

## **Anna University – Regulation 2013**

### **IT6601 Mobile Computing**

#### **16 Mark Questions**

##### **Unit – I**

1. What is a cellular system. Give their advantages and disadvantages. (MAY 2013)
2. Explain in detail about the motivation for a specialized MAC. (MAY 2013)
3. Compare the mechanism of SDMA, TDMA, FDMA and CDMA with their functions. (MAY 2013)
4. Discuss the principle and operation of cellular wireless networks in detail. (MAY 2013)
5. Compare the characteristic feature of FDMA, TDMA and CDMA mechanism. (MAY 2012)
6. List the various performance metrics used to make the decision with reference to hand off. (MAY 2012)
7. Explain clearly the various schemes for Medium Access Control with Time Division Multiple Access. (DEC 2012)
8. Compare and Contrast SDMA, FDMA, TDMA and CDMA. (DEC 2012)
9. Briefly discuss about cellular systems. (DEC 2012)

##### **Unit – II**

1. Explain in detail about the system and protocol architecture of IEEE802.11. (MAY 2013)
2. Discuss the advantages and disadvantages WLAN. (MAY 2013)
3. Describe the functions of MAC and physical layer of IEEE 802.16 in detail. (MAY 2013)
4. Write short notes on wireless local loop. (MAY 2013)
5. With a suitable diagram explain the extended service set. (MAY 2012)
6. Draw the schematic for physical layer of IEEE 802.11 infrared and explain. (MAY 2012)
7. List the two MAC sublayer defined by IEEE 802.11 standard and explain. (MAY 2012)
8. Briefly explain about the system and protocol architecture of 802.11 standard with MAC layer and its management. (DEC 2012)
9. Briefly discuss about system and protocol structure of 802.16 standard. (DEC 2012)

##### **Unit – III**

1. Explain in detail about the system and protocol architecture of Global System for Mobile communication. (MAY 2013)
2. Discuss in detail the various handover scenarios in GSM. (MAY 2013)
3. Describe the reliability and delay classes in GPRS. Also explain the GPRS procedures. (MAY 2013)
4. Explain the reference architecture of GSM. (MAY 2012)
5. Explain the network services of GPRS and mobility support in GPRS. (MAY 2012)
6. Explain the mobile services, system architecture, localization and calling of GSM in detail. (DEC 2012)
7. Discuss in detail about GPRS architecture and Data Services. (DEC 2012)
8. Explain security services of GSM system. (DEC 2012)

## **Unit – IV**

1. How can the tunneling and encapsulation be performed in mobile IP? Explain. (MAY 2013)
2. Describe the client server configuration of DHCP. (MAY 2013)
3. Discuss how snooping TCP acts as a transparent TCP and explain the role of foreign agent in it in detail. (MAY 2013)
4. What happens in the case of I-TCP if the mobile is disconnected? Discuss. (MAY 2013)
5. What are the three TDMA upgrade options for evolution of 2.5 G TDMA standards? Explain and distinguish between them. (MAY 2012)
6. List the steps for TCP adaptation. (MAY 2012)
7. Explain the goals assumptions and requirements of mobile IP. (MAY 2012)
8. What is Mobile IP? Explain agent discovery registration and encapsulation. (DEC 2012)
9. Write brief notes on Fast Retransmit / Fast Recovery. (DEC 2012)
10. Discuss in detail Indirect and Snooping TCP and differentiate them. (DEC 2012)

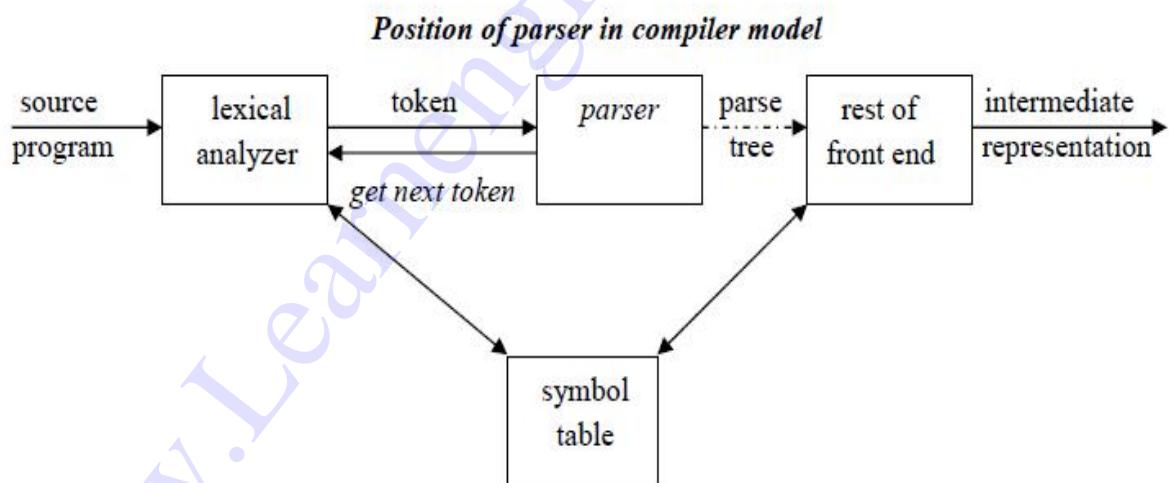
## **Unit – V**

1. Explain in detail the components and interfaces of WAP architecture. (MAY 2013)
2. Discuss the architecture of wireless telephony application in detail. (MAY 2013)
3. Describe the several standard libraries for WML Script specified by WAP. (MAY 2013)
4. Give the architecture of WAP and explain the function of each block. (MAY 2012)
5. Explain the parameters of transmission and session protocols. (MAY 2012)
6. Discuss in detail the WAP architecture and Protocol. (DEC 2012)
7. Write short notes on the following(DEC 2012)
  - a. I-mode.
  - b. Wireless markup language

## SYNTAX ANALYSIS

### THE ROLE OF THE PARSER

- The parser obtains a string of tokens from the lexical analyzer, and verifies that the string of token names can be generated by the grammar for the source language.
- The parser will report any syntax errors in an intelligible fashion and recovers from commonly occurring errors to continue processing the remainder of the program.
- The parser constructs a parse tree and passes it to the rest of the compiler for further processing.



- **The goals of the error handler in a parser**
  1. Report the presence of errors clearly and accurately.
  2. Recover from each error quickly enough to detect subsequent errors.
  3. Add minimal overhead to the processing of correct programs.

## Error-Recovery Strategies

- Once an error is detected, how should the parser recover?

### Panic-Mode Recovery

- On discovering an error, the parser discards input symbols one at a time until one of a designated set of synchronizing tokens is found.

### Phrase-Level Recovery

- On discovering an error, a parser may perform local correction on the remaining input; that is, it may replace a prefix of the remaining input by some string that allows the parser to continue.
- A typical local correction is to replace a comma by a semicolon, delete an extraneous semicolon, or insert a missing semicolon.

### Error Productions

- By anticipating common errors that might be encountered, we can augment the grammar for the language at hand with productions that generate the erroneous constructs.
- A parser constructed from a grammar augmented by these error productions detects the anticipated errors when an error production is used during parsing.
- The parser can then generate appropriate error diagnostics about the erroneous construct that has been recognized in the input.

### Global Correction

- Given an incorrect input string  $x$  and grammar  $G$ , these algorithms will find a parse tree for a related string  $y$ , such that the number of insertions, deletions, and changes of tokens required to transform  $x$  into  $y$  is as small as possible.

## CONTEXT-FREE GRAMMARS

- Grammars were introduced to systematically describe the syntax of programming language constructs like expressions and statements.
- A context-free grammar (grammar for short) consists of terminals, nonterminals, a start symbol, and productions.

### Example:

```
expr → expr op expr
expr → (expr)
expr → - expr
expr → id
op → +
op → -
op → *
op → /
op → ↑
```

#### Grammar for simple arithmetic expressions

- In the above grammar the terminal symbols are (,),-,id,+,-\*,/,↑
- The non-terminal symbols are expr and op
- Using notational shorthands the grammar can be written as

$$\begin{aligned} E &\rightarrow E A E \mid (E) \mid -E \mid id \\ A &\rightarrow + \mid - \mid * \mid / \mid ↑ \end{aligned}$$

## Derivations:

The two types of derivation are:

1. Left most derivation
2. Right most derivation.

- **In leftmost derivations**, the leftmost non-terminal in each sentinel is always chosen first for replacement.
- **In rightmost derivations**, the rightmost non-terminal in each sentinel is always chosen first for replacement.

## Example:

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

Sentence to be derived : – (id+id)

### LEFTMOST DERIVATION

$E \rightarrow -E$

$E \rightarrow - ( E )$

$E \rightarrow - ( E+E )$

$E \rightarrow - ( id+E )$

$E \rightarrow - ( id+id )$

### RIGHTMOST DERIVATION

$E \rightarrow -E$

$E \rightarrow - ( E )$

$E \rightarrow - ( E+E )$

$E \rightarrow - ( E+id )$

$E \rightarrow - ( id+id )$

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

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

## 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 :



## Ambiguity

- A grammar that produces more than one parse tree for some sentence is said to be ambiguous.
- An ambiguous grammar is one that produces more than one leftmost derivation or more than one rightmost derivation for the same sentence.

## WRITING A GRAMMAR

Grammars are capable of describing most, but not all, of the syntax of programming languages.

- For instance, the requirement that identifiers be declared before they are used, cannot be described by a context-free grammar.
- We consider several transformations that could be applied to get a grammar more suitable for parsing.
- One technique can eliminate ambiguity in the grammar, and other techniques — left-recursion elimination and left factoring — are useful for rewriting grammars so they become suitable for top-down parsing.

### Regular Expressions vs. Context-Free Grammars:

REGULAR EXPRESSION	CONTEXT-FREE GRAMMAR
It is used to describe the tokens of programming languages.	It consists of a quadruple where $S \rightarrow$ start symbol, $P \rightarrow$ production, $T \rightarrow$ terminal, $V \rightarrow$ variable or non-terminal.
It is used to check whether the given input is valid or not using <b>transition diagram</b> .	It is used to check whether the given input is valid or not using <b>derivation</b> .
The transition diagram has set of states and edges.	The context-free grammar has set of productions.
It has no start symbol.	It has start symbol.
It is useful for describing the structure of lexical constructs such as identifiers, constants, keywords, and so forth.	It is useful in describing nested structures such as balanced parentheses, matching begin-end's and so on.

## Eliminating Ambiguity

- Sometimes an ambiguous grammar can be rewritten to eliminate the ambiguity.
- As an example, we shall eliminate the ambiguity from the following "danglingelse" grammar:

$\text{stmt} \rightarrow \text{if expr then stmt}$

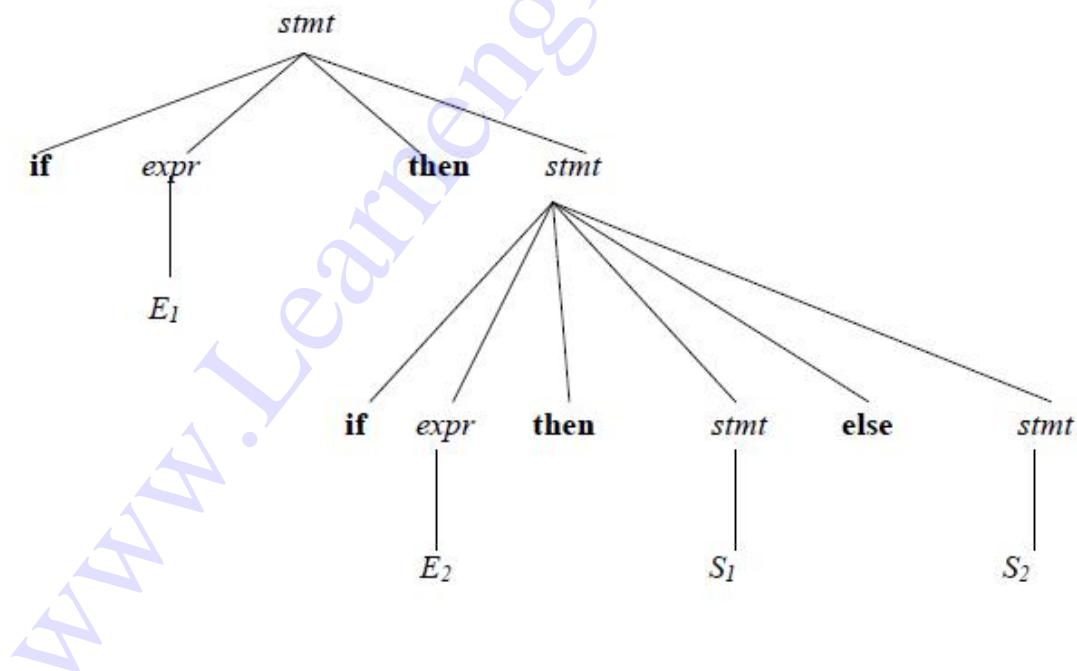
|  $\text{if expr then stmt else stmt}$

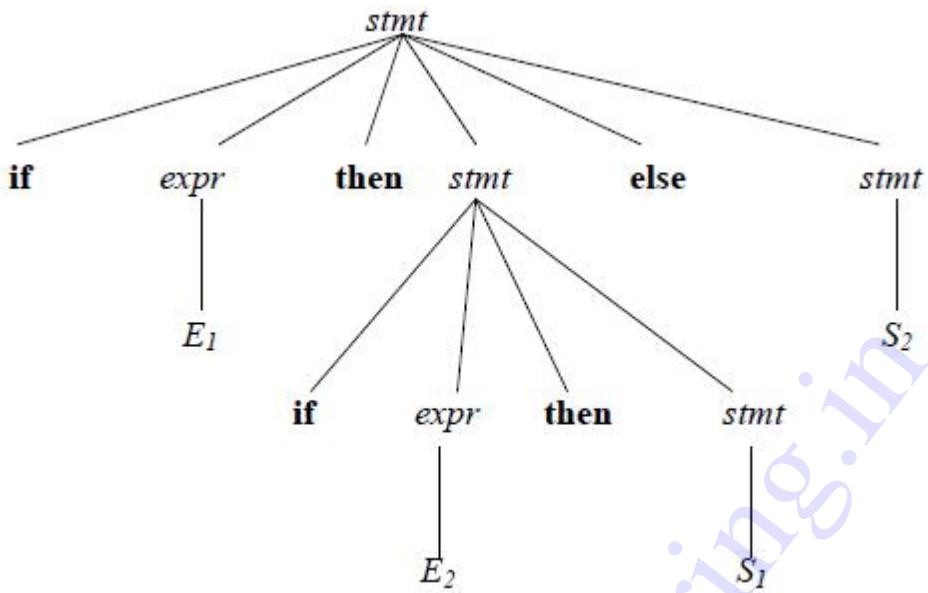
| other

- Here "other" stands for any other statement.
- The ambiguous sentence

**if E1 then if E2 then S1 else S2**

has two parse trees as follows:





- To eliminate ambiguity we can rewrite the dangling-else grammar as the following unambiguous grammar.

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

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

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

- The disambiguation rule used is “Match each else with the closest previous unmatched then”.**
- The idea is that a statement appearing between a then and an else must be "matched"; that is, the interior statement must not end with an unmatched or open then.
- A matched statement is either an if-then-else statement containing no open statements or it is any other kind of unconditional statement.

## Elimination of Left Recursion

- A grammar is left recursive if it has a nonterminal A such that there is a derivation  $A \Rightarrow A\alpha$  for some string  $\alpha$ .
- Top-down parsing methods cannot handle left-recursive grammars, so a transformation is needed to eliminate left recursion.
- If there is a production of the form  $A \rightarrow A\alpha | \beta$  then it is replaced by the following productions

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

### Example:

Consider the following grammar for arithmetic expressions

$$\begin{aligned}E &\rightarrow E + T \mid T \\T &\rightarrow T * F \mid F \\F &\rightarrow .(E) \mid id\end{aligned}$$

The above grammar is left-recursive.

Hence eliminating left-recursion we get,

$$\begin{aligned}E &\rightarrow TE' \\E' &\rightarrow +TE' \mid \epsilon \\T &\rightarrow FT' \\T' &\rightarrow *FT' \mid \epsilon \\F &\rightarrow .(E) \mid id\end{aligned}$$

### **Algorithm to eliminate left recursion from a grammar**

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  $A_i \rightarrow \delta_1 \gamma | \delta_2 \gamma | \dots | \delta_k \gamma$   
    where  $A_j \rightarrow \delta_1 | \delta_2 | \dots | \delta_k$  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, or top-down, parsing.
- If there is a production of the form  $A \rightarrow \alpha\beta_1 | \alpha\beta_2$  then replace it with the following productions

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 | \beta_2$$

### **Example:**

Consider the following grammar that abstracts the dangling else grammar:

$$S \rightarrow iEtS | iEtSeS | a$$

$$E \rightarrow b$$

Left factoring the grammar we get,

$$S \rightarrow iEtSS' | a$$

$$S' \rightarrow eS | \epsilon$$

$E \rightarrow b$

### Algorithm for left factoring a grammar

*Input.* Grammar  $G$ .

*Output.* An equivalent left-factored grammar.

*Method.* For each nonterminal  $A$  find the longest prefix  $\alpha$  common to two or more of its alternatives. If  $\alpha \neq \epsilon$ , i.e., there is a nontrivial common prefix, replace all the  $A$  productions  $A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \dots | \alpha\beta_n | \gamma$  where  $\gamma$  represents all alternatives that do not begin with  $\alpha$  by

$$\begin{aligned} A &\rightarrow \alpha A' | \gamma \\ A' &\rightarrow \beta_1 | \beta_2 | \dots | \beta_n \end{aligned}$$

Here  $A'$  is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix.  $\square$

## Parsing

- There are two types of parsing namely:
  - Top down parsing
  - Bottom up parsing

## TOP DOWN PARSING

- It constructs a parse tree for the input string, starting from the root and creates the nodes of the parse tree in pre-order .
- Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string.
- There are two types of top down parsing namely:
  1. Recursive Descent Parsing
  2. Predictive Parsing
- Recursive descent parser is a top down parser that builds a parse tree which involves backtracking.

- A predictive parser is a special kind of recursive descent parser that involves **no backtracking**.

## 1. Recursive Descent Parsing

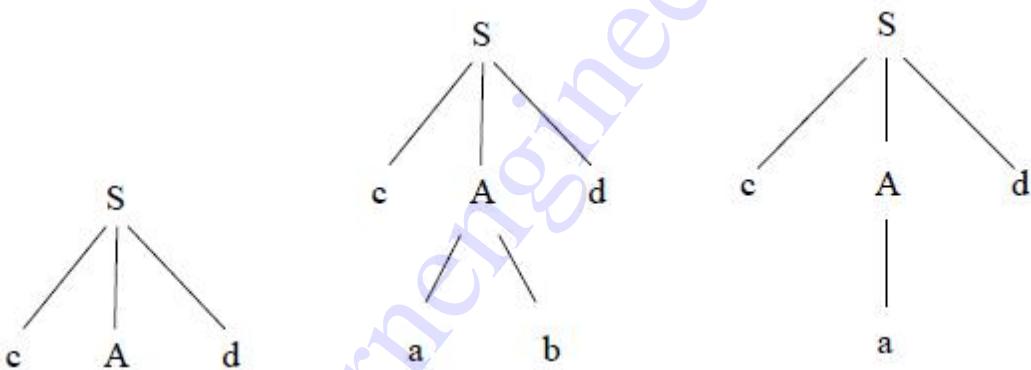
**Example:** Consider the grammar

$$S \rightarrow cAd$$

$$A \rightarrow ab \mid a$$

and the input string  $w = cad$ .

The recursive descent parser builds the parse tree as shown below:



**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.

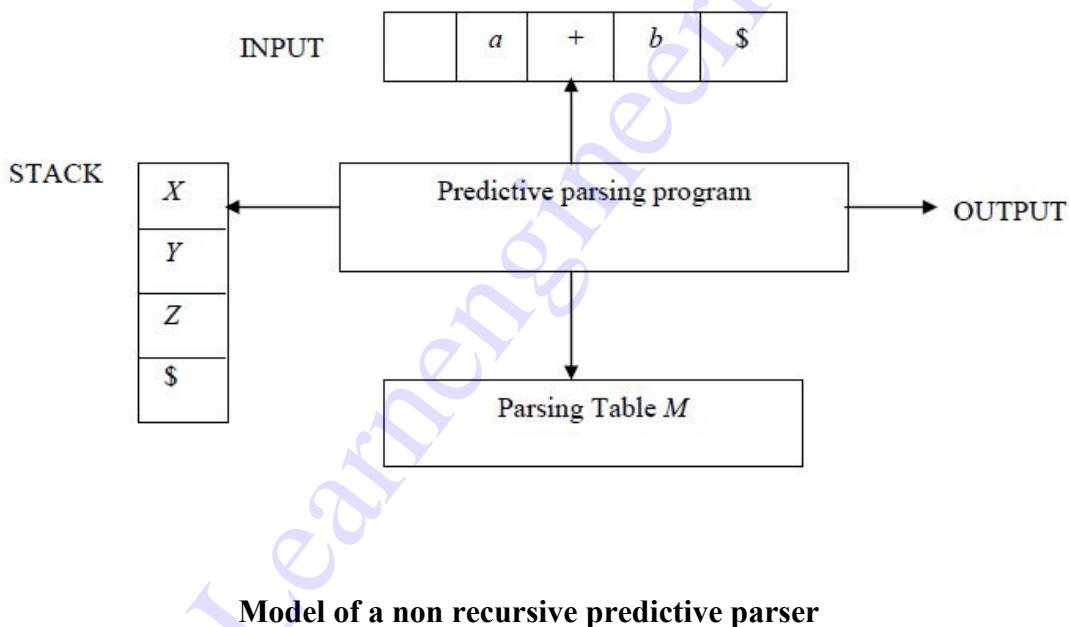
**Step3:** The second symbol 'a' of w also matches with second leaf of tree. So advance the input pointer to third symbol of w 'd'. But the third leaf of tree is b which does not match with the input symbol d. Hence

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

**Step4:** Now try the second alternative for A.

### Predictive Parsing

- A predictive parser is a special kind of recursive descent parser that involves **no backtracking**.



### Steps to construct a predictive parser for a given grammar and to parse the given input string.

1. Eliminate left recursion from the grammar if required.
2. Left factor the grammar if required.
3. Calculate FIRST(X) and FOLLOW(A) where X is a grammar symbol and A is a non-terminal.
4. Construct the parsing table M .

5. Then parse the given input string using the predictive parsing program.

### **Calculation of FIRST(X)**

To compute  $\text{FIRST}(X)$  for all grammar symbols  $X$ , apply the following rules until no more terminals or  $\epsilon$  can be added to any  $\text{FIRST}$  set.

1. If  $X$  is a terminal, then  $\text{FIRST}(X) = \{X\}$ .
2. If  $X$  is a nonterminal 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  $\text{FIRST}(Y_1), \dots, \text{FIRST}(Y_{i-1})$ .
3. If  $X \rightarrow \epsilon$  is a production, then add  $\epsilon$  to  $\text{FIRST}(X)$ .

### **Calculation of FOLLOW(A)**

To compute  $\text{FOLLOW}(A)$  for all nonterminals  $A$ , apply the following rules until nothing can be added to any  $\text{FOLLOW}$  set.

1. Place  $\$$  in  $\text{FOLLOW}(S)$ , where  $S$  is the start symbol, and  $\$$  is the input right endmarker.
2. If there is a production  $A \rightarrow \alpha B \beta$ , then everything in  $\text{FIRST}(\beta)$  except  $\epsilon$  is 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 to construct parsing table M**

**INPUT:** Grammar G.

**OUTPUT:** Parsing table M.

**METHOD:** For each production  $A \rightarrow \alpha$  of the grammar, perform steps 2 and 3

1. For each terminal a in FIRST( $\alpha$ ), add  $A \rightarrow \alpha$  to  $M[A, a]$ .
2. If  $\epsilon$  is in FIRST( $\alpha$ ), then for each terminal b in FOLLOW(A), add  $A \rightarrow \alpha$  to  $M[A, b]$ . If  $\epsilon$  is in FIRST ( $\alpha$ ) and  $\$$  is in FOLLOW(A), add  $A \rightarrow \alpha$  to  $M[A, \$]$ .
3. Mark each undefined entry as error.

### **Algorithm for the predictive parsing program**

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 nonterminal,then consult the entry  $M[X,a]$  of the parsing table M. This entry will either be an X-production or blank.

For eg. If  $M[X,a]= X \rightarrow UVW$  then the parser replaces X on top of the stack by WVU(with U on top). If  $M[X,a]=$ blank the parser calls an error recovery routine.

## Pseudocode for the predictive parsing program

**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  $X = \$$           /* stack is empty */
```

**Example:**

Construct the predictive parser for the following grammar

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

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

$$F \rightarrow .(E) \mid id$$

Show the behaviour of the parser on the sentence  $id + id^* id$ .

**Solution:**

The above grammar is left-recursive.

So eliminating left recursion we get,

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

$$F \rightarrow .(E) \mid id$$

**Calculation of FIRST(X) and FOLLOW(A)**

$$FIRST(+)=\{+\}$$

$$FIRST(E)=FIRST(T)=FIRST(F)=\{ , id\}$$

$$FIRST(*)=\{*\}$$

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

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

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

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

$$FIRST(id)=\{id\}$$

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

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

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

### Parsing Table

Non-terminal	Input Symbol					
	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	$F \rightarrow id$			$F \rightarrow (E)$		

### Moves made by predictive parser on the input $id + id * id$

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) Grammars

- Predictive parsers, can be constructed for a class of grammars called LL(1).
- The first "L" in LL(1) stands for scanning the input from left to right, the second "L" for producing a leftmost derivation, and the "1" for using one input symbol of lookahead at each step to make parsing action decisions.
- A grammar whose parsing table has no multiple-defined entries is called a LL(1) grammar.

### Example:

Check whether the given grammar is LL(1) or not.

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

$$E \rightarrow b$$

### Solution:

The above grammar has a left factor.

Hence left-factoring the grammar we get ,

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

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

$$E \rightarrow b$$

### Calculation of FIRST(X)

$$\text{FIRST}(i) = \{i\} \quad \text{FIRST}(S) = \{i, a\}$$

$$\text{FIRST}(t) = \{t\} \quad \text{FIRST}(S') = \{e, \epsilon\}$$

$$\text{FIRST}(e) = \{e\} \quad \text{FIRST}(E) = \{b\}$$

$$\text{FIRST}(a) = \{a\}$$

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

### Calculation of FOLLOW(A)

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

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

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

### Parsing Table

Non-terminal	Input Symbol					
	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow \epsilon$			$S' \rightarrow eS$
E		$E \rightarrow b$				

This parsing table has multiple-defined entries. Hence the **given grammar is not LL(1).**

## Error Recovery in Predictive Parsing

### How is an error detected in predictive parsing?

- An error is detected during predictive parsing when the terminal on top of the stack does not match the next input symbol or when nonterminal A is on top of the stack, a is the next input symbol, and  $M[A, a]$  is error (i.e., the parsing-table entry is empty).
- Panic-mode error recovery is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens appears.
- Its effectiveness depends on the choice of synchronizing set.
- The sets should be chosen so that the parser recovers quickly from errors that are likely to occur in practice.
- Several heuristics exist out of which the following is one:
- As a starting point, place all symbols in  $\text{FOLLOW}(A)$  into the synchronizing set for nonterminal A.
- If we skip tokens until an element of  $\text{FOLLOW}(A)$  is seen and pop A from the stack, it is likely that parsing can continue.

### Example:

Consider the parsing table of the grammar

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

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

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

<b>Non-terminal</b>	<b>Input Symbol</b>					
	<b>id</b>	<b>+</b>	<b>*</b>	<b>(</b>	<b>)</b>	<b>\$</b>
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	$F \rightarrow id$			$F \rightarrow (E)$		

- To perform error recovery , we place synchronizing tokens into an entry from the FOLLOW set of the nonterminal in question.
- We already know the FOLLOW values which are as follows:

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

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

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

- Once the table is filled with synchronizing tokens the parsing table M is as follows:

<b>Non-terminal</b>	<b>Input Symbol</b>					
	<b>id</b>	<b>+</b>	<b>*</b>	<b>(</b>	<b>)</b>	<b>\$</b>
E	$E \rightarrow TE'$			$E \rightarrow TE'$	synch	synch
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	synch		$T \rightarrow FT'$	synch	synch
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$	synch	synch	$F \rightarrow (E)$	synch	synch

- Suppose we have the following erroneous input string id\*+id.

- We

see  
the

STACK	INPUT	OUTPUT
\$E	id*+id \$	
\$E'T	id*+id \$	
\$E'T'F	id*+id \$	
\$E'T'id	id*+id \$	
\$E'T'	*+id \$	
\$E'T'F*	*+id\$	

will  
how  
parser

recovers from this erroneous input and continues parsing.

### Steps for error recovery

1. If the parser looks up entry  $M[A,a]$  and finds that it is blank, then the input symbol  $a$  is skipped.
2. If the entry is "synch," then the non-terminal on top of the stack is popped in an attempt to resume parsing.
3. If a token on top of the stack does not match the input symbol, then we pop the token from the stack.

### Parsing and error recovery moves made by predictive parser

\$E'T'F	+ id \$	Error, M[F,+]=synch; F has been popped
\$E'T'	+ id \$	
\$E'	+ id \$	
\$E'T+	+ id \$	
\$E'T	id \$	
\$E'T'F	id \$	
\$E'T'id	id \$	
\$E'T'	\$	
\$E'	\$	
\$E'	\$	
\$	\$	

## 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**.

## 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**.

**REDUCTION (LEFTMOST)**

abbcde ( $A \rightarrow b$ )

aAbcde ( $A \rightarrow Abc$ )

aAde ( $B \rightarrow d$ )

aABe ( $S \rightarrow aABe$ )

S

**RIGHTMOST DERIVATION**

$$S \rightarrow aABe$$

$$\rightarrow aAde$$

$$\rightarrow aAbcde$$

$$\rightarrow abbcde$$

The reductions trace out the right-most derivation in reverse.

**Handles:**

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

**Example:**

Consider the grammar:

$E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow (E)$

$E \rightarrow id$

And the input string  $id1+id2*id3$

The rightmost derivation is :

$E \rightarrow \underline{E+E}$

$\rightarrow E+\underline{E*E}$

$\rightarrow E+E*\underline{id3}$

$\rightarrow E+\underline{id2}*id3$

$\rightarrow \underline{id1}+id2*id3$

In the above derivation the underlined substrings are called **handles**.

### **Handle pruning:**

A rightmost derivation in reverse can be obtained by “**handle pruning**”.

(i.e.) if  $w$  is a sentence or string of the grammar at hand, then  $w = \gamma_n$ , where  $\gamma_n$  is the  $n^{\text{th}}$  right- sentinel form of some rightmost derivation.

### **Stack Implementation of Shift-Reduce Parsing**

Stack	Input	Action
\$	$id_1 + id_2 * id_3 \$$	shift
\$ $id_1$	$+ id_2 * id_3 \$$	reduce by $E \rightarrow id$
\$ $E$	$+ id_2 * id_3 \$$	shift
\$ $E +$	$id_2 * id_3 \$$	shift
\$ $E + id_2$	$* id_3 \$$	reduce by $E \rightarrow id$
\$ $E + E$	$* id_3 \$$	shift
\$ $E + E *$	$id_3 \$$	shift
\$ $E + E * id_3$	\$	reduce by $E \rightarrow id$
\$ $E + E * E$	\$	reduce by $E \rightarrow E * E$
\$ $E + E$	\$	reduce by $E \rightarrow E + E$
\$ $E$	\$	accept

### **Actions in shift-reduce parser:**

1. **shift** – The next input symbol is shifted onto the top of the stack.
2. **reduce** – The parser replaces the handle within a stack with a non-terminal.
3. **accept** – The parser announces successful completion of parsing.
4. **error** – The parser discovers that a syntax error has occurred and calls an error recovery routine.

### **Conflicts in shift-reduce parsing:**

There are two conflicts that occur in shift reduce parsing:

1. **Shift-reduce conflict:** The parser cannot decide whether to shift or to reduce.
2. **Reduce-reduce conflict:** The parser cannot decide which of several reductions to make.

## 1. Shift-reduce conflict:

### Example:

Consider the grammar:

$E \rightarrow E+E \mid E^*E \mid id$  and input  $id+id^*id$

Stack	Input	Action	Stack	Input	Action
\$ E+E	*id \$	Reduce by $E \rightarrow E+E$	\$E+E	*id \$	Shift
\$ E	*id \$	Shift	\$E+E*	id \$	Shift
\$ E*	id \$	Shift	\$E+E*id	\$	Reduce by $E \rightarrow id$
\$ E*id	\$	Reduce by $E \rightarrow id$	\$E+E*E	\$	Reduce by $E \rightarrow E^*E$
\$ E^*E	\$	Reduce by $E \rightarrow E^*E$	\$E+E	\$	Reduce by $E \rightarrow E^*E$
\$ E			\$E		

## 2. Reduce-reduce conflict:

Consider the grammar:

$$M \rightarrow R+R \mid R+c \mid R$$

$$R \rightarrow c$$

and input  $c+c$

Stack	Input	Action	Stack	Input	Action
\$	$c+c \$$	Shift	\$	$c+c \$$	Shift
$\$ c$	$+c \$$	Reduce by $R \rightarrow c$	$\$ c$	$+c \$$	Reduce by $R \rightarrow c$
$\$ R$	$+c \$$	Shift	$\$ R$	$+c \$$	Shift
$\$ R+$	$c \$$	Shift	$\$ R+$	$c \$$	Shift
$\$ R+c$	\$	Reduce by $R \rightarrow c$	$\$ R+c$	\$	Reduce by $M \rightarrow R+c$
$\$ R+R$	\$	Reduce by $M \rightarrow R+R$	$\$ M$	\$	
$\$ M$	\$				

### Viable prefixes:

- $\alpha$  is a viable prefix of the grammar if there is  $w$  such that  $\alpha w$  is a right sentinel form.
- The set of prefixes of right sentinel forms that can appear on the stack of a shift-reduce parser are called viable prefixes.
- The set of viable prefixes is a regular language.

## Operator-Precedence Parsing

- An efficient way of constructing shift-reduce parser is called operator-precedence parsing.
- Operator precedence parser can be constructed from a grammar called **Operator-grammar**.
- These grammars have the property that no production on right side is  $\epsilon$  or has two adjacent non-terminals.

### Example:

Consider the grammar:

$$E \rightarrow EAE \mid (E) \mid -E \mid id$$

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

Since the right side EAE has three consecutive non-terminals, the grammar can be written as follows:

$$E \rightarrow E+E \mid E-E \mid E^*E \mid E/E \mid E\uparrow E \mid -E \mid id$$

### Operator precedence relations:

There are three disjoint precedence relations namely

$<$  . ---- less than

$=$  . ---- equal to

. $>$  . ---- greater than

The relations give the following meaning:

$a < . b$  ---- a yields precedence to b

$a = b$  ---- a has the same precedence as b

$a . > b$  ---- a takes precedence over b

### Rules for binary operations:

1. If operator  $\theta_1$  has higher precedence than operator  $\theta_2$ , then make  
 $\theta_1 . > \theta_2$  and  $\theta_2 < . \theta_1$
2. If operators  $\theta_1$  and  $\theta_2$ , are of equal precedence, then make  $\theta_1 . > \theta_2$  and  
 $\theta_2 . > \theta_1$  if operators are left associative  
 $\theta_1 < . \theta_2$  and  $\theta_2 < . \theta_1$  if right associative
3. Make the following for all operators  $\theta$ :

$$\begin{aligned}\theta < . \text{id}, \text{id} . > \theta \\ \theta < . (, ( < . \theta \\ ) . > \theta, \theta . > ) \\ \theta . > \$, \$ < . \theta\end{aligned}$$

Also make

$$(=), (< . (, ) . >), (< . \text{id}, \text{id} . >), \$ < . \text{id}, \text{id} . > \$, \$ < . (, ) . > \$$$

### Example:

Operator-precedence relations for the grammar

$E \rightarrow E+E \mid E-E \mid E^*E \mid E/E \mid E \uparrow E \mid (E) \mid -E \mid \text{id}$  is given in the following table assuming

1.  $\uparrow$  is of highest precedence and right-associative
2. \* and / are of next higher precedence and left-associative, and
3. + and - are of lowest precedence and left-associative

Note that the blanks in the table denote error entries.

	+	-	*	/	$\uparrow$	id	(	)	\$
+	>	>	<	<	<	<	<	>	>
-	>	>	<	<	<	<	<	>	>
*	>	>	>	>	<	<	<	>	>
/	>	>	>	>	<	<	<	>	>
$\uparrow$	>	>	>	>	<	<	<•	>	>
id	>	>	>	>	>			>	>
(	<	<	<	<	<	<	<	=	
)	>	>	>	>	>			>	>
\$	<	<	<	<	<	<	<		

### Operator Precedence Relations

### Operator precedence parsing algorithm

**Input :** An input string  $w$  and a table of precedence relations.

**Output :** If  $w$  is well formed, a *skeletal* parse tree ,with a placeholder non-terminal E labeling all interior nodes; otherwise, an error indication.

**Method :** Initially the stack contains \$ and the input buffer the string  $w\$$ .

To parse, we execute the following program :

- (1) Set  $ip$  to point to the first symbol of  $w\$$ ;
- (2) **repeat forever**
- (3)   **if** \$ is on top of the stack and  $ip$  points to \$ **then**
- (4)       **return**
- else begin**
- (5)        let  $a$  be the topmost terminal symbol on the stack  
and let  $b$  be the symbol pointed to by  $ip$ ;

- (6)   **if**  $a < b$  or  $a = b$  **then begin**
- (7)              push  $b$  onto the stack;
- (8)              advance  $ip$  to the next input symbol;
- end;**
- (9)   **else if**  $a . > b$  **then /\*reduce\*/**
- (10)       **repeat**
- (11)              pop the stack
- (12)       **until** the top stack terminal is related by  $<$   
                    to the terminal most recently popped
- (13)       **else** error( )
- end**

### **Stack implementation of operator precedence parsing:**

Operator precedence parsing uses a stack and precedence relation table for its implementation of above algorithm. It is a shift-reduce parsing containing all four actions shift, reduce, accept and error.

The initial configuration of an operator precedence parsing is

STACK	INPUT
\$	w \$

where w is the input string to be parsed.

#### **Example:**

Consider the grammar  $E \rightarrow E+E \mid E-E \mid E^*E \mid E/E \mid E\uparrow E \mid (E) \mid id$ . Input string is **id+id\*id**. The implementation is as follows:

STACK	INPUT	COMMENT
\$	< id+id*id \$	shift id
\$ id	> +id*id \$	pop the top of the stack id
\$	< +id*id \$	shift +
\$ +	< id*id \$	shift id
\$ +id	> *id \$	pop id
\$ +	< *id \$	shift *
\$ + *	< id \$	shift id
\$ + * id	> \$	pop id
\$ + *	> \$	pop *
\$ +	> \$	pop +
\$	\$	accept

### Advantages of operator precedence parsing:

1. It is easy to implement.
2. Once an operator precedence relation is made between all pairs of terminals of a grammar , the grammar can be ignored. The grammar is not referred anymore during implementation.

### Disadvantages of operator precedence parsing:

1. It is hard to handle tokens like the minus sign (-) which has two different precedence.
2. Only a small class of grammar can be parsed using operator-precedence parser.

## LR PARSERS

- An efficient bottom-up syntax analysis technique that can be used to parse a large class of CFG is called LR( $k$ ) parsing.
- The ‘L’ is for left-to-right scanning of the input, the ‘R’ for constructing a rightmost derivation in reverse, and the ‘ $k$ ’ for the number of input symbols.
- When ‘ $k$ ’ is omitted, it is assumed to be 1.

### Advantages of LR parsing:

1. It recognizes virtually all programming language constructs for which CFG can be written.
2. It is an efficient non-backtracking shift-reduce parsing method.
3. A grammar that can be parsed using LR method is a proper superset of a grammar that can be parsed with predictive parser.
4. It detects a syntactic error as soon as possible.

### Drawbacks of LR method:

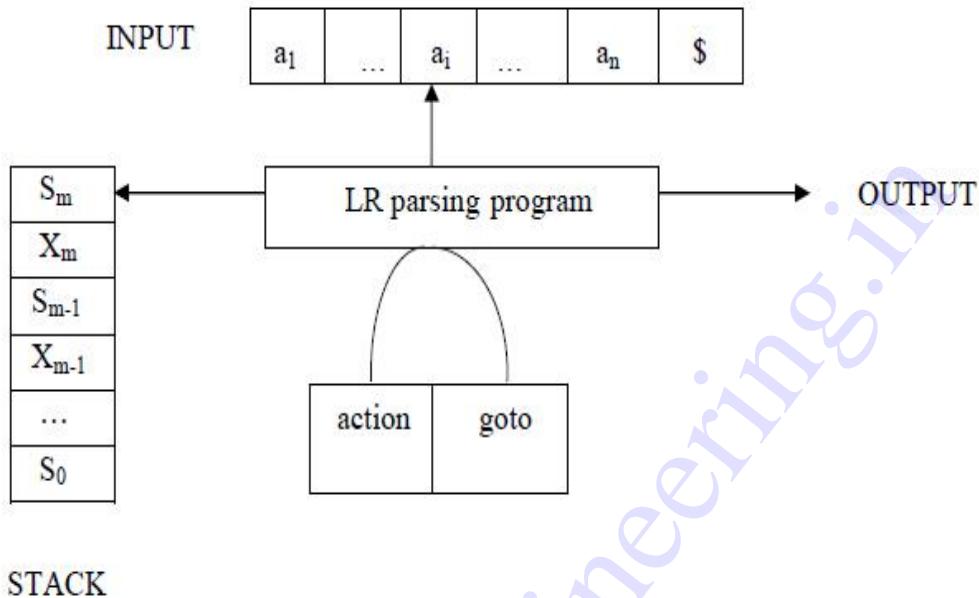
It is too much of work to construct a LR parser by hand for a programming language grammar. A specialized tool, called a LR parser generator, is needed. Example: YACC.

### Types of LR parsing method:

1. SLR- Simple LR
  - Easiest to implement, least powerful.
2. CLR- Canonical LR
  - Most powerful, most expensive.
3. LALR- Look-Ahead LR
  - Intermediate in size and cost between the other two methods.

## The LR parsing algorithm:

The schematic form of an LR parser is as follows:



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

- The driver program is the same for all LR parser.
- The parsing program reads characters from an input buffer one at a time.
- The program uses a stack to store a string of the form  $s_0X_1s_1X_2s_2\dots X_ms_m$ , where  $s_m$  is on top. Each  $X_i$  is a grammar symbol and each  $s_i$  is a state.
- The parsing table consists of two parts : *action* and *goto* functions.

**Action :** The parsing program determines  $s_m$ , the state currently on top of stack, and  $a_i$ , the current input symbol. It then consults  $\text{action}[s_m, a_i]$  in the action table which can have one of four values :

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

**Goto** : The function goto takes a state and grammar symbol as arguments and produces a state.

### **LR Parsing algorithm:**

**Input:** An input string  $w$  and an LR parsing table with functions  $action$  and  $goto$  for grammar  $G$ .

**Output:** If  $w$  is in  $L(G)$ , a bottom-up-parse for  $w$ ; otherwise, an error indication.

**Method:** Initially, the parser has  $s_0$  on its stack, where  $s_0$  is the initial state, and  $w\$$  in the input buffer.

The parser then executes the following program :

```
set ip to point to the first input symbol of  $w\$$ ;  
repeat forever begin  
    let  $s$  be the state on top of the stack and  
     $a$  the symbol pointed to by ip;  
    if  $action[s, a] = \text{shift } s'$  then begin  
        push  $a$  then  $s'$  on top of the stack;  
        advance ip to the next input symbol  
    end  
    else if  $action[s, a] = \text{reduce } A \rightarrow \beta$  then begin  
        pop  $2^* | \beta |$  symbols off the stack;  
        let  $s'$  be the state now on top of the stack;
```

