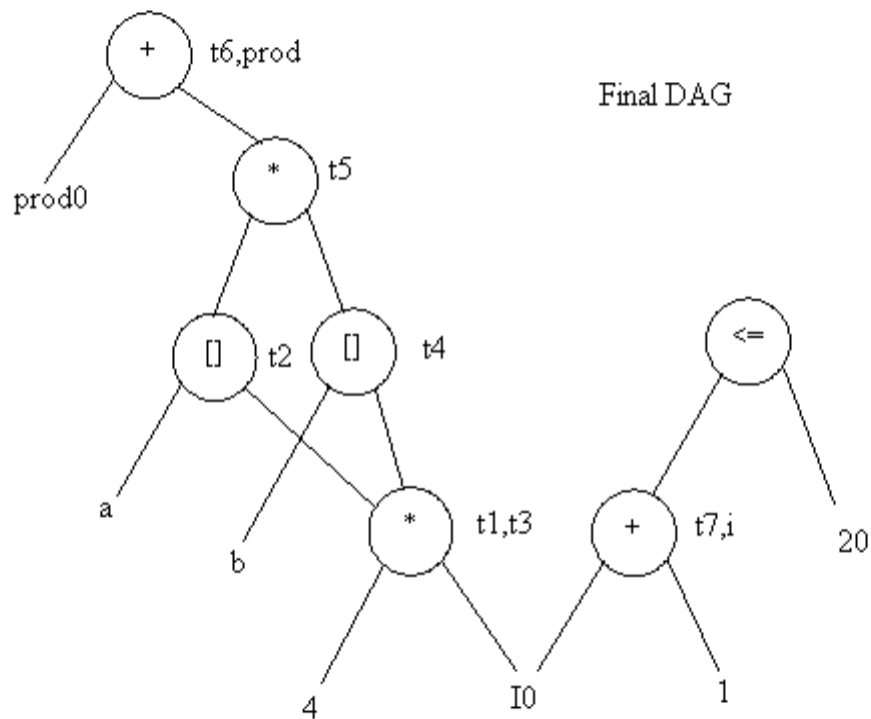




(g)



(h)



Application of DAGs:

1. We can automatically detect common sub expressions.
2. We can determine which identifiers have their values used in the block.
3. We can determine which statements compute values that could be used outside the block.

The advantage of generating code for a basic block from its dag representation is that, from a dag we can easily see how to rearrange the order of the final computation sequence than we can starting from a linear sequence of three-address statements or quadruples.

### Rearranging the order

The order in which computations are done can affect the cost of resulting object code.

For example, consider the following basic block:

```
t1 : = a + b
t2 : = c + d
t3 : = e - t2
t4 : = t1 - t3
```

Generated code sequence for basic block:

```
MOV a , R0
ADD b , R0
MOV c , R1
ADD d , R1
MOV R0 , t1
MOV e , R0
SUB R1 , R0
MOV t1 , R1
SUB R0 , R1
MOV R1 , t4
```

Rearranged basic block:

Now t1 occurs immediately before t4.

```
t2 : = c + d
t3 : = e - t2
t1 : = a + b
t4 : = t1 - t3
```

Revised code sequence:

```
MOV c , R0
ADD d , R0
MOV a , R0
SUB R0 , R1
MOV a , R0
ADD b , R0
SUB R1 , R0
MOV R0 , t4
```

In this order, two instructions MOV R0 , t1 and MOV t1 , R1 have been saved.

A Heuristic ordering for Dags

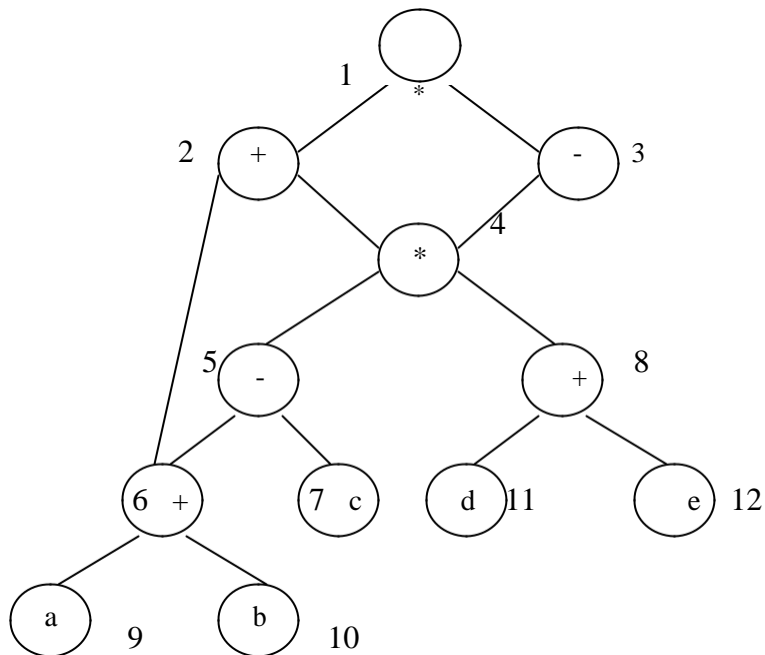
The heuristic ordering algorithm attempts to make the evaluation of a node immediately follow the evaluation of its leftmost argument.

The algorithm shown below produces the ordering in reverse.

Algorithm:

- 1) while unlisted interior nodes remain do begin
- 2)     select an unlisted node n, all of whose parents have been listed;
- 3)     list n;
- 4)     while the leftmost child m of n has no unlisted parents and is not a leaf do
  - 5)         list m;
  - 6)         n := m
- end
- end

Example: Consider the DAG shown below:



Initially, the only node with no unlisted parents is 1 so set  $n=1$  at line (2) and list 1 at line (3).

Now, the left argument of 1, which is 2, has its parents listed, so we list 2 and set  $n=2$  at line (6).

Now, at line (4) we find the leftmost child of 2, which is 6, has an unlisted parent 5. Thus we select a new  $n$  at line (2), and node 3 is the only candidate. We list 3 and proceed down its left chain, listing 4, 5 and 6. This leaves only 8 among the interior nodes so we list that.

The resulting list is 1234568 and the order of evaluation is 8654321.

## Compiler Design

Code sequence:

 $t_8 := d + e$  $t_6 := a + b$  $t_5 := t_6 - c$  $t_4 := t_5 * t_8$  $t_3 := t_4 - e$  $t_2 := t_6 + t_4$  $t_1 := t_2 * t_3$ 

This will yield an optimal code for the DAG on machine whatever be the number of registers.

## 5.2 CODE OPTIMIZATION

### 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.

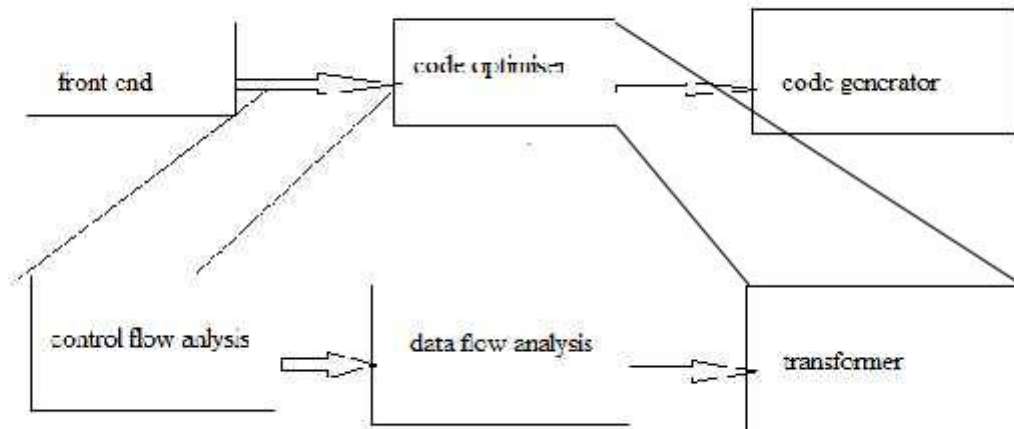
The transformation must preserve the meaning of programs. That is, the optimization must not change the output produced by a program for a given input, or cause an error such as division by zero, that was not present in the original source program. At all times we take the “safe” approach of missing 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.



## Organization for an Optimizing 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

control flow  
graph Call graph

Data flow analysis (DFA) is the process of ascertaining and collecting information prior to program execution about the possible modification, preservation, and use of certain entities (such as values or attributes of variables) in a computer program.

## PRINCIPAL SOURCES OF OPTIMISATION

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.