```
push A then goto[s', A] on top of the stack;  
output the production A → β  
end  
else if action[s, a] = accept then  
    return  
else error( )  
end
```

### CONSTRUCTING SLR(1) PARSING TABLE:

To perform SLR parsing, take grammar as input and do the following:

1. Find LR(0) items.
2. Completing the closure.
3. Compute *goto*(I,X), where, I is set of items and X is grammar symbol.

### LR(0) items:

An *LR(0)* item of a grammar G is a production of G with a dot at some position of the right side. For example, production A → XYZ yields the four items :

$$\begin{aligned} & A \rightarrow .XYZ \\ & A \rightarrow X . YZ \\ & A \rightarrow XY . Z \\ & A \rightarrow XYZ . \end{aligned}$$

### Closure operation:

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

1. Initially, every item in  $I$  is added to  $\text{closure}(I)$ .
2. If  $A \rightarrow \alpha . B\beta$  is in  $\text{closure}(I)$  and  $B \rightarrow \gamma$  is a production, then add the item  $B \rightarrow . \gamma$  to  $I$ , if it is not already there. We apply this rule until no more new items can be added to  $\text{closure}(I)$ .

### Goto operation:

$\text{Goto}(I, X)$  is defined to be the closure of the set of all items  $[A \rightarrow \alpha X . \beta]$  such that  $[A \rightarrow \alpha . X\beta]$  is in  $I$ .

### Steps to construct SLR parsing table for grammar $G$ are:

1. Augment  $G$  and produce  $G'$
2. Construct the canonical collection of set of items  $C$  for  $G'$
3. Construct the parsing action function *action* and *goto* using the following algorithm that requires  $\text{FOLLOW}(A)$  for each non-terminal of grammar.

### Algorithm for construction of SLR parsing table:

**Input :** An augmented grammar  $G'$

**Output :** The SLR parsing table functions *action* and *goto* for  $G'$

#### Method :

1. Construct  $C = \{I_0, I_1, \dots, I_n\}$ , the collection of sets of  $\text{LR}(0)$  items for  $G'$ .
2. State  $i$  is constructed from  $I_i$ . The parsing functions for state  $i$  are determined as follows:

(a) If  $[A \rightarrow a \cdot a\beta]$  is in  $I_i$  and  $\text{goto}(I_i, a) = I_j$ , then set  $\text{action}[i, a]$  to “shift j”. Here  $a$  must be terminal.

(b) If  $[A \rightarrow a \cdot]$  is in  $I_i$ , then set  $\text{action}[i, a]$  to “reduce  $A \rightarrow a$ ” for all  $a$  in  $\text{FOLLOW}(A)$ .

(c) If  $[S' \rightarrow S \cdot]$  is in  $I_i$ , then set  $\text{action}[i, \$]$  to “accept”.

If any conflicting actions are generated by the above rules, we say grammar is not SLR(1).

3. The  $\text{goto}$  transitions for state  $i$  are constructed for all non-terminals  $A$  using the rule:

If  $\text{goto}(I_i, A) = I_j$ , then  $\text{goto}[i, A] = j$ .

4. All entries not defined by rules (2) and (3) are made “error”

5. The initial state of the parser is the one constructed from the set of items containing  $[S' \rightarrow \cdot S]$ .

### Example:

Construct SLR parsing for the following grammar :

$$G : E \rightarrow E + T \mid T$$

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

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

The given grammar is :

$$G : E \rightarrow E + T \text{ ----- (1)}$$

$$E \rightarrow T \text{ ----- (2)}$$

$$T \rightarrow T * F \text{ ----- (3)}$$

$$T \rightarrow F \text{ ----- (4)}$$

$$F \rightarrow (E) \text{ ----- (5)}$$

$$F \rightarrow \text{id} \text{ ----- (6)}$$

**Step 1 :** Convert given grammar into augmented grammar.

**Augmented grammar :**

$$E' \rightarrow E$$

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow id$$

**Step 2 :** Find LR (0) items.

$$I_0 : E' \rightarrow . E$$

$$E \rightarrow . E + T$$

$$E \rightarrow . T$$

$$T \rightarrow . T * F$$

$$T \rightarrow . F$$

$$F \rightarrow . (E)$$

$$F \rightarrow . id$$

GOTO (  $I_0$  , E )

$I_1 : E' \rightarrow E .$

$$E \rightarrow E . + T$$

GOTO (  $I_0$  , T )

$I_2 : E \rightarrow T .$

$$T \rightarrow T . * F$$

GOTO (  $I_4$  , id )

$I_5 : F \rightarrow id .$

GOTO (  $I_6$  , T )

$I_9 : E \rightarrow E + T .$

$$T \rightarrow T . * F$$

GOTO ( I<sub>6</sub> , F )

I<sub>3</sub> : T → F .

GOTO ( I<sub>0</sub> , F )

I<sub>3</sub> : T → F .

GOTO ( I<sub>0</sub> , () )

I<sub>4</sub> : F → ( . E )

E → . E + T

E → . T

T → . T \* F

T → . F

F → . (E)

F → . id

GOTO ( I<sub>0</sub> , id )

I<sub>5</sub> : F → id .

GOTO ( I<sub>1</sub> , + )

I<sub>6</sub> : E → E + . T

T → . T \* F

T → . F

F → . (E)

F → . id

GOTO ( I<sub>6</sub> , () )

I<sub>4</sub> : F → ( . E )

GOTO ( I<sub>6</sub> , id )

I<sub>5</sub> : F → id .

GOTO ( I<sub>7</sub> , F )

I<sub>10</sub> : T → T \* F .

GOTO ( I<sub>7</sub> , () )

I<sub>4</sub> : F → ( . E )

E → . E + T

E → . T

T → . T \* F

T → . F

F → . (E)

F → . id

GOTO ( I<sub>7</sub> , id )

I<sub>5</sub> : F → id .

GOTO ( I <sub>2</sub> , * )	GOTO ( I <sub>8</sub> , ) )
I <sub>7</sub> : T → T * . F	I <sub>11</sub> : F → ( E ) .
F → . (E)	GOTO ( I <sub>8</sub> , + )
F → . id	I <sub>6</sub> : E → E + . T
	T → . T * F
GOTO ( I <sub>4</sub> , E )	T → . F
I <sub>8</sub> : F → ( E . )	F → . ( E )
E → E . + T	F → . id
GOTO ( I <sub>4</sub> , T )	GOTO ( I <sub>9</sub> , * )
I <sub>2</sub> : E → T .	I <sub>7</sub> : T → T * . F
T → T . * F	F → . ( E )
	F → . id
GOTO ( I <sub>4</sub> , F )	
I <sub>3</sub> : T → F .	
GOTO ( I <sub>4</sub> , ( ) )	
I <sub>4</sub> : F → ( . E )	
E → . E + T	
E → . T	
T → . T * F	
T → . F	
F → . (E)	
F → id	

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

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

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

### SLR parsing table:

	ACTION						GOTO		
	<b>id</b>	<b>+</b>	<b>*</b>	<b>(</b>	<b>)</b>	<b>\$</b>	<b>E</b>	<b>T</b>	<b>F</b>
<b>I<sub>0</sub></b>	s <sub>5</sub>				s <sub>4</sub>		1	2	3
<b>I<sub>1</sub></b>		s <sub>6</sub>				ACC			
<b>I<sub>2</sub></b>		r <sub>2</sub>	s <sub>7</sub>		r <sub>2</sub>	r <sub>2</sub>			
<b>I<sub>3</sub></b>		r <sub>4</sub>	r <sub>4</sub>		r <sub>4</sub>	r <sub>4</sub>			
<b>I<sub>4</sub></b>	s <sub>5</sub>			s <sub>4</sub>			8	2	3
<b>I<sub>5</sub></b>		r <sub>6</sub>	r <sub>6</sub>		r <sub>6</sub>	r <sub>6</sub>			
<b>I<sub>6</sub></b>	s <sub>5</sub>			s <sub>4</sub>				9	3
<b>I<sub>7</sub></b>	s <sub>5</sub>			s <sub>4</sub>					10
<b>I<sub>8</sub></b>		s <sub>6</sub>			s <sub>11</sub>				
<b>I<sub>9</sub></b>		r <sub>1</sub>	s <sub>7</sub>		r <sub>1</sub>	r <sub>1</sub>			
<b>I<sub>10</sub></b>		r <sub>3</sub>	r <sub>3</sub>		r <sub>3</sub>	r <sub>3</sub>			
<b>I<sub>11</sub></b>		r <sub>5</sub>	r <sub>5</sub>		r <sub>5</sub>	r <sub>5</sub>			

Blank entries are error entries.

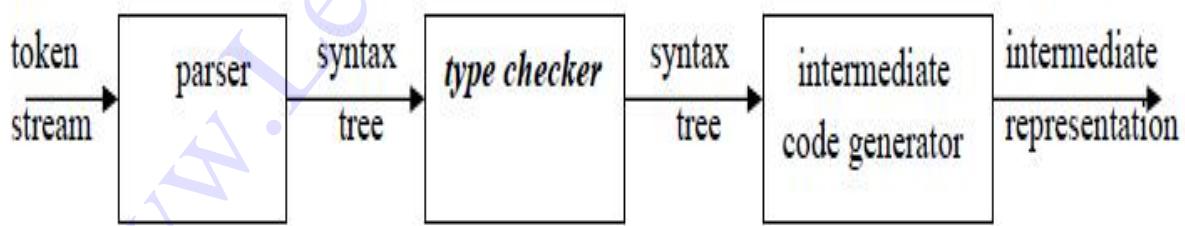
## Stack implementation:

Check whether the input id + id \* id is valid or not.

STACK	INPUT	ACTION
0	id + id * id \$	GOTO ( I <sub>0</sub> , id ) = s5 ; <b>shift</b>
0 id 5	+ id * id \$	GOTO ( I <sub>5</sub> , + ) = r6 ; <b>reduce</b> by F → id
0 F 3	+ id * id \$	GOTO ( I <sub>0</sub> , F ) = 3 GOTO ( I <sub>3</sub> , + ) = r4 ; <b>reduce</b> by T → F
0 T 2	+ id * id \$	GOTO ( I <sub>0</sub> , T ) = 2 GOTO ( I <sub>2</sub> , + ) = r2 ; <b>reduce</b> by E → T
0 E 1	+ id * id \$	GOTO ( I <sub>0</sub> , E ) = 1 GOTO ( I <sub>1</sub> , + ) = s6 ; <b>shift</b>
0 E 1 + 6	id * id \$	GOTO ( I <sub>6</sub> , id ) = s5 ; <b>shift</b>
0 E 1 + 6 id 5	* id \$	GOTO ( I <sub>5</sub> , * ) = r6 ; <b>reduce</b> by F → id
0 E 1 + 6 F 3	* id \$	GOTO ( I <sub>6</sub> , F ) = 3 GOTO ( I <sub>3</sub> , * ) = r4 ; <b>reduce</b> by T → F
0 E 1 + 6 T 9	* id \$	GOTO ( I <sub>6</sub> , T ) = 9 GOTO ( I <sub>9</sub> , * ) = s7 ; <b>shift</b>
0 E 1 + 6 T 9 * 7	id \$	GOTO ( I <sub>7</sub> , id ) = s5 ; <b>shift</b>
0 E 1 + 6 T 9 * 7 id 5	\$	GOTO ( I <sub>5</sub> , \$ ) = r6 ; <b>reduce</b> by F → id
0 E 1 + 6 T 9 * 7 F 10	\$	GOTO ( I <sub>7</sub> , F ) = 10 GOTO ( I <sub>10</sub> , \$ ) = r3 ; <b>reduce</b> by T → T * F
0 E 1 + 6 T 9	\$	GOTO ( I <sub>6</sub> , T ) = 9 GOTO ( I <sub>9</sub> , \$ ) = r1 ; <b>reduce</b> by E → E + T
0 E 1	\$	GOTO ( I <sub>0</sub> , E ) = 1 GOTO ( I <sub>1</sub> , \$ ) = <b>accept</b>

## 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.

**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.

**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.

### 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 T1 and T2 are type expressions, then their Cartesian product T1 X T2 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.

**Example:**

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

**var table: array[1...101] of row;**

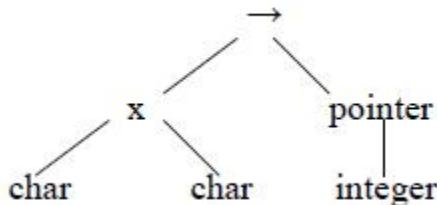
declares the type name *row* representing the type expression **record((address X integer) X(lexeme X array(1..15,char)))** and the variable *table* to be an array of records of this type.

**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:

$P \rightarrow D ; E$

$D \rightarrow D ; D \mid id : T$

$T \rightarrow char \mid integer \mid array [ num ] of T \mid \uparrow T$

$E \rightarrow literal \mid num \mid id \mid E \text{ mod } E \mid E [ E ] \mid E \uparrow$

### Translation scheme:

$P \rightarrow D ; E$

$D \rightarrow D ; D$

$D \rightarrow id : T \{ \text{addtype} (id.entry, T.type) \}$

$T \rightarrow char \{ T.type := \text{char} \}$

$T \rightarrow integer \{ T.type := \text{integer} \}$

$T \rightarrow \uparrow T1 \{ T.type := \text{pointer}(T1.type) \}$

$T \rightarrow array [ num ] of T1 \{ T.type := \text{array} ( 1 \dots num.val, T1.type ) \}$

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.  $E \rightarrow \text{literal} \quad \{ E.type := \text{char} \}$

$E \rightarrow \text{num} \quad \{ E.type := \text{integer} \}$

Here, constants represented by the tokens **literal** and **num** have type *char* and *integer*.

$$2. E \rightarrow \mathbf{id} \quad \{ E.type := \text{lookup}(\mathbf{id}.entry) \}$$

*lookup* (*e*) is used to fetch the type saved in the symbol table entry pointed to by *e*.

$$3. E \rightarrow E_1 \mathbf{mod} E_2 \quad \{ E.type := \begin{array}{l} \mathbf{if } E_1.type = \mathbf{integer} \mathbf{and} \\ \qquad E_2.type = \mathbf{integer} \mathbf{then} \mathbf{integer} \\ \mathbf{else} type\_error \end{array} \}$$

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

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

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  $E_1$ .

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

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:

$S \rightarrow id := E \quad \{ S.type := \text{if } id.type = E.type \text{ then } void$   
 $\qquad \qquad \qquad \text{else } type\_error \}$

### 2. Conditional statement:

$S \rightarrow \text{if } E \text{ then } S1 \quad \{ S.type := \text{if } E.type = boolean \text{ then } S1.type$   
 $\qquad \qquad \qquad \text{else } type\_error \}$

### 3. While statement:

$S \rightarrow \text{while } E \text{ do } S1 \quad \{ S.type := \text{if } E.type = boolean \text{ then } S1.type$   
 $\qquad \qquad \qquad \text{else } type\_error \}$

### 4. Sequence of statements:

$S \rightarrow S1 ; S2 \quad \{ S.type := \text{if } S1.type = void \text{ and }$   
 $\qquad \qquad \qquad S1.type = void \text{ then } void$   
 $\qquad \qquad \qquad \text{else } type\_error \}$

## Type checking of functions

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

$E \rightarrow E1 ( E2 ) \quad \{ E.type := \text{if } E2.type = s \text{ and }$   
 $\qquad \qquad \qquad E1.type = s \rightarrow t \text{ then } t$   
 $\qquad \qquad \qquad \text{else } type\_error \}$

## RUN-TIME ENVIRONMENTS

- The allocation and deallocation of data objects is managed by the run-time support package.
- Run-time support package consisting of routines loaded with the generated target code.
- The design of the run-time support package is influenced by the semantics of procedures.
- Each execution of a procedure is referred to as an activation of the procedure.
- If the procedure is recursive, several of its activations may be alive at the same time.
- The representation of a data object at run-time is determined by its type.

## SOURCE LANGUAGE ISSUES

We distinguish between the source text of a procedure and its activations at run time.

### Procedures:

- A procedure definition is a declaration that in its simplest form associates an identifier with a statement.
- The identifier is the procedure name, and the statement is procedure body.
- Procedures that return values are called functions in many languages.
- A complete program will also be treated as a procedure.
- When a procedure name appears within an executable statement, we say that the procedure is called at that point.
- The basic idea is that a procedure call executes the procedure body.

- Some of the identifiers appearing in a procedure definition are special and are called formal parameters of the procedure.
- Arguments known as actual parameters may be passed to a called procedure. They are substituted for the formals in the body.

```
(1) program sort(input, output);
(2)   var a : array [0..10] of integer;
(3)   procedure readarray;
(4)     var i : integer;
(5)     begin
(6)       for i := 1 to 9 do read(a[i])
(7)     end;
(8)   function partition(y, z: integer) : integer;
(9)     var i, j, x, v: integer;
(10)    begin ...
(11)    end;
(12)   procedure quicksort(m, n: integer);
(13)     var i : integer;
(14)     begin
(15)       if ( n > m ) then begin
(16)         i := partition(m,n);
(17)         quicksort(m,i-1);
(18)         quicksort(i+1,n)
(19)       end
(20)     end;
(21)   begin
(22)     a[0] := -9999; a[10] := 9999;
(23)     readarray;
(24)     quicksort(1,9)
(25)   end.
```

A Pascal program for reading and sorting integers

### Activation Trees:

- Each execution of a procedure starts at the beginning of the procedure body and eventually return control to the point immediately following the place where the procedure was called.
- This means the flow of control between procedures can be depicted using trees.

```
Execution begins...
enter readarray
leave readarray
enter quicksort(1,9)
enter partition(1,9)
leave partition(1,9)
enter quicksort(1,3)
...
leave quicksort(1,3)
enter quicksort(5,9)
...
leave quicksort(5,9)
leave quicksort(1,9)
Execution terminated.
```

#### Output suggesting activation of procedures

- Each execution of a procedure body is referred to as an activation of the procedure.
- The lifetime of an activation of a procedure p is the sequence of steps between the first and last steps in the execution of the procedure body, including time spent executing procedures called by p.
- The lifetime refers to consecutive sequence of steps during the execution of a program.
- Activation tree is a tree which is used to depict the way control enters and leaves activations.

- In an activation tree,
  - (i) each node represents an activation of a procedure.
  - (ii) The root represents the activation of the main program.
  - (iii) The node for ‘a’ is the parent of the node for ‘b’ if and only if control flows from activation ‘a’ to ‘b’.
  - (iv) 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’.

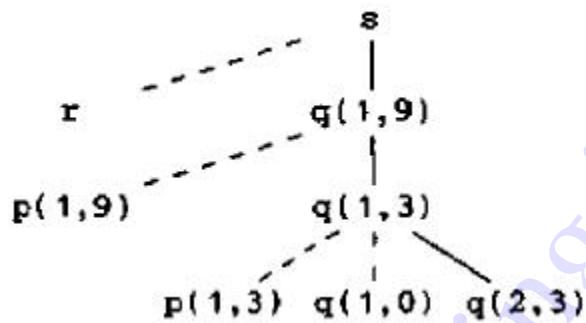


Activation tree corresponding to the above output

### Control Stacks:

- The flow of control in a program corresponds to a depth-first traversal of the activation tree that starts at the root, visits a node before its children, and recursively visits children at each node in a left-to-right order.
- We can use a stack called a control stack 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.

- Then 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 the control stack, the stack contains the nodes along the path from ‘n’ to the root.



**Control stack contains a node along a path to the root**

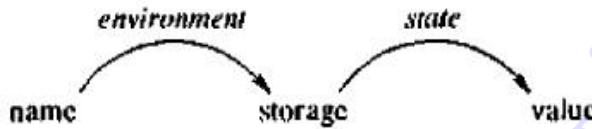
### **The Scope of a Declaration:**

- The portion of the program to which a declaration applies is called the scope of that declaration.
- An occurrence of a name in a procedure is said to be local to the procedure if it is in the scope of a declaration within the procedure. Otherwise the occurrence is said to be non-local.
- At compile time, the symbol table can be used to find the declaration that applies to an occurrence of a name. When a declaration is seen, a symbol table entry is created for it. As long as we are in the scope of the declaration, its entry is returned when the name in it is looked up.

### **Binding of Names:**

- Even if each name is declared once in a program, the same name may denote different data objects at run time. The informal term “data object” corresponds to a storage location that can hold values.

- Environment refers to a function that maps a name to a storage location and the term state refers to a function that maps a storage location to the value held there.



**Two stage mapping from names to values**

- Environment and states are different. An assignment changes the state but not the environment.
- For example, suppose that storage address 100, associated with variable pi holds 0. After the assignment  $\text{pi} := 3.14$ , the same storage address is associated with pi, but the value held there is 3.14.
- When an environment associates storage location s with a name x, we say that x is bound to s; the association itself is referred to as a binding of x.
- A binding is the dynamic counterpart of a declaration.

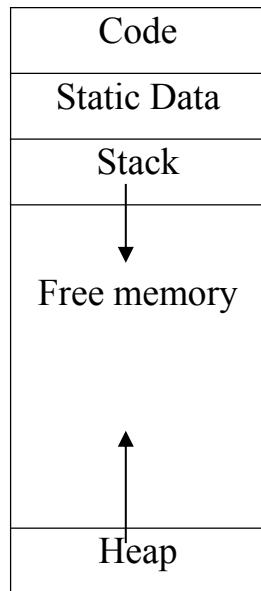
Static Notion	Dynamic Counterpart
Definition of a procedure	Activation of a procedure
Declaration of a name	Bindings of the name
Scope of a declaration	Lifetime of a binding.

## STORAGE ORGANIZATION

### Subdivision of Run-Time Memory:

Run-time storage might be subdivided to hold:

- (i) the generated target code,
  - (ii) data objects and
  - (iii) a counterpart of the control stack to keep track of procedure activation.
- The size of the generated target code is fixed at compile time. So the compiler can place it in a statically determined area (low end of memory).
  - Similarly, the size of some of data objects may also be known at compile time and these too can be placed in a statically determined area.
  - When a procedure call occurs, execution of an activation is interrupted and information about the status of the machine such as the value of the program counter and machine registers is saved on the stack.
  - When control returns from the call, this activation can be restarted after restoring the values of relevant registers and setting the program counter to the point immediately after the call.



#### Typical subdivision of run-time memory into code and data areas

- A separate area of run-time memory called a heap, holds all other information.
- The sizes of the stack and heap can change as the program executes, so we show these at opposite ends of memory as shown in fig. where they can grow toward each other as needed.

#### Activation Records:

- Information needed by a single execution of a procedure is managed using a contiguous block of storage called an activation record or frame.
- Activation record consists of the collection of fields as shown in the following figure:

Returned value
Actual parameters
Optional control link
Optional access link
Saved machine status
Local data
Temporaries

**A general activation record**

- The purpose of the fields of an activation record is as follows, starting from the field for temporaries.
  - (i) Temporary values, such as those arising from the evaluation of expressions, are stored in the field for temporaries.
  - (ii) The field for local data holds data that is local to an execution of a procedure.
  - (iii) The field for saved machine status holds information about the state of the machine just before the procedure is called. This information typically includes the value of the program counter and machine registers that have to be restored when control returns from the procedure.
  - (iv) An optional access link is used to refer to non-local data stored in another activation record.
  - (v) A optional control link points to the activation record of the caller.

- (vi) The actual parameters used by the calling procedure to supply parameters to the called procedure.
  - (vii) It is used by called procedure to return a value to the calling procedure.
- 
- The size of each of these fields can be determined at the time a procedure is called.

### **Compile-Time Layout of Local Data:**

- The amount of storage needed for a name is determined by its type. An elementary data type such as a character, integer or real can usually be stored in an integral number of bytes. Storage for an aggregate such as an array or record must be large enough to hold all its components.
- The field for local data is laid out as the declarations in a procedure are examined at compile time. Variable length data is kept outside this field.
- The storage layout for data objects is strongly influenced by the addressing constraints of the target machine.
- To store an array of 10 characters, a compiler may allocate 12 bytes leaving 2 bytes unused.
- Space left unused due to alignment considerations is referred to as padding.
- When space is at a premium, a compiler may pack data so that no padding is left; additional instructions may need to be executed at run time to position packed data so that it can be operated on as if it were properly aligned.

## STORAGE ALLOCATION STRATEGIES

The various storage allocation strategies are:

- (i) **static allocation** lays out storage for all data objects at compile time.
  - (ii) **stack allocation** manages the run-time storage as a stack.
  - (iii) **heap allocation** allocates and deallocates storage as needed at run time from a data area known as a heap.
- 
- These allocation strategies are applied to activation records.

### Static Allocation:

- During compilation, names are bound to storage.
- So there is no need of run-time support package.
- Since the bindings do not change at run-time, every time a procedure is activated, its names are bound to the same storage locations.
- This property allows the values of local names to be retained across activations of a procedure.
- The address of this storage consists of an offset from an end of the activation record for the procedure.
- The compiler must decide where the activation records go, relative to the target code and to one another.
- Once this decision is made, the position of each activation record and hence of the storage for each name in the record is fixed.

- At compile time, fill in the addresses at which the target code can find the data it operates on.
- The addresses at which information is to be saved when a procedure call occurs are also known at compile time.

### **Limitations with static allocation:**

- (i) The size of the data object and constraints on its position in memory must be known at compile time.
  - (ii) Recursive procedures are restricted, because all activations of a procedure use the same bindings for local names.
  - (iii) Data structures cannot be created dynamically, since there is no mechanism for storage allocation at run time.
- 
- Fortran was designed to permit static storage allocation. Since the sizes of executable code and the activation records are known at compile time, different memory organizations are possible.
  - A Fortran compiler might place the activation record for a procedure together with the code for that procedure.
  - On some computer systems, it is feasible to leave the relative position of the activation records unspecified and allow the link editor to link activation records and executable code.

### **Stack Allocation:**

- It is based on the idea of a control stack.
- Storage is organized as a stack and activation records are pushed and popped as activation begin and end respectively.
- Storage for the locals in each call of a procedure is contained in the activation record for that call. Thus locals are bound to fresh storage in each activation, because a new activation record is pushed onto the stack when a call is made.
- The values of locals are deleted when the activation ends. The values are lost because the storage for locals disappears when the activation record is popped.
- Suppose register ‘top’ marks the top of the stack. At run time, an activation record can be allocated and deallocated by incrementing and decrementing top respectively, by the size of the record.
- For example, if procedure ‘q’ has an activation record of size ‘a’ then top is incremented by ‘a’ just before the target code of q is executed. When control returns from q, top is decremented by ‘a’.

### **Calling Sequences:**

- Procedure calls are implemented by generating what are known as calling sequences in the target code.
- A call sequence allocates an activation record and enters information into its field.

- A return sequence restores the state of the machine so the calling procedure can continue execution.
- The code in a calling sequence is divided between the calling procedure (the caller) and the procedure it calls (the callee).
- There is no exact division of run-time tasks between the caller and callee – the source language, the target machine and the operating system impose requirements that may favor one solution over another.

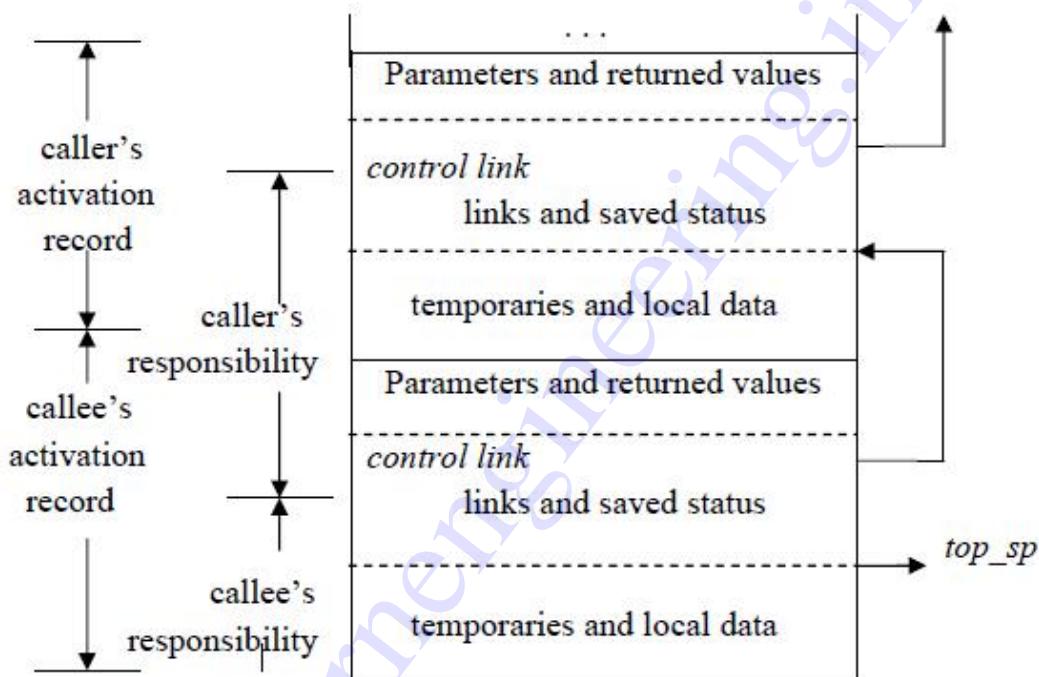
The call sequence is:

- (i) The caller evaluates actuals.
- (ii) The caller stores a return address and the old value of top\_sp into the callee's activation record. top\_sp is moved past the caller's local data and temporaries and the callee's parameter and status fields.
- (iii) The callee saves register values and other status information.
- (iv) The callee initializes its local data and begins execution.

A possible return sequence is :

- (i) The callee places a return value next to the activation record of the caller.
- (ii) Using the information in the status field, the callee restores top\_sp and other registers and branches to a return address in the caller's code.

- (iii) Although top\_sp has been decremented, the caller can copy the returned value into its own activation record and use it to evaluate an expression.

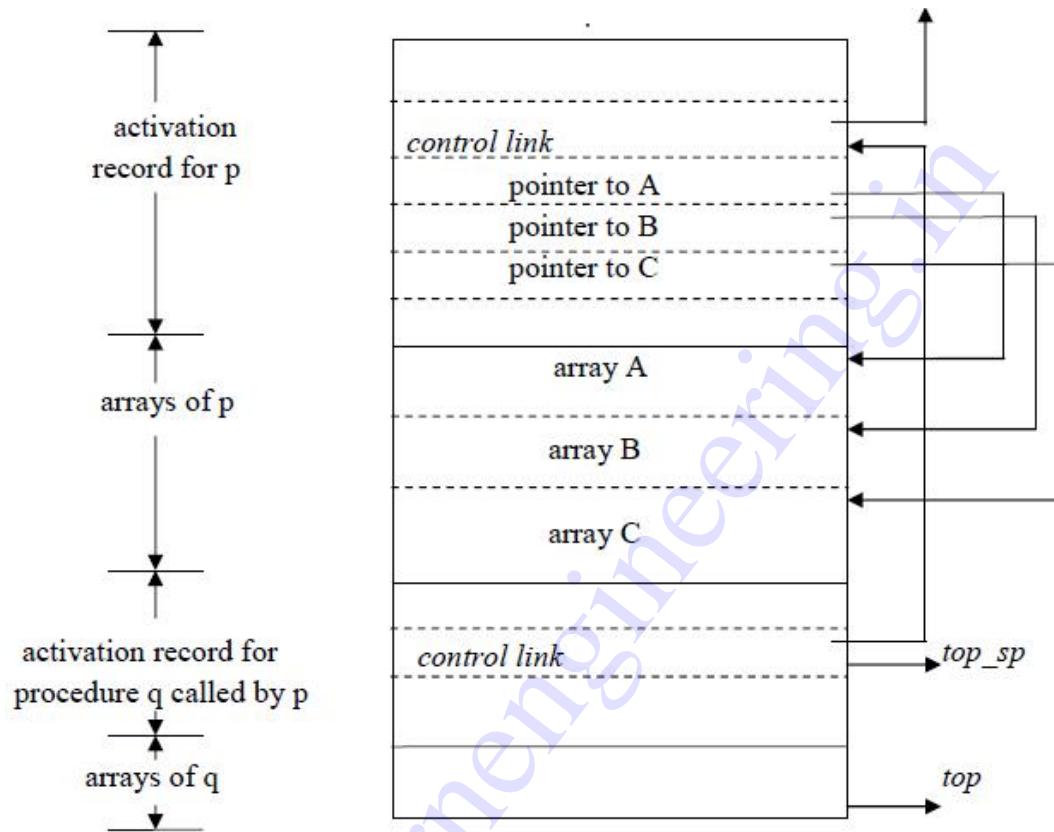


**Division of tasks between caller and callee.**

## Variable – Length Data

- A common strategy for handling variable-length data is shown in the figure below, where procedure p has three local arrays.
- The storage for these arrays is not part of the activation record for p; only a pointer to the beginning of each array appears in the activation record.

- The relative addresses of these pointers are known at compile time, so the target code can access array elements through the pointers.



### Access to dynamically allocated arrays

- The above figure is the activation record for a procedure q, called by p.
- The activation record for q begins after the arrays of p, and any variable-length arrays of q are located beyond that.
- Access to the data on the stack is through two pointers, *top* and *top-sp*.
- Here, *top* marks the actual top of stack; it points to 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.
- For consistency, we shall suppose that *top-sp* points to the end of the machine-status field.
- In above figure, *top-sp* points to the end of this field in the activation record for q.
- Within the field is a control-link to the previous value of *top\_sp* when control was in the calling activation of p.
- The code to reposition *top* and *top-sp* can be generated at compile time, using the sizes of fields in the activation records.
- When q returns, the new value of *top* is *top-sp* minus the length of the machine-status, and parameter fields in q's activation record.
- This length is known at compile time to the caller.
- After adjusting *top*, the new value of *top\_sp* can be copied from the control link of q.

### Dangling References:

- Whenever storage can be deallocated, the problem of dangling references arises.
- A dangling reference occurs when there is a reference to storage that has been deallocated.

## Limitations of Stack Allocation:

- The value of local names must be retained when an activation ends.
- A called activation outlives the caller.
  - This possibility cannot occur for those languages where activation trees correctly depict the flow of control between procedures.
  - In each of the above cases, the deallocation of activation records need not occur in a last-in-first-out fashion, so storage cannot be organized as a stack.

## Heap Allocation:

- 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 time the heap will consist of alternate areas that are free and in use.

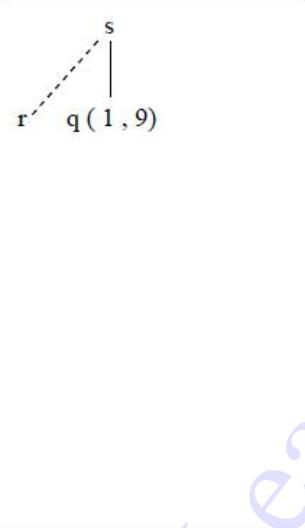
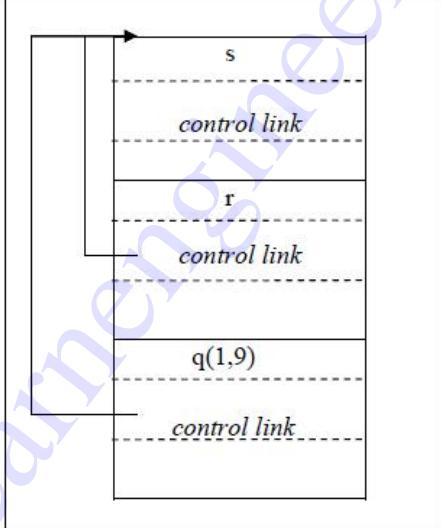
For efficiency reasons, heap allocation may be helpful to handle small activation records or records of a predictable size as a special case as follows:

- (i) For each size of interest, keep a linked list of free blocks of that size.
- (ii) If possible, fill a request for size  $s$  with a block of size  $s'$ , where  $s'$  is the smallest size greater than or equal to  $s$ . When the block

is eventually deallocated, it is returned to the linked list it came from.

- (iii) For larger blocks of storage use the heap manager.

These approach results in fast allocation and deallocation of small amounts of storage, since taking and returning a block from a linked list are efficient operations.

Position in the activation tree	Activation records in the heap	Remarks
		Retained activation record for r

## Part-A

- 1. Eliminate the left recursion from the following grammar**

$A \rightarrow Ac \mid Aad \mid bd \mid c$

[AU MAY/JUN 2007(Reg 2004)]

Solution:

$$\begin{aligned} A &\rightarrow bdA' \mid cA' \\ A' &\rightarrow cA' \mid adA' \mid \epsilon \end{aligned}$$

- 2. What are the disadvantages of operator precedence parsing?**

[AU MAY/JUN 2007(Reg 2004)]

The operator, like minus (unary and binary) has two different precedence. Hence it is hard to handle tokens like minus sign. This kind of parsing is applicable to only small class of grammars.

- 3. Draw the diagram of the general activation record and give the purpose of any two fields.**

[AU MAY/JUN 2007(Reg 2004)]

Returned value
Actual parameters
Optional control link
Optional access link
Saved machine status
Local data
Temporaries

- (i) Temporary values, such as those arising from the evaluation of expressions, are stored in the field for temporaries.
- (ii) The field for local data holds data that is local to an execution of a procedure.

**4. What is an ambiguous grammar? [AU NOV/DEC 2007(Reg 2004)]**

- A grammar that produces more than one parse tree for some sentence is said to be ambiguous.
- An ambiguous grammar is one that produces more than one leftmost derivation or more than one rightmost derivation for the same sentence.

**5. What is a predictive parser? [AU NOV/DEC 2007(Reg 2004)]**

A predictive parser is a special kind of recursive descent parser that involves **no backtracking**.

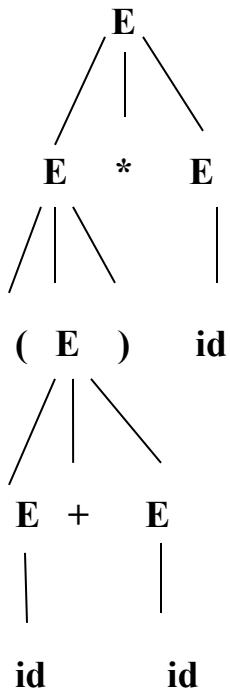
**6. Define Activation tree. [AU NOV/DEC 2007(Reg 2004)]**

Each execution of a procedure starts at the beginning of the procedure body and eventually return control to the point immediately following the place where the procedure was called. This can be depicted using an activation tree.

### 7. Construct a parse tree for $(a+b)^*c$ for the grammar

$$E \rightarrow E + E \mid E^* E \mid (E) \mid id$$

[AU APR/MAY 2008(Reg 2004)]



### 8. Eliminate left recursion for the following grammar:

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

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

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

[AU APR/MAY 2008(Reg 2004)]

Solution:

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

$$F \rightarrow .(E) \mid id$$

**9. Eliminate left recursion from the following grammar:**

$S \rightarrow (L) \mid a; L \rightarrow L, S \mid S.$

[AU NOV/DEC 2008(Reg 2004)]

Solution:

$$\begin{aligned}S &\rightarrow (L) \mid a \\L &\rightarrow SL' \\L' &\rightarrow, SL' \mid \epsilon\end{aligned}$$

**10. What is CLR?**

[AU NOV/DEC 2008(Reg 2004)]

CLR stands for canonical LR parser. It is a bottom up parsing technique. L stands for left to right scanning of the input and R stands for constricting a rightmost derivation in reverse.

**11. Derive the string and construct a syntax tree for the input string**

**ceaedbe using the grammar  $S \rightarrow SaA \mid A, A \rightarrow AbB \mid B, B \rightarrow cSd \mid e$**

[AU MAY/JUN 2009(Reg 2004)]

Solution:

$$\begin{aligned}S &\rightarrow A \\&\rightarrow B \\&\rightarrow cSd \\&\rightarrow cSaAd\end{aligned}$$

**12. List the factors to be considered for top-down parsing.**

[AU MAY/JUN 2009(Reg 2004)]

A top-down parse corresponds to a preorder traversal of the parse tree. A leftmost derivation is applied at each derivation step.

### 13. What is handle pruning?

[AU APR/MAY 2011, NOV/DEC 2011(Reg 2008)]

A rightmost derivation in reverse can be obtained by “**handle pruning**”.

### 14. What are the limitations of static allocation?

[AU NOV, DEC 2007, APR/MAY 2011(Reg 2008)]

- The size of the data object and constraints on its position in memory must be known at compile time.
- Recursive procedures are restricted, because all activations of a procedure use the same bindings for local names.
- Data structures cannot be created dynamically, since there is no mechanism for storage allocation at run time.

### 15. Mention the two rules for type checking.

[AU NOV/DEC 2011(Reg 2008)]

- i. Type synthesis
- ii. Type inference

### 16. What is a operator precedence parser?

A grammar is said to be operator precedence if it possess the following properties:

1. No production on the right side is  $\epsilon$ .
2. There should not be any production rule possessing two adjacent non terminals at the right hand side.

### 17. Mention the types of LR parser.

- SLR parser- simple LR parser
- LALR parser- lookahead LR parser
- Canonical LR parser

## 18. What are the problems with top down parsing?

The following are the problems associated with top down parsing:

- Backtracking
- Left recursion
- Left factoring
- Ambiguity

## 19.What is meant by viable prefixes?

The set of prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called viable prefixes. An equivalent definition of a viable prefix is that it is a prefix of a right sentential form that does not continue past the right end of the rightmost handle of that sentential form.

## 20. Define handle.

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

## **21.What is phrase level error recovery?**

Phrase level error recovery is implemented by filling in the blank entries in the predictive parsing table with pointers to error routines. These routines may change, insert, or delete symbols on the input and issue appropriate error messages. They may also pop from the stack.

## **22.List the properties of LR parser.**

- LR parsers can be constructed to recognize most of the programming languages for which the context free grammar can be written.
- The class of grammar that can be parsed by LR parser is a superset of class of grammars that can be parsed using predictive parsers.
- LR parsers work using non backtracking shift reduce technique yet it is efficient one.

## **23.Mention the basic issues in parsing.**

There are two important issues in parsing.

- Specification of syntax
- Representation of input after parsing.

## **24. Why are lexical and syntax analyzers separated out?**

Reasons for separating the analysis phase into lexical and syntax analyzers:

- Simpler design.
- Compiler efficiency is improved.
- Compiler portability is enhanced.

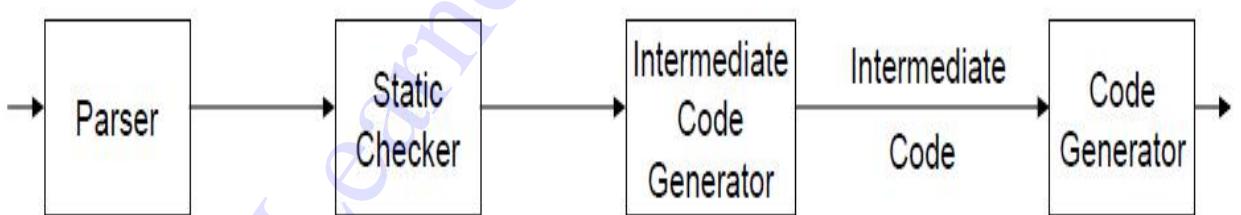
## INTERMEDIATE CODE GENERATION

### INTRODUCTION

- The front end of a compiler translates a source program into an intermediate representation.
- The back end of a compiler generates target code.
- Even though, the source program can be translated directly into target code, some benefits are there in using machine independent intermediate form.

### Benefits of intermediate representation

- Retargeting is facilitated. A compiler for a new machine can be created by attaching a back end for the new machine to an existing front end.
- A machine independent code optimizer can be applied to the intermediate representation.



**Position of intermediate code generator**

## INTERMEDIATE LANGUAGES

There are three kinds of intermediate representations.

1. Syntax Trees
2. Postfix Notation
3. Three Address code

Semantic rules for generating three-address code constructs are similar to those for constructing syntax trees or for generating postfix notations.

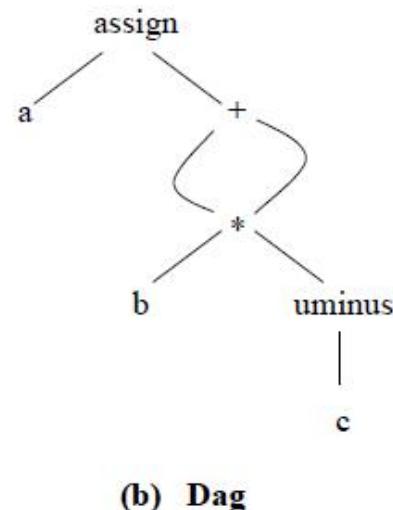
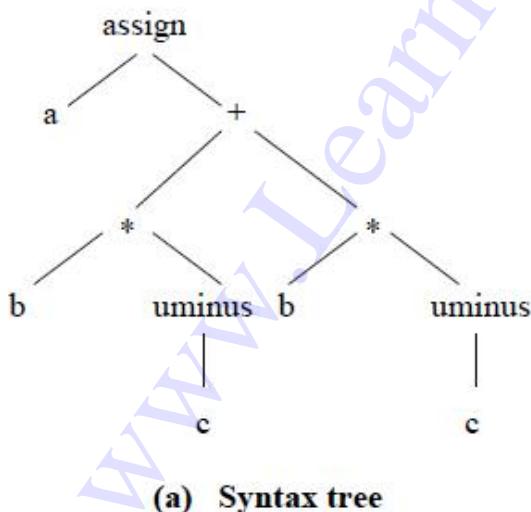
### Graphical Representations (Syntax tree and DAG)

- A syntax tree depicts the natural hierarchical structure of a source program.
- A DAG gives the same information but in a more compact way because common sub-expressions are identified.

Eg: Consider the assignment statement:

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

The syntax tree and DAG for the above statement are:



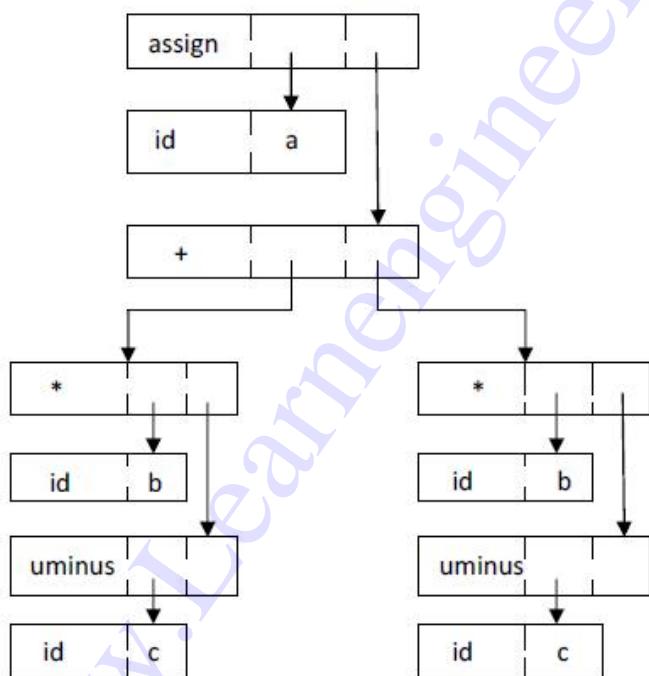
In DAG, common sub-expressions are identified and the repetition of the common sub-expression has been removed.

## Postfix Notation

- It is a linear representation of a syntax tree.
- It is a list of nodes of the tree in which a node appears immediately after its children.
- The postfix notation for the above syntax tree is:  

$$a \ b \ c \ uminus \ * \ b \ c \ uminus \ * \ + \ assign$$
- The edges in a syntax tree do not appear explicitly in postfix notation.
- They can be recovered from the order in which the nodes appear & the number of operands that the operator at a node expects.

### Two ways of representing a syntax tree:



(a) Linked List

0	id	b	
1	id	c	
2	uminus	1	
3	*	0	2
4	id	b	
5	id	c	
6	uminus	5	
7	*	4	6
8	+	3	7
9	id	a	
10	assign	9	8

(b) Array of Records

1. Syntax tree can be represented as a linked list
  - Each node is represented as a record with a field for its operator and additional fields for pointer to its children.
2. It also can be represented as an array of records.
  - Nodes are allocated from an array of records and the index or position of the node serves as the pointer to the node.
  - All the nodes in the syntax tree can be visited by following pointers, starting from root at position 10.

### Three – Address Code

It is a sequence of statements of the form

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

where x, y and z are names, constants or compiler-generated temporaries.

op stands for operator such as fixed or floating point arithmetic operator or a logical operator on boolean valued data.

#### Example:

$x + y * z$  is translated as

$t_1 := y * z$

$t_2 := x + t_1$

where  $t_1$  and  $t_2$  are compiler generated temporary names.

Three address code is a linear representation of a syntax tree or a DAG in which explicit names correspond to the interior nodes of the graph.

The three address code representation for the syntax tree and dag of the expression  $a := b * - c + b * - c$  is:

$t_1 := -c$

$t_2 := b * t_1$

$t_3 := -c$

$t_4 := b * t_3$

$t_5 := t_2 + t_4$

$a := t_5$

(a) Code for the syntax tree

$t_1 := -c$

$t_2 := b * t_1$

$t_5 := t_2 + t_4$

$a := t_5$

(b) Code for the dag

## Types of Three-Address Statements

The different types of three-address statements are :

1. Assignment statements of the form  $x := y \text{ op } z$ , where op is a binary arithmetic or logical operation, and x, y, and z are operands.
2. Assignment statements of the form  $x := \text{op } y$ , where op is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert an integer to a floating-point number.
3. Copy instructions of the form  $x := y$ , where x is assigned the value of y.
4. An unconditional jump goto L. The three-address instruction with label L is the next to be executed.
5. Conditional jumps such as if  $x \text{ relop } y \text{ goto } L$ . relop – relational operator such as  $<$ ,  $\leq$ ,  $>$ ,  $\geq$ ,  $=$ ,  $\neq$
6. Param x and call p, n for procedure calls and return y where y representing a returned value is optional.

param  $x_1$

param  $x_2$

.....