## Compiler Design

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
t6: = b [t4] +t5
```

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  $t_1$ . 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

## Compiler Design

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

## Compiler Design

```

t= limit-2;
while (i<=t) /* statement does not change limit or t */

```

## Induction Variables :

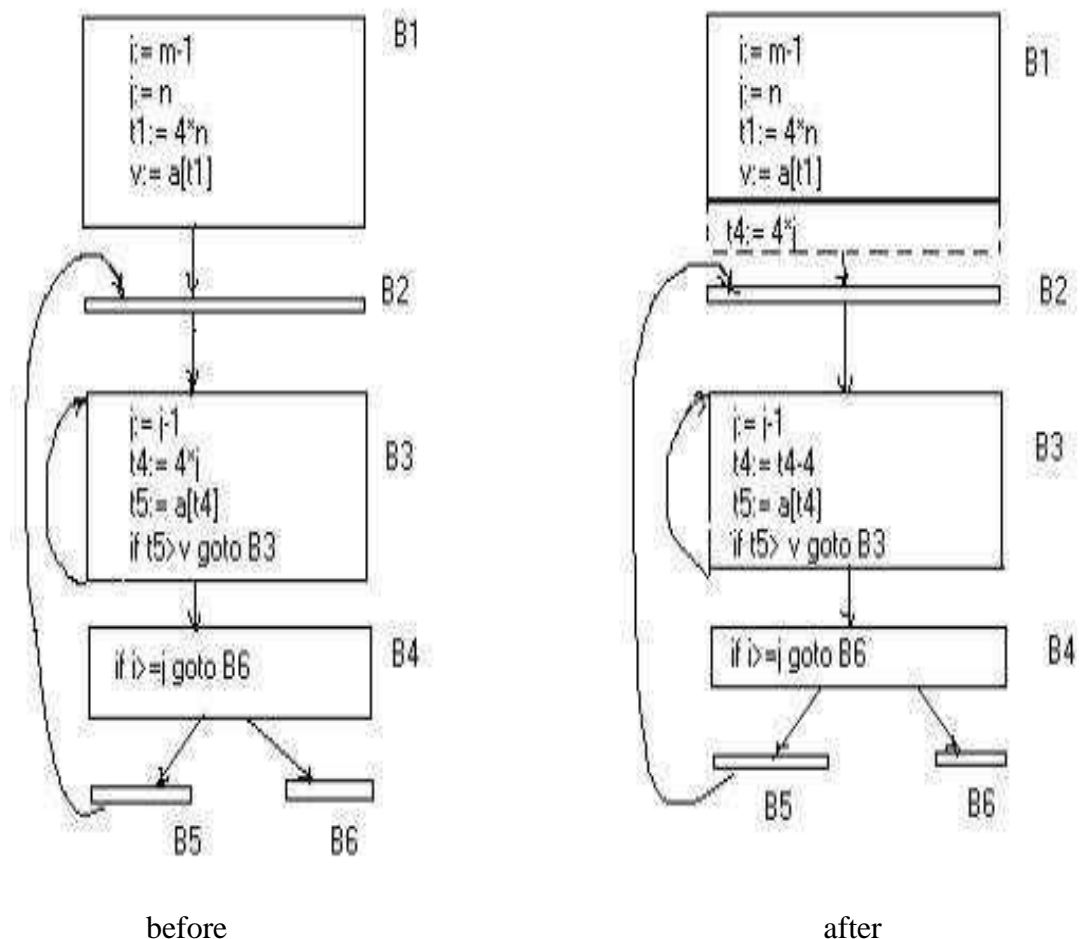
Loops are usually processed inside out. For example consider the loop around B3.

Note that the values of  $j$  and  $t_4$  remain in lock-step; every time the value of  $j$  decreases by 1, that of  $t_4$  decreases by 4 because  $4*j$  is assigned to  $t_4$ . 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  $t_4$  completely;  $t_4$  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  $t_4 := 4*j$  surely holds after such an assignment to  $t_4$  in Fig. and  $t_4$  is not changed elsewhere in the inner loop around B3, it follows that just after the statement  $j := j - 1$  the relationship  $t_4 := 4*j - 4$  must hold. We may therefore replace the assignment  $t_4 := 4*j$  by  $t_4 := t_4 - 4$ . The only problem is that  $t_4$  does not have a value when we enter block B3 for the first time. Since we must maintain the relationship  $t_4 = 4*j$  on entry to the block B3, we place an initialization of  $t_4$  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.

## 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 2<sup>nd</sup> and 4<sup>th</sup> 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
```

## Compiler Design

### 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

$t_1:=b+c$

$t_2:=x+y$

can be interchanged or reordered in its computation in the basic block when value of  $t_1$  does not affect the value of  $t_2$ .