```
param xn
call p,n
```

7. Indexed assignments of the form  $x := y[i]$  and  $x[i] := y$ .
8. Address and pointer assignments of the form  $x := \&y$ ,  $x := *y$  and  $*x := y$

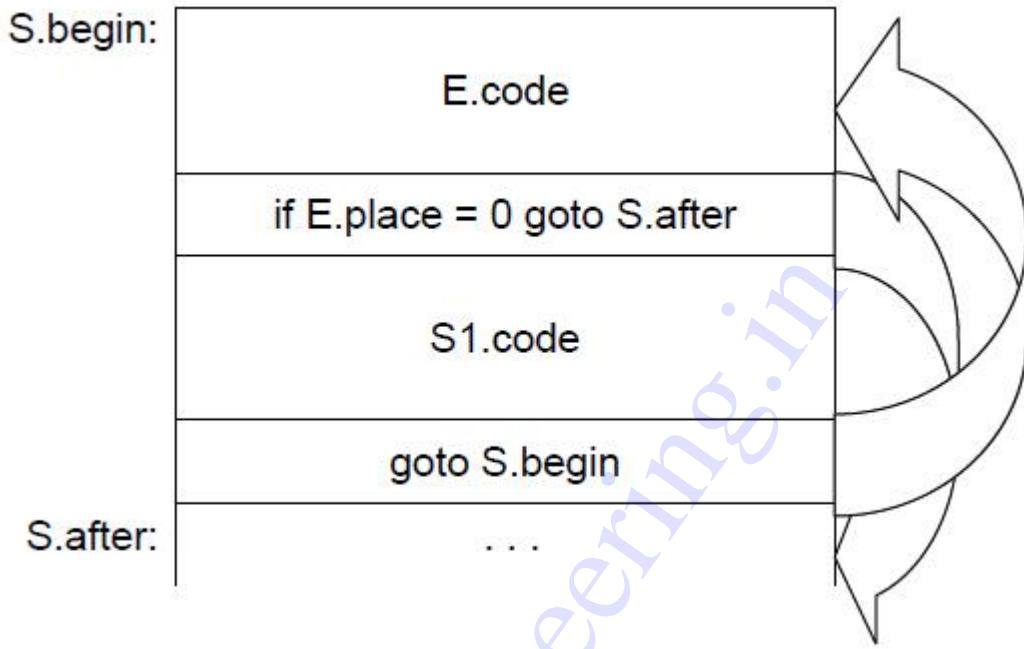
## Syntax-Directed Translation into Three-Address Code

- When three address code is generated, temporary names are made up for the interior nodes of a syntax tree.
- The value of non-terminal E on the left side of  $E \rightarrow E_1 + E_2$  will be computed into a new temporary t.
- In general, three address code for  $id := E$  consists of code to evaluate E into some temporary t, followed by the assignment  $id.place := t$ .
- The non-terminal E has two attributes:
  1. E.place - the name that will hold the value of E
  2. E.code - the sequence of three-address statements evaluating E.
- For convenience, we use the notation  $gen( x := y + z )$ . Expressions appearing instead of variables like x, y and z are evaluated when passed to gen and quoted operators or operands like '+' are taken literally.

## Syntax-directed definition to produce three-address code for assignments.

Production	Semantic Rules
$S \rightarrow id := E$	$S.code := E.code \parallel \text{gen}(id.place ':= E.place)$
$E \rightarrow E_1 + E_2$	$E.place := \text{newtemp};$ $E.code := E_1.code \parallel E_2.code \parallel$ $\text{gen}(E.place ':= E_1.place '+' E_2.place)$
$E \rightarrow E_1 * E_2$	$E.place := \text{newtemp};$ $E.code := E_1.code \parallel E_2.code \parallel$ $\text{gen}(E.place ':= E_1.place '*' E_2.place)$
$E \rightarrow - E_1$	$E.place := \text{newtemp};$ $E.code := E_1.code \parallel$ $\text{gen}(E.place ':= 'uminus' E_1.place)$
$E \rightarrow (E_1)$	$E.place := E_1.place;$ $E.code := E_1.code;$
$E \rightarrow id$	$E.place := id.place;$ $E.code := ''$

### Semantic rules generating code for a while statement



#### Production

$S \rightarrow \text{while } E \text{ do } S_1$

#### Semantic Rules

```

S.begin := newlabel;
S.after := newlabel;
S.code := gen(S.begin ':') || E.code ||
          gen( 'if' E.place '=' '0' 'goto' S.after) ||
          S1.code || gen( 'goto' S.begin) ||
          gen(S.after ':')
    
```

### Implementations of Three-Address Statements

There are three representations for implementing three address statements .They are :

1. **Quadruples**
2. **Triples**
3. **Indirect Triples**

## 1. Quadruples

- It is a record structure with 4 fields such as op, arg1, arg2 and result.
- The op field contains an internal code for the operator.
- The contents of arg1, arg2 and result are normally pointers to the symbol-table entries for the names represented by these fields.

**Eg 1:** Three-address statement

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

It is represented by placing y in arg1, z in arg2 and x in result.

**Eg 2:** Statements with unary operators like

$$x := -y \quad (\text{or}) \quad x := y \quad \text{do not use arg2}.$$

For the assignment statement  $a := b * - c + b * - c$ , the quadruples representation are shown as below:

	op	arg1	arg2	result
(0)	uminus	c		t <sub>1</sub>
(1)	*	b	t <sub>1</sub>	t <sub>2</sub>
(2)	uminus	c		t <sub>3</sub>
(3)	*	b	t <sub>3</sub>	t <sub>4</sub>
(4)	+	t <sub>2</sub>	t <sub>4</sub>	t <sub>5</sub>
(5)	:=	t <sub>5</sub>		a

## 2. Triples

- To avoid entering temporary names into symbol table, we might refer to a temporary value by position of the statement that computes it.
- In this, only three fields are used, so it is called as triples.
- The fields are op, arg1 and arg2.
- The fields arg1 and arg2 for the arguments of op are either pointers to the symbol table.

- For the assignment statement  $a := b * - c + b * - c$ , the triples representation are shown as below:

	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	(4)	a

- Parenthesized numbers represent pointers into triple structure, while symbol table pointers are represented by the names themselves.
- A ternary operators like  $x[i] := y$  requires two entries in the triple structure as shown in following figure:

$x[i] := y$

	op	arg1	arg2
(0)	[]=	x	i
(1)	assign	(0)	y

$x := y[i]$

	op	arg1	arg2
(0)	=[]	y	i
(1)	assign	x	(0)

### 3. Indirect Triples

- It is nothing but listing pointers to triples, rather than listing the triples themselves.
- The following figure shows the indirect triple representation of three address statement:

$$\mathbf{a := b * - c + b * - c}$$

	statement	op	arg1	arg2
(0)	(14)	uminus	c	
(1)	(15)	*	b	(14)
(2)	(16)	uminus	c	
(3)	(17)	*	b	(16)
(4)	(18)	+	(15)	(17)
(5)	(19)	:	(18)	a

- Indirect triples can save some space compared with quadruples if the same temporary value is used more than once.

## DECLARATIONS

- In the procedure, for each local name, we create a symbol-table entry with information like **type** and **relative address** of the storage for the name.
- The relative address consists of an offset from the base of the static data area or the field for local data in an activation record.

**Declarations in a procedure:-****Computing the types and relative addresses of declared names**

Production	Semantic Rules
$P \rightarrow D$	$\text{offset} := 0$
$D \rightarrow D ; D$	
$D \rightarrow \text{id} : T$	$\text{enter(id.name, T.type, offset);}$ $\text{offset} := \text{offset} + T.\text{width}$
$T \rightarrow \text{integer}$	$T.\text{type} := \text{integer};$ $T.\text{width} := 4$
$T \rightarrow \text{real}$	$T.\text{type} := \text{real}$ $T.\text{width} := 8$
$T \rightarrow \text{array[num] of } T_1$	$T.\text{type} := \text{array(num, } T_1.\text{type});$ $T.\text{width} := \text{num} * T_1.\text{width}$
$T \rightarrow \uparrow T_1$	$T.\text{type} := \text{pointer}(T_1.\text{type});$ $T.\text{width} := 4$

- Before the first declaration is considered, offset is set to 0.
- As each new name is seen, that name is entered in the symbol table with offset equal to the current value of offset, and offset is incremented by the width of the data object by that name.
- The procedure  $\text{enter(name, type, offset)}$  creates a symbol table entry for  $\text{name}$ , gives it  $\text{type}$  and relative address  $\text{offset}$  in its data area.
- The attribute  $\text{type}$  represents type expression constructed from the basic types integer and real by applying the type constructors pointer and array.
- If type expressions are represented by graphs, then attribute  $\text{type}$  might be a pointer to the node representing a type expression.
- The width of an integer is 4 and for real is 8.
- The width of an array is obtained by multiplying the width of each element by the number of elements in an array.
- The width of pointer is 4

### Keeping Track of Scope Information:-

- In a language with nested procedures, names local to each procedure can be assigned relative addresses using the above approach.
- When a nested procedure is seen, processing of declarations in the enclosing procedure is temporarily suspended.
- This approach will be illustrated by adding semantic rules to the following language:

$P \rightarrow D$

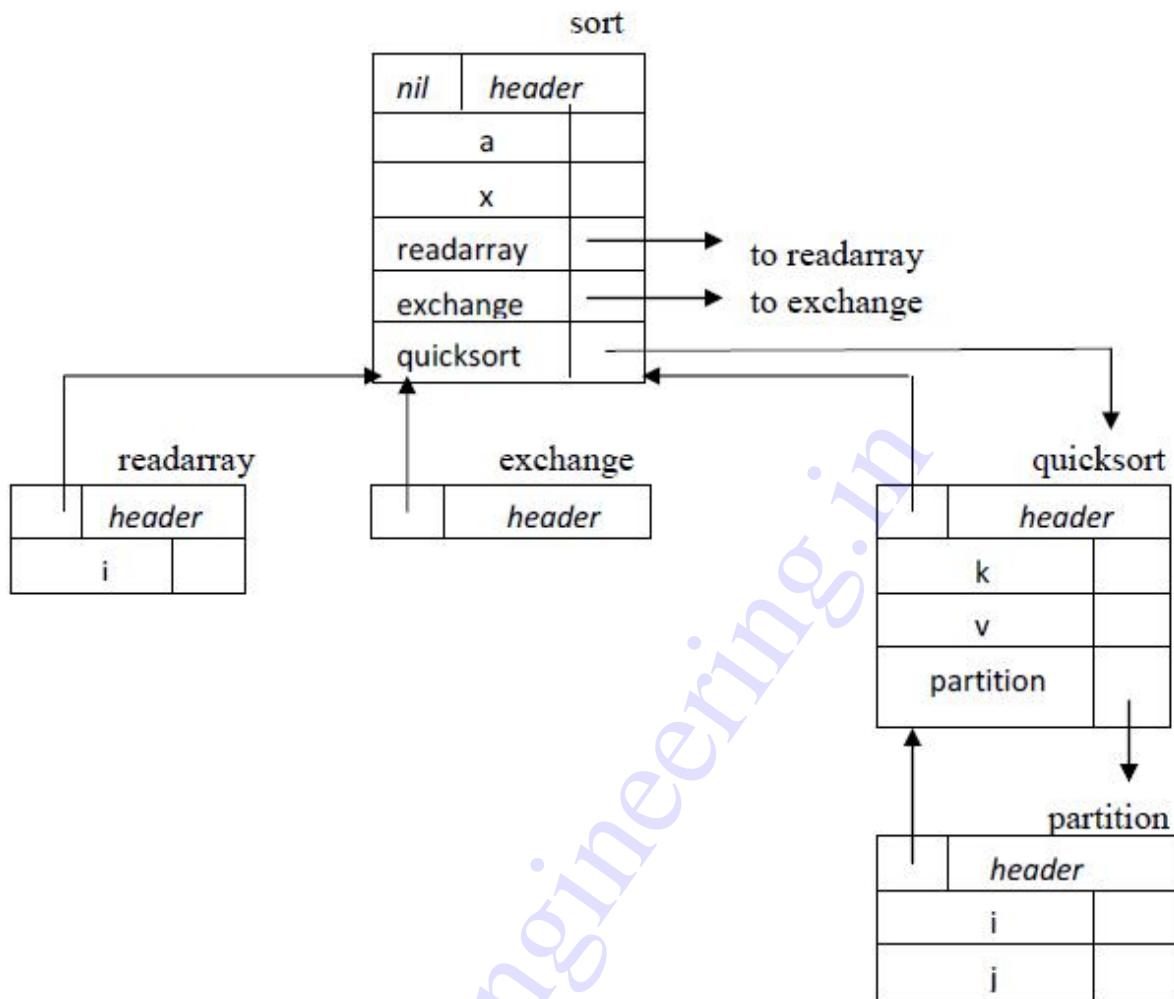
$D \rightarrow D ; D \mid id : T \mid proc\ id ; D ; S$

Where S – statements and T – types.

- If there is a nested procedure present, then there is a separate symbol table for each procedure.
- Symbol table is implemented by using linked list of entries for names.
- A new symbol table is created when a procedure declaration  $D \rightarrow proc id ; D_1 ; S$  is seen, and entries for the declarations in  $D_1$ , are created in the new table.
- The new table points back to the symbol table of the enclosing procedure.

#### **Example:**

- The symbol tables for procedures **readarray**, **exchange**, and **quicksort** point back to that for the containing procedure **sort**, consisting of the entire program.
- Since **partition** is declared within **quicksort**, its table points to that of **quicksort**.



### Symbol tables for Nested Procedures

The semantic rules are defined in terms of the following operations:

#### 1. *mktab(previous)*

- creates a new symbol table and returns a pointer to the new table.
- The argument *previous* points to a previously created symbol table and it is placed in a header for the new symbol table, along with additional information.

#### 2. *enter(table, name, type, offset)*

- creates a new entry for name *name* in the symbol table pointed to by *table*.

- Enter places type *type* and relative address *offset* in fields within the entry.

### 3. *addwidth(table, width)*

- records the cumulative width of all the entries in *table* in the header associated with this symbol table

### 4. *enterproc(table, name, newtable)*

- creates a new entry for procedure *name* in the symbol table pointed to by *table*.
- Argument *newtable* points to the symbol table for this procedure *name*.

## Processing Declarations in nested procedures

$P \rightarrow M\ D$

```
{addwidth(top(tblptr), top(offset));
pop(tblptr); pop(offset)}
```

$M \rightarrow \epsilon$

```
{ t := mkttable(nil);
push(t,tblptr); push(0,offset)}
```

$D \rightarrow D_1 ; D_2$

$D \rightarrow \text{proc id} ; N\ D_1 ; S$

```
{ t := top(tblptr);
addwidth(t.top(offset));
pop(tblptr); pop(offset);
enterproc(top(tblptr), id.name, t)}
{enter(top(tblptr),id.name,T.type,top(offset));
top(offset) := top(offset) + T.width }
```

$N \rightarrow \epsilon$

```
{ t := mkttable(top(tblptr));
push(t, tblptr); push(0, offset)}
```

## ASSIGNMENT STATEMENTS

- Expressions can be of type integer, real array and record.
- In the part of the translation of assignments into three-address code, we show how names can be looked up in the symbol table and how elements of arrays and records can be accessed.

### Names in the symbol table

- Names stood for pointers to their symbol-table entries.
- The lexeme for the name represented by **id** is given by attribute **id.name**.
- Operation **lookup(id.name)** checks if there is an entry for this occurrence of the name in the symbol table. If so, a pointer to the entry is returned; otherwise lookup will return nil to indicate that no entry was found.
- In the semantic actions of assignments, procedure **emit()** is used to emit three-address statements to an output file, rather than building up code attributes for non-terminals.

### Translation scheme to produce three-address code for assignments

$S \rightarrow id := D$	{p=lookup(id.name); if p!=nil then emit(P':=' E.place) else error }
$E \rightarrow E1 + E2$	{E.place := newtemp; emit(E.place ':=' E1.place '+' E2.place)}
$E \rightarrow E1 * E2$	{E.place := newtemp; emit(E.place ':=' E1.place '*' E2.place)}
$E \rightarrow -E1$	{E.place := newtemp; emit(E.place ':=' 'uminus' E1.place)}
$E \rightarrow ( E1 )$	{E.place := E1.place }
$E \rightarrow id$	{ p:= lookup(id.name); If p != nil then E.place := p else error}

## Reusing Temporary Names

- Temporaries can be reused by changing newtemp.
- newtemp – a new temporary name, each time a temporary is needed.
- Temporaries used to hold intermediate values in expression calculations tend to clutter up the symbol table and space has to be allotted to hold their values.
- The bulk of temporaries denoting data are generated during the syntax directed translation of expressions.
- Eg: The code generated by the rules  $E \rightarrow E_1 + E_2$  has the general form:
  - Evaluate  $E_1$  into  $t_1$
  - Evaluate  $E_2$  into  $t_2$
  - $t := t_1 + t_2$
- Keep a count  $c$ , initialized to zero.
- Whenever a temporary name is used as an operand, decrement  $c$  by 1.
- Whenever a new temporary name is generated, use  $\$c$  and increase  $c$  by 1.

**Eg:** Consider the assignment statement

$$x := a * b + c * d - e * f$$

The following figure shows the sequence of three-address statements that would be generated by semantic rules:

Statement	Value of $c$
	0
$\$0 := a * b$	1
$\$1 := c * d$	2
$\$0 := \$0 + \$1$	1
$\$1 := e * f$	2
$\$0 := \$0 - \$1$	1
$x := \$0$	0

**Three-address code with stacked temporaries**

## Addressing Array Elements

- Elements of the array can be accessed quickly if the elements are stored in a block of consecutive locations.
- If the width of each array element is  $w$ , then  $i^{\text{th}}$  element of array  $A$  begins in location

$$\text{base} + (i - \text{low}) * w \quad \text{--- (1)}$$

where  $\text{low}$  - lower bound on the subscript

base – relative address of the storage allocated for the array. i.e. base is the relative address of  $A[\text{low}]$

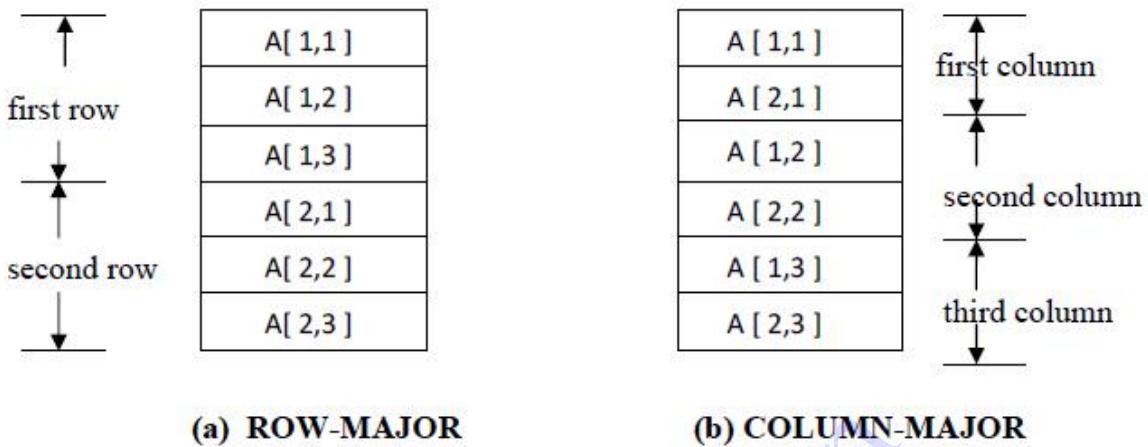
- The above expression can be partially evaluated at compile time if it is rewritten as:

$$i * w + (\text{base} - \text{low} * w) \quad \text{--- (2)}$$

The sub-expression  $c = \text{base} - \text{low} * w$  can be evaluated when the declaration of the array is seen. Assume that  $c$  is saved in symbol table entry for  $A$ , so the relative address of  $A[i]$  is obtained by simply adding  $i * w$  to  $c$ .

- A two-dimensional array is normally stored in one of two forms, either row-major (row-by-row) or column-major(column-by-column).

### Layouts for a two-dimensional array



- In the case of two-dimensional array stored in row-major form, the relative address of  $A[i_1, i_2]$  can be calculated by the formula

$$\text{base} + ((i_1 - \text{low}_1) * n_2 + i_2 - \text{low}_2) * w \quad \dots \quad (3)$$

Where  $\text{low}_1$  and  $\text{low}_2$  – lower bounds on the values of  $i_1$  and  $i_2$

$n_2$  – number of values that  $i_2$  can take.

i.e. if  $\text{high}_2$  is the upper bound on the value of  $i_2$ , then  $n_2 = \text{high}_2 - \text{low}_2 + 1$ .

Assuming that  $i_1$  and  $i_2$  are the only values that are not known at compile time, we can rewrite the above expression as:

$$((i_1 * n_2) + i_2) * w + (\text{base} - ((\text{low}_1 * n_2) + \text{low}_2) * w) \quad \dots \quad (4)$$

- The last term in this expression can be determined at compile time.
- We can generalize row- or column-major form to many dimensions.
- The generalization of row-major form is to store the elements in such a way that, as we scan down a block of storage, the rightmost subscripts appear to vary fastest, like the numbers on an odometer.
- The expression (4) generalizes to the following expression for the relative address of  $A[i_1, i_2, \dots, i_k]$

$$( ( \dots ((i_1 n_2 + i_2) n_3 + i_3) \dots ) n_k + i_k ) X w \quad \dots \quad (5)$$

$$+ \text{base} - ( ( \dots ((\text{low}_1 n_2 + \text{low}_2) n_3 + \text{low}_3) \dots ) n_k + \text{low}_k ) X w$$

- Since for all  $j$ ,  $nj = \text{high}_j - \text{low} + 1$  is assumed fixed, the term on the second line of (5) can be computed by the compiler and saved with the symbol-table entry for A.
- Column-major form generalizes to the opposite arrangement, with the leftmost subscripts varying fastest.
- Some languages permit the sizes of arrays to be specified dynamically, when a procedure is called at run-time.
- The formulas for accessing the elements of such arrays are the same as for fixed-size arrays, but the upper and lower limits are not known at compile time.
- The chief problem in generating code for array references is to relate the computation of (5) to a grammar for array references.
- Array references can be permitted in assignments if non-terminal L with the following productions is allowed .

$L \rightarrow id [Elist] | id$

$Elist \rightarrow Elist, E | E$

- In order that the various dimensional limits  $nj$  of the array be available as we group index expressions into an Elist, it is useful to rewrite the productions as

$L \rightarrow Elist] | id$

$Elist \rightarrow Elist, E | id [ E$

- That is, the array name is attached to the leftmost index expression rather than being joined to Elist when an L is formed.
- These productions allow a pointer to the symbol-table entry for the array name to be passed as a synthesized attribute array of Elist.
- We also use Elist.ndim to record the number of dimensions (index expressions) in the Elist. The function limit(array, j) returns  $nj$ , the number of elements along the  $j$  th dimension of the array whose symbol-table entry is pointed to by array.
- Finally, Elist.place denotes the temporary holding a value computed from index expressions in Elist.

- An Elist that produces the first m indices of a k-dimensional array reference  $A[i_1, i_2, \dots, i_k]$  will generate three-address code to compute

$$( \dots ((i_1 n_2 + i_2)n_3 + i_3) \dots )n_m + i_m \quad \text{-----(6)}$$

using the recurrence

$$e_1 = i_1$$

$$e_m = e_{m-1} \times n_m + i_m \quad \text{-----(7)}$$

- Thus, when  $m = k$ , a multiplication by the width w is all that will be needed to compute the term on the first line of (5).
- Note that the  $i_j$ 's here may really be values of expressions and code to evaluate those expressions will be interspersed with code to compute (6).
- An l-value L will have two attributes. L.place and L.offset.
- In the case that L is a simple name.
- L.place will be a pointer to the symbol-table entry for that name, and L.offset will be null, indicating that the l-value is a simple name rather than an array reference.
- The non-terminal E has the same translation E.place.

### Translation Scheme for Addressing Array Elements

1.  $S \rightarrow L := E$
2.  $E \rightarrow E + E$
3.  $E \rightarrow (E)$
4.  $E \rightarrow L$
5.  $L \rightarrow Elist ]$
6.  $L \rightarrow id$
7.  $Elist \rightarrow Elist, E$
8.  $Elist \rightarrow id [ E$

As in the case of expressions without array references, the three-address code itself is produced by the *emit* procedure invoked in the semantic actions. We generate a normal assignment if  $L$  is a simple name, and an indexed assignment into the location denoted by  $L$  otherwise:

(1)  $S \rightarrow L := E \{ \text{if } L.\text{offset} = \text{null} \text{ then /* } L \text{ is a simple id */}$   
 $\quad \quad \quad \text{emit}(L.\text{place} ':= E.\text{place});$   
 $\quad \quad \quad \text{else emit}(L.\text{place}'[L.\text{offset}]' ':= E.\text{place})$

The code for arithmetic expressions is exactly the same as above.

(2)  $E \rightarrow E1 + E2 \{ E.\text{place} := \text{newtemp};$   
 $\quad \quad \quad \text{emit}(E.\text{place} ':= E1.\text{place} '+' E2.\text{place}) \}$

(3)  $E \rightarrow ( E1 ) \quad \{ E.\text{place} := E1.\text{place} \}$

When an array reference  $L$  is reduced to  $E$ , we want the r-value of  $L$ . Therefore we use indexing to obtain the contents of the location  $L.\text{place}[L.\text{offset}]$ :

(4)  $E \rightarrow L \quad \{ \text{if } L.\text{offset} = \text{null} \text{ then /* } L \text{ is a simple id */}$   
 $\quad \quad \quad E.\text{place} := L.\text{place}$   
 $\quad \quad \quad \text{else begin}$   
 $\quad \quad \quad \quad \quad E.\text{place} := \text{newtemp};$   
 $\quad \quad \quad \quad \quad \text{emit}(E.\text{place} ':= L.\text{place}'[L.\text{offset}'])$   
 $\quad \quad \quad \text{end} \}$

Below,  $L.\text{offset}$  is a new temporary representing the first term of (5); function  $\text{width}(Elist.\text{array})$  returns  $w$  in (5).  $L.\text{place}$  represents the second term of (5), returned by the function  $c(Elist.\text{array})$ .

(5)  $L \rightarrow Elist ] \quad \{ L.\text{place} := \text{newtemp};$   
 $\quad \quad \quad L.\text{offset} := \text{newtemp};$   
 $\quad \quad \quad \text{emit}(L.\text{place} ':= c(Elist.\text{array}))$

*emit(L.offset ':= ' Elist.place '\*width(Elist.array)))*

A null offset indicates a simple name.

(6)  $L \rightarrow \text{id} \quad \{ L.place := \text{id.place};$   
 $L.offset := \text{null}$

When the next index expression is seen, we apply the recurrence (7). In the following action, *Elist1.place* corresponds to *e(m-1)* in (7) and *Elist.place* to *em*. Note that if *Elist1* has *m-1* components, then *Elist* on the left side of the production has *m* components.

(7)  $Elist \rightarrow Elist1, E \quad \{ t := \text{newtemp};$   
 $m := Elist1.ndim + l;$   
 $\text{emit}(t := ' Elist1.place '*' \text{limit}(Elist1.array, m));$   
 $\text{emit}(t := ' t '+' E.place);$   
 $Elist.array := Elist1.array;$   
 $Elist.place := t;$   
 $Elist.ndim := m \}$

*E.place* holds both the value of the expression *E* and the value of (6) for *m=1*.

(8)  $Elist \rightarrow \text{id} [ E \quad \{ Elist.array := \text{id.place};$   
 $Elist.place := E.place;$   
 $Elist.ndim := l \})$

For the assignment example  $x := A[y,z]$  we have the annotated parse tree shown below under the assumption that the base address for the array *A* is 0.

## BOOLEAN EXPRESSIONS

- **Use of Boolean expressions**

1. They are used to compute logical values
  2. Used as conditional expressions in statements that alter flow of control ( such as if-then, if-then-else, or while-do statement).
- Boolean expressions are composed of Boolean operators such as **and**, **or** & **not** .
  - Boolean operators are applied to elements that are Boolean variables or relational expressions.
  - Relational expressions are of the form :

$E_1 \text{ relop } E_2$

where  $E_1, E_2$  are arithmetic expressions

rellop – relational operators such as  $<$ ,  $\leq$ ,  $>$ ,  $\geq$ ,  $=$ ,  $\neq$

- Boolean expressions are generated by the following grammar:

$$E \rightarrow E \text{ or } E \mid E \text{ and } E \mid \text{not } E \mid (E) \mid \text{id relop id} \mid \text{true} \mid \text{false}$$

- **Order of Precedence:** not , and , or

### Methods of Translating Boolean Expressions

- There are two principal methods to represent the value of a Boolean expression:
  1. Numerical Representation
  2. Flow of Control Statements

### Numerical Representation:

- In this representation of Boolean expression, 1 is used to denote true and 0 to denote false.
- Boolean expressions are evaluated from left to right.
- Eg: The translation for **a or b and not c** is as follows:  
The three-address sequence is given as:

$t_1 := \text{not } c$

$t_2 := b \text{ and } t_1$

$t_3 := a \text{ or } t_2$

Consider a relational expression: **if a < b then 1 else 0**, which can be translated into three-address code sequence as: (arbitrarily statement numbers start at 100)

100: if a < b goto 103

101: t := 0

102: goto 104

103: t := 1

104:

### Translation Scheme for producing three-address code for Boolean expressions (using Numerical Representation)

In this scheme, we are going to have the following assumptions:

- **emit** – places three-address statements into an output file in the right format.
- **nextstat** – gives the index of the next three-address statement in the output sequence
- **emit** increments **nextstat** after producing each three-address statement.

$$E \rightarrow E_1 \text{ or } E_2 \quad \{ \text{ E.place := newtemp; }$$

emit(E.place ‘:=’ E<sub>1</sub>.place ‘or’ E<sub>2</sub>.place)}

E → E<sub>1</sub> **and** E<sub>2</sub> { E.place := newtemp;  
emit(E.place ‘:=’ E<sub>1</sub>.place ‘and’ E<sub>2</sub>.place)}

E → **not** E<sub>1</sub> { E.place := newtemp;  
emit(E.place ‘:=’ ‘not’ E<sub>1</sub>.place)}

E → ( E<sub>1</sub> ) { E.place := E<sub>1</sub>.place}

E → **id**<sub>1</sub> **relop** **id**<sub>2</sub> { E.place := newtemp;  
emit(‘if’ **id**<sub>1</sub>.place **relop.op** **id**<sub>2</sub>.place  
‘goto’ nextstat + 3);  
emit(E.place ‘:=’ ‘0’);  
emit(‘goto’ nextstat + 2);  
emit(E.place ‘:=’ ‘1’) }

E → **true** { E.place := newtemp;  
emit(E.place ‘:=’ ‘1’) }

E → **false** { E.place := newtemp;  
emit(E.place ‘:=’ ‘0’) }

### Translation Scheme for producing three-address code for Boolean expressions (using Numerical Representation)

**Example:**

Three- address code for the expression:

**a < b or c < d and e < f**

100: if a < b goto 103	107: t <sub>2</sub> := 1
101: t <sub>1</sub> := 0	108: if e < f goto 111
102: goto 104	109: t <sub>3</sub> := 0
103: t <sub>1</sub> := 1	110: goto 112
104: if c < d goto 107	111: t <sub>3</sub> := 1
105: t <sub>2</sub> := 0	112: t <sub>4</sub> := t <sub>2</sub> and t <sub>3</sub>
106: goto 108	113: t <sub>5</sub> := t <sub>1</sub> or t <sub>4</sub>

### Short-Circuit Code:

- We can also translate a boolean expressions into three-address code without generating code for any of the Boolean operators and without having the code necessarily evaluate the entire expression.
- This style of evaluation is called “**short-circuit**” or “**jumping**” code.
- It is possible to evaluate boolean expressions without generating code for boolean operators and, or & not if we represent the value of an expression by a position in the code sequence.

### Flow-of-Control Statements:

- Consider the translation of boolean expression into three-address code in the context of if-then, if-then-else and while-do statements .

$S \rightarrow \text{if } E \text{ then } S_1$

|  $\text{if } E \text{ then } S_1 \text{ else } S_2$

|  $\text{while } E \text{ do } S_1$

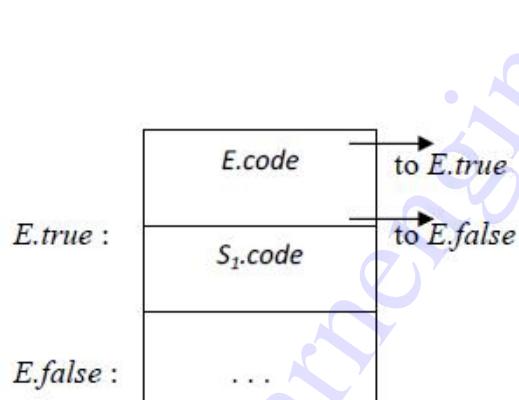
- In each of these productions, E is the Boolean expression to be translated.

- Two labels are associated with Boolean expression E, they are :

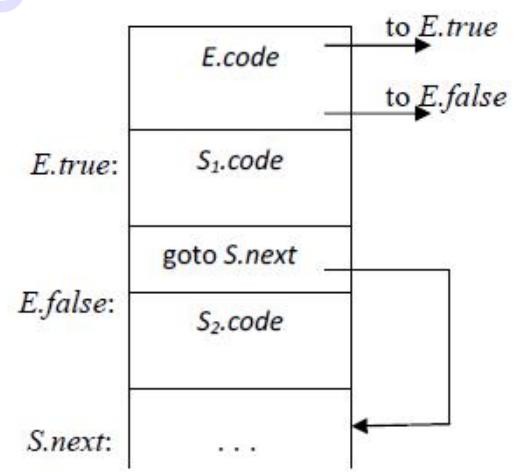
➤ E.true – the label to which control flows if E is true.

➤ E.false – the label to which control flows if E is false.

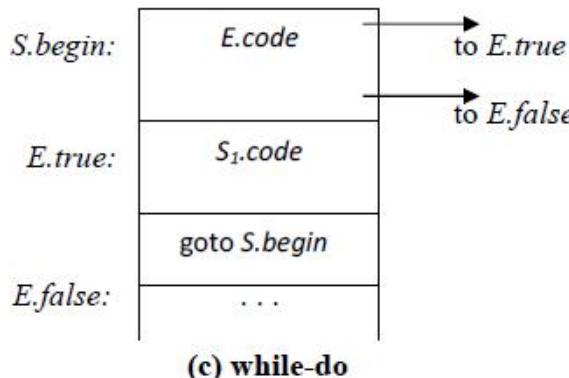
- In translating the if-then statement,  $S \rightarrow \text{if } E \text{ then } S_1$ , a new label E.true is created and attached to the first three-address instruction generated for the statement  $S_1$  as shown in the figure (a)



(a) if-then



(b) if-then-else



(c) while-do

- In translating the if-then-else statement,  $S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$ , the code for the Boolean expression  $E$  has jumps out of it to the first instruction of the code for  $S_1$  if  $E$  is true, and to the first instruction of the code for  $S_2$  if  $E$  is false as illustrated in figure (b)

As with the if-then statement, an inherited attribute  $S.next$  gives the label of the three-address instruction to be execute next after executing the code for  $S$ .

- The code for  $S \rightarrow \text{while } E \text{ do } S_1$  is formed as shown in the figure (c)
- A new label  $S.begin$  is created and attached to the first instruction generated for  $E$ .
- Another new label  $E.true$  is attached to the first instruction for  $S_1$ .
- The code for  $E$  generates a jump to this label if  $E$  is true, a jump to  $S.next$  if  $E$  is false; again, we set  $E.false$  to be  $S.next$ .
- After the code for  $S_1$  we place the instruction  $\text{goto } S.begin$ , which causes a jump back to the beginning of the code for boolean expression.

## A syntax-directed definition for flow-of-control statements

<b>Production</b>	<b>Semantic Rules</b>
$S \rightarrow \text{if } E \text{ then } S_1$	<pre> E.true := newlabel; E.false := S.next; S<sub>1</sub>.next := S.next; S.code := E.code    gen(E.true':')   S<sub>1</sub>.code </pre>
$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$	<pre> E.true := newlabel; E.false := newlabel; S<sub>1</sub>.next := S.next; S<sub>2</sub>.next := S.next; S.code := E.code    gen(E.true':')    S<sub>1</sub>.code    gen('goto' S.next)    gen( E.false ':')    S<sub>2</sub>.code </pre>
$S \rightarrow \text{while } E \text{ do } S_1$	<pre> S.begin := newlabel; E.true := newlabel; E.false := S.next; S<sub>1</sub>.next := S.begin; S.code := gen(S.begin ':')    E.code    gen(E.true ':')    S<sub>1</sub>.code    gen('goto' S.begin) </pre>

The syntax-directed definition for  $S \rightarrow \text{if } E \text{ then } S_1$  appears in the above table. The code for E generates a jump to E.true if E is true and a jump to S.next if E is false. Therefore we set E.false to S.next.

**Syntax-directed definition to produce three-address code for booleans.**

## Production

## Semantic Rules

$E \rightarrow E_1 \text{ or } E_2$

$E_1.\text{true} := E.\text{true};$   
 $E_1.\text{false} := \text{newlabel};$   
 $E_2.\text{true} := E.\text{true};$   
 $E_2.\text{false} := E.\text{false};$   
 $E.\text{code} := E_1.\text{code} \parallel \text{gen}(E_1.\text{false} ::) \parallel E_2.\text{code}$

$E \rightarrow E_1 \text{ and } E_2$

$E_1.\text{true} := \text{newlabel};$   
 $E_1.\text{false} := E.\text{false};$   
 $E_2.\text{true} := E.\text{true};$   
 $E_2.\text{false} := E.\text{false};$   
 $E.\text{code} := E_1.\text{code} \parallel \text{gen}(E_1.\text{true} ::) \parallel E_2.\text{code}$

$E \rightarrow \text{not } E_1$

$E_1.\text{true} := E.\text{fasle};$   
 $E_1.\text{false} := E.\text{true};$   
 $E.\text{code} := E_1.\text{code}$

$E \rightarrow ( E_1 )$

$E_1.\text{true} := E.\text{true};$   
 $E_1.\text{false} := E.\text{false};$   
 $E.\text{code} := E_1.\text{code}$

$E \rightarrow \text{id}_1 \text{ relop id}_2$   
 $\text{'goto' } E.\text{true}) \parallel$

$E.\text{code} := \text{gen}(\text{'if' } \text{id}_1.\text{place } \text{relop}.\text{op } \text{id}_2..\text{place}$   
 $\text{gen}(\text{'goto' } E.\text{false}))$

$E \rightarrow \text{true}$

$E.\text{code} := \text{gen}(\text{'goto' } E.\text{true})$

$E \rightarrow \text{false}$

$E.\text{code} := \text{gen}(\text{'goto' } E.\text{false})$

## CASE STATEMENTS

- The ‘switch’ or ‘case’ statement is available in a variety of languages.
- The syntax for switch-case statement:

```
switch expression  
begin  
    case value : statement  
    case value : statement  
    .....  
    case value : statement  
    default : statement  
end.
```

- The intended translation of a switch is code to:
  - i. Evaluate the expression.
  - ii. Find which case value is the same as the value of the evaluate expression.
  - iii. Execute the statement associated with the value found.
- Step (ii) is an n-way branch, which can be implemented in one of several ways:
- If the number of cases not too great, then we can use sequence of **conditional goto's** each of which tests for an individual value and transfers to the code for the corresponding statement.

### Syntax-Directed Translation of Case Statements:

Consider the following switch statement:

**switch E**

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

With a syntax-directed translation scheme, it is convenient to translate this case case statement into intermediate code that has the following form:

```
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:
```

- To translate into the above form, when we see the keyword **switch**, we generate two labels **test** and **next** and a new temporary t.
- Then parse the expression E, we generate code to evaluate E into t.
- After processing E, we generate the jump **goto test**.

### Another Translation of a Case Statement:

code to evaluate E into t  
if t = V<sub>1</sub> goto L<sub>1</sub>  
code for S<sub>1</sub>  
goto next  
L<sub>1</sub>: if t = V<sub>2</sub> goto L<sub>2</sub>  
code for S<sub>2</sub>  
goto next  
L<sub>2</sub>:  
.....  
L<sub>n-2</sub>: if t = V<sub>n-1</sub> goto L<sub>n-1</sub>  
code for S<sub>n-1</sub>  
goto next  
L<sub>n-1</sub>: code for S<sub>n</sub>  
next:

## BACK PATCHING

- The easiest way to implement the syntax-directed definition is to use two passes.
  - Construct a parse tree (syntax tree) for the input
  - Walk the tree in depth-first order, computing the translations given in the definition.
- The main problem with generating code for Boolean expression and flow-of-control statements in a single pass is that during one-single pass we may not know the labels that control must go to at the time the jump statements are generated.
- To get around this problem by generating a series of branching statements with the targets of jumps.
- Each statement will be put on a list of goto statements whose labels will be filled in when the proper label can be determined.
- This subsequent filling in of labels is called “**Backpatching**”.

### Functions to manipulate list of labels:

#### 1. *makelist(i)*

- Creates a new list containing only i, an index into the array of quadruples.
- It returns a pointer to the list it has made.

#### 2. *merge(p<sub>1</sub>, p<sub>2</sub>)*

- Concatenates the lists pointed to by p<sub>1</sub> and p<sub>2</sub>, and returns a pointer to the concatenated list.

#### 3. *backpatch(p, i)*

- Inserts ‘i’ as the target label for each of the statements on the list pointed to by p.

### Boolean Expressions:

- Construct a translation scheme suitable for producing quadruples for Boolean expressions during bottom-up parsing.
- Insert a marker non-terminal M into the grammar to cause a semantic action to pick up, at appropriate times, the index of the next quadruple to be generated.

Eg: Consider the following grammar:

- (1)  $E \rightarrow E_1 \text{ or } M E_2$
- (2)      |  $E_1 \text{ and } M E_2$
- (3)      |  $\text{not } E_1$
- (4)      |  $( E_1 )$
- (5)      |  $\text{id}_1 \text{ relop } \text{id}_2$
- (6)      |  $\text{true}$
- (7)      |  $\text{false}$
- (8)  $M \rightarrow \epsilon$

- Synthesized attributes *truelist* and *falselist* of non-terminal E are used to generate jumping code for Boolean expressions.
- As code is generated for E, jumps to true and false exits are left incomplete, with the label field unfilled.
- These incomplete jumps are placed on lists pointed to by *E.truelist* and *E.falselist* as appropriate.

Consider the production  $E \rightarrow E_1 \text{ and } M E_2$ .

- If  $E_1$  is false, then E is also false, so the statements on  $E_1.\text{falselist}$  become part of E.*falselist*.
- If  $E_1$  is true, however we must test  $E_2$ , so the target for the statements  $E_1.\text{truelist}$  must be the beginning of the code generated or  $E_2$

- This target is obtained using the marker nonterminal M.
- Attribute M.quad records the number of the first statement of E<sub>2</sub>.code.
- With the production M → ε, the semantic action associated  
 $\{ M.\text{quad} := \text{nextquad} \}$
- The variable nextquad holds the index of the next quadruple to follow.

### The translation scheme is as follows:

- (1) E → E<sub>1</sub> **or** M E<sub>2</sub>      { backpatch(E<sub>1</sub>.falselist, M.quad);  
 E.truelist := merge(E<sub>1</sub>.truelist, E<sub>2</sub>.truelist);  
 E.falselist := E<sub>2</sub>.falselist }
- (2) E → E<sub>1</sub> **and** M E<sub>2</sub>      { backpatch(E<sub>1</sub>.truelist, M.quad);  
 E.truelist := E<sub>2</sub>.truelist ;  
 E.falselist := merge(E<sub>1</sub>.falselist, E<sub>2</sub>.falselist) }
- (3) E → **not** E<sub>1</sub>      { E.truelist := E<sub>1</sub>.falselist ;  
 E.falselist := E<sub>1</sub>.truelist }
- (4) E → ( E<sub>1</sub> )      { E.truelist := E<sub>1</sub>.truelist ;  
 E.falselist := E<sub>1</sub>.falselist }
- (5) E → **id**<sub>1</sub> **relop** **id**<sub>2</sub>      { E.truelist := makelist(nextquad);  
 E.falselist := makelist(nextquad + 1 );  
 emit( ‘if **id**<sub>1</sub>.place **relop**.op **id**<sub>2</sub>.place ‘goto’)  
 emit(‘goto’) }
- (6) E → **true**      { E.truelist := makelist(nextquad);  
 emit(‘goto’) }

(7)  $E \rightarrow \text{false}$

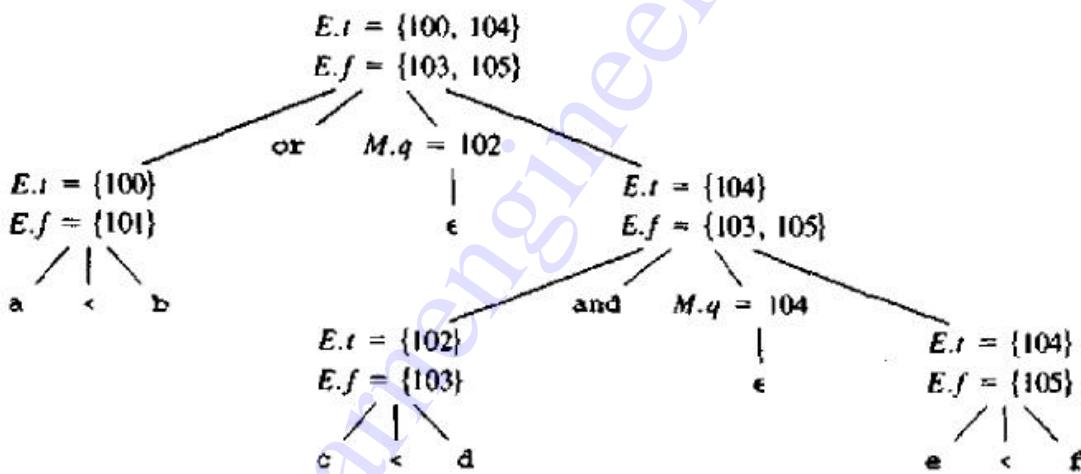
{ E.falselist := makelist(nextquad);  
emit('goto') }

(8)  $M \rightarrow \epsilon$

{ M.quad := nextquad }

Example: Consider the expression  $a < b$  or  $c < d$  and  $e < f$ .

An annotated parse tree for the above Boolean expression is shown in the figure below:



- In response to the reduction of  $a < b$  to E by above production (5), the two quadruples:

100: if a < b goto

101: goto

are generated. The marker nonterminal M in the production  $E \rightarrow E_1 \text{ or } M E_2$  records the value of nextquad, which at this time is 102.

- The reduction of  $c < d$  to E by production (5) generates the quadruples:

102: if c < d goto

103: goto

we have now seen  $E_1$  in the production  $E \rightarrow E_1 \text{ and } M E_2$ . The marker non-terminal in this production records the current value of nextquad, which is now 104.

- Reducing  $e < f$  into  $E$  by production (5) generates

104: if  $e < f$  goto

105: goto

We now reduce by  $E \rightarrow E_1 \text{ and } M E_2$ . The corresponding semantic action calls backpatch( $\{102\}, 104$ ) where  $\{102\}$  as argument denotes a pointer to the list containing only 102, that list being the one pointed to by  $E_1.\text{truelist}$ . This call to backpatch fills in 104 in statement 102.

The six statements generated so far are thus:

100: if  $a < b$  goto

101: goto

102: if  $c < d$  goto 104

103: goto

104: if  $e < f$  goto

105: goto

The semantic action associated with the final reduction by  $E \rightarrow E_1 \text{ or } M E_2$  calls backpatch( $\{101\}, 102$ ) which leaves the statements looking like:

100: if  $a < b$  goto

101: goto 102

102: if  $c < d$  goto 104

103: goto

104: if  $e < f$  goto

105: goto

The entire expression is true if and only if the goto's of statements 100 or 104 are reached, and is false if and only if the goto's of statements 103 or 105 are reached. These instructions will have their targets filled in later in the compilation, when it is seen what must be done depending on the truth or falsehood of the expression.

## PROCEDURE CALL

- Procedures are important in programming constructs.
- Procedure is imperative for a compiler to generate good code for procedure calls and returns.
- Run-time routines that handle procedure argument passing, calls and returns are part of the run-time support package.
- Consider a grammar for a simple procedure call statement:

$$S \rightarrow \text{call id} ( Elist )$$

$$Elist \rightarrow Elist, E$$

$$Elist \rightarrow E$$

### Calling Sequences:

#### Actions that take place when a procedure call occurs:

- Space must be allocated for the activation record of the called procedure.
- The arguments of the called procedure must be evaluated and made available to the called procedure in a known place.
- Environment pointers must be established to enable the called procedure to access data in enclosing blocks.
- The state of calling procedure must be saved, so it can resume execution after the call and also the return address is stored.

- Return address is usually the location of the instruction that follows the call in the calling procedure.
- Finally, a jump to the beginning of the called procedure must be generated.

#### **Actions that take place when a procedure returns :**

- If the called procedure is a function, the result must be stored in a known place.
- Activation record of calling procedure must be stored.
- A jump to calling procedure's return address must be generated.

**S → call id (Elist)**

```
{ for each item p on queue do  
    emit('param' p);  
    emit('call' id.place) }
```

\* Code for S is the code for Elist.

\* Elist evaluates the arguments followed by a param p statement for each argument, followed by a call statement.

\* The values of argument- evaluating statements are saved in a queue.

## Part-A

### 1. Write the properties of intermediate language.

[AU MAY/JUN 2007(Reg 2004)]

- 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.

### 1. What is back patching?

[AU MAY/JUN 2007(Reg 2004)],APR/MAY 2008(Reg 2004),MAY/JUN 2009(Reg 2004)]

Backpatching is the activity of filling up unspecified information of labels using appropriate semantic actions in during the code generation process.

### 2. What are the notations used to represent intermediate languages?

[AU NOV/DEC 2007(Reg 2004)]

There are three types of notations used to represent intermediate languages:

- Syntax tree
- Postfix Notation
- Three address code

### 3. Give the ways of representing three address statements.

[AU NOV/DEC 2007(Reg 2004)]

Three address statements are represented as

- Quadruples

- Triples
- Indirect Triples

**4. Translate the expression  $a-(b+c)$  into three address code.**

[AU NOV/DEC 2008(Reg 2004)]

Solution:  $t1 := b+c$   
 $t2 := a-t1$

**5. List out the three functions that are used to manipulate the list of labels in back patching.**

[AU NOV/DEC 2008(Reg 2004)]

- **mklist(i)** creates the new list. The index i is passed as an argument to this function where I is an index to the array of quadruple.
- **merge\_list(p1,p2)** this function concatenates two lists pointed by p1 and p2. It returns the pointer to the concatenated list.
- **backpatch(p,i)** inserts i as target label for the statement pointed by pointer p.

**6. Why is it necessary to generate intermediate code instead of generating target program itself?**

[AU MAY/JUN 2009(Reg 2004)]

- Retargeting is facilitated. A compiler for a new machine can be created by attaching a back end for the new machine to an existing front end.
- A machine independent code optimizer can be applied to the intermediate representation.

**7. List out the benefits of using machine-independent intermediate forms.**

[AU APR/MAY 2011(Reg 2008)]

- Retargeting is facilitated. A compiler for a new machine can be created by attaching a back end for the new machine to an existing front end.

- A machine independent code optimizer can be applied to the intermediate representation.

**8. What is a syntax tree? Draw the syntax tree for the following statement**

**a := b \* -c + b \* -c**

[AU APR/MAY 2011(Reg 2008),NOV/DEC 2011(Reg 2008)]

**9. What are the types of three address statements?**

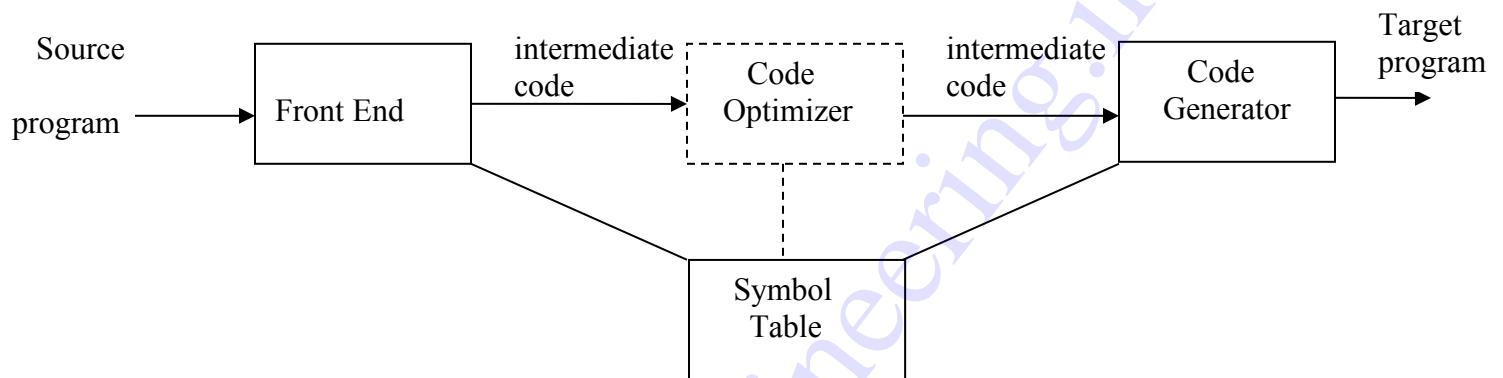
[AU NOV/DEC 2011(Reg 2008)]

1.  $x := y \text{ op } z$
2.  $x := \text{op } y$
3.  $x := y$
4. goto L
5. if  $x \text{ relop } y \text{ goto } L$
6. param  $x_1$   
param  $x_2$   
.....  
param  $x_n$   
call p,n
7.  $x := y[i] \text{ and } x[i] := y$
8.  $x := &y, x := *y, *x := y$

## CODE GENERATION

### INTRODUCTION

- The final phase of a compiler is code generation.
- It takes as input an intermediate representation of the source program and produces as output an equivalent target program.



- If a code optimizer is present before code generator within a compiler then the compiler is called “Optimizing” compilers.
- A phase tries to transform the intermediate code into a form from which more efficient target code can be produced.
- The output code must be correct and of high quality, meaning that it should make effective use of the resources of the target machine. The code generator itself should run efficiently.

## ISSUES IN THE DESIGN OF A CODE GENERATOR

While the details are dependent on the target language and the operating system, issues such as

- (i) Input to Code Generator
- (ii) Target Programs
- (iii) Memory Management
- (iv) Instruction Selection
- (v) Register Allocation
- (vi) Choice of Evaluation Order
- (vii) Approaches to Code Generation.

### **(i) Input to Code Generator**

- The input to the code generator consists of the intermediate representation of the source program produced by the front end, together with information in the symbol table that is used to determine the run-time addresses of the data objects denoted by the names in the intermediate representation.
- There are several choices for intermediate languages:
  - Linear representations such as postfix notation
  - Three-address representations such as quadruples
  - Virtual machine representations such as stack machine code
  - Graphical representations such as trees and DAGs.

### **(ii) Target Programs**

- The output of the code generator is the target program.
- The output of code generator may be in different form:

- Absolute machine language
- Relocatable machine language
- Assembly language.

### **Absolute Machine Language:**

- It can be placed in a fixed location in memory and immediately executed.
- Small program can be compiled and executed quickly.

### **Relocatable Machine Language:**

- Producing it as output allows subprograms to be compiled separately.
- A set of relocatable object modules can be linked together and loaded for execution by a linking loader.

### **Assembly Language:**

- It makes the process of code generation easier.
- It can generate symbolic instructions and use the macro facilities of the assembler to help generate code.

### **(iii) Memory Management**

- Mapping of names in the source program to addresses of data objects in run-time memory is done by the front-end and code generator.
- From the symbol table information, a relative address can be determined for the name in a data area for the procedure.
- If the machine code is being generated, labels in the three-address statements have to be converted to addresses of instructions.

#### (iv) Instruction Selection

- The instruction set of the target machine determines the difficulty of instruction selection.
- The uniformity , completeness of the instruction set , instruction speed and machine idioms are important factors.

**Example:** Every three-address statement of the form

**x := y + z,**      where x, y and z are statically allocated, can be translated into code sequence

```
MOV y, R0      /* load y into register R0 */
ADD z, R0      /* add z to R0 */
MOV R0, x      /* store R0 into x */
```

But , this kind of statement-by-statement code generation often produces poor code. For example, the sequence of statements

a := b + c

d := a + e

would be translated into

```
MOV b, R0
ADD c, R0
MOV R0, a
MOV a, R0
ADD e, R0
MOV R0, d
```

The fourth statement is redundant and so is the third if ‘a’ is not subsequently used.

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

#### (v) Register Allocation

- Instructions involving register operands are usually shorter and faster than those involving operands in memory. Therefore, efficient utilization of registers is particularly important in generating good code.
- The use of registers is subdivided into two sub-problems:
  - (a) During register allocation, the set of variables that will reside in registers at a point in the program are selected.
  - (b) During a subsequent register assignment phase. We pick the specific register that a variable will reside in.

#### **(vi) Choice of Evaluation Order**

- The order in which computations are performed can affect the efficiency of the target code.
- Some computation orders require fewer registers to hold intermediate results.
- Picking a best order is difficult problem.

#### **(vii) Approaches to Code Generation**

- The important criterion for a code generator is that it produce correct code.
- Correctness takes on special significance because of the number of special cases that a code generator might face.
- Given the premium on correctness, designing a code generator so it can be easily implemented, tested and maintained is an important design goal.

## **THE TARGET MACHINE**

- 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 four bytes to a word and ‘ $n$ ’ general purpose registers R0, R1,…….Rn-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 following op-code:

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.
- The address modes together with their assembly-language forms and associated costs are as follows:

<b>MODE</b>	<b>FORM</b>	<b>ADDRESS</b>	<b>ADDED COST</b>
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

where contents(a) denotes the contents of the register or memory address represented by a.

- A memory location M or a register R represents itself when used as a source or destination.

For example, the instruction

**MOV R0, M**

stores the contents of R0 into memory location M.

- An address offset  $c$  from the value in register R is written as  $c(R)$ . Thus,

**MOV 4(R0), M**

stores the value  $\text{contents}(4 + \text{contents}(R0))$  into memory location M.

- Indirect versions of the last two modes are indicated by prefix \*. Thus,

**MOV \*4(R0), M**

stores the value  $\text{contents}(\text{contents}(4 + \text{contents}(R0)))$  into memory location M.

- A final address mode allows the source to be a constant:

MODE	FORM	CONSTANT	ADDED COST
Literal	$\#c$	$c$	1

Thus the instruction **MOV #1, R0** loads the constant 1 into register R0.

### Instruction Costs:

- For simplicity, we take the cost of an instruction to be one plus the costs associated with the source and destination address modes.
- This cost corresponds to the length (in words) of the instruction.

- Address modes involving registers have cost zero, while those with a memory location or literal in them have cost one, because such operands have to be stored with the instruction.
- If space is important, then minimize the instruction length.
- The time taken to fetch an instruction from memory exceeds the time spent executing the instruction.
- Therefore, by minimizing the instruction length leads to minimize the time taken to perform the instruction as well.

**Some examples follow:**

- The instruction `MOV R0, R1` copies the contents of register R0 into register R1. This instruction has cost one, since it occupies only one word of memory.
- The (store) instruction `MOV R5, M` copies the contents of register R5 into memory location M. This instruction has cost two, since the address of memory location M is in the word following the instruction.
- The instruction `ADD #1, R3` adds the constant 1 to the content of register R3, and has cost two, since the constant 1 must appear in the next word following the instruction.
- The instruction `SUB 4( R0 ), * 12( R1 )` stores the value  $\text{contents}(\text{contents}(12 + \text{contents}(R1))) - \text{contents}(\text{contents}(4 + R0))$  into the destination `*12( R1 )`. The cost of this instruction is three, since the constants 4 and 12 are stored in the next two words following the instruction.

Some of the difficulties in generating code for this machine can be seen by considering what code to generate for a three-address statement of the form

$a := b + c$  where  $b$  and  $c$  are simple variables in distinct memory locations denoted by these names.

This statement can be implemented by many different instruction sequences.

1. MOV b, R0  
ADD c, R0      cost = 6  
MOV R0, a

2. MOV b, a  
ADD c, a      cost = 6

Assuming R0, R1 and R2 contain the addresses of  $a$ ,  $b$  and  $c$  respectively.

3. MOV \*R1, \*R0  
ADD \*R2, \*R0      cost = 2

Assuming R1 and R2 contain the values of  $b$  and  $c$  respectively and that the value of  $b$  is not needed after the assignment, we can use:

4. ADD R2, R1  
MOV R1, a      cost = 3

In order to generate good code for this 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.
- Storage for names local to the procedure also appears in the activation record.
- There are two standard storage allocation strategies.
  - (i) **Static Allocation** - the position of an activation record in memory is fixed at compile time.
  - (ii) **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.
- An activation record for a procedure has fields to hold parameters, results, machine-status information, local data, temporaries and like.
- Since run-time allocation and deallocation of activation records occurs as part of the procedure call and return sequences, we focus on the following three-address statements:
  - (i) call
  - (ii) return
  - (iii) halt
  - (iv) action – a placeholder for other statements.
- The size and layout of activation records are communicated to the code generator via the information about names that is in the symbol table.
- Assume that the run-time memory is divided into areas for code, static data and a stack.

## Static Allocation

Consider the code needed to implement static allocation.

- A call statement in the intermediate code is implemented by a sequence of two target-machine instructions.
- A MOV instruction saves the return address, and a GOTO transfers control to the target code for the called procedure:

```
MOV      #here + 20, callee.static_area  
GOTO    callee.code_area
```

- The attributes callee.static\_area and callee.code\_area are constants referring to the address of the activation record and the first instruction for the called procedure respectively.
- The source #here + 20 in the MOV instruction is the literal return address; it is the address of the instruction following the GOTO instruction.
- The code for a procedure ends with a return to the calling procedure, except that the first procedure has no caller, so its final instruction is HALT, which presumably returns control to the operating system.
- A return from procedure callee is implemented by GOTO \*callee.static\_area which transfers control to the address saved at the beginning of the activation record.

## Stack Allocation

- Static allocation can become stack allocation by using relative addresses for storage in activation records.
- The position of the record for an activation of a procedure is not known until run time.
- In stack allocation, this position is usually stored in a register, so words in the activation record can be accessed as offsets from the value in this register.

- Relative addresses in an activation record can be taken as offsets from any known position in the activation record.
- Use positive offsets by maintaining in a register SP a pointer to the beginning of the activation record on top of the stack.
- When a procedure call occurs, the calling procedure increments SP and transfers control to the called procedure.
- After control returns to the caller, it decrements SP, thereby deallocating the activation record of the called procedure.

The code for the first procedure initializes the stack by setting SP to the start of the stack area in memory:

```
MOV #stackstart, SP      /* initialize the stack */
code for the first procedure
HALT                  /* terminate execution */
```

A procedure call sequence increments SP, saves the return address, and transfers control to the called procedure:

```
ADD #caller.recordsize, SP
MOV #here + 16, *SP      /* save return address */
GOTO callee.code_area
```

The attribute caller.recordsize represents the size of an activation record, so the ADD instruction leaves SP pointing to the beginning of the next activation record. The source #here + 16 in the MOV instruction is the address of the instruction following the GOTO; it is saved in the address pointed to by SP.

The return sequence consists of two parts.

- (i) The called procedure transfers control to the return address using

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

The reason for using  $*0(SP)$  in the GOTO instruction is that we need two levels of indirection:

$0(SP)$  is the address of the first word in the activation record and

$*0(SP)$  is the return address saved there.

The second part of the return sequence is in the caller, which decrements SP, thereby restoring SP to its previous value. (i.e) After the subtraction SP points to the beginning of the activation record of the caller:

```
SUB #caller.recordsize, SP.
```

## Run-Time Addresses for Names

- The storage allocation strategies and the layout of local data in an activation record for a procedure determine how the storage for names is accessed.
- Assume that a name in a three-address statement is really a pointer to a symbol table entry for the name.
- This approach has a significant advantage:
- It makes the compiler more portable, since the front end need not be changed even if the compiler is moved to different machine where a different run-time organization is needed.
- Generating the specific sequence of access steps while generating intermediate code can be of significant advantage in an optimizing compiler.

Consider the simple three-address copy statement  $x := 0$ .

After the declarations in a procedure are processed, suppose the symbol-table entry for  $x$  contains a relative address 12 for  $x$ .

First consider the case in which  $x$  is in a statically allocated area beginning at address static. Then the actual run-time address of  $x$  is static + 12.

Although the compiler can determine the value of static + 12 at compile time, the position of the static area may not be known when intermediate code to access the name is generated.

The assignment  $x := 0$  then translates into

$\text{static}[12] := 0$

If the static area starts at address 100, the target code for this statement is

$\text{MOV } \#0, 112$ .

Suppose in Pascal language, a display is used to access nonlocal names and the display is kept in registers, and that  $x$  is local to an active procedure whose display pointer is in register R3.

Then we may translate the copy  $x := 0$  into three-address statements

$t1 := 12 + R3$   
 $*t1 := 0$

where  $t1$  – address of  $x$ .

This sequence can be implemented by the single machine instruction

$\text{MOV } \#0, 12(R3)$

The value in register R3 cannot be determined at compile time.

## BASIC BLOCKS AND FLOW GRAPHS

### **Flow Graph:**

A graph representation of three-address statements, called a flow graph, is useful for understanding code-generation algorithms.

Nodes in the flow graph represent computations and the edges represent the flow of control.

### **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 halt or possibility of branching except at the end.

The following sequence of three-address statements forms a basic block:

```
t1 := a * a  
t2 := a * b  
t3 := 2 * t2  
t4 := t1 + t3  
t5 := b * b  
t6 := t4 + t5
```

A **name in a basic block** is said to be **live** at a given point if its value is used after that point in the program, perhaps in another basic block.

## **Algorithm for partitioning a sequence of three-address statements 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. Determine the set of leaders, the first statements of basic blocks.

#### **Rules to follow:**

- (i) The first statement is a leader.
  - (ii) Any statement that is the target of a conditional or unconditional goto is a leader.
  - (iii) 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.

**Example:**

Consider the fragment of source code shown below. It computes the 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
```

**Step 1:** Generation of three-address code

- (3) prod := 0
- (4) i := 1
- (5) t<sub>1</sub> := 4 \* i
- (6) t<sub>2</sub> := a [t<sub>1</sub>] /\* computes a[i] \*/
- (7) t<sub>3</sub> := 4 \* i
- (8) t<sub>4</sub> := b [t<sub>3</sub>] /\* computes b[i] \*/
- (9) t<sub>5</sub> := t<sub>2</sub> \* t<sub>4</sub>
- (10) t<sub>6</sub> := prod + t<sub>5</sub>
- (11) prod := t<sub>6</sub>
- (12) t<sub>7</sub> := i + 1
- (13) i := t<sub>7</sub>
- (14) if i <= 20 goto (3)

Apply the above algorithm to the above three-address statements.

Statement 1 is a leader by rule (i) and statement 3 is a leader by rule (ii), since the last statement can jump to it. By rule (iii) the statement following 12 is a leader. Therefore, statements 1 and 2 form a basic block. The remaining of the program beginning with statement 3 forms a second basic block.

```
prod := 0  
i := 1
```

B<sub>1</sub>

```
t1 := 4 * i  
t2 := a [t1]  
t3 := 4 * i  
t4 := b [t3]  
t5 := t2 * t4  
t6 := prod + t5  
prod := t6  
t7 := i + 1  
i := t7  
if i <= 20 goto (3)
```

B<sub>2</sub>

## Transformations on Basic Blocks

There are two important classes of local transformations that can be applied to basic blocks.

1. Structure-Preserving Transformations
2. Algebraic Transformations.

### 1. Structure-Preserving Transformations

The primary structure-preserving transformations on basic blocks are:

- (i) Common Sub-Expression Elimination
- (ii) Dead-Code Elimination
- (iii) Renaming Of Temporary Variables
- (iv) Interchange Of Two Independent Adjacent Statements.

#### (i) Common Sub-expression Elimination:

Consider the basic block:

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

The second and fourth statements compute the same expression, and hence this basic block may be transformed into the equivalent block

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

**Note:**

- Although the first and third statements appear to have the same expression on the right, the second statement redefines b.
- Therefore, the value of b in the third statement is different from the value of b in the first.

**(ii) 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.

**(iii) Renaming Temporary Variables:**

- Suppose there is a statement  $t := b + c$  where t is a temporary.
- If we change this statement to  $u := b + c$ , where u is a new temporary variable, and change all uses of this instance of t to u, then the value of the basic block is not changed.

**(iv) Interchange of Statements:**

Suppose a block consists of two adjacent statements:

$$\begin{aligned}t_1 &:= b + c \\t_2 &:= x + y\end{aligned}$$

Then interchange the two statements without affecting the value of the block if and only if neither x nor y is  $t_1$  and neither b nor c is  $t_2$ .

## 2. Algebraic Transformations

- Algebraic transformations can be used to change the set of expressions computed by a basic block into an algebraically equivalent set.
- The useful ones are those that simplify expressions or replace expensive operations by cheaper ones.

### Example:

Statements such as

$$x := x + 0 \text{ (or) } x := x * 1$$

can be eliminated from a basic block without changing the set of expressions it computes.

The exponentiation operator in the statement

$$x := y^{**} 2$$

usually requires a function call to implement.

Using an algebraic transformation, this statement can be replaced by the cheaper, but equivalent statement

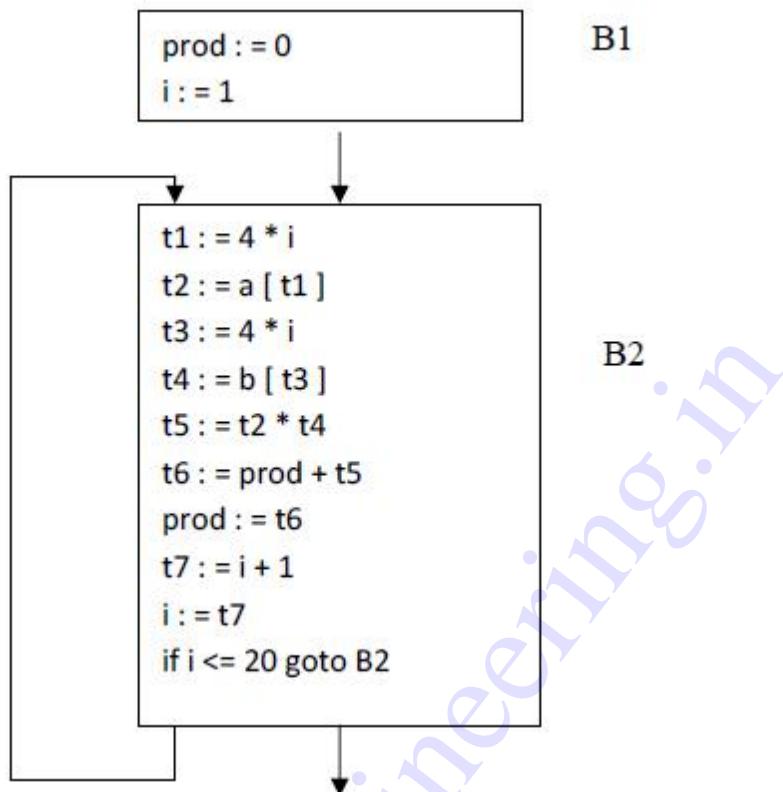
$$x := y * y.$$

## FLOW GRAPHS

- Add the flow-of-control information to the set of basic blocks making up a program by constructing a directed graph called a flow graph.
- The nodes of the flow graph are the basic blocks.
- One node is distinguished as initial - it is the block whose leader is the first statement.
- There is a directed edge from block  $B_1$  to block  $B_2$  if  $B_2$  can immediately follow  $B_1$  in some execution sequence, if
  - there is a conditional or unconditional jump from the last statement of  $B_1$  to the first statement of  $B_2$  or
  - $B_2$  immediately follows  $B_1$  in the order of the program, and  $B_1$  does not end in an unconditional jump.

(i) We say that  $B_1$  is a predecessor of  $B_2$  and  $B_2$  is a successor of  $B_1$

The flow graph for the above three address statement is shown below.  $B_1$  is the initial node. Note that in the last statement, the jump to statement 3 has been replaced by an equivalent jump to the beginning of block  $B_2$ .



**Flow graph for the above example**

## Representation of Basic Blocks

Basic blocks can be represented by a variety of data structures.

- a. After partitioning the three-address statements, each basic block can be represented by a record consisting of a count of the number of quadruples in the block, followed by a pointer to the leader of the block, and by the lists of predecessors and successors of the block.
- b. An alternative is to make a linked list of the quadruples in each block.

## Loops

- A loop is a collection of nodes in a flow graph such that
  1. All nodes in the collection are ***strongly connected*** - from any node in the loop to any other, there is a path of length one or more, wholly within the loop and
  2. The collection of nodes has a ***unique entry*** – a node in the loop such that the only way to reach a node of the loop from a node outside the loop is to first go through the entry.
- A loop that contains no other loops is called an ***inner loop***.

## NEXT-USE INFORMATION

- Collect next-use information about names in basic blocks.
- If the name in a register is no longer needed, then the register can be assigned to some other name.

## Computing Next Uses

- To determine for each three-address statement  $x := y \text{ op } z$  what the next uses of  $x$ ,  $y$  and  $z$  are.

### Algorithm to determine next uses

1. Make a backward pass over each basic block.
2. Scan a stream of three-address statements to find the ends of basic blocks.
3. Having found the end of a basic block, scan backwards to the beginning, recording (in the symbol table) for each name  $x$  whether  $x$  has a next use in the block and if not, whether it is live on exit from that block.
4. If no live-variable analysis has been done, assume all non-temporary variables are live on exit, to be conservative.
5. If the algorithm generating intermediate code or optimizing the code permit certain temporaries to be used across blocks, these must be considered live.

### Example:

Suppose we reach three-address statement    **i: x := y op z**

in our backward scan, then do the following:

1. Attach to statement i the information currently found in the symbol table regarding the next use and liveliness 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 the next uses of y and z to i.

### Storage for Temporary Names

- It may be useful in an optimizing compiler to create a distinct name each time a temporary is needed, space has to be allocated to hold the values of these temporaries.
- Pack two temporaries into the same location if they are not live simultaneously.
- Since all temporaries are defined and used within basic blocks, next-use information can be applied to pack temporaries.

### Example: The six temporaries in the basic block

```
t1 := a * a  
t2 := a * b  
t3 := 2 * t2  
t4 := t1 + t3  
t5 := b * b  
t6 := t4 + t5
```

can be packed into two locations.

These locations correspond to t1 and t2 in :

```
t1 := a * a  
t2 := a * b  
t2 := 2 * t2  
t1 := t1 + t2  
t2 := b * b  
t1 := t1 + t2
```

## A SIMPLE CODE GENERATOR

- The code generation strategy generates target code for a sequence of three-address statement.

### Register and Address Descriptors

The code generation algorithm uses descriptors to keep track of register contents and addresses for names.

#### 1. Register Descriptor.

- A register descriptor keeps track of what is currently in each register.
- It is consulted whenever a new register is needed. Assume that initially the register descriptor shows that all registers are empty.

#### 2. Address Descriptor.

- An address descriptor keeps track of the location where the current value of the name can be found at run time.
- The location might be a register , a stack location, a memory address or some set of these, since when copied a value also stays where it was.

## A Code-Generation Algorithm

- The code-generation algorithm takes a sequence of three-address statements constituting a basic block as input.
- It performs the following actions for each three-address statement of the form

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

1. Invoke a function “*getreg*” to determine the location L where the result of the computation  $y \text{ op } z$  should be stored. L will usually be a register, but it could also be a memory location.
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  $\text{MOV } y', L$  to place a copy of y in L.
3. Generate the instruction  $\text{OP } z', L$  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 L is a register, update its descriptor to indicate that it contains the value of x and remove x from all other register descriptors.
4. If the current values of y and/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 no longer will contain y and/or z respectively.

## The Function *getreg*

The function *getreg* returns the location L to hold the value of x for the assignment  
 $x := y \text{ op } z$ .

A simple, easy-to-implement scheme based on next-use information as follows:

1. If the name y is in a register that holds the value of no other names and y is not live and has no next use after execution of  $x := y \text{ op } z$ , then return the register of y for L. Update the address descriptor of y to indicate that y is no longer in L.
2. Failing (1), return an empty register for L if there is one.
3. Failing (2), if x has a next use in the block or op is an operator such as indexing, that requires a register, find an occupied register R. Store the value of R into a memory location M, update the address descriptor for M and return R.
4. If x is not used in the block, or no suitable occupied register can be found, select the memory location of x as L.

**Example:** The assignment statement

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

might be translated into following three-address code sequence:

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

with d live at the end.

The code-generation algorithm will produce the code sequence as shown below:

<b>STATEMENTS</b>	<b>CODE GENERATED</b>	<b>REGISTER DESCRIPTOR</b>	<b>ADDRESS DESCRIPTOR</b>
		Registers 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.

## Generating Code for Other Types of Statements

Indexing and pointer operations in three-address statements are handled in the same manner as binary operations. The following table shows the code sequences generated for the indexed assignment statements  $a := b [ i ]$  and  $a [ i ] := b$ , assuming  $b$  is statically allocated.

Statement	i in Register $R_i$		i in Memory $M_i$		i in Stack	
	Code	Cost	Code	Cost	Code	Cost
$a := b [ i ]$	MOV $b(R_i), R$	2	MOV $M_i, R$ MOV $b(R), R$	4	MOV $S_i(A), R$ MOV $b(R), R$	4
$a [ i ] := b$	MOV $b, a(R_i)$	3	MOV $M_i, R$ MOV $b, a(R)$	5	MOV $S_i(A), R$ MOV $b, a(R)$	5

The current location of “i” determines the code sequence.

Three cases are covered depending on

- whether “i” is in register  $R_i$ ,
- whether “i” is in memory location  $M_i$  or
- whether “i” is on the stack at offset  $S_i$

and the pointer to the activation record for  $i$  is in register  $A$ . The register  $R$  is the register returned when the function getreg is called.

For the first assignment, prefer to leave ‘a’ in register  $R$  if ‘a’ has a next use in the block and register  $R$  is available. In the second assignment, assume ‘a’ is statically allocated.

The following table shows the code sequences generated for the pointer assignments  $a := *p$  and  $*p := a$ . The current location of  $p$  determines the code sequence.

Statement	p in register $R_p$		p in memory $M_p$		p in stack	
	Code	Cost	Code	Cost	Code	Cost
$a := *p$	MOV $*R_p, a$	2	MOV $M_p, R$ MOV $*R, R$	3	MOV $S_p(A), R$ MOV $*R, R$	3
$*p := a$	MOV $a, *R_p$	2	MOV $M_p, R$ MOV $a, *R$	4	MOV $a, R$ MOV $R, *S_p(A)$	4

Three cases are covered depending on

- whether “p” is in register  $R_p$ ,
- whether “p” is in memory location  $M_p$  or
- whether “p” is on the stack at offset  $S_p$

and the pointer to the activation record for  $p$  is in register A. The register R is the register returned when the function getreg is called. In the second assignment, assume ‘a’ is statically allocated.

## Conditional Statements

Machines implement conditional jumps in one of two ways:

1. If the value of a designated register meets one of the six conditions:
  - i. negative
  - ii. zero
  - iii. positive
  - iv. nonnegative
  - v. nonzero
  - vi. nonpositive.

A three-address statement such as if  $x < y$  goto  $z$  can be implemented by subtracting  $y$  from  $x$  in register  $R$  and then jumping to  $z$  if the value in register  $R$  is negative.

2. A second approach uses a set of condition codes to indicate whether the last quantity computed or loaded into a register is negative , zero or positive.

- A compare instruction has the desirable property that it sets the condition code without actually computing a value. (i.e)  $CMP x, y$  sets the condition code to positive if  $x > y$  and so on.

- A conditional jump machine instruction makes the jump if a designated condition  $<, =, >, \leq, =, \geq$  is met.
- The instruction  $CJ \leq z$  to mean “jump to  $z$  if the condition code is negative or zero”
- For example, if  $x < y$  goto  $z$  could be implemented by

$CMP x, y$

$CJ < z$

If we are generating code for a machine with condition codes it is useful to maintain a condition-code descriptor. This descriptor tells the name that last set the condition code, or the pair of names compared, if the condition code was last set that way.

Thus we could implement       $x := y + z$   
                                  If  $x < 0$  goto  $z$

by

$MOV y, R0$   
 $ADD z, R0$   
 $MOV R0, x$   
 $CJ < z$

If we were aware that the condition code was determined by  $x$  after  $ADD z, R0$ .

## **DAG REPRESENTATION OF BASIC BLOCKS**

- Directed Acyclic Graphs (DAG) are useful data structures for implementing transformations on basic blocks.
- DAG gives a picture of how the value computed by each statement in a basic block.
- Constructing a DAG from three-address statements is a good way of determining common sub-expressions within a block, determining which names are used inside the block but evaluated outside the block, and determining which statements of the block could have their computed value used outside the block.

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

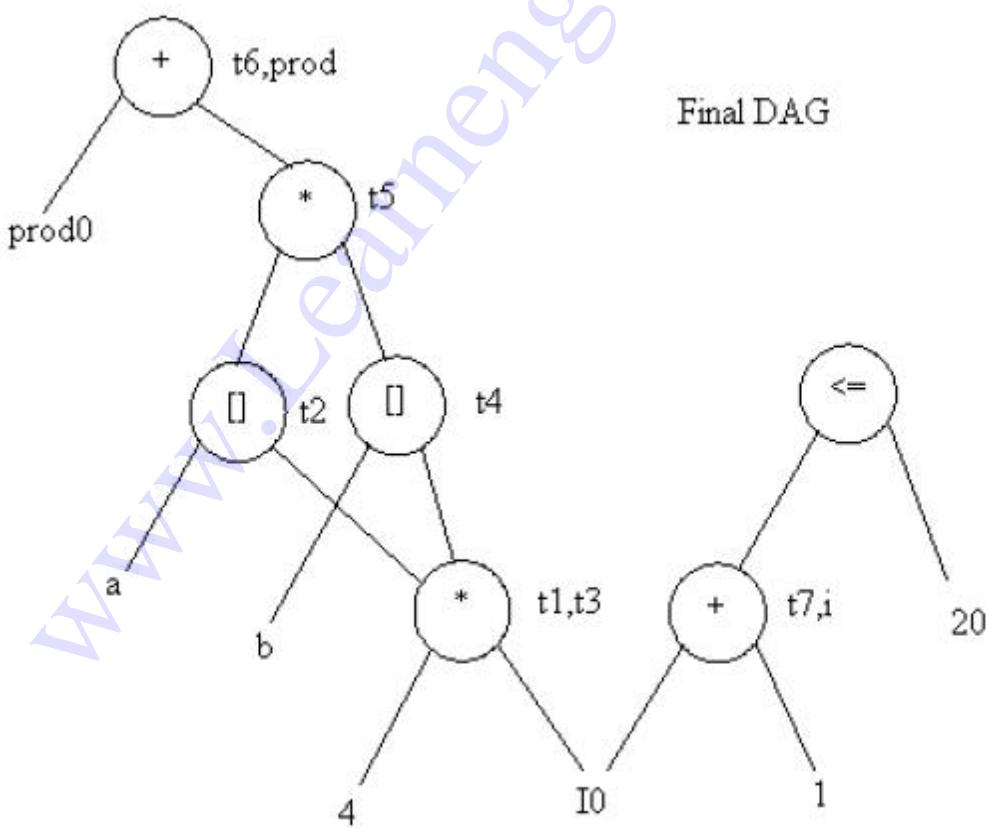
- Leaves are labeled by unique identifiers, either variable names or constants.
- Interior nodes are labeled by an operator symbol.
- Nodes are also optionally given a sequence of identifiers for labels. The intention is that interior nodes represent computed values, and the identifiers labeling a node are deemed to have that value.

It is important not to confuse dags with flow graphs. Each node of a flow graph can be represented by a dag, since each node of the flow graph stands for a basic block.

**Example:** Given the three-address statement:

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)

The corresponding DAG is shown as:



## DAG Construction

### Algorithm for constructing a DAG for basic block:

**Input:** A basic block.

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

- (i) A label for each node. For leaves the label is an identifier or constants and for interior nodes, an operator symbol.
- (ii) For each node a (possible empty) list of attached identifiers (constants not permitted here).

#### Method:

The dag construction process is to do the following steps (1) through (3) for each statement of the basic block.

Suppose the current three-address statement is either

- (i)  $x := y \text{ op } z$  (ii)  $x := \text{op } y$  or (iii)  $x := y$ .

We refer to these cases (i) (ii) and (iii).

We treat a relational operator like “if  $i \leq 20$  goto as case (i)” with  $x$  undefined.

(1) If node(y) is undefined, create a leaf labeled y and let node (y) be this node. In case(i), if node (z) is undefined, create a leaf labeled z and let that leaf be node (z).

(2) In case (i), determine if there is a node labeled op, whose left child is node(y) and whose right child is node (z). If not, create such a node. In either event, let n be the node found or created.

In case(ii), determine whether there is a node labeled op, whose lone child is node(y). If not, create such a node and let n be the node found or created.

In case (iii), let n be node(y).

(3) Delete x from the list of attached identifiers for node(x). Append x to the list of attached identifiers for the node n found in (2) and set node(x) to n.

## Applications of DAG

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.
4. Reconstruct a simplified list of quadruples taking advantage of common sub-expressions and not performing assignments of the form  $x := y$  unless absolutely necessary.
5. Evaluate the interior nodes of the dag in any order that is a topological sort of the dag.

### Reconstruction of basic block from the above DAG

```
t1 := 4 * i.  
t2 := a [t1]  
t4 := b [ t1 ]  
t5 := t2 * t4  
prod := prod + t5  
i := i + 1  
if i <= 20 goto (1)
```

**Note:** The ten statements of original basic block have been reduced to seven by taking advantage of the common sub-expressions exposed during the dag construction process and by eliminating unnecessary assignments.

## REGISTER ALLOCATION AND ASSIGNMENT

1. Global Register Allocation
2. Register Allocation by Graph Coloring

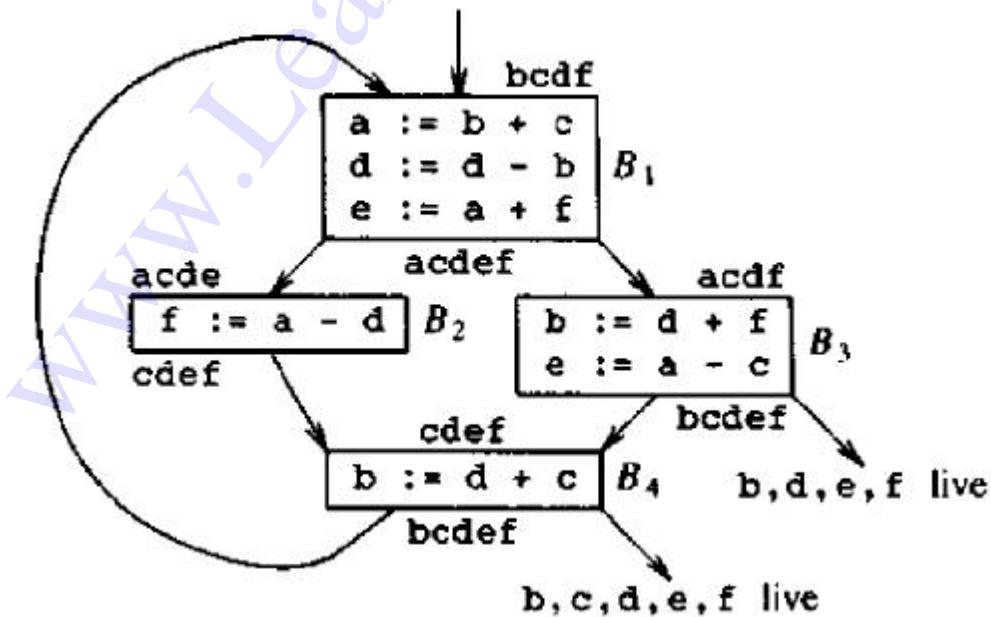
### 1. Global Register Allocation

- The strategy is to assign some fixed number of registers to hold the most active values in each inner loop.
- For this an approximate formula for allocating a register to x within a loop L is given as:

$$\sum_{\text{Blocks } B \text{ in } L} (\text{use}(x, B) + 2 * \text{live}(x, B))$$

Blocks B in L

- Consider the basic blocks in the inner loop in the following flow graph:



- To evaluate the above formula for  $x=a$ , we get

$$\text{use}(a, B_1) = 0$$

$$\text{use}(a, B_2) = 1$$

$$\text{use}(a, B_3) = 1$$

$$\text{use}(a, B_4) = 0$$

$$\text{Hence } \sum_{\text{Blocks } B \text{ in } L} (\text{use}(a, B)) = 2$$

Blocks B in L

- Similarly  $\sum_{\text{Blocks } B \text{ in } L} 2 * \text{live}(a, B) = 2$
- Hence the value of  $\sum_{\text{Blocks } B \text{ in } L} (\text{use}(a, B) + 2 * \text{live}(a, B)) = 4$  for  $x=a$ .
- Similarly the values of b,c,d,e and f are 6,3,6,4 and 4 respectively.
- Assuming that there are only three registers available namely R0,R1,R2
- R0 is assigned to a , R1 is assigned to b and R2 to d.
- R0 may also be assigned to e or f.

## 2. Register Allocation by Graph Coloring

- When a register is required for allocation and all registers are in use,then the contents of one of the used registers must be stored(spilled) into memory location to free a register.
- Graph coloring is used to allocate registers and manage register spills.
- For each procedure, a register-interference graph is constructed.

- The graph is colored using  $k$  colors where  $k$  is the number of assignable registers.
- A graph is said to be colored if no two adjacent nodes have the same color.
- A color represents a register and the coloring makes sure that no two symbolic registers that can interfere with each other are assigned the same physical register.

## GENERATING CODE FROM DAGS

There are two ways to generate code for a basic block from its DAG representation. They are:

1. Heuristic ordering for DAGs
2. Optimal ordering for DAGs

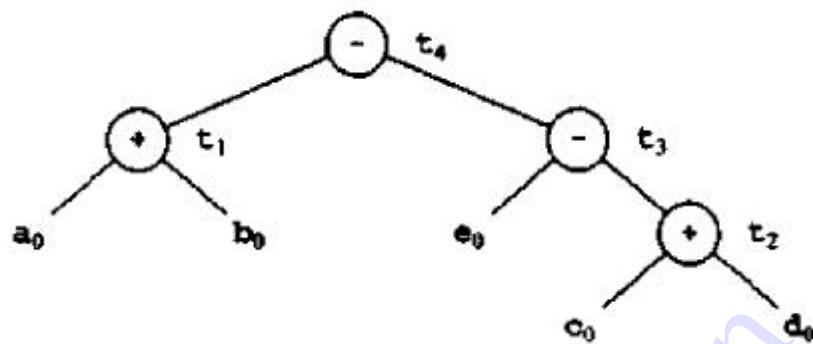
### Heuristic ordering for DAGs

Consider the following expression  $(a+b)-(e-(c+d))$

The three address code is as follows:

$$\begin{aligned}t_1 &:= a + b \\t_2 &:= c + d \\t_3 &:= e - t_2 \\t_4 &:= t_1 - t_3\end{aligned}$$

The DAG for the above three-address code is as follows:



- Applying the code generation algorithm, the target for the above three-address statements is as follows: ( assume we have 2 registers R0 and R1 and only t4 is live on exit.)

```

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
  
```

- Now to generate code from a DAG, we use a node listing algorithm that will be used to produce the target code.

```
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
    begin
5)     list m;
6)     n := m
  end
end
```

#### Node listing algorithm

- By applying the above algorithm to the DAG we get 1234 .
- Now reversing this list, we get 4321
- Reorder the three-address statements according to this list. We get

$$\begin{aligned}t_2 &:= c + d \\t_3 &:= e - t_2 \\t_1 &:= a + b \\t_4 &:= t_1 - t_3\end{aligned}$$

- Now generate the target code for the above sequence . The target code is as follows:

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

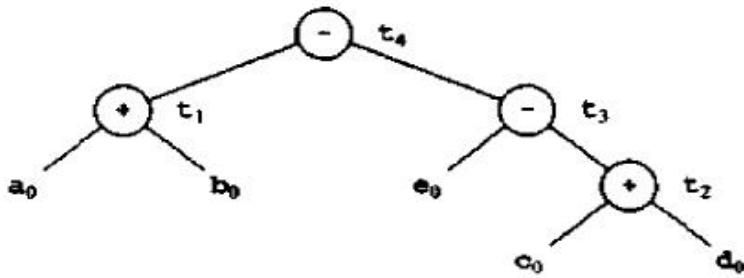
## Optimal Ordering for DAGs

- This technique has two parts :
  1. Using the Labeling algorithm , label each node of the tree.
  2. Generate the target code using a tree traversal whose order is governed by the computed node labels.

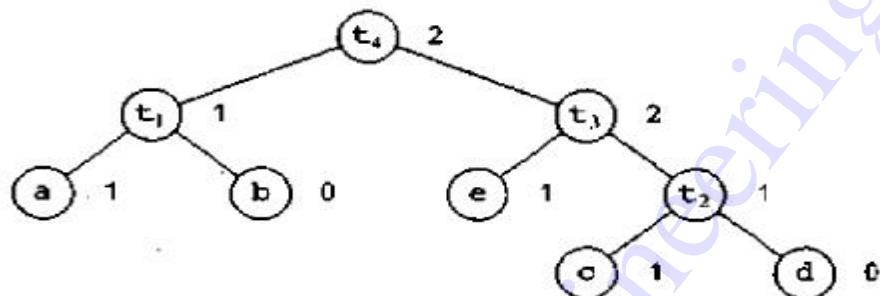
### 1. Labeling Algorithm

```
(1) if  $n$  is a leaf then  
(2)   if  $n$  is the leftmost child of its parent then  
(3)      $label(n) := 1$   
(4)   else  $label(n) := 0$   
   else begin /*  $n$  is an interior node */  
(5)     let  $n_1, n_2, \dots, n_k$  be the children of  $n$  ordered by  $label$ ,  
         so  $label(n_1) \geq label(n_2) \geq \dots \geq label(n_k)$ ;  
(6)      $label(n) := \max_{1 \leq i \leq k} (label(n_i) + i - 1)$   
   end
```

Consider the following DAG ,



Applying the Labeling algorithm we get the labeled tree as follows

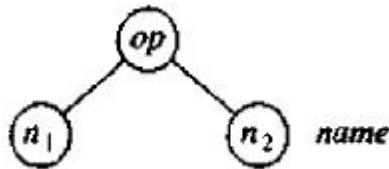


## 2. Code Generation from a labeled tree

- The code generation algorithm is to call the routine gencode on the root of the labeled tree.
- We have five cases.
  - Case 0 :** n is a leaf and the leftmost child of its parent.Hence generate just the load instruction.

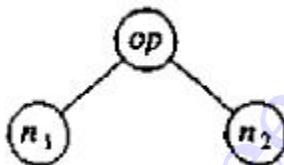


- Case 1:** For the subtree of the form



We generate code to evaluate n1 into register R = top(rstack) followed by the instruction op name , R.

iii. **Case 2:** For the subtree of the form



where n1 can be evaluated without stores but n2 is harder to evaluate than n1,then swap the two registers on rstack,then evaluate n2 into R=top(rstack).Remove R from rstack and evaluate n1 into S=top(rstack).Generate the instruction op R,S.

iv. **Case 3:** this is similar to case 2 except that n1 is harder to evaluate than n2 and is evaluated first. There is no need to swap registers.

v. **Case 4:** This is when both the subtrees require r or more registers to evaluate without stores.

The code generated for the above labeled tree is as follows:

```
gencode(t4)      [R1R0]          /* case 2 */
  gencode(t3)      [R0R1]          /* case 3 */
    gencode(e)        [R0R1]          /* case 0 */
      print MOV e,R1
    gencode(t2)      [R0]            /* case 1 */
      gencode(c)        [R0]            /* case 0 */
        print MOV c,R0
        print ADD d,R0
        print SUB R0,R1
      gencode(t1)      [R0]            /* case 1 */
        gencode(a)        [R0]            /* case 0 */
          print MOV a,R0
          print ADD b,R0
        print SUB R1,R0
```

## PART-A

### 1. What is a DAG? What are the applications of DAG?

[AU MAY/JUN 2007(Reg 2004), NOV/DEC 2008(Reg 2004),NOV/DEC 2011(Reg 2008)]

Directed acyclic graph (DAG) is a useful data structure for implementing transformations on basic blocks. A DAG for the basic block has the following information:

- A label for each node. For leaves, the label is an identifier and for interior nodes, an operator symbol.
- For each node, a possibly empty list of attached identifiers

#### Applications of DAG

- Determining the common sub-expressions.
- Determining which identifiers have their values used in the block
- Determining which statements of the block compute value outside the block.

### 2. Give the primary structure preserving transformations on Basic Blocks.

[AU MAY/JUN 2007(Reg 2004),APR/MAY 2011(Reg 2008)]

- Common sub expression elimination
- Dead-code elimination
- Renaming of temporary variables
- Interchange of two independent adjacent statements

### 3. What are basic blocks and flow graphs?

[AU NOV/DEC 2007(Reg 2004),NOV/DEC 2011(Reg 2008)]

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

A graph representation of three-address statements, called a flow graph, is useful for understanding code-generation algorithms.

Nodes in the flow graph represent computations and the edges represent the flow of control.

- 4. Give the syntax directed translation for the following statement  
Call p1(int a,int b).** [AU APR/MAY 2008(Reg 2004)]

- 5. How can you find the leaders in a basic block?** [AU APR/MAY 2008(Reg 2004),MAY/JUN 2009(Reg 2004)]

- (iv) The first statement is a leader.
- (v) Any statement that is the target of a conditional or unconditional goto is a leader.
- (vi) Any statement that immediately follows a goto or conditional goto statement is a leader.

- 6. What is constant folding?** [AU NOV/DEC 2008(Reg 2004)]

Deducing at compile time that the value of an expression is a constant and using the constant instead of the expression is called constant folding.

- 7. What is a flow graph?** [AU NOV/DEC 2008(Reg 2004)]

A flow graph is a directed graph in which the flow control information is added to the basic blocks.

- The nodes in the flow graph are basic blocks
- The block whose leader is the first statement is called initial block.
- There is a directed edge from block B1 to block B2 if B2 immediately follows B1 in the some execution sequence. We can say that B1 is a predecessor of B2 and B2 is a successor of B1.

**8. List the issues in code generation. [AU MAY/JUN 2009(Reg 2004)]**

- a. Input to Code Generator
- b. Target Programs
- c. Memory Management
- d. Instruction Selection
- e. Register Allocation
- f. Choice of Evaluation Order
- g. Approaches to Code Generation.

**9. What is the purpose of next-use information? [AU APR/MAY 2011(Reg 2008)]**

If the name in a register is no longer needed, then the register can be assigned to some other name.

**10. Mention the properties of a code generator .**

- The code generator should produce the correct and high quality code. In other words, the code generated should be such that it should make effective use of the resources of the target machine.
- Code generator should run efficiently.

**11. How do you calculate the cost of an instruction?**

The cost of an instruction can be computed as one plus cost associated with the source and destination addressing modes given by added cost.

MOV R0,R1	1
MOV R1,M	2
SUB 5(R0),*10(R1)	3

**12.What is the need of storage for temporary names?**

It may be useful in an optimizing compiler to create a distinct name each time a temporary is needed, space has to be allocated to hold the values of these temporaries.

**13. Name the descriptors used in code generation algorithm.**

- Register descriptor
- Address descriptor

**14.What is the use of the function getreg?**

The function getreg returns the location L to hold the value of x for the assignment  $x := y \text{ op } z$ .

**15. How will you generate code from DAGs?**

Code is generated from DAGs using

- Heuristic ordering for DAGs
- Optimal ordering for DAGs

**16.What is an optimizing compiler?**

If a code optimizer is present before code generator within a compiler then the compiler is called “Optimizing” compilers.

www.Learnengineering.in

## CODE OPTIMIZATION

### INTRODUCTION

- To create an efficient target language program, a programmer needs more than an optimizing compiler.
- We review the options available to a programmer and a compiler for creating efficient target programs.