### 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  $<=$ ,  $>=$ ,  $<$ ,  $>$ ,  $+$  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  $x:=y*y$

## Compiler Design

The compiler writer should examine the language carefully to determine what rearrangements of computations are permitted, since computer arithmetic does not 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  $(a+b)-c$ .

## 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,node1 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 goin 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.



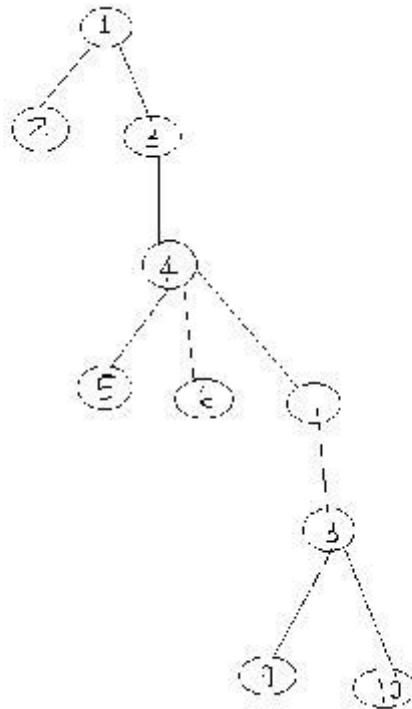
## Compiler Design

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 descendents 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  $d \text{ dom } m$ .



$$D(1) = \{1\}$$

$$D(2) = \{1, 2\}$$

$$D(3) = \{1, 3\}$$

$$D(4) = \{1, 3, 4\}$$

$$D(5) = \{1, 3, 4, 5\}$$

$$D(6) = \{1, 3, 4, 6\}$$

$$D(7) = \{1, 3, 4, 7\}$$

$$D(8) = \{1, 3, 4, 7, 8\}$$

$$D(9) = \{1, 3, 4, 7, 8, 9\}$$

$$D(10) = \{1, 3, 4, 7, 8, 10\}$$



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,

$7 \rightarrow 4$       $4 \text{DOM} 7$

$10 \rightarrow 7$       $7 \text{DOM} 10$

$4 \rightarrow 3$

$8 \rightarrow 3$

$9 \rightarrow 1$

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 U {m};
    push m onto stack
end;
```

stack := empty;

## Compiler Design

```

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 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 disjoint 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.

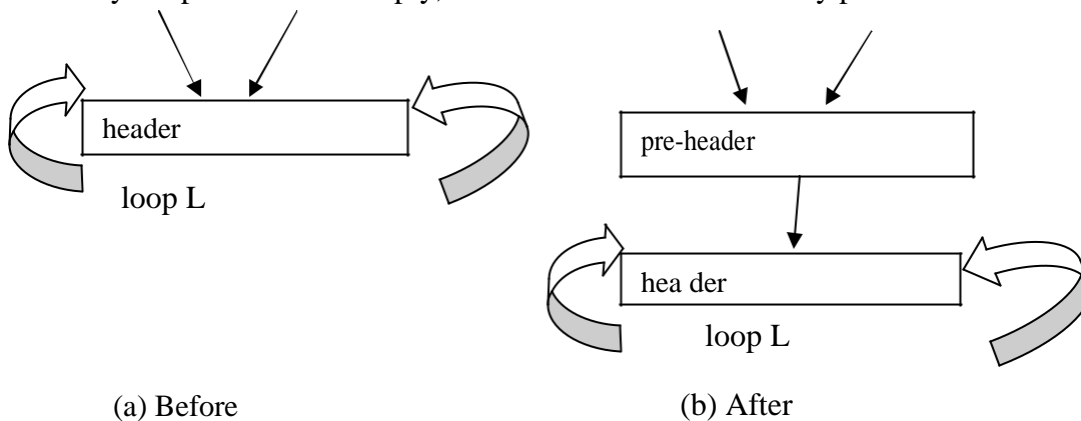
## Pre-Headers:

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 form 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.

**VINAYAKA MISSIONS RESEARCH FOUNDATION, SALEM.  
SCHOOL OF ARTS & SCIENCE - AVIT CAMPUS, CHENNAI.**

**DEPARTMENT OF COMPUTER SCIENCE  
QUESTION BANK**

**BOARD : COMPUTER SCIENCE**  
**PROGRAM : M.Sc COMPUTER SCIENCE - BATCH (2018-2020)**  
**REGULATION : 2017**  
**YEAR/ SEMESTER : I YEAR / II SEM**  
**COURSE TITLE : COMPILER DESIGN**

**UNIT – I**

**PART - A (6 MARKS)**

- 1) What is a compiler? Give a short note on phases of compiler.
- 2) Explain the variety of Intermediate forms.
- 3) Give the classification of a compiler
- 4) Describe the properties of parse trees.
- 5) Give short note on deterministic and non-deterministic automata.
- 6) Differentiate between tokens and patterns.
- 7) Describe about the structure of a compiler.
- 8) Give short note on lexical analyser
- 9) Differentiate between compiler and interpreter.
- 10) Give a short note on regular expression.