#### 1. Criteria for Code – Improving Transformations.

#### 2. Getting Better Performance.

#### 3. An Organization for an Optimizing Compiler.

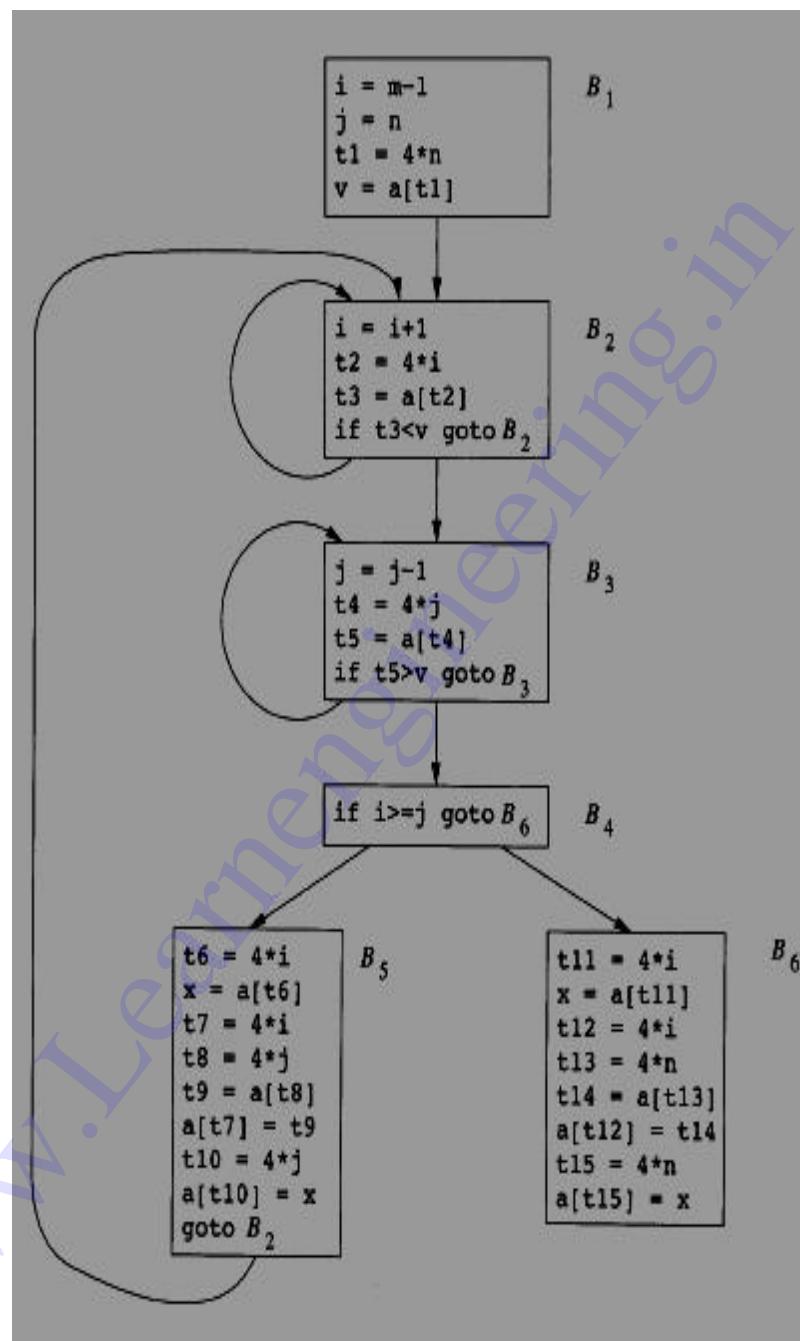
#### Example: C code for Quick Sort

```
void quicksort(int m, int n)
    /* recursively sorts a[m] through a[n] */
{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /* swap a[i], a[j] */
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* swap a[i], a[n] */
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

**Three – address code for the above fragment is as follows:**

(1)	$i = m-1$	(16)	$t7 = 4*i$
(2)	$j = n$	(17)	$t8 = 4*j$
(3)	$t1 = 4*n$	(18)	$t9 = a[t8]$
(4)	$v = a[t1]$	(19)	$a[t7] = t9$
(5)	$i = i+1$	(20)	$t10 = 4*j$
(6)	$t2 = 4*i$	(21)	$a[t10] = x$
(7)	$t3 = a[t2]$	(22)	goto (5)
(8)	if $t3 < v$ goto (5)	(23)	$t11 = 4*i$
(9)	$j = j-1$	(24)	$x = a[t11]$
(10)	$t4 = 4*j$	(25)	$t12 = 4*i$
(11)	$t5 = a[t4]$	(26)	$t13 = 4*n$
(12)	if $t5 > v$ goto (9)	(27)	$t14 = a[t13]$
(13)	if $i >= j$ goto (23)	(28)	$a[t12] = t14$
(14)	$t6 = 4*i$	(29)	$t15 = 4*n$
(15)	$x = a[t6]$	(30)	$a[t15] = x$

The flow graph for the above three-address code is as follows:



Flow graph for the Quick sort fragment

## THE PRINCIPAL SOURCES OF OPTIMIZATION

- 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.
- There are two main categories of optimization. They are:
  1. **Function-Preserving Transformations**
  2. **Loop Optimizations**

### 1. Function-Preserving Transformations.

- (i) Common sub-expression elimination
- (ii) Copy propagation
- (iii) Dead-code elimination
- (iv) Constant folding

are examples of function-preserving transformations.

### (i) Common Sub-Expression Elimination

#### Local Common Sub-Expression Elimination

The diagram illustrates the transformation of code within basic block  $B_5$  through local common sub-expression elimination. On the left, labeled 'Before', the code is:

```
t6 = 4*i  
x = a[t6]  
t7 = 4*i  
t8 = 4*j  
t9 = a[t8]  
a[t7] = t9  
t10 = 4*j  
a[t10] = x  
goto B2
```

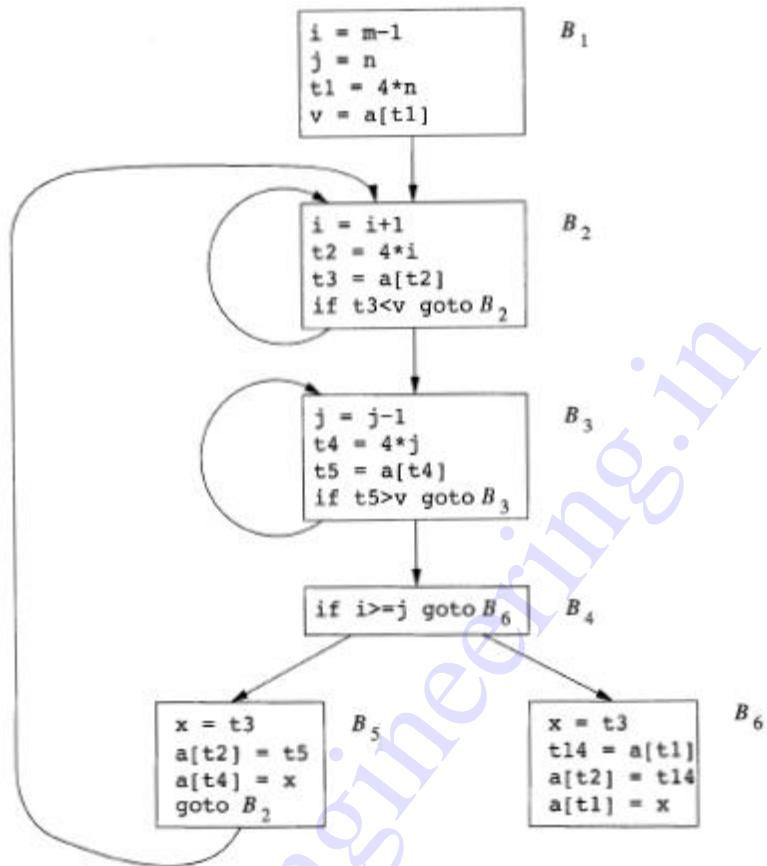
On the right, labeled 'After', the code is simplified by eliminating the redundant computation of  $t7$  and  $t10$ :

```
t6 = 4*i  
x = a[t6]  
t8 = 4*j  
t9 = a[t8]  
a[t6] = t9  
a[t8] = x  
goto B2
```

Before

After

- When sub-expressions are common within a basic block it is local.
- When sub-expressions are common between basic blocks it is global
- The following flow graph shows basic blocks  $B_5$  and  $B_6$  after both local and global common sub-expression elimination



Flow graph of B<sub>5</sub> and B<sub>6</sub> after common sub-expression elimination.

## (ii) Copy Propagation

- The idea behind the copy-propagation transformation is to use ‘g’ for ‘f’, wherever possible after the copy statement  $f = g$ .
- For example, the assignment  $x := t3$  in block B<sub>5</sub> is a copy. Copy propagation applied to B<sub>5</sub> yields:

### B<sub>5</sub> before copy propagation

$x = t3$

$a[t2] = t5$

$a[t4] = x$

goto B<sub>2</sub>

### B<sub>5</sub> after copy propagation

```
x= t3  
a[t2] = t5  
a[t4] = t3  
goto B2
```

- After applying copy propagation, it is possible to eliminate  $x = t3$ . It turns the copy statement into dead code.

### (iii) Dead-Code Elimination

- A variable is live at a point in a program if its value can be used subsequently; otherwise it is dead at that point.
- Dead or useless code statements that compute values that never get used.

#### Example:

```
debug = false
```

```
.....
```

```
.....
```

```
if (debug ) print .....
```

```
.....
```

- Since the name *debug* had been assigned to false value the print statement will not work at any time. (i.e) The print statement becomes ‘dead code’ because it cannot be reached.
- We can eliminate such dead code, by eliminating both the test and printing statement from the object code.

### (iv) Constant Folding

Deducing at compiler time that the value of an expression is a constant and using the constant instead is known as constant folding.

## 2. Loop Optimization

- Usually within inner loops the 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.
- Three techniques are important for loop optimization:
  - (i) Code-motion
  - (ii) Induction variable elimination
  - (iii) Reduction in strength

### (i) Code-motion

- This transformation takes an expression that yield the same result independent of the number of times a loop is executed and places the expression before the loop.

#### Example:

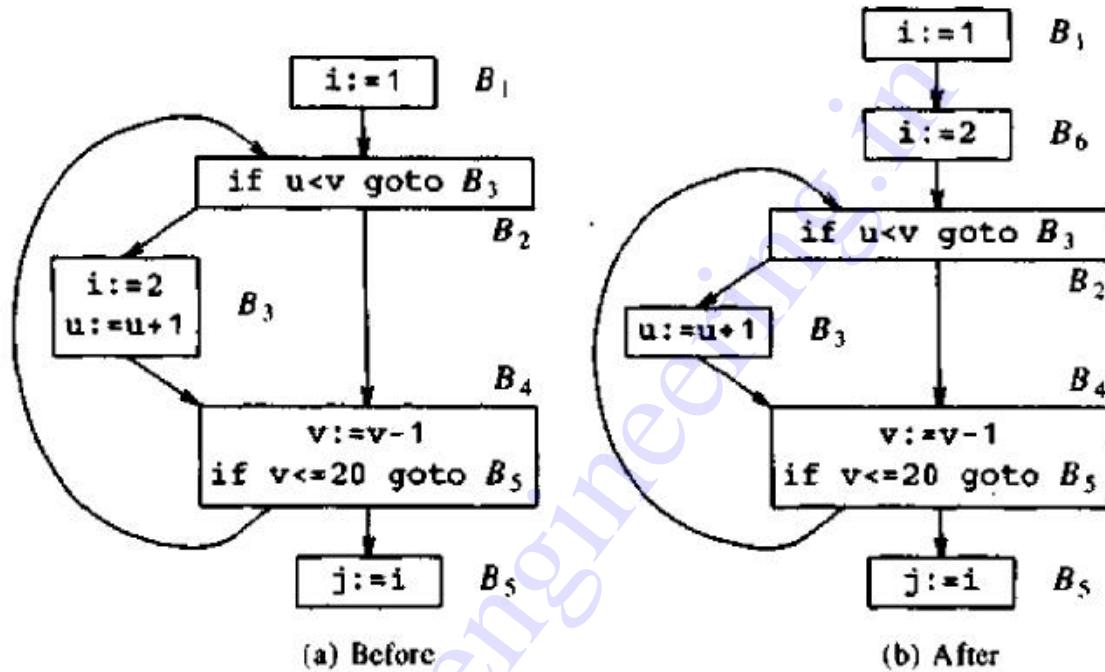
Evaluation of limit-2 is a loop-invariant computation in the following while statement:

```
while ( i <= limit-2 ) /* statement does not change limit */
```

Code motion will result in the equivalent of

**t = limit - 2;**

**while ( i <= t ) /\* statement does not change limit or t \*/**

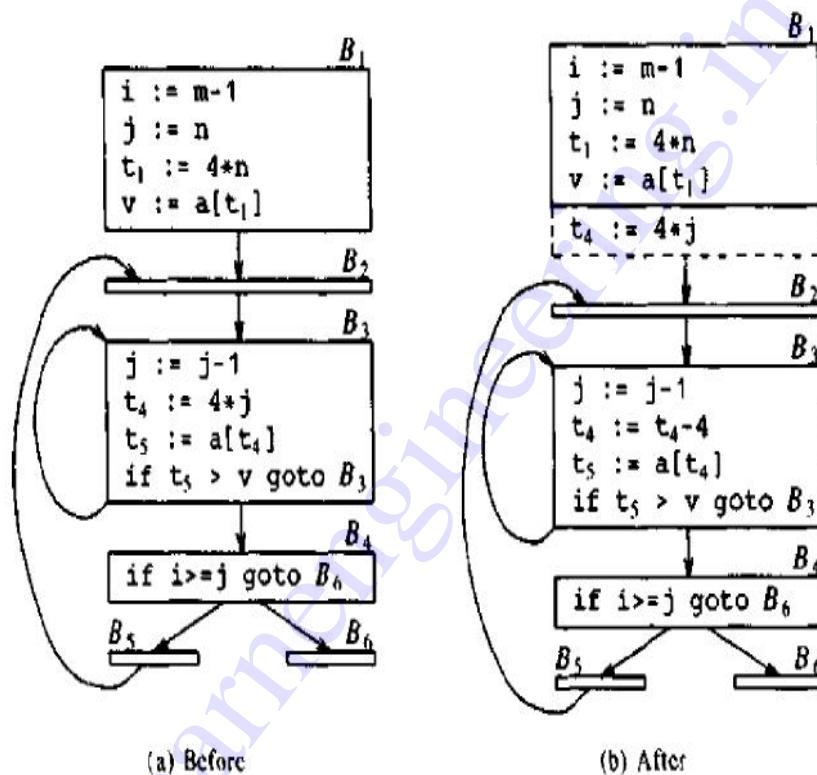


### Illegal Code Motion

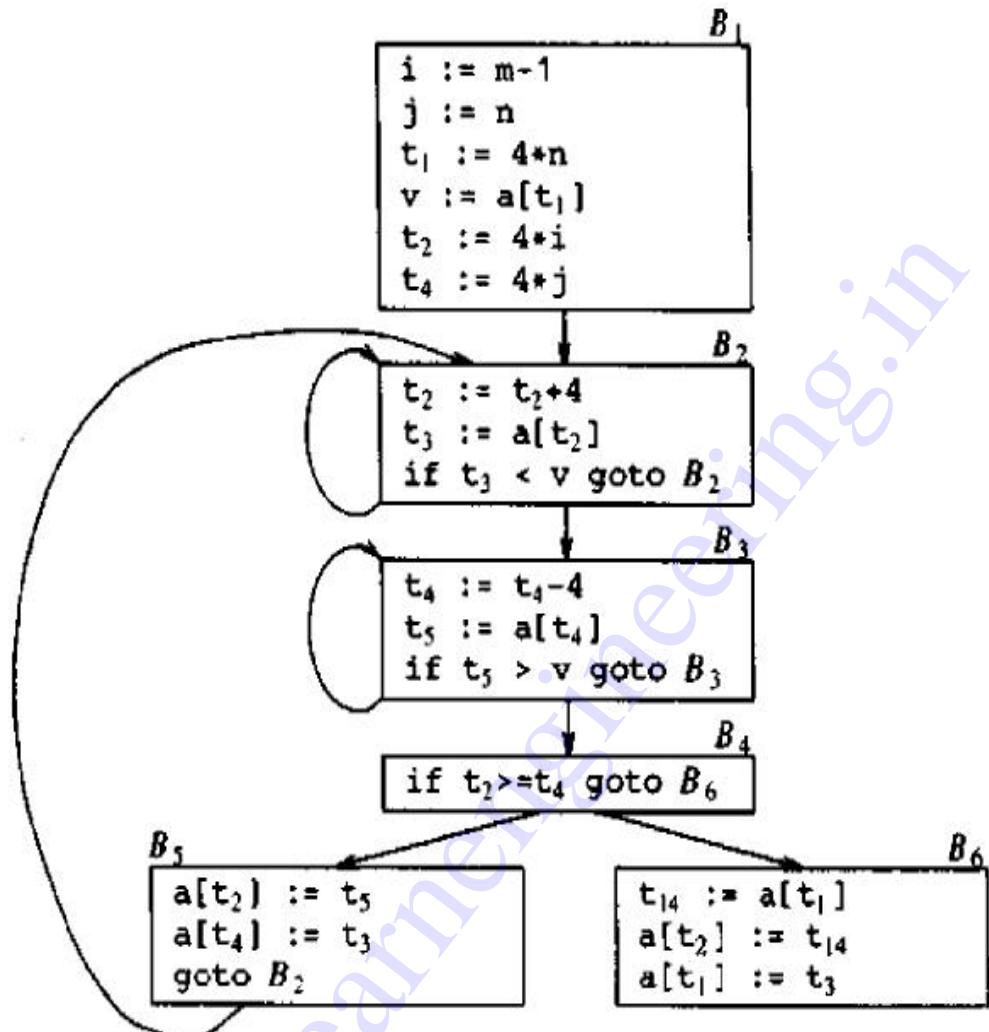
#### (ii) Induction variable elimination and reduction in strength

- A variable  $x$  is said to be an "induction variable" if there is a positive or negative constant  $c$  such that each time  $x$  is assigned, its value increases by  $c$ .
- The transformation of replacing an expensive operation, such as multiplication, by a cheaper one, such as addition, is known as *strength reduction*

- But induction variables not only allow us sometimes to perform a strength reduction; often it is possible to eliminate all but one of a group of induction variables whose values remain in lock step as we go around the loop.



Strength Reduction applied to  $4*j$  in block  $B_3$



Flowgraph after Induction Variable Elimination

## PEEPHOLE OPTIMIZATION

- A simple but effective technique for locally improving the quality of target code is peephole optimization, a method for trying to improve the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing those instruction by a shorter or faster sequence, whenever possible.
- Peephole optimization can also be applied directly after intermediate code generation to improve the intermediate representation.
- The peephole is a small, sliding window on a program. The code in the peephole need not be contiguous, although some implementations do require this.

**The characteristic of peephole optimizations are:**

1. Redundant-instruction elimination
2. Flow-of-control optimizations
3. Algebraic simplifications
4. Use of machine idioms

### 1. Redundant –instruction elimination

If we see the instruction sequence

**(1) MOV R0, a (Store)**

**(2) MOV a, R0 (Load)**

We can delete instruction (2) because whenever (2) is executed, (1) will ensure that the value of a is already in register R0. Note that if (2) had a label we could not be sure that (1) was always executed immediately before (2) and

so we could not remove (2). Put another way, (1) and (2) have to be in the same basic block for this transformation to be safe.

### Unreachable Code

- Another opportunity for peephole optimization is the removal of unreachable instructions.
- An unlabeled instruction immediately following an unconditional jump may be removed.

For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable debug is 1. In C, the source code might look like:

```
# define debug 0
.....
if (debug) {
    print debugging information
}
```

In the intermediate representation the if-statement may be translated as:

if debug = 1 goto L1

goto L2   -----> (1)

L1: print debugging information

L2:

One obvious peephole optimization is to eliminate jumps over jumps. Thus, (1) can be replaced by:

```
if debug != 1 goto L2  
print debugging information -----> (2)
```

L2:

Since debug is set to 0 at the beginning of the program, constant propagation should replace the (2) by:

```
if 0 != 1 goto L2  
print debugging information -----> (3)
```

L2:

As the argument of the first statement of (3) evaluates to a constant true it can be replaced by goto L2. Then all the statements that print debugging aids are manifestly unreachable and can be eliminated one at a time.

## 2. Flow-of-Control Optimizations

- Simple intermediate code-generation algorithms frequently produce jumps to jumps, jumps to conditional jumps, or conditional jumps to jumps.
- These unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations.

We can replace the sequence

```
goto L1  
.....  
L1: goto L2
```

by the sequence

    goto L2

.....

L1: goto L2

If there are now no jumps to **L1**, then it may be possible to eliminate the statement **L1: goto L2** provided it is preceded by an unconditional jump.

Similarly, the sequence

    if a < b goto L1

.....

L1: goto L2

can be replaced by

    if a < b goto L2

.....

L1: goto L2

Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto. Then the sequence

    goto L1

.....

L1: if a < b goto L2                         -----> (1)

L3:

may be replaced by

```
if a < b goto L2  
goto L3 -----> (2)  
.....  
L3:
```

While the number of instruction in (1) and (2) is the same, We sometimes skip the unconditional jump in (2) but never in (1) Thus (2) is superior than (1) in execution time.

### 3. Algebraic Simplification

There is no end to the amount of algebraic simplification that can be attempted through peephole optimization.

Eg: Statements such as  $x := x + 0$  (or)

$x := x * 1$

are often produced by straightforward intermediate code-generation algorithms, and they can be eliminated easily through peephole optimization.

## Reduction in Strength

- Reduction-in-strength transformations can be applied in the peephole to replace 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.

## 4. Use of Machine Idioms

- The target machine may have hardware instructions to implement certain specific operations efficiently.
- Detecting situations that permit the use of these instructions can reduce execution time significantly.
- For example, some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value. The use of the modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like  $x=x+1$ .

## OPTIMIZATION OF BASIC BLOCKS

Many of the structure preserving transformations can be implemented by constructing a DAG for a basic block.

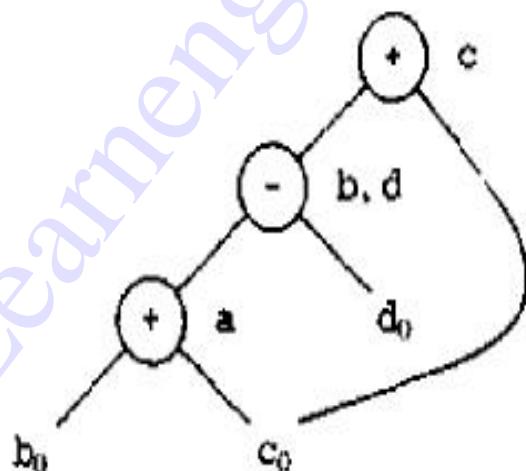
**Example:** A DAG for the block:

$$a := b + c$$

$$b := a - d \quad \xrightarrow{\hspace{1cm}} (1)$$

$$c := b + c$$

$$d := a - d$$



DAG for the above basic block

For example, if b is not live on exit, we could use:

$$a := b + c$$

$$d := a - d$$

$$c := d + c$$

However, if both b and d are live on exit, then a fourth statement must be used to copy the value from one to the other.

When we look for common sub-expressions, we really are looking for expressions that are guaranteed to compute the same value, no matter how that value is computed. Thus, the dag method will miss the fact that the expression computed by the first and the fourth statements in the sequence

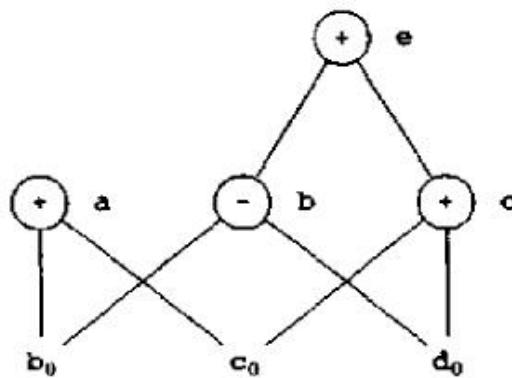
$$a := b + c$$

$$b := b - d \quad \xrightarrow{\hspace{1cm}} (2)$$

$$c := c + d$$

$$e := b + c$$

is the same namely,  $b + c$ . The dag for the sequence (2) is shown in below figure



### The use of Algebraic Identities:

- Algebraic identities represent another important class of optimizations on basic block.

For example, apply arithmetic identities such as

$$\mathbf{x + 0 = 0 + x = x}$$

$$\mathbf{x - 0 = x}$$

$$\mathbf{x * 1 = 1 * x = x}$$

$$\mathbf{x / 1 = x}$$

- Another class of algebraic optimizations includes reduction in strength, that is replacing a more expensive operator by a cheaper one as in

$$\mathbf{x^{**} 2 = x * x}$$

$$\mathbf{2.0 * x = x + x}$$

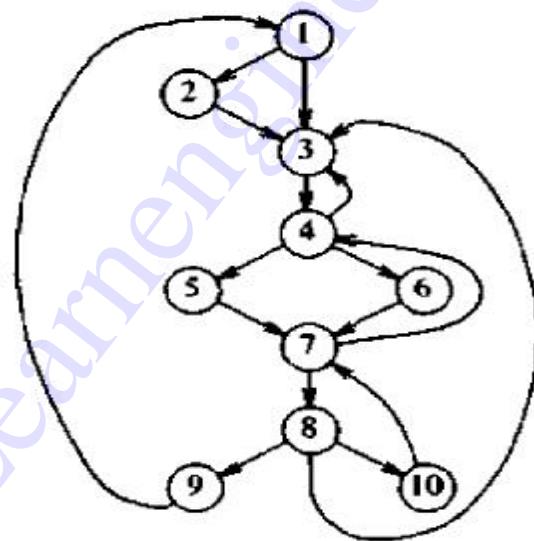
$$\mathbf{x / 2 = x * 0.5}$$

- A third class of related optimizations is constant folding. 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.

## LOOPS IN FLOW GRAPHS

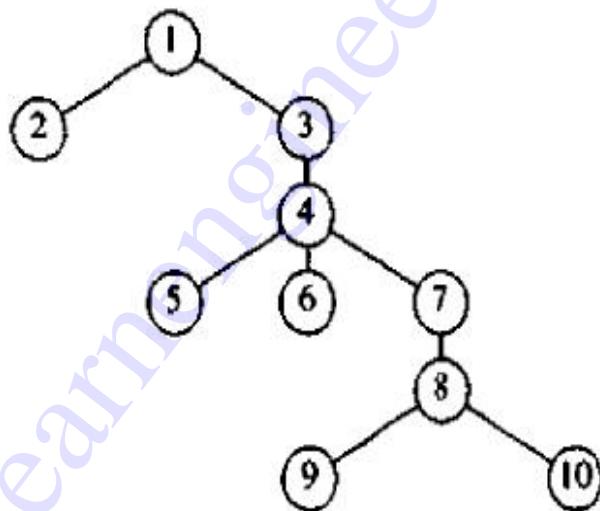
### Dominators

- We say node d of a flow graph **dominates** node n, written  $d \text{ dom } n$ , if every path from the entry node of the flow graph to n goes through d.
- Every node dominates itself.
- Dominator information is represented using a **dominator tree**
- Consider the flow graph with entry node 1.



- The entry node dominates every node (this statement is true for every flow graph).
- Node 2 dominates only itself, since control can reach any other node along a path that begins with  $1 \rightarrow 3$ .

- Node 3 dominates all but 1 and 2.
- Node 4 dominates all but 1, 2 and 3.
- Nodes 5 and 6 dominate only themselves, since flow of control can skip around either by going through the other.
- Finally, 7 dominates 7, 8, 9, and 10;
- 8 dominates 8, 9, and 10.
- 9 and 10 dominate only themselves.
- The dominator tree is as follows:



## Natural Loops

- A natural loop is defined by two essential properties:
  1. It must have a single-entry node, called the header. This entry node dominates all nodes in the loop, or it would not be the sole entry to the loop.

2. There must be a back edge that enters the loop header. Otherwise, it is not possible for the flow of control to return to the header directly from the "loop".
- 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 for constructing a natural loop** is as follows:

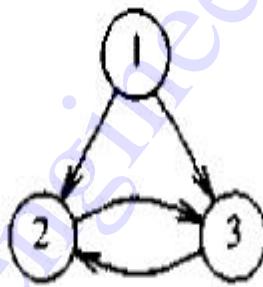
```
procedure insert(m);
if m is not in loop then begin
    loop := loop ∪ {m};
    push m onto stack
end;

/* main program follows */

stack := empty;
loop := { d };
insert(n);
while stack is not empty do begin
    pop m, the first element of stack, off stack;
    for each predecessor p of m do insert(p)
end
```

## Reducible Flow graphs

- A flow graph G is **reducible** if and only if we can partition the edges into two disjoint groups, called as the **forward edges** and **back edges**.
  1. The forward edges form an acyclic graph in which every node can be reached from the initial node G.
  2. The back edges consist only of edges whose heads dominate their tails.



Example for non-reducible flow graph

## INTRODUCTION TO GLOBAL DATA FLOW ANALYSIS

- In order to do code optimization and a good code generation, a compiler needs to collect information about the program as a whole and to distribute this information to each block in the flow graph.
- To perform transformations like dead code elimination and constant folding, compiler must know where a variable was last defined before reaching a given block.
- An optimizing compiler collects the data flow information by a process known as *data-flow analysis*.

- Data flow information can be collected by setting up and solving systems of equations that relate information at various points in a program.
- A typical equation has the form

$$\text{out}[S] = \text{gen}[S] \cup (\text{in}[S] - \text{kill}[S])$$

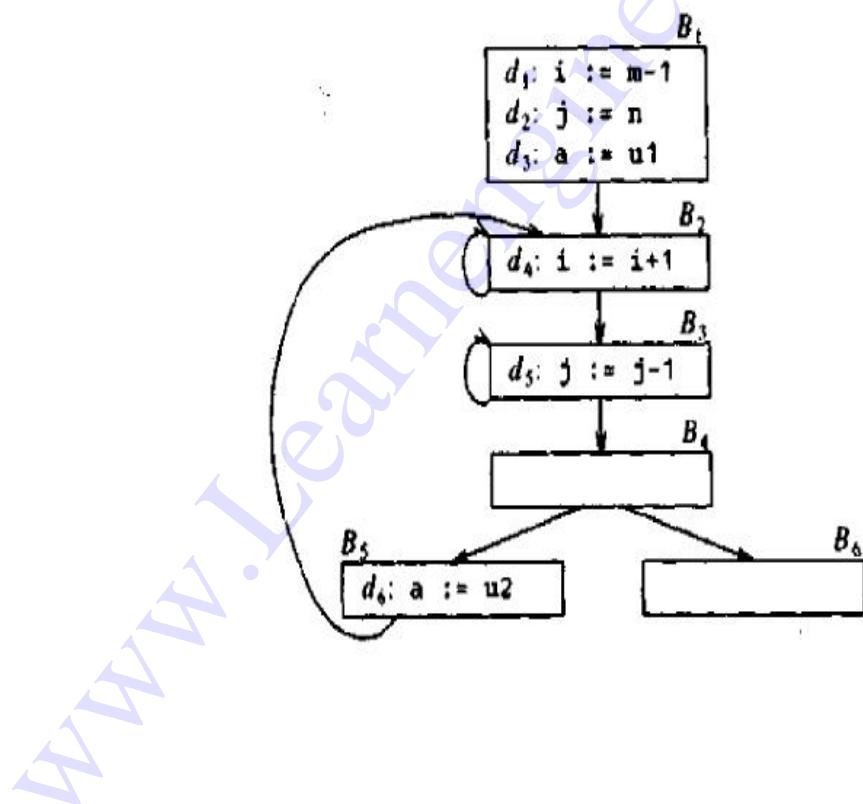
and can be read as “ the information at the end of a statement is either generated within the statement, or enters at the beginning and is not killed as control flows through the statement”. Such equations are called ***data-flow equations***.

### **Three-factors to set up and solve the data-flow equations:**

- The notion of generating and killing depend on the desired information, (i.e) on the data-flow analysis problem to be solved. Moreover, for some problems instead of defining  $\text{out}[S]$  in terms of  $\text{in}[S]$ , we need to proceed backwards and define  $\text{in}[S]$  in terms of  $\text{out}[S]$ .
- Since the data flows along the control path, data flow analysis affected by control constructs in a program.
- There are subtleties that go along with such statements as procedure calls, assignments through pointer variables and assignments to array variables.

### Points and Paths:

- Within a basic block, we discuss the point between two adjacent statements, as well as the point before the first statement and after the last.
- A path from  $p_1$  to  $p_n$  is a sequence of points  $p_1, p_2, \dots, p_n$  such that for each  $i$  between 1 and  $n - 1$ , either
  - $p_i$  is the point immediately preceding a statement and  $p_{i+1}$  is the point immediately following that statement in the same block, or
  - $p_i$  is the end of some block and  $p_{i+1}$  is the beginning of a successor block.



## Reaching Definitions:

- A definition of a variable  $x$  is a statement that assigns, or may assign, a value to  $x$ .
- The most common forms of definitions are assignments to  $x$  and statements that read a value from an I/O device and store it in  $x$ . These statements certainly define a value for  $x$ , and they referred to as **unambiguous definitions** of  $x$ .
- There are certain other kinds of statements that may define a value for  $x$ ; they are called **ambiguous definitions**.
- The usual forms of ambiguous definitions of  $x$  are:
  - i. A call of a procedure with  $x$  as a parameter or a procedure can access ' $x$ ' because  $x$  is in the scope of the procedure.
  - ii. An assignment through a pointer that could refer to  $x$ . For example, the assignment  $*q := y$  is a definition of  $x$  if it is possible that  $q$  points to  $x$ .
- We say a definition ' $d$ ' reaches a point ' $p$ ' if there is a path from the point immediately following ' $d$ ' to ' $p$ ', such that ' $d$ ' is not killed along that path.
- A point can be reached by an unambiguous definition and an ambiguous definition of the same variable appearing later along one path.

## Data-Flow Analysis of Structured Programs:

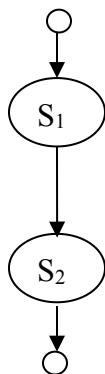
$S \rightarrow id := E$

|  $S ; S$

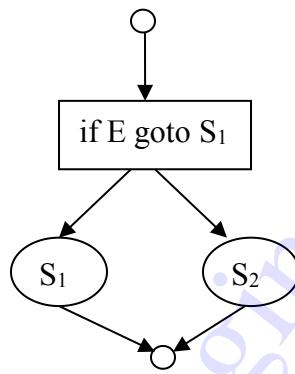
| if  $E$  then  $S$  else  $S$

| do  $S$  while  $E$  i

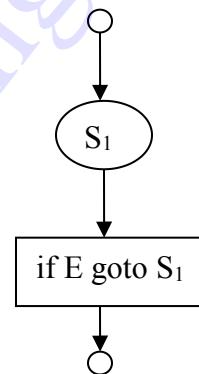
$E \rightarrow id + id \mid id$



$S_1 ; S_2$



if  $E$  then  $S_1$  else  $S_2$

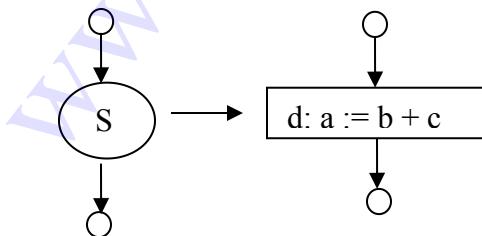


do  $S_1$  while  $E$

### Some structured control constructs

## Data-flow equations for reaching definitions.

Figure (i)



$$\begin{aligned} \text{gen}[S] &= \{ d \} \\ \text{kill}[S] &= D_a - \{ d \} \\ \text{out}[S] &= \text{gen}[S] \cup (\text{in}[S] - \text{kill}[S]) \end{aligned}$$

Observe the rule  $d : a := b + c$  in the figure(i) for a single assignment of variable ‘a’. Surely that assignment is a definition of “a” say definition d. Then d is the only definition sure to reach the end of the statement regardless of whether it reaches the beginning. Thus

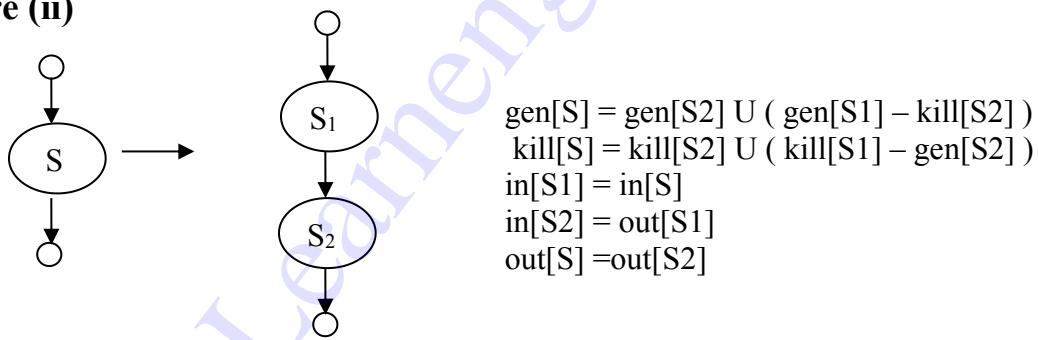
$$\text{gen}[S] = \{ d \}$$

On the other hand, d “kills” all other definitions of ‘a’ so we write

$$\text{kill}[S] = D_a - \{ d \}$$

where  $D_a$  is the set of all definitions in the program for variable ‘a’.

**Figure (ii)**



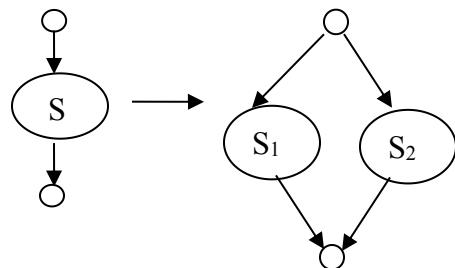
The rule for a cascade of statements, illustrated in figure(ii) is a bit more subtle. The definition d is generated by  $S = S_1 ; S_2$ ? First of all, if it is generated by  $S_2$ , then it is surely generated by S. If d is generated by  $S_1$ , it will reach the end of S provided it is not killed by  $S_2$ . Thus, we write

$$\text{gen}[S] = \text{gen}[S_2] \cup (\text{gen}[S_1] - \text{kill}[S_2])$$

Similar reasoning applies to the killing of a definition, so we have

$$\text{kill}[S] = \text{kill}[S2] \cup (\text{kill}[S1] - \text{gen}[S2])$$

**Figure (iii)**



$$\begin{aligned} \text{gen}[S] &= \text{gen}[S1] \cup \text{gen}[S2] \\ \text{kill}[S] &= \text{kill}[S1] \cap \text{kill}[S2] \end{aligned}$$

$$\begin{aligned} \text{in}[S1] &= \text{in}[S] \\ \text{in}[S2] &= \text{in}[S] \\ \text{out}[S] &= \text{out}[S1] \cup \text{out}[S2] \end{aligned}$$

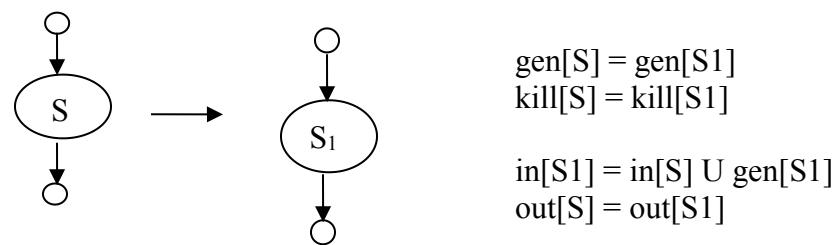
For the if-statement, illustrated in figure.(iii), note that if either branch of the “if” generates a definition, then that definition reaches the end of the statement S. Thus

$$\text{gen}[S] = \text{gen}[S1] \cup \text{gen}[S2]$$

In order to “kill” definition d, the variable defined by d must be killed along any path from the beginning to the end of S. It must be killed along either branch, so

$$\text{kill}[S] = \text{kill}[S1] \cap \text{kill}[S2]$$

**Figure (iv)**



Consider the rules for loops in figure (iv). The loop does not affect gen or kill.

If definition d is generated within  $S_1$ , then it reaches both the end of  $S_1$  and the end of S.

If d is generated within S, it can only be generated within  $S_1$ .

If d is killed by  $S_1$ , then going around the loop will not help; the variable of d gets redefined within  $S_1$  each time around.

If d is killed by S, then it must surely be killed by  $S_1$ .

Conclude that

$$\text{gen}[S] = \text{gen}[S_1]$$

$$\text{kill}[S] = \text{kill}[S_1].$$

### **Computation of ‘in’ and ‘out’:**

‘in’ is an inherited attribute and ‘out’ is an synthesized attribute depending on ‘in’.

in[S] - be the set of definitions reaching beginning of S, taking into account the flow of control throughout the entire program, including statements outside of ‘S’ or within which ‘S’ is nested.

out[S] – It is defined similarly for the end of S.

### **CODE - IMPROVING TRANSFORMATIONS**

1. Common sub-expression elimination
2. Copy propagation
3. Dead code elimination
4. Code Motion
5. Induction variable elimination and reduction in strength

**Note :** Explanations with examples must be given

## PART-A

### 1. What is code motion? [AU MAY/JUN 2007(Reg 2004), APR/MAY 2008(Reg 2004)]

It decreases the amount of code in a loop. Taking the expression which yield the same result independent of the number of times a loop is executed (a loop-invariant computation and places it before the loop.

### 2. What are the criteria used for code-improving transformations?

[AU NOV/DEC 2008(Reg 2004)]

- Must preserve the meaning of programs
- Must, on the average, speed up programs by a measurable amount
- Must be worth the effort.

### 3. Define dead code elimination. [AU APR/MAY 2011(Reg 2008)]

A variable is live at a point in a program if its value can be used subsequently; otherwise it is dead at that point.

Dead or useless code statements that compute values that never get used.

### 4. What is loop optimization? [AU APR/MAY 2011(Reg 2008)]

- Usually within inner loops the 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.
- Three techniques are important for loop optimization:

- (iv) Code-motion
- (v) Induction variable elimination
- (vi) Reduction in strength

**5. When does dangling reference occur? [AU NOV/DEC 2011(Reg 2008)]**

Whenever storage can be deallocated, the problem of dangling references arises. A dangling reference occurs when there is a reference to storage that has been deallocated. It is a logical error to use dangling references, since the value of deallocated storage is undefined according to the semantics of most languages.

**6. What do you mean by machine dependent and machine independent optimization?**

- The machine dependent optimization is based on the characteristics of the target machine for the instruction set used and addressing modes used for the instructions to produce the efficient target code.
- The machine independent optimization is based on the characteristics of the programming languages for appropriate programming structure and usage of efficient arithmetic properties in order to reduce the execution time.

**7. Write a short note on global data flow analysis.**

In order to do code optimization and a good code generation, a compiler needs to collect information about the program as a whole and to distribute this information to each block in the flow graph.

To perform transformations like dead code elimination and constant folding, compiler must know where a variable was last defined before reaching a given block.

An optimizing compiler collects the data flow information by a process known as ***data-flow analysis***.

## 8. What is a reducible flow graph?

A flow graph  $G$  is **reducible** if and only if we can partition the edges into two disjoint groups, called as the **forward edges** and **back edges**.

- The forward edges form an acyclic graph in which every node can be reached from the initial node  $G$ .
- The back edges consist only of edges whose heads dominate their tails.

## 9. List the properties of natural loops.

A natural loop is defined by two essential properties:

3. It must have a single-entry node, called the header. This entry node dominates all nodes in the loop, or it would not be the sole entry to the loop.

4. There must be a back edge that enters the loop header. Otherwise, it is not possible for the flow of control to return to the header directly from the "loop".

## 10. Define peephole optimization.

A simple but effective technique for locally improving the quality of target code is peephole optimization, a method for trying to improve the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing those instruction by a shorter or faster sequence, whenever possible.

## 11. What are the function preserving transformations?

- Common sub-expression elimination
- Copy propagation
- Dead-code elimination
- Constant folding

## 12. Define an induction variable.

A variable  $x$  is said to be an "induction variable" if there is a positive or negative constant  $c$  such that each time  $x$  is assigned, its value increases by  $c$ .

### **13. Define strength reduction.**

The transformation of replacing an expensive operation, such as multiplication, by a cheaper one, such as addition, is known as strength reduction

### **14. What are the characteristics of peephole optimization?**

- i. Redundant-instruction elimination
- ii. Flow-of-control optimizations
- iii. Algebraic simplifications
- iv. Use of machine idioms

**UNIT-II LEXICAL ANALYSIS**  
**2 MARKS**

**1. What is Lexical Analysis?**

The first phase of compiler is Lexical Analysis. This is also known as linear analysis in which the stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of characters having a collective meaning.

**2. What is a lexeme? Define a regular set.**

- A Lexeme is a sequence of characters in the source program that is matched by the pattern for a token.
- A language denoted by a regular expression is said to be a regular set

**3. What is a sentinel? What is its usage?**

A Sentinel is a special character that cannot be part of the source program. Normally we use 'eof' as the sentinel. This is used for speeding-up the lexical analyzer.

**4. What is a regular expression? State the rules, which define regular expression?**

Regular expression is a method to describe regular language

**Rules:**

- 1)  $\epsilon$ -is a regular expression that denotes  $\{\epsilon\}$  that is the set containing the empty string
- 2) If  $a$  is a symbol in  $\Sigma$ , then  $a$  is a regular expression that denotes  $\{a\}$
- 3) Suppose  $r$  and  $s$  are regular expressions denoting the languages  $L(r)$  and  $L(s)$ . Then,
  - a)  $(r)/(s)$  is a regular expression denoting  $L(r) \cup L(s)$ .
  - b)  $(r)(s)$  is a regular expression denoting  $L(r)L(s)$
  - c)  $(r)^*$  is a regular expression denoting  $L(r)^*$ .
  - d)  $(r)$  is a regular expression denoting  $L(r)$ .

**5. What are the Error-recovery actions in a lexical analyzer?**

1. Deleting an extraneous character
2. Inserting a missing character
3. Replacing an incorrect character by a correct character
4. Transposing two adjacent characters

**6. Construct Regular expression for the language**

$$L = \{w \in \{a,b\}^*/w \text{ ends in } abb\}$$

Ans:  $\{a/b\}^*abb$ .

**7. What is recognizer?**

Recognizers are machines. These are the machines which accept the strings belonging to certain language. If the valid strings of such language are accepted by the machine then it is said that the corresponding language is accepted by that machine, otherwise it is rejected.

**8. Differentiate compiler and interpreter.**

Compiler produces a target program whereas an interpreter performs the operations implied by the source program.

**9. Write short notes on buffer pair.**

Concerns with efficiency issues

Used with a lookahead on the input

It is a specialized buffering technique used to reduce the overhead required to process an input character. Buffer is divided into two N-character halves. Use two pointers. Used at times when the lexical analyzer needs to look ahead several characters beyond the lexeme for a pattern before a match is announced.

**10. Differentiate tokens, patterns, lexeme.**

- Tokens- Sequence of characters that have a collective meaning.
- Patterns- There is 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 sequence of characters in the source program that is matched by the pattern for a token.

**11. List the operations on languages.**

- **Union** -  $L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
- **Concatenation** -  $LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
- **Kleene Closure** -  $L^*$  (zero or more concatenations of L)
- **Positive Closure** -  $L^+ (one \text{ or } more \text{ concatenations of } L)$

**12. Write a regular expression for an identifier.**

An identifier is defined as a letter followed by zero or more letters or digits. The regular expression for an identifier is given as **letter (letter | digit)\***

**13. Mention the various notational shorthands for representing regular expressions.**

- One or more instances (+)
- Zero or one instance (?)
- Character classes ([abc] where a,b,c are alphabet symbols denotes the regular expressions a | b | c.)
- Non regular sets

**14. What is the function of a hierarchical analysis?**

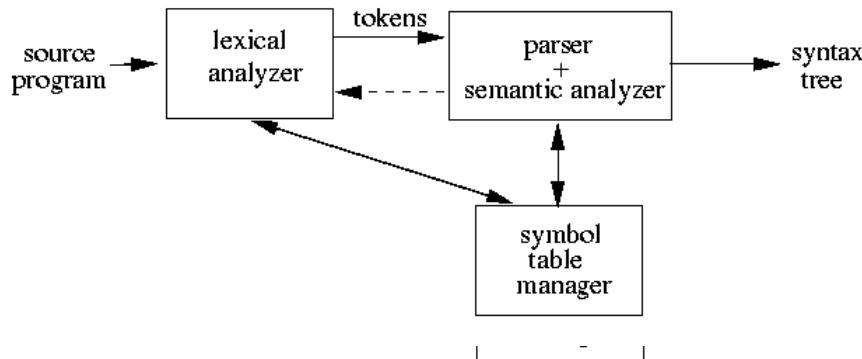
Hierarchical analysis is one in which the tokens are grouped hierarchically into nested collections with collective meaning. Also termed as Parsing.

### 15. What does a semantic analysis do?

Semantic analysis is one in which certain checks are performed to ensure that components of a program fit together meaningfully. Mainly performs type checking.

### 16 MARKS

#### 1) What are roles and tasks of a lexical analyzer?



Main Task: Take a token sequence from the scanner and verify that it is a syntactically correct program.

Secondary Tasks:

- Process declarations and set up symbol table information accordingly, in preparation for semantic analysis.
- Construct a syntax tree in preparation for intermediate code generation.

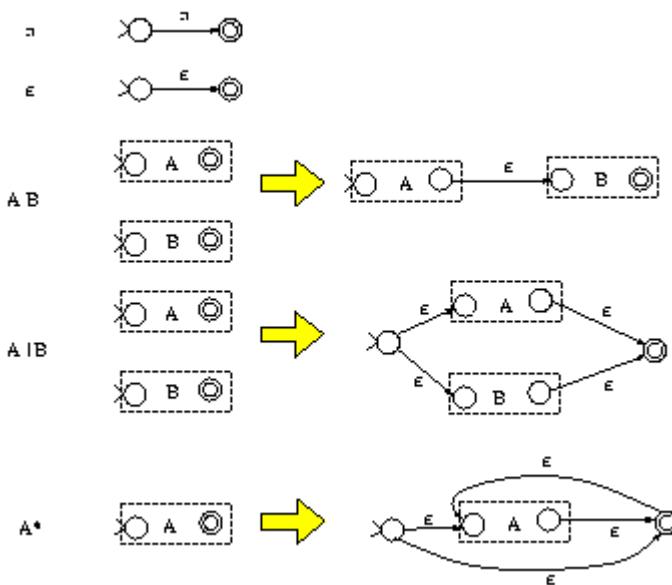
### 2. Converting a Regular Expression into a Deterministic Finite Automaton

The task of a scanner generator, such as JLex, is to generate the transition tables or to synthesize the scanner program given a scanner specification (in the form of a set of REs). So it needs to convert REs into a single DFA. This is accomplished in two steps: first it converts REs into a non-deterministic finite automaton (NFA) and then it converts the NFA into a DFA.

An NFA is similar to a DFA but it also permits multiple transitions over the same character and transitions over  $\epsilon$ . In the case of multiple transitions from a state over the same character, when we are at this state and we read this character, we have more than one choice; the NFA succeeds if at least one of these choices succeeds. The  $\epsilon$  transition doesn't consume any input characters, so you may jump to another state for free.

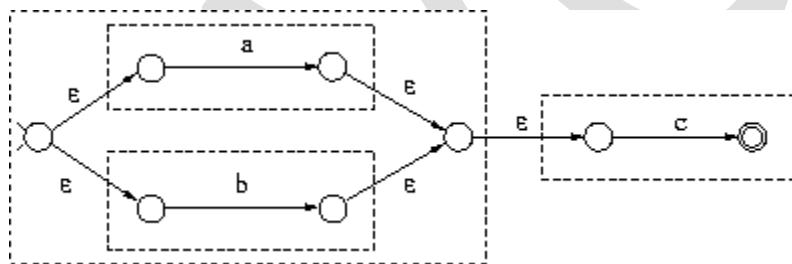
Clearly DFAs are a subset of NFAs. But it turns out that DFAs and NFAs have the same expressive power. The problem is that when converting a NFA to a DFA we may get an exponential blowup in the number of states.

We will first learn how to convert a RE into a NFA. This is the easy part. There are only 5 rules, one for each type of RE:



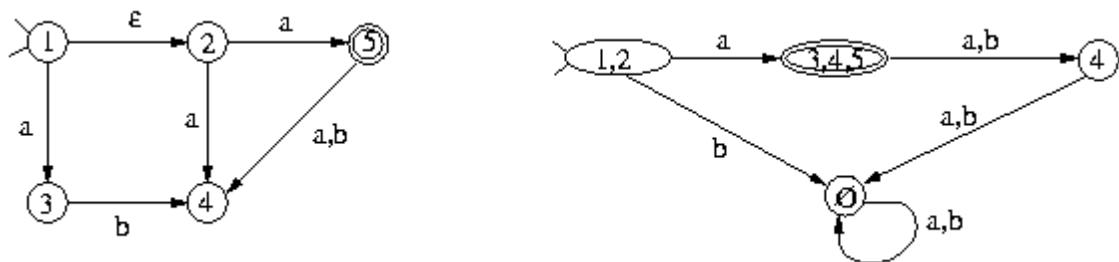
As it can be shown inductively, the above rules construct NFAs with only one final state. For example, the third rule indicates that, to construct the NFA for the RE  $AB$ , we construct the NFAs for  $A$  and  $B$ , which are represented as two boxes with one start state and one final state for each box. Then the NFA for  $AB$  is constructed by connecting the final state of  $A$  to the start state of  $B$  using an empty transition.

For example, the RE  $(a|b)c$  is mapped to the following NFA:



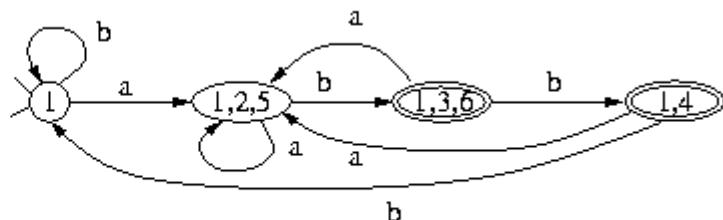
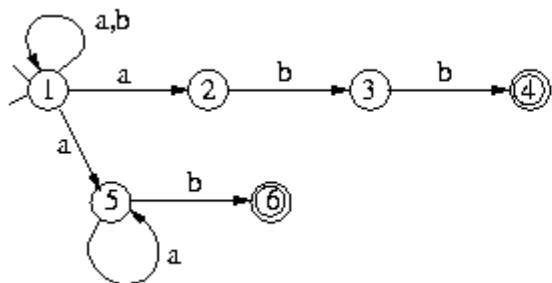
The next step is to convert a NFA to a DFA (called *subset construction*). Suppose that you assign a number to each NFA state. The DFA states generated by subset construction have sets of numbers, instead of just one number. For example, a DFA state may have been assigned the set  $\{5, 6, 8\}$ . This indicates that arriving to the state labeled  $\{5, 6, 8\}$  in the DFA is the same as arriving to the state 5, the state 6, or the state 8 in the NFA when parsing the same input. (Recall that a particular input sequence when parsed by a DFA, leads to a unique state, while when parsed by a NFA it may lead to multiple states.)

First we need to handle transitions that lead to other states for free (without consuming any input). These are the  $\epsilon$  transitions. We define the *closure* of a NFA node as the set of all the nodes reachable by this node using zero, one, or more  $\epsilon$  transitions. For example, The closure of node 1 in the left figure below



is the set  $\{1, 2\}$ . The start state of the constructed DFA is labeled by the closure of the NFA start state. For every DFA state labeled by some set  $\{s_1, \dots, s_n\}$  and for every character  $c$  in the language alphabet, you find all the states reachable by  $s_1, s_2, \dots$ , or  $s_n$  using  $c$  arrows and you union together the closures of these nodes. If this set is not the label of any other node in the DFA constructed so far, you create a new DFA node with this label. For example, node  $\{1, 2\}$  in the DFA above has an arrow to a  $\{3, 4, 5\}$  for the character  $a$  since the NFA node 3 can be reached by 1 on  $a$  and nodes 4 and 5 can be reached by 2. The  $b$  arrow for node  $\{1, 2\}$  goes to the error node which is associated with an empty set of NFA nodes.

The following NFA recognizes  $(a|b)^*(abb|a^+b)$ , even though it wasn't constructed with the above RE-to-NFA rules. It has the following DFA:



## UNIT III SYNTAX ANALYSIS

### **1. Define parser.**

Hierarchical analysis is one in which the tokens are grouped hierarchically into nested collections with collective meaning.

Also termed as Parsing.

### **2. Mention the basic issues in parsing.**

There are two important issues in parsing.

- Specification of syntax
- Representation of input after parsing.

### **3. Why lexical and syntax analyzers are separated out?**

Reasons for separating the analysis phase into lexical and syntax analyzers:

- Simpler design.
- Compiler efficiency is improved.
- Compiler portability is enhanced.

### **4. Define a context free grammar.**

A context free grammar G is a collection of the following

- V is a set of non terminals
- T is a set of terminals
- S is a start symbol
- P is a set of production rules

G can be represented as  $G = (V, T, S, P)$

Production rules are given in the following form

Non terminal  $\rightarrow (V \cup T)^*$

### **5. Briefly explain the concept of derivation.**

Derivation from S means generation of string w from S. For constructing derivation two things are important.

i) Choice of non terminal from several others.

ii) Choice of rule from production rules for corresponding non terminal.

Instead of choosing the arbitrary non terminal one can choose

i) either leftmost derivation – leftmost non terminal in a sentinel form

ii) or rightmost derivation – rightmost non terminal in a sentinel form

**6. Define ambiguous grammar.**

A grammar G is said to be ambiguous if it generates more than one parse tree for some sentence of language L(G).

i.e. both leftmost and rightmost derivations are same for the given sentence.

**7. What is a operator precedence parser?**

A grammar is said to be operator precedence if it possess the following properties:

1. No production on the right side is  $\epsilon$ .
2. There should not be any production rule possessing two adjacent non terminals at the right hand side.

**8. List the properties of LR parser.**

1. LR parsers can be constructed to recognize most of the programming languages for which the context free grammar can be written.
2. The class of grammar that can be parsed by LR parser is a superset of class of grammars that can be parsed using predictive parsers.
3. LR parsers work using non backtracking shift reduce technique yet it is efficient one.

**9. Mention the types of LR parser.**

- SLR parser- simple LR parser
- LALR parser- lookahead LR parser
- Canonical LR parser

**10. What are the problems with top down parsing?**

The following are the problems associated with top down parsing:

- Backtracking
- Left recursion
- Left factoring
- Ambiguity

**11. Write the algorithm for FIRST and FOLLOW.****FIRST**

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

## FOLLOW

1. Place \$ in FOLLOW(S), where S is the start symbol and \$ is the input right endmarker.
2. If there is a production  $A \rightarrow \alpha B \beta$ , then everything in FIRST( $\beta$ ) except for  $\epsilon$  is placed in FOLLOW(B).
3. If there is a production  $A \rightarrow \alpha B$ , or a production  $A \rightarrow \alpha B \beta$  where FIRST( $\beta$ ) contains  $\epsilon$ , then everything in FOLLOW(A) is in FOLLOW(B).

**12. List the advantages and disadvantages of operator precedence parsing.**

### Advantages

This type of parsing is simple to implement.

### Disadvantages

1. The operator like minus has two different precedence(unary and binary). Hence it is hard to handle tokens like minus sign.
2. This kind of parsing is applicable to only small class of grammars.

**13. What is dangling else problem?**

Ambiguity can be eliminated by means of dangling-else grammar which is shown below:

```
stmt → if expr then stmt
| if expr then stmt else stmt
| other
```

**14. Write short notes on YACC.**

YACC is an automatic tool for generating the parser program.

YACC stands for Yet Another Compiler Compiler which is basically the utility available from UNIX.

Basically YACC is LALR parser generator.

It can report conflict or ambiguities in the form of error messages.

**15. What is meant by handle pruning?**

A rightmost derivation in reverse can be obtained by handle pruning.

If w is a sentence of the grammar at hand, then  $w = \gamma_n$ , where  $\gamma_n$  is the nth right-sentential form of some as yet unknown rightmost derivation

$$S = \gamma_0 \Rightarrow \gamma_1 \dots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = w$$

**16. Define LR(0) items.**

An LR(0) item of a grammar G is a production of G with a dot at some position of the right side. Thus, production  $A \rightarrow XYZ$  yields the four items

$A \rightarrow .XYZ$

$A \rightarrow X.YZ$

$A \rightarrow XY.Z$

$A \rightarrow XYZ.$

**17. What is meant by viable prefixes?**

The set of prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called viable prefixes. An equivalent definition of a viable prefix is that it is a prefix of a right sentential form that does not continue past the right end of the rightmost handle of that sentential form.

**18. Define handle.**

A handle of a string is a substring that matches the right side of a production, and whose reduction to the nonterminal on the left side of the production represents one step along the reverse of a rightmost derivation.

A handle of a right – sentential form  $\gamma$  is a production  $A \rightarrow \beta$  and a position of  $\gamma$  where the string  $\beta$  may be found and replaced by  $A$  to produce the previous right-sentential form in a rightmost derivation of  $\gamma$ . That is , if  $S \Rightarrow^* \alpha Aw \Rightarrow^* \alpha \beta w$ , then  $A \rightarrow \beta$  in the position following  $\alpha$  is a handle of  $\alpha \beta w$ .

**19. What are kernel & non-kernel items?**

**Kernel items**, which include the initial item,  $S' \rightarrow .S$ , and all items whose dots are not at the left end.

**Non-kernel items**, which have their dots at the left end.

**20. What is phrase level error recovery?**

Phrase level error recovery is implemented by filling in the blank entries in the predictive parsing table with pointers to error routines. These routines may change, insert, or delete symbols on the input and issue appropriate error messages. They may also pop from the stack.

**PART B**

- 1) Construct a predictive parsing table for the grammar  
 $E \rightarrow E + T / F$

$T \rightarrow T * F / F$

$F \rightarrow (E) / id$

- 2) Give the LALR parsing table for the grammar.

$S \rightarrow L = R / R$

$L \rightarrow * R / id$

$R \rightarrow L$

- 3) Consider the grammar

$E \rightarrow TE'$

$E' \rightarrow + TE' / E$

$T \rightarrow FT'$

$T' \rightarrow *FT' / E$

$F \rightarrow (E) / id$

Construct a predictive parsing table for the grammar shown above. Verify whether the input string

$id + id * id$  is accepted by the grammar or not.

- 4) Consider the grammar.

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E) / id$

Construct an LR parsing table for the above grammar. Give the moves of the LR parser on

$id * id + id$

- 5) For the grammar given below, calculate the operator precedence relation and

the precedence functions

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

$\mid E * E \mid E \mid E \mid$

$\mid E ^ E \mid (E) \mid$

$\mid -E \mid id$

- 6) Compare top down parsing and bottom up parsing methods.
- 7) What are LR parsers? Explain with a diagram the LR parsing algorithm.
- 8) What are parser generators?
- 9) Explain recursive descent parser with appropriate examples.
- 10) Compare SLR, LALR and LR parses.

## UNIT IV - SYNTAX DIRECTED TRANSLATION & RUN TIME ENVIRONMENT

### **1. List the different storage allocation strategies.**

The strategies are:

- Static allocation
- Stack allocation
- Heap allocation

### **2. What are the contents of activation record?**

The activation record is a block of memory used for managing the information needed by a single execution of a procedure. Various fields of activation record are:

- Temporary variables
- Local variables
- Saved machine registers
- Control link
- Access link
- Actual parameters
- Return values

### **3. What is dynamic scoping?**

In dynamic scoping a use of non-local variable refers to the non-local data declared in most recently called and still active procedure. Therefore each time new findings are set up for local names called procedure. In dynamic scoping symbol tables can be required at run time.

### **4. Define symbol table.**

Symbol table is a data structure used by the compiler to keep track of semantics of the variables. It stores information about scope and binding information about names.

### **5. What is code motion?**

Code motion is an optimization technique in which amount of code in a loop is decreased. This transformation is applicable to the expression that yields the same result independent of the number of times the loop is executed. Such an expression is placed before the loop.

### **6. What are the properties of optimizing compiler?**

The source code should be such that it should produce minimum amount of target code.

There should not be any unreachable code.

Dead code should be completely removed from source language.

The optimizing compilers should apply following code improving transformations on source language.

- i) common subexpression elimination
- ii) dead code elimination
- iii) code movement
- iv) strength reduction

**7. What are the various ways to pass a parameter in a function?**

- Call by value
- Call by reference
- Copy-restore
- Call by name

**8. Suggest a suitable approach for computing hash function.**

Using hash function we should obtain exact locations of name in symbol table.

The hash function should result in uniform distribution of names in symbol table.

The hash function should be such that there will be minimum number of collisions. Collision is such a situation where hash function results in same location for storing the names.

**9. Define bottom up parsing?**

It attempts to construct a parse tree for an input string is beginning at leaves and working up towards the root (i.e.) reducing a string „w“ to the start symbol of a grammar. At each reduction step, a particular substring matching the right side of a production is replaced by the symbol on the left of that production. It is a rightmost derivation and it's also known as shifts reduce parsing.

**10. What are the functions used to create the nodes of syntax trees?**

- Mknnode (op, left, right)
- Mkleaf (id,entry)
- Mkleaf (num, val)

**11. What are the functions for constructing syntax trees for expressions?**

- i) The construction of a syntax tree for an expression is similar to the translation of the expression into postfix form.
- ii) Each node in a syntax tree can be implemented as a record with several fields.

**12. Give short note about call-by-name?**

Call by name, at every reference to a formal parameter in a procedure body the name of the corresponding actual parameter is evaluated. Access is then made to the effective parameter.

**13. How parameters are passed to procedures in call-by-value method?**

This mechanism transmits values of the parameters of call to the called program. The transfer is one way only and therefore the only way to returned can be the value of a function.

```

Main ( )
{
    print (5);
}
Int
Void print (int n)
{
    printf ("%d", n);
}

```

**14. Define static allocations and stack allocations**

**Static allocation** is defined as lays out for all data objects at compile time.

Names are bound to storage as a program is compiled, so there is no need for a run time support package.

**Stack allocation** is defined as process in which manages the run time as a Stack. It is based on the idea of a control stack; storage is organized as a stack, and activation records are pushed and popped as activations begin and end.

**15. What are the difficulties with top down parsing?**

- a) Left recursion
- b) Backtracking
- c) The order in which alternates are tried can affect the language accepted
- d) When failure is reported. We have very little idea where the error actually occurred.

**16. Define top down parsing?**

It can be viewed as an attempt to find the left most derivation for an input string. It can be viewed as attempting to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder.

**17. Define a syntax-directed translation?**

Syntax-directed translation specifies the translation of a construct in terms of Attributes associated with its syntactic components. Syntax-directed translation uses a context free grammar to specify the syntactic structure of the input. It is an input- output mapping.

**18. Define an attribute. Give the types of an attribute?**

An attribute may represent any quantity, with each grammar symbol, it associates a set of attributes and with each production, a set of semantic rules for computing values of the attributes associated with the symbols appearing in that production.

**Example:** a type, a value, a memory location etc.,

- i) Synthesized attributes.
- ii) Inherited attributes.

**19. Give the 2 attributes of syntax directed translation into 3-addr code?**

- i) E.place, the name that will hold the value of E and
- ii) E.code , the sequence of 3-addr statements evaluating E.

**20. Write the grammar for flow-of-control statements?**

The following grammar generates the flow-of-control statements, if-then, if-then-else, and while-do statements.

```

S -> if E
      then S1
      | If E then S1
          else S2
          | While E do S1.
    
```

**PART B**

- 1) Discuss in detail about the run time storage arrangement.
- 2) What are different storage allocation strategies. Explain.
- 3) Write in detail about the issues in the design of code generator.
- 4) Explain how declarations are done in a procedure using syntax directed translations.
- 5) What is a three address code? Mention its types. How would you implement these address statements? Explain with suitable examples.
- 6) Write syntax directed translation for arrays.



## UNIT V CODE OPTIMISATION AND CODE GENERATION

### 1. Define code generations with ex?

It is the final phase in compiler model and it takes as an input an intermediate representation of the source program and output produces as equivalent target programs. Then intermediate instructions are each translated into a sequence of machine instructions that perform the same task.

### 2. What are the issues in the design of code generator?

- Input to the generator
- Target programs
- Memory management
- Instruction selection
- Register allocation
- Choice of evaluation order
- Approaches to code generation.

### 3. Give the variety of forms in target program.

- Absolute machine language.
- Relocatable machine language.
- Assembly language.

### 4. Give the factors of instruction selections.

- Uniformity and completeness of the instruction sets
- Instruction speed and machine idioms
- Size of the instruction sets.

### 5. What are the sub problems in register allocation strategies?

- During register allocation, we select the set of variables that will reside in register at a point in the program.
- During a subsequent register assignment phase, we pick the specific register that a variable reside in.

### 6. Give the standard storage allocation strategies.

- Static allocation
- Stack allocation.

### 7. Define static allocations and stack allocations

**Static allocation** is defined as lays out for all data objects at compile time. Names are bound to storage as a program is compiled, so there is no need for a Run time support package.

**Stack allocation** is defined as process in which manages the run time as a Stack. It is based on the idea of a control stack; storage is organized as a stack, And activation records are pushed and popped as activations begin and end.

**8. Write the addressing mode and associated costs in the target machine.**

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

**9. Define basic block and flow graph.**

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

A flow graph is defined as the adding of flow of control information to the Set of basic blocks making up a program by constructing a directed graph.

**10. Write the step to partition a sequence of 3 address statements into basic blocks.**

1. First determine the set of leaders, the first statement of basic blocks.

- The rules we can use are the following.
- The first statement is a leader.
- Any statement that is the target of a conditional or unconditional goto is a leader.
- Any statement that immediately follows a goto or conditional goto statement is a leader.

2. For each leader, its basic blocks consists of the leader and all statements

Up to but not including the next leader or the end of the program.

**11. Give the important classes of local transformations on basic blocks**

- Structure preservation transformations
- Algebraic transformations.

**12. Describe algebraic transformations.**

It can be used to change the set of expressions computed by a basic blocks into A algebraically equivalent sets. The useful ones are those that simplify the Expressions place expensive operations by cheaper ones.

$$X = X + 0$$

$$X = X * 1$$

**13. What is meant by register descriptors and address descriptors?**

A register descriptor keeps track of what is currently in each register. It is Consulted whenever a new register is needed.

An address descriptor keeps track of the location where ever the current Value of the name can be found at run time. The location might be a register, a Stack location, a memory address,

**14. What are the actions to perform the code generation algorithms?**

- Invoke a function get reg to determine the location L.
- Consult the address descriptor for y to determine y", the current location of y.
- If the current values of y and/or z have no next uses, are not live on exit from the block, and are in register, alter the register descriptor.

**15. Write the code sequence for the  $d:=(a-b)+(a-c)+(a-c)$ .**

Statement	Code generation	Register descriptor	Address descriptor
$t:=a-b$	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

**16. Write the labels on nodes in DAG.**

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

- Leaves are labeled by unique identifiers, either variable names or constants.
- Interior nodes are labeled by an operator symbol.
- Nodes are also optionally given a sequence of identifiers for labels.

**17. Give the applications of DAG.**

- Automatically detect the common sub expressions
- Determine which identifiers have their values used in the block.
- Determine which statements compute values that could be used outside the blocks.

**18. Define Peephole optimization.**

A Statement by statement code generation strategy often produces target code that contains redundant instructions and suboptimal constructs. “Optimizing” is misleading because there is no guarantee that the resulting code is optimal. It is a method for trying to improve the performance of the target program by examining the short sequence of target instructions and replacing this instructions by shorter or faster sequence.

**19. Write the characteristics of peephole optimization?**

- Redundant-instruction elimination
- Flow-of-control optimizations.
- Algebraic simplifications
- Use of machine idioms

**20. What are the structure preserving transformations on basic blocks?**

- Common sub-expression elimination
- Dead-code elimination
- Renaming of temporary variables
- Interchange of two independent adjacent statement

**21. Define Common sub-expression elimination with ex.**

It is defined as the process in which eliminate the statements which has the same expressions. Hence this basic block may be transformed into the equivalent Block.

**Ex:**

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

**After elimination:**

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

**22. Define Dead-code elimination with ex.**

It is defined as the process in which the statement  $x=y+z$  appear in a basic block, where  $x$  is a dead that is never subsequently used. Then this statement maybe safely removed without changing the value of basic blocks.

**23. Define Renaming of temporary variables with ex.**

We have the statement  $u:=b + c$ , where  $u$  is a new temporary variable, and change all uses of this instance of  $t$  to  $u$ , then the value of the basic block is not changed.

**24. Define reduction in strength with ex.**

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machines. Certain machine instructions are cheaper than others and can often be used as special cases of more expensive operators. Ex:

$X^2$  is invariably cheaper to implement as  $x*x$  than as a call to an exponentiation routine.

**25. Define use of machine idioms.**

The target machine may have harder instructions to implement certain specific operations efficiently. Detecting situations that permit the use of these instructions can reduce execution time significantly.

**26. Define code optimization and optimizing compiler**

The term **code-optimization** refers to techniques a compiler can employ in an attempt to produce a better object language program than the most obvious for a given source program.

Compilers that apply code-improving transformations are called **Optimizing-compilers**.

**PART B:**

1. What are the issues in the design of code generator? Explain in detail.
2. Discuss about the run time storage management.
3. Explain basic blocks and flow graphs.
4. Explain about transformation on a basic block.
5. Write a code generation algorithm. Explain about the descriptor and function getreg(). Give an example.
6. Explain peephole optimization
7. Explain DAG representation of basic blocks.
8. Explain principle sources of code optimization in details.
9. Explain the Source language issues with details.
10. Explain the Storage organization strategies with examples.
11. Explain storage allocation strategy.
12. Explain about Parameter passing.
13. Explain the non local names in runtime storage managements.
14. Explain about activation records and its purpose.
15. Explain about Optimization of basic blocks.
16. Explain the various approaches to compiler development.
17. Explain simple code generator with suitable example.
18. Discuss about the following:
  - a) Copy Propagation
  - b) Dead-code Elimination
  - c) Code motion

## **UNIT I INTRODUCTION TO COMPILER**

**1. What is a Complier?**

A Complier is a program that reads a program written in one language-the source language-and translates it in to an equivalent program in another language-the target language . As an important part of this translation process, the compiler reports to its user the presence of errors in the source program

**2. State some software tools that manipulate source program?**

- i. Structure editors
- ii. Pretty printers
- iii. Static
- iv. checkers
- v. Interpreters.

**3. What are the cousins of compiler? April/May 2004, April/May 2005**

The following are the cousins of

- i. Preprocessors
- ii. Assemblers
- iii. Loaders
- iv. Link editors.

**4. What are the main two parts of compilation? What are they performing?**

The two main parts are

- **Analysis** part breaks up the source program into constituent pieces and creates an intermediate representation of the source program.
- **Synthesis** part constructs the desired target program from the intermediate representation

**5. What is a Structure editor?**

A structure editor takes as input a sequence of commands to build a source program .The structure editor not only performs the text creation and modification functions of an ordinary text editor but it also analyzes the program text putting an appropriate hierarchical structure on the source program.

**6. What are a Pretty Printer and Static Checker?**

- A Pretty printer analyses a program and prints it in such a way that the structure of the program becomes clearly visible.
- A static checker reads a program, analyses it and attempts to discover potential bugs without running the program.

**7. How many phases does analysis consists?**

Analysis consists of three phases

- i .Linear analysis
- ii. Hierarchical analysis
- iii. Semantic analysis

**8. What happens in linear analysis?**

This is the phase in which the stream of characters making up the source program is read from left to right and grouped in to tokens that are sequences of characters having collective

**9. What happens in Hierarchical analysis?**

This is the phase in which characters or tokens are grouped hierarchically in to nested collections with collective meaning.

**10. What happens in Semantic analysis?**

This is the phase in which certain checks are performed to ensure that the components of a program fit together meaningfully.

**11. State some compiler construction tools?Arpil /May 2008**

- i. Parse generator
- ii. Scanner generators
- iii. Syntax-directed translation engines
- iv. Automatic code generator
- v. Data flow engines.

**12. What is a Loader? What does the loading process do?**

A Loader is a program that performs the two functions

- i. Loading
- ii .Link editing

The process of loading consists of taking relocatable machine code, altering the relocatable address and placing the altered instructions and data in memory at the proper locations.

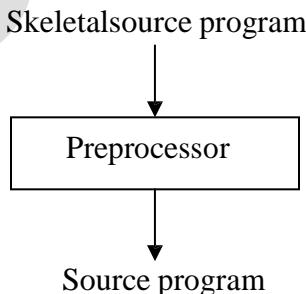
**13. What does the Link Editing does?**

**Link editing:** This allows us to make a single program from several files of relocatable machine code. These files may have been the result of several compilations, and one or more may be library files of routines provided by the system and available to any program that needs them.

**14. What is a preprocessor?**

A preprocessor is one, which produces input to compilers. A source program may be divided into modules stored in separate files. The task of collecting the source program is sometimes entrusted to a distinct program called a preprocessor.

The preprocessor may also expand macros into source language statements.



**15. State some functions of Preprocessors**

- i) Macro processing
- ii) File inclusion
- iii) Relational Preprocessors

**16. What is a Symbol table?**

A Symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier. The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly.

**17. State the general phases of a compiler**

- i. Lexical analysis
- ii. Syntax analysis
- iii. Semantic analysis
- iv. Intermediate code generation
- v. Code optimization
- vi. Code generation

**18. What is an assembler?**

Assembler is a program, which converts the source language in to assembly language.

**19. What is the need for separating the analysis phase into lexical analysis and parsing?**

**(Or) What are the issues of lexical analyzer?**

- Simpler design is perhaps the most important consideration. The separation of lexical analysis from syntax analysis often allows us to simplify one or the other of these phases.
- Compiler efficiency is improved.
- Compiler portability is enhanced.

**20. What is Lexical Analysis?**

The first phase of compiler is Lexical Analysis. This is also known as linear analysis in which the stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of characters having a collective meaning.

**21. What is a lexeme? Define a regular set. Nov/Dec 2006**

- A Lexeme is a sequence of characters in the source program that is matched by the pattern for a token.
- A language denoted by a regular expression is said to be a regular set

**22. What is a sentinel? What is its usage? April/May 2004**

A Sentinel is a special character that cannot be part of the source program. Normally we use 'eof' as the sentinel. This is used for speeding-up the lexical analyzer.

**23. What is a regular expression? State the rules, which define regular expression?**

Regular expression is a method to describe regular language

**Rules:**

- 1)  $\epsilon$ -is a regular expression that denotes  $\{\epsilon\}$  that is the set containing the empty string
- 2) If  $a$  is a symbol in  $\Sigma$ ,then  $a$  is a regular expression that denotes  $\{a\}$
- 3) Suppose  $r$  and  $s$  are regular expressions denoting the languages  $L(r)$  and  $L(s)$  Then,
  - a)  $(r)/(s)$  is a regular expression denoting  $L(r) \cup L(s)$ .
  - b)  $(r)(s)$  is a regular expression denoting  $L(r)L(s)$
  - c)  $(r)^*$  is a regular expression denoting  $L(r)^*$ .
  - d)  $(r)$  is a regular expression denoting  $L(r)$ .

**24. What are the Error-recovery actions in a lexical analyzer?**

1. Deleting an extraneous character
2. Inserting a missing character
3. Replacing an incorrect character by a correct character
4. Transposing two adjacent characters

**25. Construct Regular expression for the language**

$L = \{w \in \{a,b\}^*/w \text{ ends in } abb\}$

Ans:  $\{a/b\}^*abb$ .

**26. What is recognizer?**

Recognizers are machines. These are the machines which accept the strings belonging to certain language. If the valid strings of such language are accepted by the machine then it is said that the corresponding language is accepted by that machine, otherwise it is rejected.

**16 MARKS**

**1.PHASES OF COMPILER**

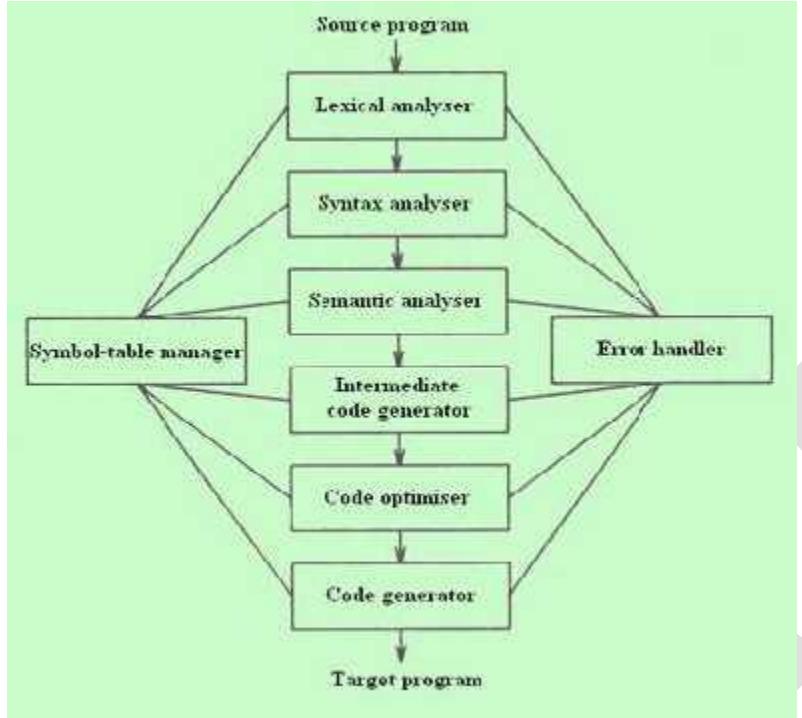
A Compiler operates in phases, each of which transforms the source program from one representation into another. The following are the phases of the compiler:

**Main phases:**

- 1) Lexical analysis
- 2) Syntax analysis
- 3) Semantic analysis
- 4) Intermediate code generation
- 5) Code optimization
- 6) Code generation

**Sub-Phases:**

- 1) Symbol table management
- 2) Error handling

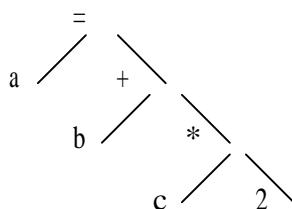


### LEXICAL ANALYSIS:

- It is the first phase of the compiler. It gets input from the source program and produces tokens as output.
- It reads the characters one by one, starting from left to right and forms the tokens.
- **Token** : It represents a logically cohesive sequence of characters such as keywords,
  - operators, identifiers, special symbols etc.
  - Example:  $a + b = 20$
  - Here,  $a, b, +, =, 2, 0$  are all separate tokens.
  - Group of characters forming a token is called the **Lexeme**.
- The lexical analyser not only generates a token but also enters the lexeme into the symbol table if it is not already there.

### SYNTAX ANALYSIS:

- It is the second phase of the compiler. It is also known as parser.
- It gets the token stream as input from the lexical analyser of the compiler and generates
  - syntax tree as the output.
- Syntax tree:
  - It is a tree in which interior nodes are operators and exterior nodes are operands.
- Example: For  $a=b+c*2$ , syntax tree is



## **SEMANTIC ANALYSIS:**

- It is the third phase of the compiler.
- It gets input from the syntax analysis as parse tree and checks whether the given syntax is correct or not.
- It performs type conversion of all the data types into real data types.

## **INTERMEDIATE CODE GENERATION:**

- It is the fourth phase of the compiler.
- It gets input from the semantic analysis and converts the input into output as intermediate code such as three address code.
- The three-address code consists of a sequence of instructions, each of which has atmost three operands.  
Example:  $t1=t2+t3$

## **CODE OPTIMIZATION:**

- It is the fifth phase of the compiler.
- It gets the intermediate code as input and produces optimized intermediate code as output.
- This phase reduces the redundant code and attempts to improve the intermediate code so that faster-running machine code will result.
- During the code optimization, the result of the program is not affected.
- To improve the code generation, the optimization involves
  - deduction and removal of dead code (unreachable code).
  - calculation of constants in expressions and terms.
  - collapsing of repeated expression into temporary string.
  - loop unrolling.
  - moving code outside the loop.
  - removal of unwanted temporary variables.

## **CODE GENERATION:**

- It is the final phase of the compiler.
- It gets input from code optimization phase and produces the target code or object code as result.
- Intermediate instructions are translated into a sequence of machine instructions that perform the same task.
- The code generation involves
  - allocation of register and memory
  - generation of correct references
  - generation of correct data types
  - generation of missing code

## **SYMBOL TABLE MANAGEMENT:**

- Symbol table is used to store all the information about identifiers used in the program.
- It is a data structure containing a record for each identifier, with fields for the attributes of the identifier.
- It allows to find the record for each identifier quickly and to store or retrieve data from that record.
- Whenever an identifier is detected in any of the phases, it is stored in the symbol table.

## ERROR HANDLING:

- Each phase can encounter errors. After detecting an error, a phase must handle the error so that compilation can proceed.
- In lexical analysis, errors occur in separation of tokens.
  - In syntax analysis, errors occur during construction of syntax tree.
  - In semantic analysis, errors occur when the compiler detects constructs with right syntactic structure but no meaning and during type conversion.
  - In code optimization, errors occur when the result is affected by the optimization.
  - In code generation, it shows error when code is missing etc.

## 2.COMPLIER CONSTRUCTION TOOLS

These are specialized tools that have been developed for helping implement various phases of a compiler. The following are the compiler construction tools:

### 1) Parser Generators:

- These produce syntax analyzers, normally from input that is based on a context-free grammar.
- It consumes a large fraction of the running time of a compiler.
- Example-YACC (Yet another Compiler-Compiler).

### 2) Scanner Generator:

- These generate lexical analyzers, normally from a specification based on regular expressions.
- The basic organization of lexical analyzers is based on finite automaton.

### 3) Syntax-Directed Translation:

- These produce routines that walk the parse tree and as a result generate intermediate code.
- Each translation is defined in terms of translations at its neighbor nodes in the tree.

### 4) Automatic Code Generators:

- It takes a collection of rules to translate intermediate language into machine language. The rules must include sufficient details to handle different possible access methods for data.

### 5) Data-Flow Engines:

- It does code optimization using data-flow analysis, that is, the gathering of information about how values are transmitted from one part of a program to each other part.

Shri Vishnu Engineering College For Women

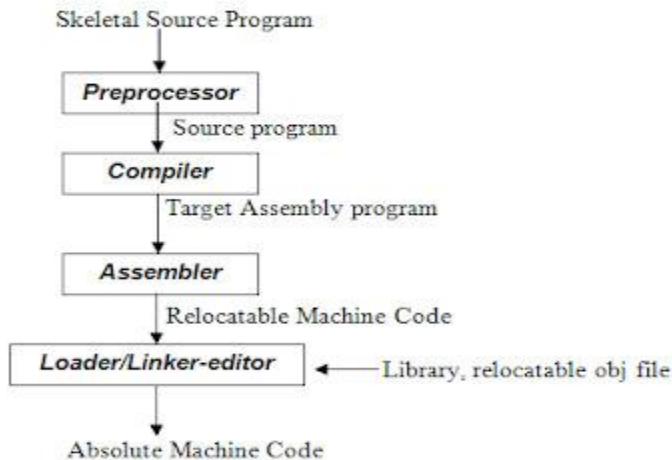
# COMPILER DESIGN LECTURE NOTES



**DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING  
SHRI VISHNU ENGINEERING COLLEGE FOR WOMEN**  
(Approved by AICTE, Accredited by NBA, Affiliated to JNTU Kakinada)  
**BHIMAVARAM – 534 202**

## UNIT -1

### 1.1 OVERVIEW OF LANGUAGE PROCESSING SYSTEM



**Fig 1.1 Language -processing System**

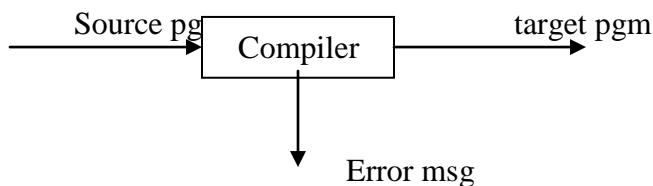
### 1.2 Preprocessor

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

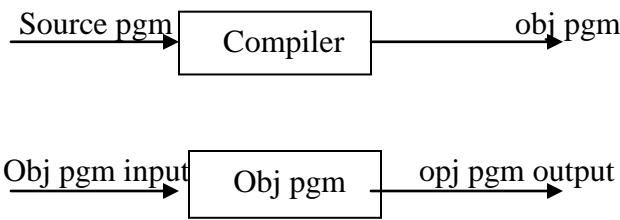
1. *Macro processing*: A preprocessor may allow a user to define macros that are short hands for longer constructs.
2. *File inclusion*: A preprocessor may include header files into the program text.
3. *Rational preprocessor*: these preprocessors augment older languages with more modern flow-of-control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

### 1.3 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.

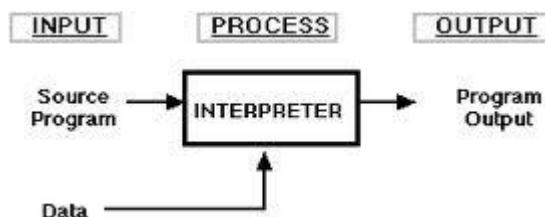


Executing a program written in HLL programming language is basically of two parts. the source program must first be compiled translated into an object program. Then the results object program is loaded into a memory executed.



**1.4 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 assembler were written to automate the translation of assembly language into machine language. The input to an assembler program is called source program, the output is a machine language translation (object program).

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



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

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

#### *Advantages:*

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

***Disadvantages:***

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

**2 *Loader and Link-editor:***

Once the assembler procedures 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 execute. 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 over come this 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.

**1.6 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. Beside program translation, the translator performs another very important role, the error-detection. Any violation of d HLL specification would be detected and reported to the programmers. Important role of 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.

**1.7 TYPE OF TRANSLATORS:-**

- INTERPRETOR
- COMPILER
- PREPROSSESSOR

## 1.8 LIST OF COMPILERS

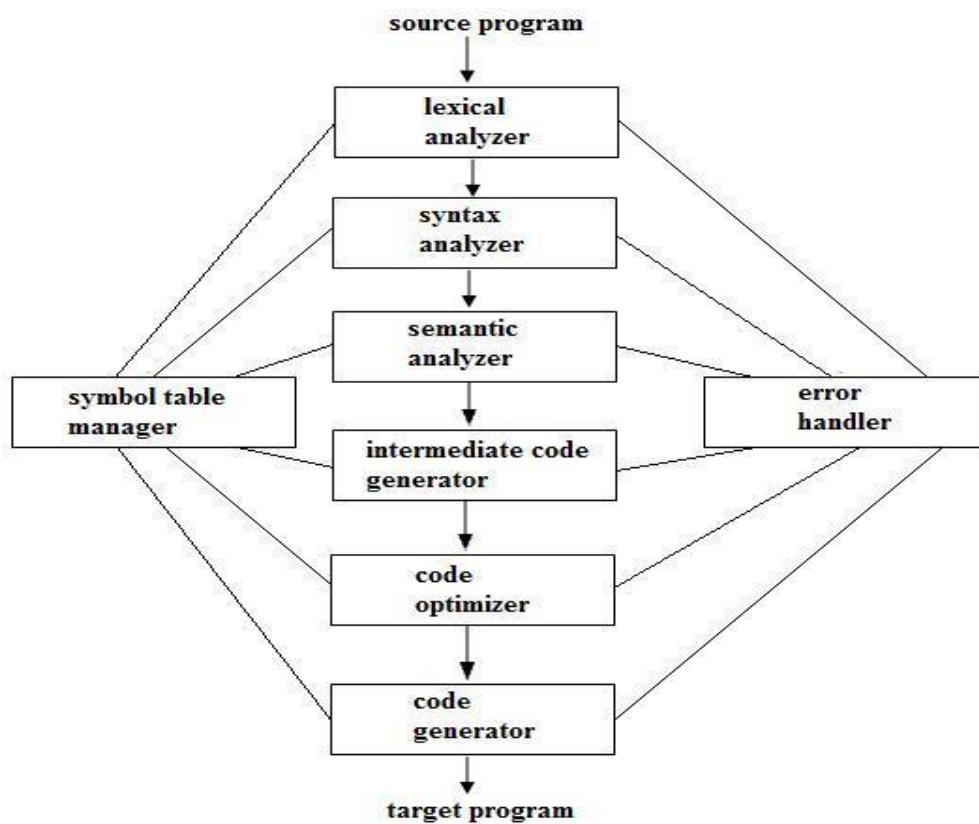
1. Ada compilers
- 2 .ALGOL compilers
- 3 .BASIC compilers
- 4 .C# compilers
- 5 .C compilers
- 6 .C++ compilers
- 7 .COBOL compilers
- 8 .D compilers
- 9 .Common Lisp compilers
10. ECMAScript interpreters
11. Eiffel compilers
12. Felix compilers
13. Fortran compilers
14. Haskell compilers
- 15 .Java compilers
16. Pascal compilers
17. PL/I compilers
18. Python compilers
19. Scheme compilers
20. Smalltalk compilers
21. CIL compilers

## 1.9 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’.

**Fig 1.5 Phases of a compiler****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**.

**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 program and records essential information about 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 same job as the original, but in a way that saves time and / or spaces.

- 1, Local Optimization:-

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

If **A > B** goto **L2**

Goto **L3**

**L2 :**

This can be replaced by a single statement

If **A < B** goto **L3**

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

**A := B + C + D**

**E := B + C + F**

Might be evaluated as

**T1 := B + C**

**A := T1 + D**

**E := T1 + F**

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

## 2, 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 :-

Cg 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 :-

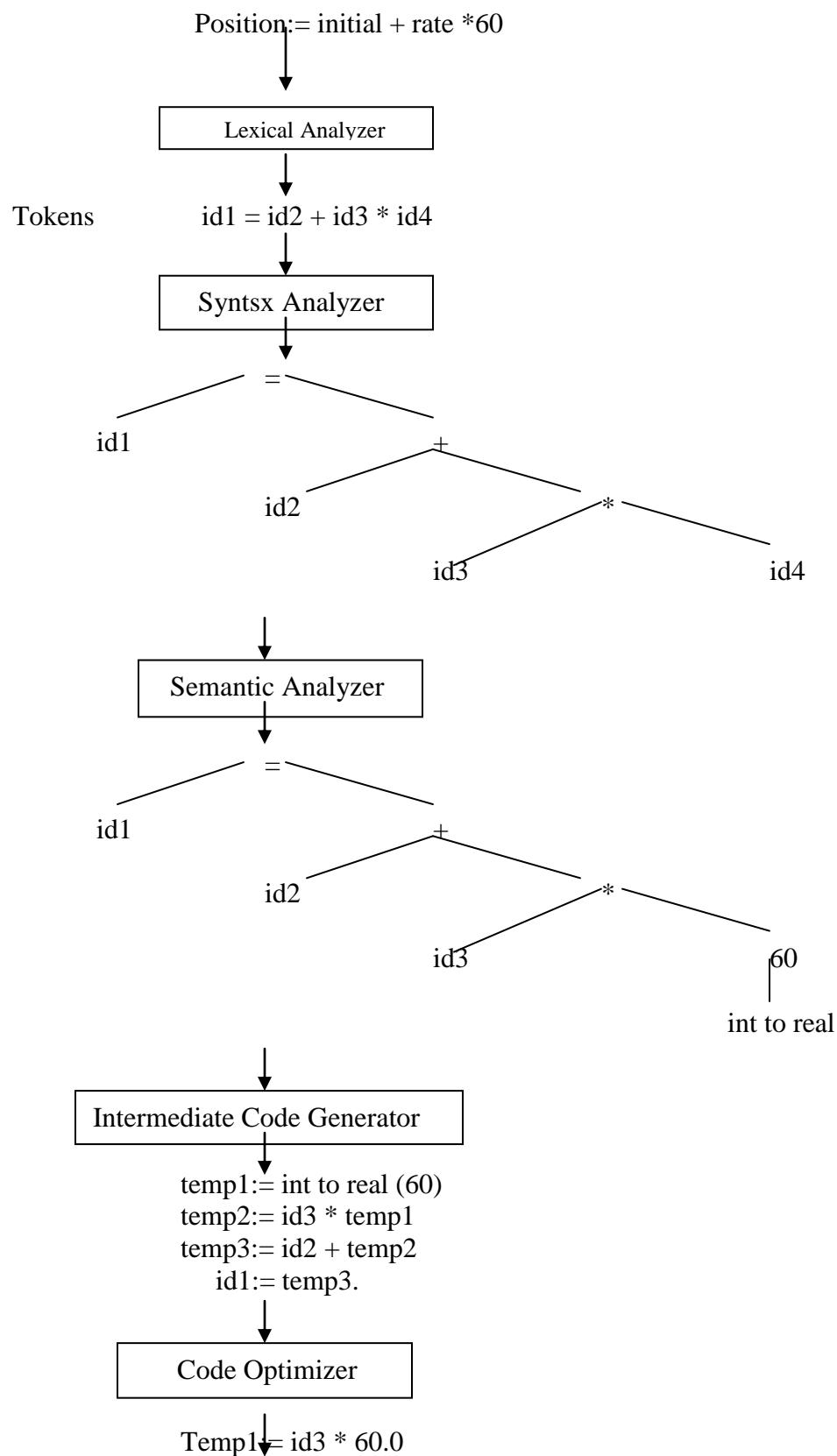
A compiler needs to collect information about all the data objects that appear in the source program. The information about data objects is collected by the early phases of the compiler-lexical and syntactic analyzers. The data structure used to record this information is called as Symbol Table.

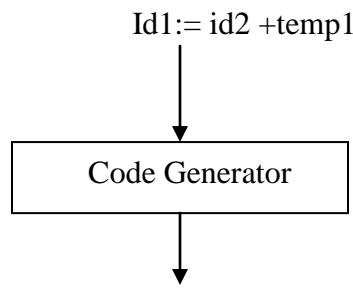
### Error Handling :-

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:





```

MOVF id3, r2
MULF *60.0, r2
MOVF id2, r2
ADDF r2, r1
MOVF r1, id1
  
```

### **1.10 TOKEN**

LA reads the source program one character at a time, carving the source program into a sequence of automatic units called ‘Tokens’.

1, Type of the token.

2, Value of the token.

Type : variable, operator, keyword, constant

Value : Name of variable, current variable (or) pointer to symbol table.

**If the symbols given in the standard format the LA accepts and produces token as output.** Each token is a sub-string of the program that is to be treated as a single unit. Token are two types.

1, Specific strings such as IF (or) semicolon.

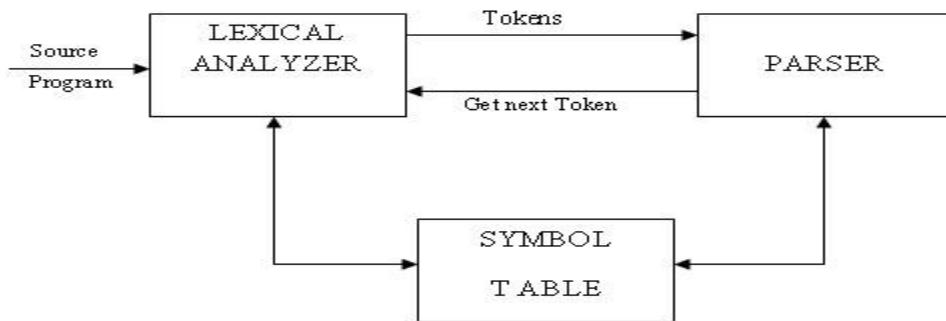
2, Classes of string such as identifiers, label, constants.

**UNIT -2****LEXICAL ANALYSIS****2.1 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.

**2.2 ROLE OF LEXICAL ANALYZER**

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



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

LA may also perform certain secondary tasks as the user interface. One such task is striping 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.

### 2.3 LEXICAL ANALYSIS VS PARSING:

Lexical analysis	Parsing
<p>A Scanner simply turns an input String (say a file) into a list of tokens. These tokens represent things like identifiers, parentheses, operators etc.</p> <p>The lexical analyzer (the "lexer") parses individual symbols from the source code file into tokens. From there, the "parser" proper turns those whole tokens into sentences of your grammar</p>	<p>A parser converts this list of tokens into a Tree-like object to represent how the tokens fit together to form a cohesive whole (sometimes referred to as a sentence).</p> <p>A parser does not give the nodes any meaning beyond structural cohesion. The next thing to do is extract meaning from this structure (sometimes called contextual analysis).</p>

### 2.4 TOKEN, LEXEME, PATTERN:

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

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

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

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

**Example:**

Description of token

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

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

## **2.5 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 your 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. A 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.

## **2.6 DIFFERENCE BETWEEN COMPILER AND INTERPRETER**

- A compiler converts the high level instruction into machine language while an interpreter converts the high level instruction into an intermediate form.
- Before execution, entire program is executed by the compiler whereas after translating the first line, an interpreter then executes it and so on.
- List of errors is created by the compiler after the compilation process while an interpreter stops translating after the first error.
- An independent executable file is created by the compiler whereas interpreter is required by an interpreted program each time.
- The compiler produces object code whereas interpreter does not produce object code.
- In the process of compilation the program is analyzed only once and then the code is generated whereas source program is interpreted every time it is to be executed and every time the source program is analyzed. hence interpreter is less efficient than compiler.
- Examples of interpreter: A *UPS Debugger* is basically a graphical source level debugger but it contains built-in C interpreter which can handle multiple source files. Example of compiler: *Borland C compiler* or *Turbo C* compiler compiles the programs written in C or C++.

## 2.7 REGULAR EXPRESSIONS

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

Component of regular expression..

<b>X</b>	<b>the character x</b>
.	<b>any character, usually accept a new line</b>
[x y z]	<b>any of the characters x, y, z, .....</b>
R?	<b>a R or nothing (=optionally as R)</b>
R*	<b>zero or more occurrences.....</b>
R+	<b>one or more occurrences .....</b>
R1R2	<b>an R1 followed by an R2</b>
R2R1	<b>either an R1 or an R2.</b>

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 .

- is a regular expression denoting { € }, that is, the language containing only the empty string.
- For each ‘a’ in  $\Sigma$ , 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<sub>r</sub>UL<sub>s</sub>  
 R.S means L<sub>r</sub>.L<sub>s</sub>  
 R\* denotes L<sub>r</sub>\*

## 2.8 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 | .....| Z | a | b |.....| z|  
 Digits - 0 | 1 | 2 | .... | 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)<sup>+</sup>

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
Any number	number	pointer to table entry
<	relop	LT

$\leq$	relop	LE
$=$	relop	ET
$< >$	relop	NE

## 2.9 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.

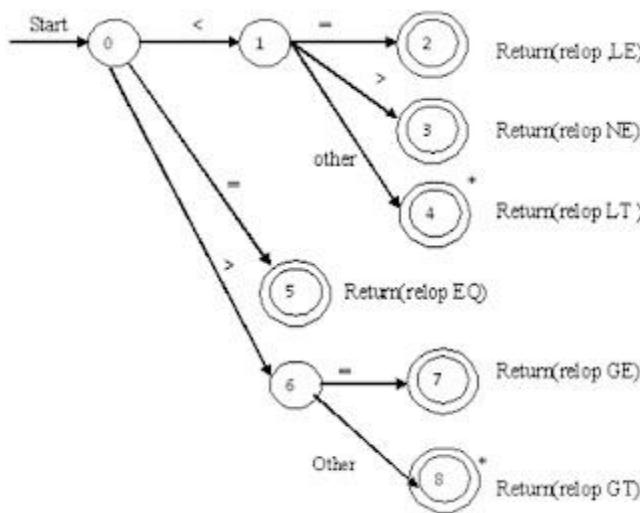
If we are in one state s, and the next input symbol is a, we look for an edge out of state s labeled by a. if we find such an edge ,we advance the forward pointer and enter the state of the transition diagram to which that edge leads.

### Some important conventions about transition diagrams are

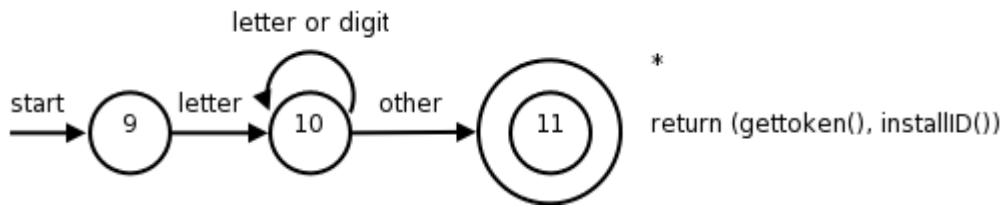
1. Certain states are said to be accepting or final .These states indicates that a lexeme has been found, although the actual lexeme may not consist of all positions b/w the lexeme Begin and forward pointers we always indicate an accepting state by a double circle.

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.



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



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

If	=	if
Then	=	then
Else	=	else
Relop	=	<   <=   =   >   >=
Id	=	letter (letter   digit) *
Num	=	digit

## 2.10 AUTOMATA

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

1, an automation in which the output depends only on the input is **called an automation without memory**.

2, an automation in which the output depends on the input and state also is **called as automation with memory**.

3, an automation in which the output depends only on the state of the machine is **called a Moore machine**.

3, an automation in which the output depends on the state and input at any instant of time is **called a mealy machine**.

## 2.11 DESCRIPTION OF AUTOMATA

1, an automata has a mechanism to read input from input tape,

2, any language is recognized by some automation, Hence these automation are basically language ‘acceptors’ or ‘language recognizers’.

### Types of Finite Automata

- Deterministic Automata
- Non-Deterministic Automata.

## 2.12 DETERMINISTIC AUTOMATA

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

- 1, it has no transitions on input  $\epsilon$ ,

2, each input symbol has at most one transition from any state.

DFA formally defined by 5 tuple notation  $M = (Q, \Sigma, \delta, q_0, F)$ , where

$Q$  is a finite ‘set of states’, which is non empty.

$\Sigma$  is ‘input alphabets’, indicates input set.

$q_0$  is an ‘initial state’ and  $q_0$  is in  $Q$  ie,  $q_0, \Sigma, Q$

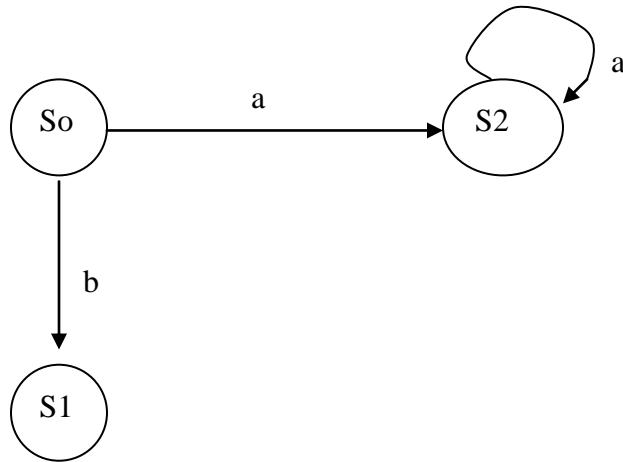
$F$  is a set of ‘Final states’,

$\delta$  is a ‘transmission function’ or mapping function, using this function the next state can be determined.

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

**Regular expression → NFA → DFA → Minimized DFA**

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



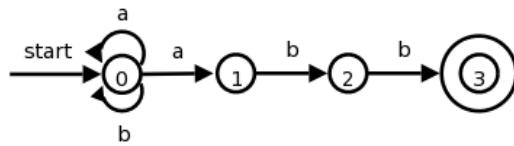
From state  $S_0$  for input ‘a’ there is only one path going to  $S_2$ . similarly from  $S_0$  there is only one path for input going to  $S_1$ .

## 2.13 NONDETERMINISTIC AUTOMATA

■ A NFA is a mathematical model that consists of

- A set of states  $S$ .
- A set of input symbols  $\Sigma$ .
- A transition for move from one state to an other.
- A state  $s_0$  so that is distinguished as the start (or initial) state.
- A set of states  $F$  distinguished as accepting (or final) state.
- A number of transition to a single symbol.

- ⊕ A NFA can be diagrammatically represented by a labeled directed graph, called a transition graph, In which the nodes are the states and the labeled edges represent the transition function.
- ⊕ This graph looks like a transition diagram, but the same character can label two or more transitions out of one state and edges can be labeled by the special symbol  $\epsilon$  as well as by input symbols.
- ⊕ The transition graph for an NFA that recognizes the language  $(a \mid b)^* abb$  is shown



## 2.14 DEFINITION OF CFG

It involves four quantities.

CFG contain terminals, N-T, start symbol and production.

- ⊕ Terminal are basic symbols form which string are formed.
- ⊕ N-terminals are synthetic variables that denote sets of strings
- ⊕ In a Grammar, one N-T are distinguished as the start symbol, and the set of string it denotes is the language defined by the grammar.
- ⊕ The production of the grammar specify the manner in which the terminal and N-T can be combined to form strings.
- ⊕ Each production consists of a N-T, followed by an arrow, followed by a string of one terminal and terminals.

## 2.15 DEFINITION OF SYMBOL TABLE

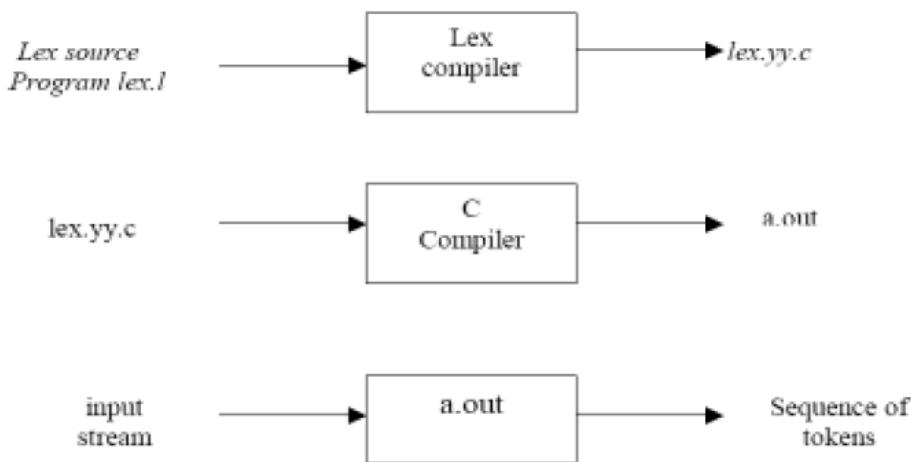
- ⊕ An extensible array of records.
- ⊕ The identifier and the associated records contains collected information about the identifier.

FUNCTION identify (Identifier name)

RETURNING a pointer to identifier information contains

- ⊕ The actual string
- ⊕ A macro definition
- ⊕ A keyword definition
- ⊕ A list of type, variable & function definition
- ⊕ A list of structure and union name definition
- ⊕ A list of structure and union field selected definitions.

## 2.16 Creating a lexical analyzer with Lex



## 2.17 Lex specifications:

A Lex program (the .l file ) consists of three parts:

**declarations**  
`%%`  
**translation rules**  
`%%`  
**auxiliary procedures**

1. The *declarations* section includes declarations of variables,manifest constants(A manifest constant is an identifier that is declared to represent a constant e.g. `# define PIE 3.14`), and regular definitions.

2. The *translation rules* of a Lex program are statements of the form :

<i>p1</i>	{action 1}
<i>p2</i>	{action 2}
<i>p3</i>	{action 3}
...	...
...	...

where each *p* is a regular expression and each *action* is a program fragment describing what action the lexical analyzer should take when a pattern *p* matches a lexeme. In Lex the actions are written in C.

3. The third section holds whatever *auxiliary procedures* are needed by the *actions*.Alternatively these procedures can be compiled separately and loaded with the lexical analyzer.

Note: You can refer to a sample lex program given in page no. 109 of chapter 3 of the book: *Compilers: Principles, Techniques, and Tools* by Aho, Sethi & Ullman for more clarity.

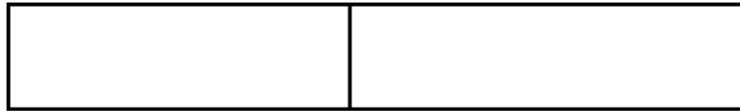
## 2.18 INPUT BUFFERING

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

Buffering techniques:

1. Buffer pairs
2. Sentinels

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



**Token beginnings      look ahead pointer**

Token beginnings look ahead pointer The distance which the lookahead pointer may have to travel past the actual token may be large. For example, in a PL/I program we may see: DECALRE (ARG1, ARG2... ARG n) Without knowing whether DECLARE is a keyword or an array name until we see the character that follows the right parenthesis. In either case, the token itself ends at the second E. If the lookahead pointer travels beyond the buffer half in which it began, the other half must be loaded with the next characters from the source file. Since the buffer shown in above figure is of limited size there is an implied constraint on how much lookahead can be used before the next token is discovered. In the above example, if the lookahead traveled to the left half and all the way through the left half to the middle, we could not reload the right half, because we would lose characters that had not yet been grouped into tokens. While we can make the buffer larger if we chose or use another buffering scheme, we cannot ignore the fact that overhead is limited.

## UNIT -3

### SYNTAX ANALYSIS

#### **3.1 ROLE OF THE PARSER**

Parser obtains a string of tokens from the lexical analyzer and verifies that it can be generated by the language for the source program. The parser should report any syntax errors in an intelligible fashion. The two types of parsers employed are:

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

Therefore there are two types of parsing methods— top-down parsing and bottom-up parsing

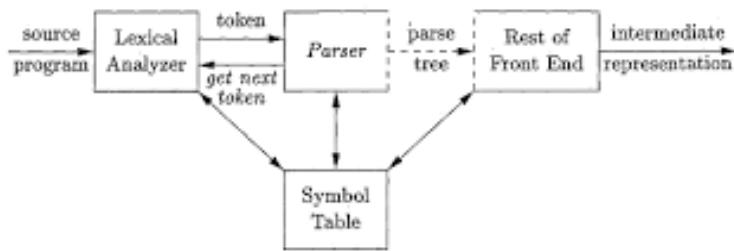


Figure 4.1: Position of parser in compiler model

#### **3.2 TOP-DOWN PARSING**

A program that performs syntax analysis is called a parser. A syntax analyzer takes tokens as input and output error message if the program syntax is wrong. The parser uses symbol-look-ahead and an approach called top-down parsing without backtracking. Top-down parsers check to see if a string can be generated by a grammar by creating a parse tree starting from the initial symbol and working down. Bottom-up parsers, however, check to see if a string can be generated from a grammar by creating a parse tree from the leaves, and working up. Early parser generators such as YACC creates bottom-up parsers whereas many of Java parser generators such as JavaCC create top-down parsers.

#### **3.3 RECURSIVE DESCENT PARSING**

Typically, top-down parsers are implemented as a set of recursive functions that descent through a parse tree for a string. This approach is known as recursive descent parsing, also known as LL(k) parsing where the first L stands for left-to-right, the second L stands for

leftmost-derivation, and  $k$  indicates  $k$ -symbol lookahead. Therefore, a parser using the single symbol look-ahead method and top-down parsing without backtracking is called LL(1) parser. In the following sections, we will also use an extended BNF notation in which some regulation expression operators are to be incorporated.

A syntax expression defines sentences of the form  $, \text{ or } .$  A syntax of the form defines sentences that consist of a sentence of the form followed by a sentence of the form followed by a sentence of the form  $.$  A syntax of the form defines zero or one occurrence of the form  $.$  A syntax of the form defines zero or more occurrences of the form  $.$

A usual implementation of an LL(1) parser is:

- initialize its data structures,
- get the lookahead token by calling scanner routines, and
- call the routine that implements the start symbol.

Here is an example.

```
proc syntaxAnalysis()
begin
    initialize(); // initialize global data and structures
    nextToken(); // get the lookahead token
    program(); // parser routine that implements the start symbol
end;
```

### 3.4 FIRST AND FOLLOW

To compute  $\text{FIRST}(X)$  for all grammar symbols  $X$ , apply the following rules until no more terminals or  $e$  can be added to any  $\text{FIRST}$  set.

1. If  $X$  is terminal, then  $\text{FIRST}(X)$  is  $\{X\}.$
2. If  $X \rightarrow e$  is a production, then add  $e$  to  $\text{FIRST}(X).$
3. If  $X$  is nonterminal 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  $e$  is in all of  $\text{FIRST}(Y_1), \dots, \text{FIRST}(Y_{i-1})$  that is,  
 $Y_1 \dots Y_{i-1} =^* e.$  If  $e$  is in  $\text{FIRST}(Y_j)$  for all  $j=1, 2, \dots, k$ , then add  $e$  to  $\text{FIRST}(X).$  For example, everything in  $\text{FIRST}(Y_j)$  is surely in  $\text{FIRST}(X).$  If  $y_1$  does not derive  $e$ , then we add nothing more to  $\text{FIRST}(X)$ , but if  $Y_1 =^* e$ , then we add  $\text{FIRST}(Y_2)$  and so on.

## Shri Vishnu Engineering College For Women

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

1. Place \$ in FOLLOW(S), where S is the start symbol and \$ in the input right endmarker.
2. If there is a production A=>aBs where FIRST(s) except e is placed in FOLLOW(B).
3. If there is a production A->aB or a production A->aBs where FIRST(s) contains e, then everything in FOLLOW(A) is in FOLLOW(B).

Consider the following example to understand the concept of First and Follow. Find the first and follow of all nonterminals in the Grammar-

E -> TE'

E' -> +TE'|e

T -> FT'

T' -> \*FT'|e

F -> (E)|id

Then:

FIRST(E)=FIRST(T)=FIRST(F)={(),id}

FIRST(E')={+,e}

FIRST(T')={\*,e}

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

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

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

For example, id and left parenthesis are added to FIRST(F) by rule 3 in definition of FIRST with i=1 in each case, since FIRST(id)=(id) and FIRST(')= {{}} by rule 1. Then by rule 3 with i=1, the production T -> FT' implies that id and left parenthesis belong to FIRST(T) also.

To compute FOLLOW, we put \$ in FOLLOW(E) by rule 1 for FOLLOW. By rule 2 applied to production F-> (E), right parenthesis is also in FOLLOW(E). By rule 3 applied to production E-> TE', \$ and right parenthesis are in FOLLOW(E').

### **3.5 CONSTRUCTION OF PREDICTIVE PARSING TABLES**

For any grammar G, the following algorithm can be used to construct the predictive parsing table. The algorithm is

Input : Grammar G

Output : Parsing table M

Method

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

### **3.6.LL(1) GRAMMAR**

The above algorithm can be applied to any grammar G to produce a parsing table M. For some Grammars, for example if G is left recursive or ambiguous, then M will have at least one multiply-defined entry. A grammar whose parsing table has no multiply defined entries is said to be LL(1). It can be shown that the above algorithm can be used to produce for every LL(1) grammar G a parsing table M that parses all and only the sentences of G. LL(1) grammars have several distinctive properties. No ambiguous or left recursive grammar can be LL(1). There remains a question of what should be done in case of multiply defined entries. One easy solution is to eliminate all left recursion and left factoring, hoping to produce a grammar which will produce no multiply defined entries in the parse tables. Unfortunately there are some grammars which will give an LL(1) grammar after any kind of alteration. In general, there are no universal rules to convert multiply defined entries into single valued entries without affecting the language recognized by the parser.

The main difficulty in using predictive parsing is in writing a grammar for the source language such that a predictive parser can be constructed from the grammar. Although left recursion elimination and left factoring are easy to do, they make the resulting grammar hard to read and difficult to use for translation purposes. To alleviate some of this difficulty, a common organization for a parser in a compiler is to use a predictive parser for control

constructs and to use operator precedence for expressions. However, if an LR parser generator is available, one can get all the benefits of predictive parsing and operator precedence automatically.

### **3.7. ERROR RECOVERY IN PREDICTIVE PARSING**

The stack of a nonrecursive predictive parser makes explicit the terminals and nonterminals that the parser hopes to match with the remainder of the input. We shall therefore refer to symbols on the parser stack in the following discussion. An error is detected during predictive parsing when the terminal on top of the stack does not match the next input symbol or when nonterminal A is on top of the stack, a is the next input symbol, and the parsing table entry  $M[A,a]$  is empty.

Panic-mode error recovery is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens appears. Its effectiveness depends on the choice of synchronizing set. The sets should be chosen so that the parser recovers quickly from errors that are likely to occur in practice. Some heuristics are as follows

- As a starting point, we can place all symbols in  $\text{FOLLOW}(A)$  into the synchronizing set for nonterminal A. If we skip tokens until an element of  $\text{FOLLOW}(A)$  is seen and pop A from the stack, it is likely that parsing can continue.
- It is not enough to use  $\text{FOLLOW}(A)$  as the synchronizing set for A. For example, if semicolons terminate statements, as in C, then keywords that begin statements may not appear in the FOLLOW set of the nonterminal generating expressions. A missing semicolon after an assignment may therefore result in the keyword beginning the next statement being skipped. Often, there is a hierarchical structure on constructs in a language; e.g., expressions appear within statements, which appear within blocks, and so on. We can add to the synchronizing set of a lower construct the symbols that begin higher constructs. For example, we might add keywords that begin statements to the synchronizing sets for the nonterminals generating expressions.
- If we add symbols in  $\text{FIRST}(A)$  to the synchronizing set for nonterminal A, then it may be possible to resume parsing according to A if a symbol in  $\text{FIRST}(A)$  appears in the input.

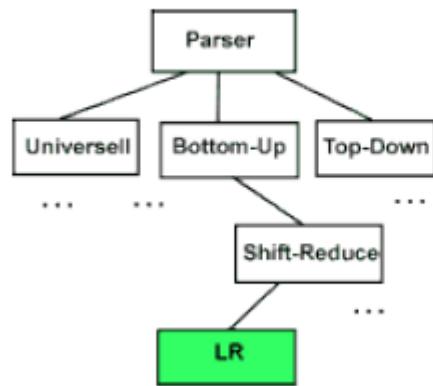
- If a nonterminal can generate the empty string, then the production deriving e can be used as a default. Doing so may postpone some error detection, but cannot cause an error to be missed. This approach reduces the number of nonterminals that have to be considered during error recovery.
- If a terminal on top of the stack cannot be matched, a simple idea is to pop the terminal, issue a message saying that the terminal was inserted, and continue parsing. In effect, this approach takes the synchronizing set of a token to consist of all other tokens.

## UNIT 4

### LR PARSER

#### **4.1 LR PARSING INTRODUCTION**

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



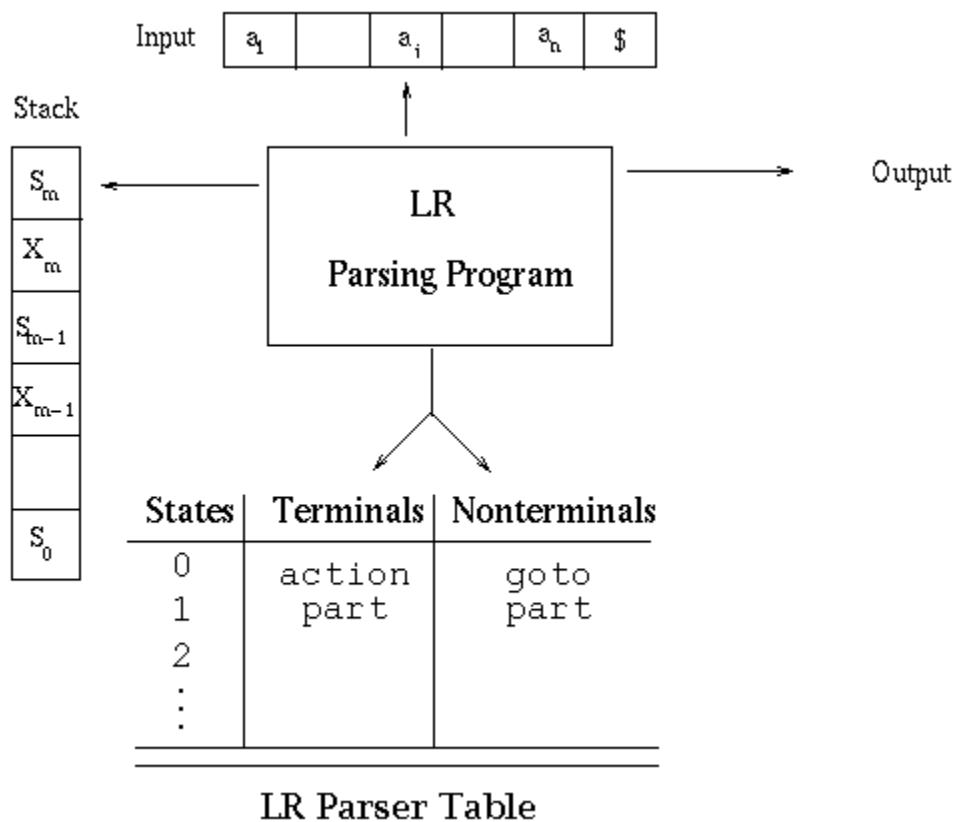
#### **4.2 WHY LR PARSING:**

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

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

### 4.3.MODELS OF LR PARSERS

The schematic form of an LR parser is shown below.



The program uses a stack to store a string of the form  $s_0X_1s_1X_2\dots X_ms_m$  where  $s_m$  is on top. Each  $X_i$  is a grammar symbol and each  $s_i$  is a symbol representing a state. Each state symbol summarizes the information contained in the stack below it. The combination of the state symbol on top of the stack and the current input symbol are used to index the parsing table and determine the shift/reduce parsing decision. The parsing table consists of two parts: a parsing action function *action* and a goto function *goto*. The program driving the LR parser behaves as follows: It determines  $s_m$  the state currently on top of the stack and  $a_i$  the current input symbol. It then consults  $\text{action}[s_m, a_i]$ , which can have one of four values:

- shift  $s$ , where  $s$  is a state
- reduce by a grammar production  $A \rightarrow b$
- accept
- error

The function goto takes a state and grammar symbol as arguments and produces a state.

For a parsing table constructed for a grammar G, the goto table is the transition function of a deterministic finite automaton that recognizes the viable prefixes of G. Recall that the viable prefixes of G are those prefixes of right-sentential forms that can appear on the stack of a shift-reduce parser because they do not extend past the rightmost handle.

A configuration of an LR parser is a pair whose first component is the stack contents and whose second component is the unexpanded input:

$(s_0 \ X_1 \ s_1 \ X_2 \ s_2 \dots \ X_m \ s_m, a_i \ a_{i+1} \dots \ a_n \$)$

This configuration represents the right-sentential form

$X_1 \ X_1 \dots \ X_m \ a_i \ a_{i+1} \dots \ a_n$

in essentially the same way a shift-reduce parser would; only the presence of the states on the stack is new. Recall the sample parse we did (see Example 1: Sample bottom-up parse) in which we assembled the right-sentential form by concatenating the remainder of the input buffer to the top of the stack. The next move of the parser is determined by reading  $a_i$  and  $s_m$ , and consulting the parsing action table entry  $\text{action}[s_m, a_i]$ . Note that we are just looking at the state here and no symbol below it. We'll see how this actually works later.

The configurations resulting after each of the four types of move are as follows:

If  $\text{action}[s_m, a_i] = \text{shift } s$ , the parser executes a shift move entering the configuration

$(s_0 \ X_1 \ s_1 \ X_2 \ s_2 \dots \ X_m \ s_m \ a_i \ s, a_{i+1} \dots \ a_n \$)$

Here the parser has shifted both the current input symbol  $a_i$  and the next symbol.

If  $\text{action}[s_m, a_i] = \text{reduce } A \rightarrow b$ , then the parser executes a reduce move, entering the configuration,

$(s_0 \ X_1 \ s_1 \ X_2 \ s_2 \dots \ X_m \ s_m \ r \ A \ s, a_{i+1} \dots \ a_n \$)$

where  $s = \text{goto}[s_m, A]$  and  $r$  is the length of  $b$ , the right side of the production. The parser first popped  $2r$  symbols off the stack ( $r$  state symbols and  $r$  grammar symbols), exposing state  $s_m - r$ . The parser then pushed both  $A$ , the left side of the production, and  $s$ , the entry for  $\text{goto}[s_m - r, A]$ , onto the stack. The current input symbol is not changed in a reduce move.

The output of an LR parser is generated after a reduce move by executing the semantic action associated with the reducing production. For example, we might just print out the production reduced.

If  $\text{action}[s_m, a_i] = \text{accept}$ , parsing is completed.

#### 4.4.OPERATOR PRECEDENCE PARSING

##### Precedence Relations

Bottom-up parsers for a large class of context-free grammars can be easily developed using operator grammars. Operator grammars have the property that no production right side is empty or has two adjacent nonterminals. This property enables the implementation of efficient operator-precedence parsers. These parser rely on the following three precedence relations:

##### Relation Meaning

$a < \cdot b$  a yields precedence to b

$a = \cdot b$  a has the same precedence as b

$a \cdot > b$  a takes precedence over b

These operator precedence relations allow to delimit the handles in the right sentential forms:  $<\cdot$  marks the left end,  $=\cdot$  appears in the interior of the handle, and  $\cdot >$  marks the right end.

	<b>id</b>	<b>+</b>	<b>*</b>	<b>\$</b>
<b>id</b>		<b>&gt;</b>	<b>&gt;</b>	<b>&gt;</b>
<b>+</b>	<b>&lt;</b>	<b>&gt;</b>	<b>&lt;</b>	<b>&gt;</b>
<b>*</b>	<b>&lt;</b>	<b>&gt;</b>	<b>&gt;</b>	<b>&gt;</b>
<b>\$</b>	<b>&lt;</b>	<b>&lt;</b>	<b>&lt;</b>	<b>&gt;</b>

Example: The input string:

id1 + id2 \* id3

after inserting precedence relations becomes

$\$ < \cdot id1 \cdot > + < \cdot id2 \cdot > * < \cdot id3 \cdot > \$$

Having precedence relations allows to identify handles as follows:

- scan the string from left until seeing  $\cdot >$
- scan backwards the string from right to left until seeing  $<\cdot$
- everything between the two relations  $<\cdot$  and  $\cdot >$  forms the handle

#### **4.5 OPERATOR PRECEDENCE PARSING ALGORITHM**

*Initialize: 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 \$ is on the top of the stack and ip points to \$ then return*

*else*

*Let a be the top terminal on the stack, and b the symbol pointed to*

*by ip*

*if  $a < \cdot b$  or  $a = \cdot b$  then*

*push b onto the stack*

*advance ip to the next input symbol*

*else if  $a \cdot > b$  then*

*repeat*

*pop the stack*

*until the top stack terminal is related by  $<\cdot$*

*to the terminal most recently popped*

*else error()*

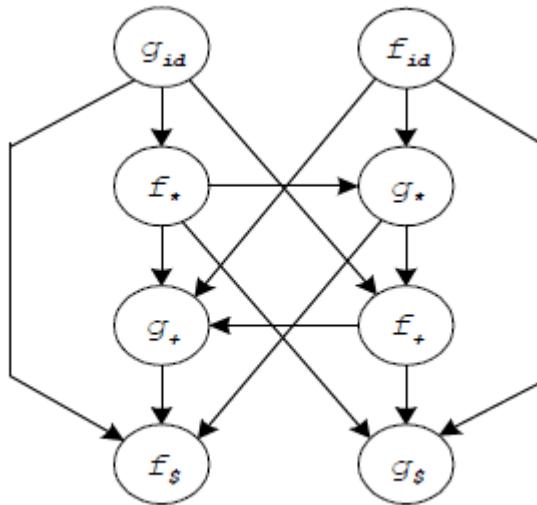
*end*

#### **4.6 ALGORITHM FOR CONSTRUCTING PRECEDENCE FUNCTIONS**

1. Create functions fa for each grammar terminal a and for the end of string symbol;
2. Partition the symbols in groups so that fa and gb are in the same group if  $a = \cdot b$  ( there can be symbols in the same group even if they are not connected by this relation)
3. Create a directed graph whose nodes are in the groups, next for each symbols a and b do: place an edge from the group of gb to the group of fa if  $a < \cdot b$ , otherwise if  $a \cdot > b$  place an edge from the group of fa to that of gb;
4. If the constructed graph has a cycle then no precedence functions exist. When there are no cycles collect the length of the longest paths from the groups of fa and gb Example:

	<b>id</b>	<b>+</b>	<b>*</b>	<b>\$</b>
<b>id</b>	>	>	>	>
<b>+</b>	<	>	<	>
<b>*</b>	<	>	>	>
<b>\$</b>	<	<	<	>

Consider the above table Using the algorithm leads to the following graph:



#### 4.7 SHIFT REDUCE PARSING

A shift-reduce parser uses a parse stack which (conceptually) contains grammar symbols.

During the operation of the parser, symbols from the input are shifted onto the stack. If a prefix of the symbols on top of the stack matches the RHS of a grammar rule which is the correct rule to use within the current context, then the parser reduces the RHS of the rule to its LHS, replacing the RHS symbols on top of the stack with the nonterminal occurring on the LHS of the rule. This shift-reduce process continues until the parser terminates, reporting either success or failure. It terminates with success when the input is legal and is accepted by the parser. It terminates with failure if an error is detected in the input. The parser is nothing but a stack automaton which may be in one of several discrete states. A state is usually represented simply as an integer. In reality, the parse stack contains states, rather than

grammar symbols. However, since each state corresponds to a unique grammar symbol, the state stack can be mapped onto the grammar symbol stack mentioned earlier.

The operation of the parser is controlled by a couple of tables:

#### **4.8 ACTION TABLE**

The action table is a table with rows indexed by states and columns indexed by terminal symbols. When the parser is in some state  $s$  and the current lookahead terminal is  $t$ , the action taken by the parser depends on the contents of  $\text{action}[s][t]$ , which can contain four different kinds of entries:

*Shift  $s'$*

*Shift state  $s'$  onto the parse stack.*

*Reduce  $r$*

*Reduce by rule  $r$ . This is explained in more detail below.*

*Accept*

*Terminate the parse with success, accepting the input.*

*Error*

Signal a parse error

#### **4.9 GOTO TABLE**

The goto table is a table with rows indexed by states and columns indexed by nonterminal symbols. When the parser is in state  $s$  immediately after reducing by rule  $N$ , then the next state to enter is given by  $\text{goto}[s][N]$ .

The current state of a shift-reduce parser is the state on top of the state stack. The detailed operation of such a parser is as follows:

1. Initialize the parse stack to contain a single state  $s_0$ , where  $s_0$  is the distinguished initial state of the parser.
2. Use the state  $s$  on top of the parse stack and the current lookahead  $t$  to consult the action table entry  $\text{action}[s][t]$ :
  - If the action table entry is *shift  $s'$*  then push state  $s'$  onto the stack and advance the input so that the lookahead is set to the next token.
  - If the action table entry is *reduce  $r$*  and rule  $r$  has  $m$  symbols in its RHS, then pop  $m$  symbols off the parse stack. Let  $s'$  be the state now revealed on top of the parse stack and  $N$  be the LHS nonterminal for rule  $r$ . Then consult the goto table and

push the state given by  $\text{goto}[s'][N]$  onto the stack. The lookahead token is not changed by this step.

- If the action table entry is accept, then terminate the parse with success.
- If the action table entry is error, then signal an error.

3. Repeat step (2) until the parser terminates.

For example, consider the following simple grammar

0)  $\$S: \text{stmt} <\text{EOF}>$

1)  $\text{stmt}: \text{ID} ':=' \text{expr}$

2)  $\text{expr}: \text{expr} '+' \text{ID}$

3)  $\text{expr}: \text{expr} '-' \text{ID}$

4)  $\text{expr}: \text{ID}$

which describes assignment statements like  $a := b + c - d$ . (Rule 0 is a special augmenting production added to the grammar).

One possible set of shift-reduce parsing tables is shown below (sn denotes shift n, rn denotes reduce n, acc denotes accept and blank entries denote error entries):

#### Parser Tables

**Parser Tables**

<b>Action Table</b>					<b>Goto Table</b>	
<b>ID</b>	<b>'':'</b>	<b>'+'</b>	<b>'-'</b>	<b>&lt;EOF&gt;</b>	<b>stmt</b>	<b>expr</b>
<b>0</b>	s1					g2
<b>1</b>		s3				
<b>2</b>					s4	
<b>3</b>	s5					g6
<b>4</b>	acc	acc	acc	acc	acc	
<b>5</b>	r4	r4	r4	r4	r4	
<b>6</b>	r1	r1	s7	s8	r1	
<b>7</b>	s9					
<b>8</b>	s10					
<b>9</b>	r2	r2	r2	r2	r2	
<b>10</b>	r3	r3	r3	r3	r3	

A trace of the parser on the input  $a := b + c - d$  is shown below:

Stack	Remaining Input	Action
0/\$\$ a := b + c - d		s1
0/\$\$ 1/a := b + c - d		s3
0/\$\$ 1/a 3/ := b + c - d		s5
0/\$\$ 1/a 3/ := 5/b + c - d		r4
0/\$\$ 1/a 3/ := + c - d		g6 ON expr
0/\$\$ 1/a 3/ := 6/expr + c - d		s7
0/\$\$ 1/a 3/ := 6/expr 7/+ c - d		s9
0/\$\$ 1/a 3/ := 6/expr 7/+ 9/c - d		r2
0/\$\$ 1/a 3/ := - d		g6 ON expr
0/\$\$ 1/a 3/ := 6/expr - d		s8
0/\$\$ 1/a 3/ := 6/expr 8/- d		s10
0/\$\$ 1/a 3/ := 6/expr 8/- 10/d<EOF>		r3
0/\$\$ 1/a 3/ := <EOF>		g6 ON expr
0/\$\$ 1/a 3/ := 6/expr <EOF>		r1
0/\$\$ <EOF>		g2 ON stmt
0/\$\$ 2/stmt <EOF>		s4
0/\$\$ 2/stmt 4/<EOF>		accept

Each stack entry is shown as a state number followed by the symbol which caused the transition to that state.

#### 4.10 SLR PARSER

An  $LR(0)$  item (or just *item*) of a grammar  $G$  is a production of  $G$  with a dot at some position of the right side indicating how much of a production we have seen up to a given point.

For example, for the production  $E \rightarrow E + T$  we would have the following items:

[ $E \rightarrow .E + T$ ]

[ $E \rightarrow E. + T$ ]

[ $E \rightarrow E. + T$ ]

[ $E \rightarrow E + T.$ ]

Stack	State	Comments
Empty	[ $E \rightarrow .E$ ]	can't go anywhere from here
	e-transition	so we follow an e-transition
Empty	[ $F \rightarrow .(E)$ ]	now we can shift the (
(	[ $F \rightarrow (.E)$ ]	building the handle (E); This state says: "I have ( on the stack and expect the input to give me tokens that can eventually be reduced to give me the rest of the handle, E.)"

#### **4.11 CONSTRUCTING THE SLR PARSING TABLE**

To construct the parser table we must convert our NFA into a DFA. The states in the LR table will be the e-closures of the states corresponding to the items SO...the process of creating the LR state table parallels the process of constructing an equivalent DFA from a machine with e-transitions. Been there, done that - this is essentially the subset construction algorithm so we are in familiar territory here.

We need two operations: closure()

and goto().

closure()

If I is a set of items for a grammar  $G$ , then closure(I) is the set of items constructed from I by the two rules: Initially every item in I is added to closure(I)

If  $A \rightarrow a.Bb$  is in closure(I), and  $B \rightarrow g$  is a production, then add the initial item  $[B \rightarrow .g]$  to I, if it is not already there. Apply this rule until no more new items can be added to closure(I).

From our grammar above, if I is the set of one item  $\{[E' \rightarrow .E]\}$ , then closure(I) contains:

I0:  $E' \rightarrow .E$

$E \rightarrow .E + T$

$E \rightarrow .T$

$T \rightarrow .T * F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

goto()

goto( $I, X$ ), where I is a set of items and X is a grammar symbol, is defined to be the closure of the set of all items  $[A \rightarrow aX.b]$  such that  $[A \rightarrow a.Xb]$  is in I. The idea here is fairly intuitive: if I is the set of items that are valid for some viable prefix g, then goto( $I, X$ ) is the set of items that are valid for the viable prefix  $gX$ .

#### **4.12 SETS-OF-ITEMS-CONSTRUCTION**

To construct the canonical collection of sets of LR(0) items for

*augmented grammar  $G'$* .

*procedure items( $G'$ )*

*begin*