**PART - B (10 MARKS)**

- 1) Discuss briefly about structure of a compiler.
- 2) Discuss about the implementation of lexical analyser.
- 3) Explain about regular expression and its property.
- 4) Construct and optimize the REGEX of  $(a/b)^* abb$
- 5) Explain about DFA and NFA
- 6) Prepare the NFA state transition table in the given automata diagrammatic representation.

**UNIT – II**

**PART - A (6 MARKS)**

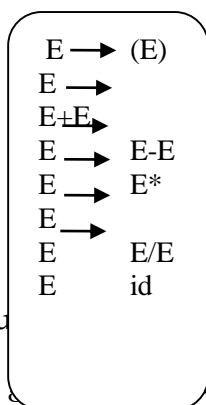
- 1) Write about syntax analyser and its significance.

## Compiler Design

- 2) Write down the procedure of bottom up parsing and explain it.
- 3) Describe the Top down parsing techniques with give an example.
- 4) What is parsing? Explain its concepts.
- 5) Define a context free grammar.
- 6) List the merits and demerits of operator precedence parsing
- 7) What do you mean by handle pruning?
- 8) Write the algorithm for FIRST and FOLLOW.
- 9) Write a short note on YACC and Explain it briefly.
- 10) Mention the types of LR parser and explain it.

**PART - B (10 MARKS)**

- 1) List the properties of LR parser and explain the types of LR parser
- 2) Explain about syntax analyser and its techniques.
- 3) Explain runtime environment with suitable example.
- 4) Optimize the following grammar with bottom up parsing techniques.



- 5) Describe the various optimization strategies in detail.
- 6) Why LR parsing is active?

**UNIT - III****PART - A (6 MARKS)**

- 1) What is intermediate code generation and explain its benefits.
- 2) What are the different types of three address statements?
- 3) Describe about syntax tree.
- 4) Write about parse tree with give an example.
- 5) Give the syntax - directed definition for if else statement.
- 6) Write the code generation algorithm.
- 7) What is meant by syntax directed information? Explain it.

- 8) What are the three functions used for back patching?
- 9) Define Quadruple and explain its merits.
- 10) What is type checking? explain it.

**PART - B (10 MARKS)**

- 1) Explain type checking of expressions and statements.
- 2) How could you generate the intermediate code for the flow of control statements?
- 3) Explain syntax tree functions with give an example.
- 4) Give the semantic rules for declarations in a procedure.
- 5) Explain in detail about the recursive evaluators.
- 6) Explain about the parse tree with give an example.

**UNIT - IV****PART - A (6 MARKS)**

- 1) Describe the state allocation of space.
- 2) Explain the variable length data on stack
- 3) List the various storage allocation and define them.
- 4) What is dynamic storage allocation.
- 5) Define back patching with different functions.
- 6) Define Boolean expression and short circuit code
- 7) Define data structures used for symbol table.
- 8) What are the intermediate languages?

**PART - B (10 MARKS)**

- 1) What are the different parameter passing methods in a procedure call?
- 2) Explain in details about the static allocation.
- 3) Explain the common sub expression eliminations.
- 4) Describe the procedure calling methods with give an example.
- 5) Clearly mention the heap allocation methods.
- 6) Explain about the generating code for assignment statements.

**UNIT - V****PART - A (6 MARKS)**

- 1) Compare the code generation and optimization

## Compiler Design

- 2) Write about the generating code for indexed assignments.
- 3) What is basic block?
- 4) What are the methods available in loop optimization?
- 5) Write down the characteristics of peephole optimization.
- 6) Describe the runtime storage management.
- 7) What is a DAG?
- 8) List out the optimization techniques.
- 9) Distinguish between basic blocks and flow graphs.

**PART - B (10 MARKS)**

- 1) Describe the peephole optimization techniques with give an example.
- 2) Explain DAG representation of the basic block with an example.
- 3) Discuss run time storage management of code generator.
- 4) Explain the optimization techniques of basic blocks.
- 5) Write down the details description of Basic blocks and flow graph functions with an example.