```

 $C := \{closure(\{[S' \rightarrow .S]\})\};$ 
repeat
for each set of items in C and each grammar symbol X
such that  $goto(I, X)$  is not empty and not in C do
add  $goto(I, X)$  to C;
until no more sets of items can be added to C
end;

```

#### **4.13 ALGORITHM FOR CONSTRUCTING AN SLR PARSING TABLE**

**Input:** augmented grammar  $G'$

**Output:** SLR parsing table functions action and goto for  $G'$

**Method:**

*Construct  $C = \{I_0, I_1, \dots, I_n\}$  the collection of sets of LR(0) items for  $G'$ .*

*State  $i$  is constructed from  $I_i$ :*

*if  $[A \rightarrow a.ab]$  is in  $I_i$  and  $goto(I_i, a) = I_j$ , then set  $action[i, a]$  to "shift j". Here  $a$  must be a terminal.*

*if  $[A \rightarrow a.]$  is in  $I_i$ , then set  $action[i, a]$  to "reduce  $A \rightarrow a$ " for all  $a$  in  $FOLLOW(A)$ . Here  $A$  may*

*not be  $S'$ .*

*if  $[S' \rightarrow S.]$  is in  $I_i$ , then set  $action[i, \$]$  to "accept"*

If any conflicting actions are generated by these rules, the grammar is not SLR(1) and the algorithm fails to produce a parser. The goto transitions for state  $i$  are constructed for all nonterminals  $A$  using the rule: If  $goto(I_i, A) = I_j$ , then  $goto[i, A] = j$ .

All entries not defined by rules 2 and 3 are made "error".

The initial state of the parser is the one constructed from the set of items containing  $[S' \rightarrow .S]$ .

Let's work an example to get a feel for what is going on,

An Example

- (1)  $E \rightarrow E * B$
- (2)  $E \rightarrow E + B$
- (3)  $E \rightarrow B$
- (4)  $B \rightarrow 0$
- (5)  $B \rightarrow 1$

The Action and Goto Table The two LR(0) parsing tables for this grammar look as follows:

	action				goto		
state	*	+	0	1	\$	E	B
0			s1	s2		3	4
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	s5	s6			acc		
4	r3	r3	r3	r3	r3		
5			s1	s2		7	
6			s1	s2			8
7	r1	r1	r1	r1	r1		
8	r2	r2	r2	r2	r2		

**UNIT -5****5.1 CANONICAL LR PARSING**

By splitting states when necessary, we can arrange to have each state of an LR parser indicate exactly which input symbols can follow a handle  $a$  for which there is a possible reduction to  $A$ . As the text points out, sometimes the FOLLOW sets give too much information and doesn't (can't) discriminate between different reductions.

The general form of an LR( $k$ ) item becomes  $[A \rightarrow a.b, s]$  where  $A \rightarrow ab$  is a production and  $s$  is a string of terminals. The first part ( $A \rightarrow a.b$ ) is called the core and the second part is the lookahead. In LR(1)  $|s|$  is 1, so  $s$  is a single terminal.

$A \rightarrow ab$  is the usual righthand side with a marker; any  $a$  in  $s$  is an incoming token in which we are interested. Completed items used to be reduced for every incoming token in  $\text{FOLLOW}(A)$ , but now we will reduce only if the next input token is in the lookahead set  $s$ . If we get two productions  $A \rightarrow a$  and  $B \rightarrow a$ , we can tell them apart when  $a$  is a handle on the stack if the corresponding completed items have different lookahead parts. Furthermore, note that the lookahead has no effect for an item of the form  $[A \rightarrow a.b, a]$  if  $b$  is not  $e$ . Recall that our problem occurs for completed items, so what we have done now is to say that an item of the form  $[A \rightarrow a., a]$  calls for a reduction by  $A \rightarrow a$  only if the next input symbol is  $a$ . More formally, an LR(1) item  $[A \rightarrow a.b, a]$  is valid for a viable prefix  $g$  if there is a derivation  $S \Rightarrow^* g abw$ , where  $g = sa$ , and either  $a$  is the first symbol of  $w$ , or  $w$  is  $e$  and  $a$  is  $\$$ .

**5.2 ALGORITHM FOR CONSTRUCTION OF THE SETS OF LR(1) ITEMS**

*Input:* grammar  $G'$

*Output:* sets of LR(1) items that are the set of items valid for one or more viable prefixes of  $G'$

*Method:*

*closure(I)*

*begin*

*repeat*

*for each item*  $[A \rightarrow a.Bb, a]$  *in I,*

*each production*  $B \rightarrow g$  *in G',*

*and each terminal*  $b$  *in FIRST(ba)*

*such that  $[B \rightarrow .g, b]$  is not in  $I$  do  
 add  $[B \rightarrow .g, b]$  to  $I$ ;  
 until no more items can be added to  $I$ ;  
 end;*

**5.3 goto(I, X)**

*begin  
 let  $J$  be the set of items  $[A \rightarrow aX.b, a]$  such that  
 $[A \rightarrow a.Xb, a]$  is in  $I$   
 return closure( $J$ );  
 end;  
 procedure items( $G'$ )  
 begin  
 $C := \{closure(\{S' \rightarrow .S, \$\})\};$   
 repeat  
 for each set of items  $I$  in  $C$  and each grammar symbol  $X$  such  
 that  $goto(I, X)$  is not empty and not in  $C$  do  
 add  $goto(I, X)$  to  $C$   
 until no more sets of items can be added to  $C$ ;  
 end;*

An example,

Consider the following grammar,

$S' \rightarrow S$

$S \rightarrow CC$

$C \rightarrow cC$

$C \rightarrow d$

Sets of LR(1) items

**I0:**  $S' \rightarrow .S, \$$

$S \rightarrow .CC, \$$

$C \rightarrow .Cc, c/d$

$C \rightarrow .d, c/d$

**I1:**  $S' \rightarrow S., \$$

**I2:**  $S \rightarrow C.C, \$$

$C \rightarrow .Cc, \$$

$C \rightarrow .d, \$$

Shri Vishnu Engineering College For Women

I3: C->c.C,c/d

C->.Cc,c/d

C->.d,c/d

I4: C->d.,c/d

I5: S->CC.,\$

I6: C->c.C,\$

C->.cC,\$

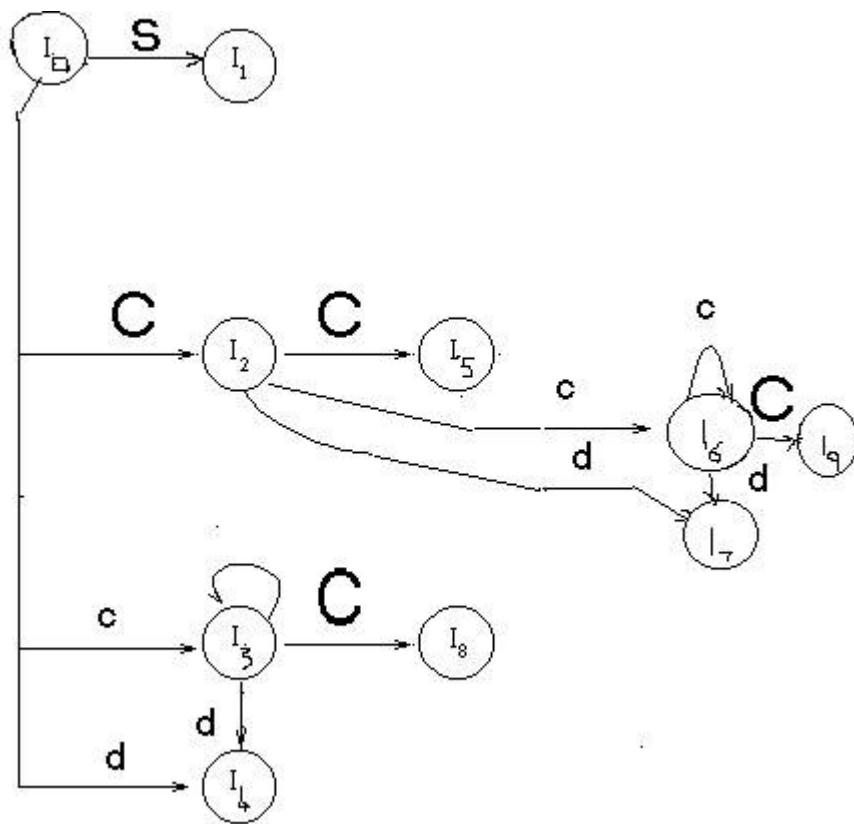
C->.d,\$

I7: C->d.,\$

I8: C->cC.,c/d

I9: C->cC.,\$

Here is what the corresponding DFA looks like



Parsing Table:state	c	d	\$	S	C
0	S3 S4			1 2	
1			acc		
2	S6 S7			5	
3	S3 S4			8	
4	R3 R3				
5			R1		
6	S6 S7			9	
7			R3		
8	R2 R2				
9			R2		

#### 5.4 ALGORITHM FOR CONSTRUCTION OF THE CANONICAL LR PARSING TABLE

**Input:** grammar  $G'$

**Output:** canonical LR parsing table functions action and goto

1. Construct  $C = \{I_0, I_1, \dots, I_n\}$  the collection of sets of LR(1) items for  $G'$ . State  $i$  is constructed from  $I_i$ .
2. if  $[A \rightarrow a.ab, b \gtreqless]$  is in  $I_i$  and  $\text{goto}(I_i, a) = I_j$ , then set  $\text{action}[i, a]$  to "shift  $j$ ". Here  $a$  must be a terminal.
3. if  $[A \rightarrow a., a]$  is in  $I_i$ , then set  $\text{action}[i, a]$  to "reduce  $A \rightarrow a$ " for all  $a$  in  $\text{FOLLOW}(A)$ . Here  $A$  may *not* be  $S$ .
4. if  $[S' \rightarrow S.]$  is in  $I_i$ , then set  $\text{action}[i, \$]$  to "accept"
5. If any conflicting actions are generated by these rules, the grammar is not LR(1) and the algorithm fails to produce a parser.
6. The goto transitions for state  $i$  are constructed for all *nonterminals*  $A$  using the rule: If  $\text{goto}(I_i, A) = I_j$ , then  $\text{goto}[i, A] = j$ .
7. All entries not defined by rules 2 and 3 are made "error".
8. The initial state of the parser is the one constructed from the set of items containing  $[S' \rightarrow .S, \$]$ .

### **5.5.LALR PARSER:**

We begin with two observations. First, some of the states generated for LR(1) parsing have the same set of core (or first) components and differ only in their second component, the lookahead symbol. Our intuition is that we should be able to merge these states and reduce the number of states we have, getting close to the number of states that would be generated for LR(0) parsing. This observation suggests a hybrid approach: We can construct the canonical LR(1) sets of items and then look for sets of items having the same core. We merge these sets with common cores into one set of items. The merging of states with common cores can never produce a shift/reduce conflict that was not present in one of the original states because shift actions depend only on the core, not the lookahead. But it is possible for the merger to produce a reduce/reduce conflict.

Our second observation is that we are really only interested in the lookahead symbol in places where there is a problem. So our next thought is to take the LR(0) set of items and add lookahead only where they are needed. This leads to a more efficient, but much more complicated method.

### **5.6 ALGORITHM FOR EASY CONSTRUCTION OF AN LALR TABLE**

Input:  $G'$

Output: LALR parsing table functions with action and goto for  $G'$ .

Method:

1. Construct  $C = \{I_0, I_1, \dots, I_n\}$  the collection of sets of LR(1) items for  $G'$ .
2. For each core present among the set of LR(1) items, find all sets having that core and replace these sets by the union.
3. Let  $C' = \{J_0, J_1, \dots, J_m\}$  be the resulting sets of LR(1) items. The parsing actions for state  $i$  are constructed from  $J_i$  in the same manner as in the construction of the canonical LR parsing table.
4. If there is a conflict, the grammar is not LALR(1) and the algorithm fails.
5. The goto table is constructed as follows: If  $J$  is the union of one or more sets of LR(1) items, that is,  $J = I_0 \cup I_1 \cup \dots \cup I_k$ , then the cores of  $\text{goto}(I_0, X)$ ,  $\text{goto}(I_1, X)$ , ...,  $\text{goto}(I_k, X)$  are the same, since  $I_0, I_1, \dots, I_k$  all have the same core. Let  $K$  be the union of all sets of items having the same core as  $\text{goto}(I_1, X)$ .

6. Then goto(J, X) = K.

Consider the above example,

I3 & I6 can be replaced by their union

I36:C->c.C,c/d/\$

C->.Cc,C/D/\$

C->.d,c/d/\$

I47:C->d.,c/d/\$

I89:C->Cc.,c/d/\$

Parsing Table

state	c	d	\$	S	C
0	S36	S47		1	2
1			Accept		
2	S36	S47			5
36	S36	S47			89
47	R3	R3			
5			R1		
89	R2	R2	R2		

## 5.7 HANDLING ERRORS

The LALR parser may continue to do reductions after the LR parser would have spotted an error, but the LALR parser will never do a shift after the point the LR parser would have discovered the error and will eventually find the error.

## 5.8 DANGLING ELSE

The dangling else is a problem in [computer programming](#) in which an optional else clause in an If–then(–else) statement results in nested conditionals being ambiguous. Formally, the [context-free grammar](#) of the language is [ambiguous](#), meaning there is more than one correct parse tree.

In many [programming languages](#) one may write conditionally executed code in two forms: the if-then form, and the if-then-else form – the else clause is optional:

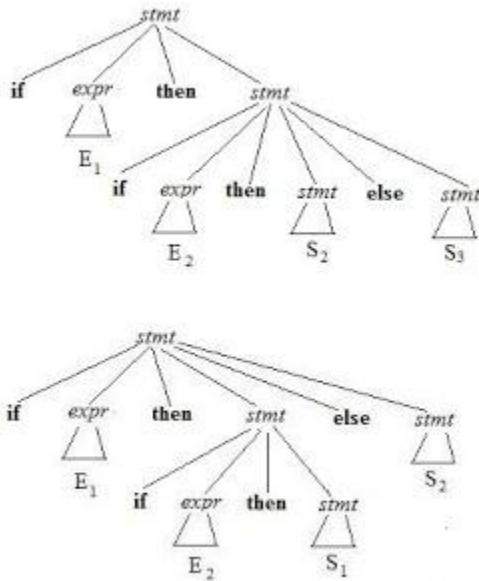


Fig 2.4 Two parse trees for an ambiguous sentence

Consider the grammar:

$S ::= E \$$

$E ::= E + E$

|  $E * E$

|  $( E )$

| id

| num

and four of its LALR(1) states:

I0:  $S ::= . E \$ ?$

$E ::= . E + E + * \$$

I1:  $S ::= E . \$ ?$

I2:  $E ::= E * . E + * \$$

$E ::= . E * E + * \$$

$E ::= E . + E + * \$$

$E ::= . E + E + * \$$

$E ::= . ( E ) + * \$$

$E ::= E . * E + * \$$

$E ::= . E * E + * \$$

$E ::= . id + * \$$

$E ::= . ( E ) + * \$$

$E ::= . num + * \$$

I3:  $E ::= E * E . + * \$$

$E ::= . id + * \$$

$E ::= E . + E + * \$$

$E ::= . num + * \$$

$$E ::= E \cdot * E \quad + * \$$$

Here we have a shift-reduce error. Consider the first two items in I3. If we have  $a^*b+c$  and we parsed  $a^*b$ , do we reduce using  $E ::= E \cdot * E$  or do we shift more symbols? In the former case we get a parse tree  $(a^*b)+c$ ; in the latter case we get  $a^*(b+c)$ . To resolve this conflict, we can specify that  $*$  has higher precedence than  $+$ . The precedence of a grammar production is equal to the precedence of the rightmost token at the rhs of the production. For example, the precedence of the production  $E ::= E \cdot * E$  is equal to the precedence of the operator  $*$ , the precedence of the production  $E ::= ( E )$  is equal to the precedence of the token  $)$ , and the precedence of the production  $E ::= \text{if } E \text{ then } E \text{ else } E$  is equal to the precedence of the token  $\text{else}$ . The idea is that if the look ahead has higher precedence than the production currently used, we shift. For example, if we are parsing  $E + E$  using the production rule  $E ::= E + E$  and the look ahead is  $*$ , we shift  $*$ . If the look ahead has the same precedence as that of the current production and is left associative, we reduce, otherwise we shift. The above grammar is valid if we define the precedence and associativity of all the operators. Thus, it is very important when you write a parser using CUP or any other LALR(1) parser generator to specify associativities and precedence's for most tokens (especially for those used as operators). Note: you can explicitly define the precedence of a rule in CUP using the `%prec` directive:

$$E ::= \text{MINUS } E \quad \%prec \text{ UMINUS}$$

where `UMINUS` is a pseudo-token that has higher precedence than `TIMES`, `MINUS` etc, so that  $-1^2$  is equal to  $(-1)^2$ , not to  $-(1^2)$ .

Another thing we can do when specifying an LALR(1) grammar for a parser generator is error recovery. All the entries in the ACTION and GOTO tables that have no content correspond to syntax errors. The simplest thing to do in case of error is to report it and stop the parsing. But we would like to continue parsing finding more errors. This is called *error recovery*. Consider the grammar:

$$S ::= L = E ;$$

$$| \{ SL \} ;$$

$$| \text{error} ;$$

$$SL ::= S ;$$

$$| SL S ;$$

The special token error indicates to the parser what to do in case of invalid syntax for S (an invalid statement). In this case, it reads all the tokens from the input stream until it finds the first semicolon. The way the parser handles this is to first push an error state in the stack. In case of an error, the parser pops out elements from the stack until it finds an error state where it can proceed. Then it discards tokens from the input until a restart is possible. Inserting error handling productions in the proper places in a grammar to do good error recovery is considered very hard.

## **5.9 LR ERROR RECOVERY**

An LR parser will detect an error when it consults the parsing action table and find a blank or error entry. Errors are never detected by consulting the goto table. An LR parser will detect an error as soon as there is no valid continuation for the portion of the input thus far scanned. A canonical LR parser will not make even a single reduction before announcing the error. SLR and LALR parsers may make several reductions before detecting an error, but they will never shift an erroneous input symbol onto the stack.

## **5.10 PANIC-MODE ERROR RECOVERY**

We can implement panic-mode error recovery by scanning down the stack until a state s with a goto on a particular nonterminal A is found. Zero or more input symbols are then discarded until a symbol a is found that can legitimately follow A. The parser then stacks the state GOTO(s, A) and resumes normal parsing. The situation might exist where there is more than one choice for the nonterminal A. Normally these would be nonterminals representing major program pieces, e.g. an expression, a statement, or a block. For example, if A is the nonterminal stmt, a might be semicolon or }, which marks the end of a statement sequence. This method of error recovery attempts to eliminate the phrase containing the syntactic error. The parser determines that a string derivable from A contains an error. Part of that string has already been processed, and the result of this processing is a sequence of states on top of the stack. The remainder of the string is still in the input, and the parser attempts to skip over the remainder of this string by looking for a symbol on the input that can legitimately follow A. By removing states from the stack, skipping over the input, and pushing GOTO(s, A) on the stack, the parser pretends that it has found an instance of A and resumes normal parsing.

### 5.11 PHRASE-LEVEL RECOVERY

Phrase-level recovery is implemented by examining each error entry in the LR action table and deciding on the basis of language usage the most likely programmer error that would give rise to that error. An appropriate recovery procedure can then be constructed; presumably the top of the stack and/or first input symbol would be modified in a way deemed appropriate for each error entry. In designing specific error-handling routines for an LR parser, we can fill in each blank entry in the action field with a pointer to an error routine that will take the appropriate action selected by the compiler designer.

The actions may include insertion or deletion of symbols from the stack or the input or both, or alteration and transposition of input symbols. We must make our choices so that the LR parser will not get into an infinite loop. A safe strategy will assure that at least one input symbol will be removed or shifted eventually, or that the stack will eventually shrink if the end of the input has been reached. Popping a stack state that covers a non terminal should be avoided, because this modification eliminates from the stack a construct that has already been successfully parsed.

## **UNIT 6**

### **SEMANTIC ANALYSIS**

#### **6.1 SEMANTIC ANALYSIS**

- Semantic Analysis computes additional information related to the meaning of the program once the syntactic structure is known.
- In typed languages as C, semantic analysis involves adding information to the symbol table and performing type checking.
- The information to be computed is beyond the capabilities of standard parsing techniques, therefore it is not regarded as syntax.
- As for Lexical and Syntax analysis, also for Semantic Analysis we need both a Representation Formalism and an Implementation Mechanism.
- As representation formalism this lecture illustrates what are called Syntax Directed Translations.

#### **6.2 SYNTAX DIRECTED TRANSLATION**

- The Principle of Syntax Directed Translation states that the meaning of an input sentence is related to its syntactic structure, i.e., to its Parse-Tree.
- By Syntax Directed Translations we indicate those formalisms for specifying translations for programming language constructs guided by context-free grammars.
  - We associate Attributes to the grammar symbols representing the language constructs.
  - Values for attributes are computed by Semantic Rules associated with grammar productions.
- Evaluation of Semantic Rules may:
  - Generate Code;
  - Insert information into the Symbol Table;
  - Perform Semantic Check;
  - Issue error messages;
  - etc.

There are two notations for attaching semantic rules:

1. **Syntax Directed Definitions.** High-level specification hiding many implementation details (also called **Attribute Grammars**).
2. **Translation Schemes.** More implementation oriented: Indicate the order in which semantic rules are to be evaluated.

### **Syntax Directed Definitions**

- **Syntax Directed Definitions** are a generalization of context-free grammars in which:
  1. Grammar symbols have an associated set of **Attributes**;
  2. Productions are associated with **Semantic Rules** for computing the values of attributes.
    - Such formalism generates **Annotated Parse-Trees** where each node of the tree is a record with a field for each attribute (e.g.,  $X.a$  indicates the attribute  $a$  of the grammar symbol  $X$ ).
    - The value of an attribute of a grammar symbol at a given parse-tree node is defined by a semantic rule associated with the production used at that node.

We distinguish between two kinds of attributes:

1. **Synthesized Attributes.** They are computed from the values of the attributes of the children nodes.
2. **Inherited Attributes.** They are computed from the values of the attributes of both the siblings and the parent nodes

### **Syntax Directed Definitions: An Example**

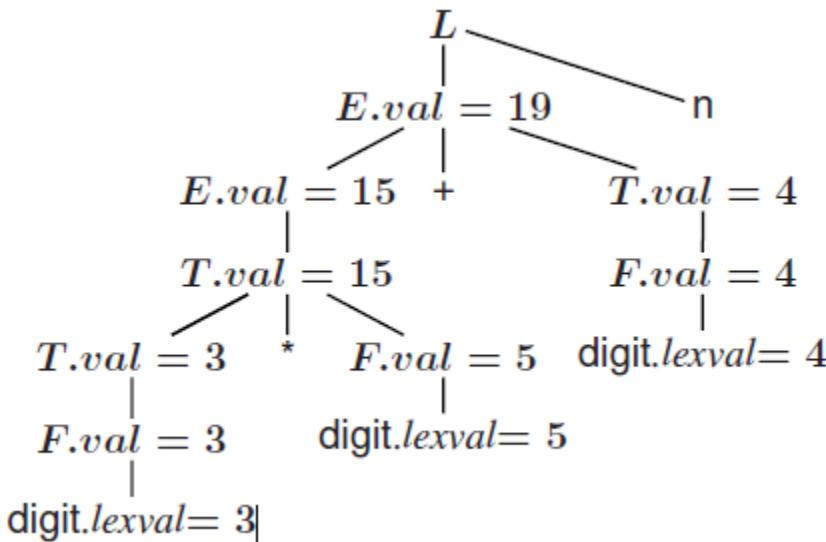
- **Example.** Let us consider the Grammar for arithmetic expressions. The Syntax Directed Definition associates to each non terminal a synthesized attribute called *val*.

PRODUCTION	SEMANTIC RULE
$L \rightarrow E\text{n}$	$print(E.\text{val})$
$E \rightarrow E_1 + T$	$E.\text{val} := E_1.\text{val} + T.\text{val}$
$E \rightarrow T$	$E.\text{val} := T.\text{val}$
$T \rightarrow T_1 * F$	$T.\text{val} := T_1.\text{val} * F.\text{val}$
$T \rightarrow F$	$T.\text{val} := F.\text{val}$
$F \rightarrow (E)$	$F.\text{val} := E.\text{val}$
$F \rightarrow \text{digit}$	$F.\text{val} := \text{digit}.lexval$

### 6.3 S-ATTRIBUTED DEFINITIONS

**Definition.** An **S-Attributed Definition** is a Syntax Directed Definition that uses only synthesized attributes.

- **Evaluation Order.** Semantic rules in a S-Attributed Definition can be evaluated by a bottom-up, or PostOrder, traversal of the parse-tree.
- **Example.** The above arithmetic grammar is an example of an S-Attributed Definition. The annotated parse-tree for the input  $3*5+4\text{n}$  is:



## 6.4 L-attributed definition

**Definition:** A SDD its *L-attributed* if each inherited attribute of  $X_i$  in the RHS of  $A ! X_1 : \dots : X_n$  depends only on

1. attributes of  $X_1; X_2; \dots; X_{i-1}$  (symbols to the left of  $X_i$  in the RHS)
2. inherited attributes of  $A$ .

### Restrictions for translation schemes:

1. Inherited attribute of  $X_i$  must be computed by an action before  $X_i$ .
2. An action must not refer to synthesized attribute of any symbol to the right of that action.
3. Synthesized attribute for  $A$  can only be computed after all attributes it references have been completed (usually at end of RHS).

## 6.5 SYMBOL TABLES

A symbol table is a major data structure used in a compiler. Associates attributes with identifiers used in a program. For instance, a type attribute is usually associated with each identifier. A symbol table is a necessary component Definition (declaration) of identifiers appears once in a program. Use of identifiers may appear in many places of the program text. Identifiers and attributes are entered by the analysis phases. When processing a definition (declaration) of an identifier. In simple languages with only global variables and implicit declarations. The scanner can enter an identifier into a symbol table if it is not already there.

In block-structured languages with scopes and explicit declarations:

- The parser and/or semantic analyzer enter identifiers and corresponding attributes
- Symbol table information is used by the analysis and synthesis phases
- To verify that used identifiers have been defined (declared)
- To verify that expressions and assignments are semantically correct – type checking
- To generate intermediate or target code

### ✓ Symbol Table Interface

The basic operations defined on a symbol table include:

- allocate – to allocate a new empty symbol table
- free – to remove all entries and free the storage of a symbol table
- insert – to insert a name in a symbol table and return a pointer to its entry

- lookup – to search for a name and return a pointer to its entry
- set\_attribute – to associate an attribute with a given entry
- get\_attribute – to get an attribute associated with a given entry

Other operations can be added depending on requirement For example, a delete operation removes a name previously inserted Some identifiers become invisible (out of scope) after exiting a block

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

#### Basic Implementation Techniques

- First consideration is how to insert and lookup names
- Variety of implementation techniques
- Unordered List
- Simplest to implement
- Implemented as an array or a linked list
- Linked list can grow dynamically – alleviates problem of a fixed size array
- Insertion is fast  $O(1)$ , but lookup is slow for large tables –  $O(n)$  on average
- Ordered List
- If an array is sorted, it can be searched using binary search –  $O(\log_2 n)$
- Insertion into a sorted array is expensive –  $O(n)$  on average
- Useful when set of names is known in advance – table of reserved words
- Binary Search Tree
- Can grow dynamically
- Insertion and lookup are  $O(\log_2 n)$  on average

## 6.6 HASH TABLES AND HASH FUNCTIONS

- ✓ A hash table is an array with index range: 0 to  $TableSize - 1$
- ✓ Most commonly used data structure to implement symbol tables
- ✓ Insertion and lookup can be made very fast –  $O(1)$
- ✓ A hash function maps an identifier name into a table index

- ✓ A hash function,  $h(name)$ , should depend solely on  $name$
- ✓  $h(name)$  should be computed quickly
- ✓  $h$  should be uniform and randomizing in distributing names
- ✓ All table indices should be mapped with equal probability.
- ✓ Similar names should not cluster to the same table index

## 6.7 HASH FUNCTIONS

- \_ Hash functions can be defined in many ways . . .
- \_ A string can be treated as a sequence of integer words
- \_ Several characters are fit into an integer word
- \_ Strings longer than one word are folded using exclusive-or or addition
- \_ Hash value is obtained by taking integer word modulo  $TableSize$
- \_ We can also compute a hash value character by character:
- \_  $h(name) = (c_0 + c_1 + \dots + c_{n-1}) \bmod TableSize$ , where  $n$  is  $name$  length
- \_  $h(name) = (c_0 * c_1 * \dots * c_{n-1}) \bmod TableSize$
- \_  $h(name) = (c_{n-1} + \dots + c_2 + c_1 + c_0) \bmod TableSize$
- \_  $h(name) = (c_0 * c_1 * \dots * c_{n-1}) \bmod TableSize$

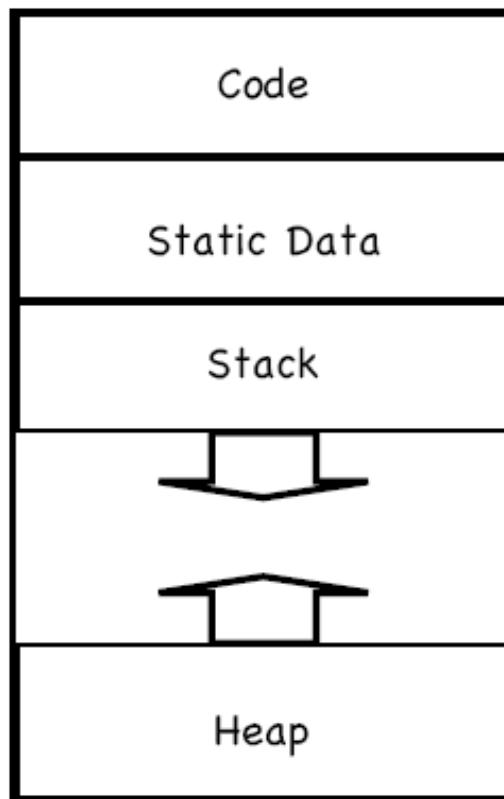
## 6.8 RUNTIME ENVIRONMENT

- Runtime organization of different storage locations
- Representation of scopes and extents during program execution.
- Components of executing program reside in blocks of memory (supplied by OS).
- Three kinds of entities that need to be managed at runtime:
  - Generated code for various procedures and programs.
- forms text or code segment of your program: size known at compile time.
  - Data objects:
- Global variables/constants: size known at compile time
- Variables declared within procedures/blocks: size known
- Variables created dynamically: size unknown.
  - Stack to keep track of procedure activations.
- Subdivide memory conceptually into code and data areas:

- Code: Program
- instructions
  - Stack: Manage activation of procedures at runtime.
  - Heap: holds variables created dynamically

## 6.9 STORAGE ORGANIZATION

1 *Fixed-size objects can be placed in predefined locations.*



## 2. Run-time stack and heap

The STACK is used to store:

- Procedure activations.
- The status of the machine just before calling a procedure, so that the status can be restored when the called procedure returns.
- The HEAP stores data allocated under program control (e.g. by malloc() in C).

### Activation records

Any information needed for a single activation of a procedure is stored in the ACTIVATION RECORD (sometimes called the STACK FRAME). Today, we'll assume the stack grows DOWNWARD, as on, e.g., the Intel architecture. The activation record gets pushed for each procedure call and popped for each procedure return.

## 6.9 STATIC ALLOCATION

Statically allocated names are bound to storage at compile time. Storage bindings of statically allocated names never change, so even if a name is local to a procedure, its name is always bound to the same storage. The compiler uses the type of a name (retrieved from the symbol table) to determine storage size required. The required number of bytes (possibly aligned) is set aside for the name. The address of the storage is fixed at compile time.

Limitations:

- The size required must be known at compile time.
- Recursive procedures cannot be implemented as all locals are statically allocated.
- No data structure can be created dynamically as all data is static.

### ❖ Stack-dynamic allocation

- ✓ Storage is organized as a stack.
- ✓ Activation records are pushed and popped.
- ✓ Locals and parameters are contained in the activation records for the call.
- ✓ This means locals are bound to fresh storage on every call.
- ✓ If we have a stack growing downwards, we just need a stack\_top pointer.
- ✓ To allocate a new activation record, we just increase stack\_top.
- ✓ To deallocate an existing activation record, we just decrease stack\_top.

### ❖ Address generation in stack allocation

The position of the activation record on the stack cannot be determined statically. Therefore the compiler must generate addresses RELATIVE to the activation record. If we have a downward-growing stack and a stack\_top pointer, we generate addresses of the form stack\_top + offset

## 6.10 HEAP ALLOCATION

Some languages do not have tree-structured allocations. In these cases, activations have to be allocated on the heap. This allows strange situations, like callee activations that live longer than their callers' activations. This is not common. Heap is used for allocating space for objects created at run time. For example: nodes of dynamic data structures such as linked lists and trees.

- Dynamic memory allocation and deallocation based on the requirements of the program `malloc()` and `free()` in C programs

*new()* and *delete()* in C++ programs

*new()* and garbage collection in Java programs

- Allocation and deallocation may be *completely manual* (C/C++), *semi-automatic* (Java), or *fully automatic* (Lisp)

## 6.11 PARAMETERS PASSING

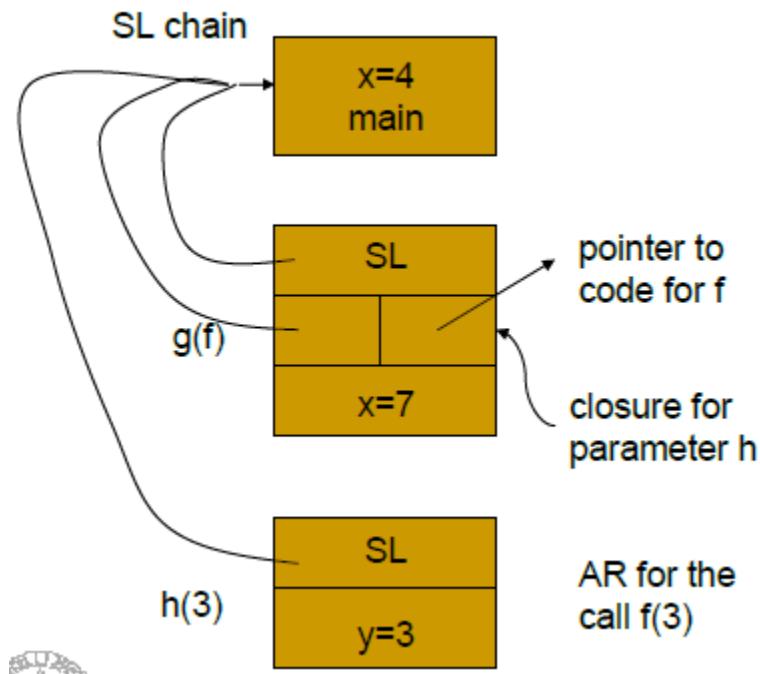
A language has first-class functions if functions can be declared within any scope passed as arguments to other functions returned as results of functions. □ In a language with first-class functions and static scope, a function value is generally represented by a closure, a pair consisting of a pointer to function code and a pointer to an activation record. □ Passing functions as arguments is very useful in structuring of systems using upcalls.

An example:

```
main()
{
    int x = 4;
    int f (int y) {
        return x*y;
    }
    int g (int →int h){
        int x = 7;
        return h(3) + x;
    }
}
```

```
g(f); // returns 12
}
```

### Passing Functions as Parameters – Implementation with Static Scope



An example:

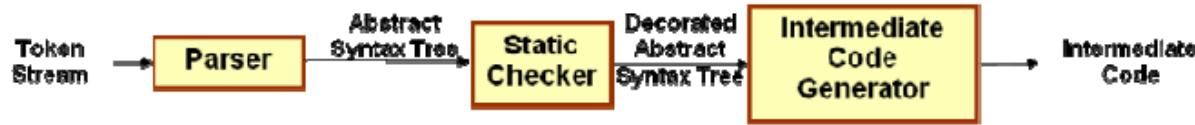
```
main()
{ int x = 4;
  int f (int y) {
    return x*y;
  }
  int g (int → int h){
    int x = 7;
    return h(3) + x;
  }
  g(f); // returns 12
}
```

## **UNIT 7** **INTERMEDIATE CODE**

### **7.1. INTERMEDIATE CODE GENERATION**

In the analysis-synthesis model of a compiler, the front end analyzes a source program and creates an intermediate representation, from which the back end generates target code. This facilitates *retargeting*: enables attaching a back end for the new machine to an existing front end.

#### **Logical Structure of a Compiler Front End**



A compiler front end is organized as in figure above, where parsing, static checking, and intermediate-code generation are done sequentially; sometimes they can be combined and folded into parsing. All schemes can be implemented by creating a syntax tree and then walking the tree.

#### **Static Checking**

This includes type checking which ensures that operators are applied to compatible operands. It also includes any syntactic checks that remain after parsing like

- flow-of-control checks
  - Ex: Break statement within a loop construct
- Uniqueness checks
  - Labels in case statements
- Name-related checks

#### **Intermediate Representations**

We could translate the source program directly into the target language. However, there are benefits to having an intermediate, machine-independent representation.

- A clear distinction between the machine-independent and machine-dependent parts of the compiler
- Retargeting is facilitated the implementation of language processors for new machines will require replacing only the back-end.
- We could apply machine independent code optimization techniques

Intermediate representations span the gap between the source and target languages.

- ***High Level Representations***

- closer to the source language
- easy to generate from an input program
- code optimizations may not be straightforward

- ***Low Level Representations***

- closer to the target machine
- Suitable for register allocation and instruction selection
- easier for optimizations, final code generation

There are several options for intermediate code. They can be either

- Specific to the language being implemented

P-code for Pascal

Byte code for Java

## 7.2 LANGUAGE INDEPENDENT 3-ADDRESS CODE

IR can be either an actual language or a group of internal data structures that are shared by the phases of the compiler. C used as intermediate language as it is flexible, compiles into efficient machine code and its compilers are widely available. In all cases, the intermediate code is a linearization of the syntax tree produced during syntax and semantic analysis. It is formed by breaking down the tree structure into sequential instructions, each of which is equivalent to a single, or small number of machine instructions. Machine code can then be generated (access might be required to symbol tables etc). TAC can range from high- to low-level, depending on the choice of operators. In general, it is a statement containing at most 3 addresses or operands.

The general form is  $x := y \text{ op } z$ , where “op” is an operator, x is the result, and y and z are operands. **x, y, z** are variables, constants, or “temporaries”. A three-address instruction

consists of at most 3 addresses for each statement.

It is a linearized representation of a binary syntax tree. Explicit names correspond to interior nodes of the graph. E.g. for a looping statement, syntax tree represents components of the statement, whereas three-address code contains labels and jump instructions to represent the flow-of-control as in machine language. A TAC instruction has at most one operator on the RHS of an instruction; no built-up arithmetic expressions are permitted.

e.g.  $x + y * z$  can be translated as

$$t1 = y * z$$

$$t2 = x + t1$$

Where  $t1$  &  $t2$  are compiler-generated temporary names.

Since it unravels multi-operator arithmetic expressions and nested control-flow statements, it is useful for target code generation and optimization.

### Addresses and Instructions

- TAC consists of a sequence of instructions, each instruction may have up to three addresses, prototypically  $t1 = t2 \text{ op } t3$
- Addresses may be one of:
  - A name. Each name is a symbol table index. For convenience, we write the names as the identifier.
  - A constant.
  - A compiler-generated temporary. Each time a temporary address is needed, the compiler generates another name from the stream  $t1, t2, t3$ , etc.
- Temporary names allow for code optimization to easily move Instructions
- At target-code generation time, these names will be allocated to registers or to memory.
- TAC Instructions
  - Symbolic labels will be used by instructions that alter the flow of control.

The instruction addresses of labels will be filled in later.

$$L: t1 = t2 \text{ op } t3$$

- Assignment instructions:  $x = y \text{ op } z$

- Includes binary arithmetic and logical operations

- Unary assignments:  $x = \text{op } y$

- Includes unary arithmetic op (-) and logical op (!) and type conversion

- Copy instructions:  $x = y$
- Unconditional jump: goto L

- L is a symbolic label of an instruction

- Conditional jumps:

if x goto L If x is true, execute instruction L next

ifFalse x goto L If x is false, execute instruction L next

- Conditional jumps:

if x relop y goto L

– Procedure calls. For a procedure call  $p(x_1, \dots, x_n)$

param  $x_1$

...

param  $x_n$

call p, n

– Function calls :  $y = p(x_1, \dots, x_n)$   $y = \text{call } p, n$ , return y

– Indexed copy instructions:  $x = y[i]$  and  $x[i] = y$

- Left: sets x to the value in the location i memory units beyond y
- Right: sets the contents of the location i memory units beyond x to y

– Address and pointer instructions:

- $x = \&y$  sets the value of x to be the location (address) of y.
- $x = *y$ , presumably y is a pointer or temporary whose value is a location. The value of x is set to the contents of that location.
- $*x = y$  sets the value of the object pointed to by x to the value of y.

Example: Given the statement **do**  $i = i+1;$  **while** ( $a[i] < v$ ); , the TAC can be written as below in two ways, using either symbolic labels or position number of instructions for labels.

### **Types of three address code**

There are different types of statements in source program to which three address code has to be generated. Along with operands and operators, three address code also use labels to provide flow of control for statements like if-then-else, for and while. The different types of three address code statements are:

#### **Assignment statement**

$a = b \text{ op } c$

In the above case b and c are operands, while op is binary or logical operator. The result of applying op on b and c is stored in a.

#### **Unary operation**

$a = \text{op } b$  This is used for unary minus or logical negation.

Example:  $a = b * (-c) + d$

Three address code for the above example will be

$t1 = -c$

$t2 = t1 * b$

$t3 = t2 + d$

$a = t3$

#### **Copy Statement**

$a = b$

The value of b is stored in variable a.

#### **Unconditional jump**

`goto L`

Creates label L and generates three-address code ‘`goto L`’

v. Creates label L, generate code for expression exp, If the exp returns value true then go to the statement labelled L. exp returns a value false go to the statement immediately following the if statement.

#### **Function call**

For a function fun with n arguments a1,a2,a3....an ie.,

`fun(a1, a2, a3,...an),`

the three address code will be

Param a1

Param a2

...

Param an

Call fun, n

Where param defines the arguments to function.

### **Array indexing**

In order to access the elements of array either single dimension or multidimension, three address code requires base address and offset value. Base address consists of the address of first element in an array. Other elements of the array can be accessed using the base address and offset value.

Example:  $x = y[i]$

Memory location m = Base address of y + Displacement i

$x$  = contents of memory location m

similarly  $x[i] = y$

Memory location m = Base address of x + Displacement i

The value of y is stored in memory location m

### **Pointer assignment**

$x = \&y$  x stores the address of memory location y

$x = *y$  y is a pointer whose r-value is location

$*x = y$  sets r-value of the object pointed by x to the r-value of y

Intermediate representation should have an operator set which is rich to implement most of the

operations of source language. It should also help in mapping to restricted instruction set of target machine.

### **Data Structure**

Three address code is represented as record structure with fields for operator and operands.

These

records can be stored as array or linked list. Most common implementations of three address code are-

Quadruples, Triples and Indirect triples.

### **7.3 QUADRUPLES-**

Quadruples consists of four fields in the record structure. One field to store operator op, two fields to store operands or arguments arg1 and arg2 and one field to store result res.  $\text{res} = \text{arg1 op arg2}$

Example:  $a = b + c$

b is represented as arg1, c is represented as arg2, + as op and a as res.

Unary operators like '-' do not use arg2. Operators like param do not use arg2 nor result. For conditional and unconditional statements res is label. Arg1, arg2 and res are pointers to symbol table or literal table for the names.

Example:  $a = -b * d + c + (-b) * d$

Three address code for the above statement is as follows

$t1 = -b$

$t2 = t1 * d$

$t3 = t2 + c$

$t4 = -b$

$t5 = t4 * d$

$t6 = t3 + t5$

$a = t6$

Quadruples for the above example is as follows

Op	Arg1	Arg2	Res
-	B		t1
*	t1	d	t2
+	t2	c	t3
-	B		t4
*	t4	d	t5
+	t3	t5	t6
=	t6		a

## 7.4 TRIPLES

Triples uses only three fields in the record structure. One field for operator, two fields for operands named as arg1 and arg2. Value of temporary variable can be accessed by the position of the statement the computes it and not by location as in quadruples.

Example:  $a = -b * d + c + (-b) * d$

Triples for the above example is as follows

Stmt no	Op	Arg1	Arg2
(0)	-	b	
(1)	*	d	(0)
(2)	+	c	(1)
(3)	-	b	
(4)	*	d	(3)
(5)	+	(2)	(4)
(6)	=	a	(5)

Arg1 and arg2 may be pointers to symbol table for program variables or literal table for constant or pointers into triple structure for intermediate results.

Example: Triples for statement  $x[i] = y$  which generates two records is as follows

Stmt no	Op	Arg1	Arg2
(0)	[]=	x	i
(1)	=	(0)	y

Triples for statement  $x = y[1]$  which generates two records is as follows

Stmt no	Op	Arg1	Arg2
(0)	=[]	y	i
(1)	=	x	(0)

Triples are alternative ways for representing syntax tree or Directed acyclic graph for program defined names.

### Indirect Triples

Indirect triples are used to achieve indirection in listing of pointers. That is, it uses pointers to triples than listing of triples themselves.

Example:  $a = -b * d + c + (-b) * d$

	Stmt no	Stmt no	Op	Arg1	Arg2
(0)	(10)	(10)	-	b	
(1)	(11)	(11)	*	d	(0)
(2)	(12)	(12)	+	c	(1)
(3)	(13)	(13)	-	b	
(4)	(14)	(14)	*	d	(3)
(5)	(15)	(15)	+	(2)	(4)
(6)	(16)	(16)	=	a	(5)

Conditional operator and operands. Representations include quadruples, triples and indirect triples.

## 7.5 SYNTAX TREES

Syntax trees are high level IR. They depict the natural hierarchical structure of the source program. Nodes represent constructs in source program and the children of a node represent meaningful components of the construct. Syntax trees are suited for static type checking.

Variants of Syntax Trees: DAG

A directed acyclic graph (DAG) for an expression identifies the common sub expressions (sub expressions that occur more than once) of the expression. DAG's can be constructed by using the same techniques that construct syntax trees.

A DAG has leaves corresponding to atomic operands and interior nodes corresponding to operators. A node N in a DAG has more than one parent if N represents a common sub expression, so a DAG represents expressions concisely. It gives clues to compiler about the generating efficient code to evaluate expressions.

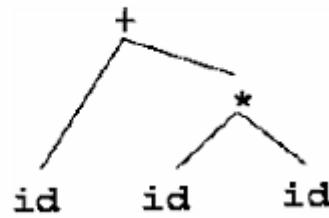
Example 1: Given the grammar below, for the input string id + id \* id , the parse tree,

syntax tree and the DAG are as shown.

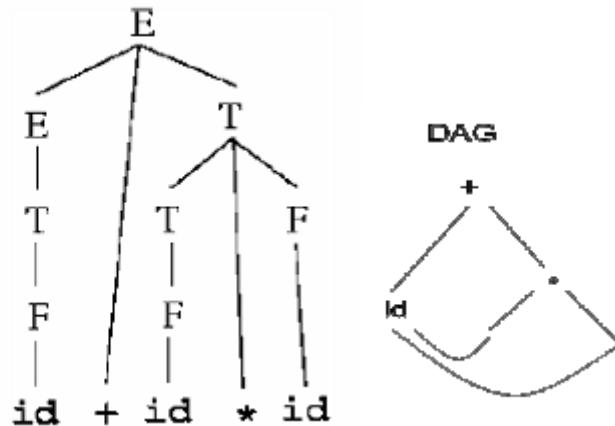
Syntax tree:

Grammar :

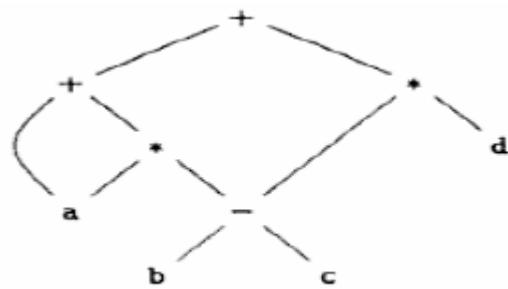
$$\begin{array}{l} E \rightarrow E + T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow ( E ) \mid id \end{array}$$



Parse tree:



Example : DAG for the expression  $a + a * (b - c) + (b - c) * d$  is shown below.



Using the SDD to draw syntax tree or DAG for a given expression:-

- Draw the parse tree
- Perform a post order traversal of the parse tree
- Perform the semantic actions at every node during the traversal
  - Constructs a DAG if before creating a new node, these functions check whether an identical node already exists. If yes, the existing node is returned.

SDD to produce Syntax trees or DAG is shown below.

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

For the expression  $a + a * ( b - c ) + (b - c) * d$ , steps for constructing the DAG is as below.

- 1)  $p_1 = \text{Leaf}(\text{id}, \text{entry}-a)$
- 2)  $p_2 = \text{Leaf}(\text{id}, \text{entry}-a) = p_1$
- 3)  $p_3 = \text{Leaf}(\text{id}, \text{entry}-b)$
- 4)  $p_4 = \text{Leaf}(\text{id}, \text{entry}-c)$
- 5)  $p_5 = \text{Node}( ' - ', p_3, p_4 )$
- 6)  $p_6 = \text{Node}( '*' , p_1, p_5 )$
- 7)  $p_7 = \text{Node}( '+', p_1, p_6 )$
- 8)  $p_8 = \text{Leaf}(\text{id}, \text{entry}-b) = p_3$
- 9)  $p_9 = \text{Leaf}(\text{id}, \text{entry}-c) = p_4$
- 10)  $p_{10} = \text{Node}( ' - ', p_3, p_4 ) = p_5$
- 11)  $p_{11} = \text{Leaf}(\text{id}, \text{entry}-d)$
- 12)  $p_{12} = \text{Node}( '*' , p_5, p_{11} )$
- 13)  $p_{13} = \text{Node}( '+', p_7, p_{12} )$

## 7.6 BASIC BLOCKS AND FLOW GRAPHS

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. Flow graph of a program can be used as a vehicle to collect information about the intermediate program. Some register-assignment algorithms use flow graphs to find the inner loops where a program is expected to spend most of its time.

### 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 halt or possibility of branching except at the end. The following sequence of three-address statements forms a basic block:

```
t1 := a*a
t2 := a*b
t3 := 2*t2
t4 := t1+t3
t5 := b*b
t6 := t4+t5
```

A three-address statement  $x := y+z$  is said to define  $x$  and to use  $y$  or  $z$ . A name in a basic block is said to live at a given point if its value is used after that point in the program, perhaps in another basic block.

The following algorithm can be used to partition a sequence of three-address statements into basic blocks.

**Algorithm 1: 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 the following:

- I) The first statement is a leader.
- II) Any statement that is the target of a conditional or unconditional goto is a leader.

III) 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.

Example 3: Consider the fragment of source code shown in fig. 7; it computes the dot product of two vectors a and b of length 20. A list of three-address statements performing this computation on our target machine is shown in fig. 8.

```

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

```

Let us apply Algorithm 1 to the three-address code in fig 8 to determine its basic blocks. Statement (1) is a leader by rule (I) and statement (3) is a leader by rule (II), since the last statement can jump to it. By rule (III) the statement following (12) is a leader. Therefore, statements (1) and (2) form a basic block. The remainder of the program beginning with statement (3) forms a second basic block.

- (1) prod := 0
- (2) i := 1
- (3) t1 := 4\*i
- (4) t2 := a [ t1 ]
- (5) t3 := 4\*i
- (6) t4 := b [ t3 ]
- (7) t5 := t2\*t4
- (8) t6 := prod + t5
- (9) prod := t6
- (10) t7 := i+1

- (11)  $i := t7$
- (12) if  $i \leq 20$  goto (3)

## 7.7 TRANSFORMATIONS ON BASIC BLOCKS

A basic block computes a set of expressions. These expressions are the values of the names live on exit from block. Two basic blocks are said to be equivalent if they compute the same set of expressions. A number of transformations can be applied to a basic block without changing the set of expressions computed by the block. Many of these transformations are useful for improving the quality of code that will be ultimately generated from a basic block. There are two important classes of local transformations that can be applied to basic blocks; these are the structure-preserving transformations and the algebraic transformations.

## 7.8 STRUCTURE-PRESERVING TRANSFORMATIONS

The primary structure-preserving transformations on basic blocks are:

1. Common sub-expression elimination
2. Dead-code elimination
3. Renaming of temporary variables
4. Interchange of two independent adjacent statements

We assume basic blocks have no arrays, pointers, or procedure calls.

### 1. Common sub-expression elimination

Consider the basic block

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

The second and fourth statements compute the same expression, namely  $b+c-d$ , and hence this basic block may be transformed into the equivalent block

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

Although the 1st and 3rd statements in both cases appear to have the same expression

on the right, the second statement redefines b. Therefore, the value of b in the 3rd statement is different from the value of b in the 1st, and the 1st and 3rd statements do not compute the same expression.

## 2. 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.

## 3. Renaming temporary variables

Suppose we have a statement  $t := b + c$ , where t is a temporary. If we change this statement to  $u := b + c$ , where u is a new temporary variable, and change all uses of this instance of t to u, then the value of the basic block is not changed.

## 4. Interchange of statements

Suppose we have a block with the two adjacent statements

$$\begin{aligned} t1 &:= b + c \\ t2 &:= x + y \end{aligned}$$

Then 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. A normal-form basic block permits all statement interchanges that are possible.

## 7.9 DAG REPRESENTATION OF BASIC BLOCKS

The goal is to obtain a visual picture of how information flows through the block. The leaves will show the values entering the block and as we proceed up the DAG we encounter uses of these values def (and redef) of values and uses of the new values.

Formally, this is defined as follows.

1. Create a leaf for the initial value of each variable appearing in the block. (We do not know what that the value is, not even if the variable has ever been given a value).
2. Create a node N for each statement s in the block.
  - i. Label N with the operator of s. This label is drawn inside the node.
  - ii. Attach to N those variables for which N is the last def in the block. These additional labels are drawn along side of N.
  - iii. Draw edges from N to each statement that is the last def of an operand used by N.

2. Designate as output nodes those N whose values are live on exit, an officially-mysterious term meaning values possibly used in another block. (Determining the live on exit values requires global, i.e., inter-block, flow analysis.) As we shall see in the next few sections various basic-block optimizations are facilitated by using the DAG.

#### Finding Local Common Subexpressions

As we create nodes for each statement, proceeding in the static order of the statements, we might notice that a new node is just like one already in the DAG in which case we don't need a new node and can use the old node to compute the new value in addition to the one it already was computing. Specifically, we do not construct a new node if an existing node has the same children in the same order and is labeled with the same operation.

Consider computing the DAG for the following block of code.

$$a = b + c$$

$$c = a + x$$

$$d = b + c$$

$$b = a + x$$

The DAG construction is explained as follows (the movie on the right accompanies the explanation).

1. First we construct leaves with the initial values.
2. Next we process  $a = b + c$ . This produces a node labeled  $+$  with  $b$  attached and having  $c_0$  and  $c_1$  as children.
3. Next we process  $c = a + x$ .
4. Next we process  $d = b + c$ . Although we have already computed  $b + c$  in the first statement, the  $c$ 's are not the same, so we produce a new node.
5. Then we process  $b = a + x$ . Since we have already computed  $a + x$  in statement 2, we do not produce a new node, but instead attach  $b$  to the old node.
6. Finally, we tidy up and erase the unused initial values.

You might think that with only three computation nodes in the DAG, the block could be reduced to three statements (dropping the computation of  $b$ ). However, this is wrong. Only if  $b$  is dead on exit can we omit the computation of  $b$ . We can, however, replace the last statement with the simpler  $b = c$ . Sometimes a combination of techniques finds

improvements that no single technique would find. For example if  $a-b$  is computed, then both  $a$  and  $b$  are incremented by one, and then  $a-b$  is computed again, it will not be recognized as a common subexpression even though the value has not changed. However, when combined with various algebraic transformations, the common value can be recognized.

### **7.10 DEAD CODE ELIMINATION**

Assume we are told (by global flow analysis) that certain values are dead on exit. We examine each root (node with no ancestor) and delete any that have no live variables attached. This process is repeated since new roots may have appeared.

For example, if we are told, for the picture on the right, that only  $a$  and  $b$  are live, then the root  $d$  can be removed since  $d$  is dead. Then the rightmost node becomes a root, which also can be removed (since  $c$  is dead).

#### The Use of Algebraic Identities

Some of these are quite clear. We can of course replace  $x+0$  or  $0+x$  by simply  $x$ . Similar Considerations apply to  $1*x$ ,  $x*1$ ,  $x-0$ , and  $x/1$ .

#### **Strength reduction**

Another class of simplifications is strength reduction, where we replace one operation by a cheaper one. A simple example is replacing  $2*x$  by  $x+x$  on architectures where addition is cheaper than multiplication. A more sophisticated strength reduction is applied by compilers that recognize induction variables (loop indices). Inside a for  $i$  from 1 to  $N$  loop, the expression  $4*i$  can be strength reduced to  $j=j+4$  and  $2^i$  can be strength reduced to  $j=2*j$  (with suitable initializations of  $j$  just before the loop). Other uses of algebraic identities are possible; many require a careful reading of the language

reference manual to ensure their legality. For example, even though it might be advantageous to convert  $((a + b) * f(x)) * a$  to  $((a + b) * a) * f(x)$

it is illegal in Fortran since the programmer's use of parentheses to specify the order of operations can not be violated.

Does

$$a = b + c$$

$$x = y + c + b + r$$

Shri Vishnu Engineering College For Women

contain a common sub expression of  $b+c$  that need be evaluated only once?

The answer depends on whether the language permits the use of the associative and commutative law for addition. (Note that the associative law is invalid for floating point numbers.)

**UNIT-8**  
**OPTIMIZATION**

### **8.1 PRINCIPLE SOURCES OF OPTIMIZATION**

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. Common sub expression elimination, copy propagation, deadcode elimination, and constant folding are common examples of such function-preserving transformations. The other transformations come up primarily when global optimizations are performed. Frequently, a program will include several calculations of the same value, such as an offset in an array. Some of these duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language. For example, block B5 recalculates  $4*i$  and  $4*j$ .

**Common Sub expressions** 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 re computing the expression if we can use the previously computed value. For example, the assignments to t7 and t10 have the common sub expressions  $4*I$  and  $4*j$ , respectively, on the right side in Fig. They have been eliminated in Fig by using t6 instead of t7 and t8 instead of t10. This change is what would result if we reconstructed the intermediate code from the dag for the basic block.

Example: the above Fig shows the result of eliminating both global and local common sub expressions from blocks B5 and B6 in the flow graph of Fig. We first discuss the transformation of B5 and then mention some subtleties involving arrays.

After local common sub expressions are eliminated B5 still evaluates  $4*i$  and  $4*j$ , as

Shown in the earlier fig. Both are common sub expressions; in particular, the three statements  $t8 := 4*j$ ;  $t9 := a[t[8]]$ ;  $a[t8] := x$  in B5 can be replaced by  $t9 := a[t4]$ ;  $a[t4] := x$  using  $t4$  computed in block B3. In Fig. observe that as control passes from the evaluation of  $4*j$  in B3 to B5, there is no change in  $j$ , so  $t4$  can be used if  $4*j$  is needed.

Another common sub expression comes to light in B5 after  $t4$  replaces  $t8$ . The new expression  $a[t4]$  corresponds to the value of  $a[j]$  at the source level. Not only does  $j$  retain its value as control leaves B3 and then enters B5, but  $a[j]$ , a value computed into a temporary  $t5$ , does too because there are no assignments to elements of the array  $a$  in the interim. The statement  $t9 := a[t4]$ ;  $a[t6] := t9$  in B5 can therefore be replaced by

$a[t6] := t5$  The expression in blocks B1 and B6 is not considered a common sub expression although  $t1$  can be used in both places. After control leaves B1 and before it reaches B6, it can go through B5, where there are assignments to  $a$ . Hence,  $a[t1]$  may not have the same value on reaching B6 as it did in leaving B1, and it is not safe to treat  $a[t1]$  as a common sub expression.

### Copy Propagation

Block B5 in Fig. can be further improved by eliminating  $x$  using two new transformations. One concerns assignments of the form  $f := g$  called copy statements, or copies for short. Had we gone into more detail in Example 10.2, copies would have arisen much sooner, because the algorithm for eliminating common sub expressions introduces them, as do several other algorithms. For example, when the common sub expression in  $c := d + e$  is eliminated in Fig., the algorithm uses a new variable  $t$  to hold the value of  $d + e$ . Since control may reach  $c := d + e$  either after the assignment to  $a$  or after the assignment to  $b$ , it would be incorrect to replace  $c := d + e$  by either  $c := a$  or by  $c := b$ . The idea behind the copy-propagation transformation is to use  $g$  for  $f$ , wherever possible after the copy statement  $f := g$ . For example, the assignment  $x := t3$  in block B5 of Fig. is a copy. Copy propagation applied to B5 yields:

$x := t3$

$a[t2] := t5$

$a[t4] := t3$

goto B2 Copies introduced during common subexpression elimination. This may not appear to be an improvement, but as we shall see, it gives us the opportunity to eliminate the assignment to  $x$ .

## 8.2 DEAD-CODE ELIMINATIONS

A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. A related idea is dead or useless code, statements that compute values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations. For example, we discussed the use of debug that is set to true or false at various points in the program, and used in statements like If (debug) print. By a data-flow analysis, it may be possible to deduce that each time the program reaches this statement, the value of debug is false. Usually, it is because there is one particular statement Debug :=false

That we can deduce to be the last assignment to debug prior to the test no matter what sequence of branches the program actually takes. If copy propagation replaces debug by false, then theprint statement is dead because it cannot be reached. 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, copy propagation followed by dead-code elimination removes the assignment to x and transforms 1.1 into

```
a [t2] := t5
a [t4] := t3
goto B2
```

## 8.3 PEEPHOLE OPTIMIZATION

A statement-by-statement code-generations strategy often produce target code that contains redundant instructions and suboptimal constructs .The quality of such target code can be improved by applying “optimizing” transformations to the target program.

A simple but effective technique for improving the target code is peephole optimization, a method for trying to improving the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence, whenever possible.

The peephole is a small, moving window on the target program. The code in the peephole need not contiguous, although some implementations do require this. We shall give the following examples of program transformations that are characteristic of peephole optimizations:

- Redundant-instructions elimination
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms

### **REDUNTANT LOADS AND STORES**

If we see the instructions sequence

(1) (1) MOV R0,a

(2) (2) MOV a,R0

-we can delete instructions (2) because whenever (2) is executed. (1) will ensure that the value of a is already in register R0.If (2) had a label we could not be sure that (1) was always executed immediately before (2) and so we could not remove (2).

### **UNREACHABLE CODE**

Another opportunity for peephole optimizations is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable debug is 1.In C, the source code might look like:

```
#define debug 0
....
If ( debug ) {
    Print debugging information
}
In the intermediate representations the if-statement may be translated as:
If debug =1 goto L2
Goto L2
L1: print debugging information
```

L2: .....(a)

One obvious peephole optimization is to eliminate jumps over jumps .Thus no matter what the value of debug; (a) can be replaced by:

If debug ≠1 goto L2

Print debugging information

L2: .....(b)

As the argument of the statement of (b) evaluates to a constant true it can be replaced by

If debug ≠0 goto L2

Print debugging information

L2: .....(c)

As the argument of the first statement of (c) evaluates to a constant true, it can be replaced by goto L2. Then all the statement that print debugging aids are manifestly unreachable and can be eliminated one at a time.

#### **8.4 FLOW-OF-CONTROL OPTIMIZATIONS**

The unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the jump sequence

goto L2

....

L1 : gotoL2

by the sequence

goto L2

....

L1 : goto L2

If there are now no jumps to L1, then it may be possible to eliminate the statement L1:goto L2 provided it is preceded by an unconditional jump .Similarly, the sequence if a < b goto L1

....

L1 : goto L2

can be replaced by

if a < b goto L2

....

L1 : goto L2

Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto.

Then the sequence

goto L1

.....

L1:if a<b goto L2

L3: .....(1)

may be replaced by

if a<b goto L2

goto L3

.....

L3: .....(2)

While the number of instructions in (1) and (2) is the same, we sometimes skip the unconditional jump in (2), but never in (1). Thus (2) is superior to (1) in execution time

## 8.5 REGISTER ALLOCATION

Instructions involving register operands are usually shorter and faster than those involving operands in memory. Therefore, efficient utilization of register is particularly important in generating good code. The use of registers is often subdivided into two sub problems:

1. During register allocation, we select the set of variables that will reside in registers at a point in the program.
2. During a subsequent register assignment phase, we pick the specific register that a variable will reside in. Finding an optimal assignment of registers to variables is difficult, even with single register values. Mathematically, the problem is NP-complete. The problem is further complicated because the hardware and/or the operating system of the target machine may require that certain register usage conventions be observed.

Certain machines require register pairs (an even and next odd numbered register) for some operands and results. For example, in the IBM System/370 machines integer multiplication and integer division involve register pairs. The multiplication instruction is of the form M x, y where x, is the multiplicand, is the even register of an even/odd register pair.

The multiplicand value is taken from the odd register pair. The multiplier y is a single register. The product occupies the entire even/odd register pair.

The division instruction is of the form D x, y where the 64-bit dividend occupies an even/odd register pair whose even register is x; y represents the divisor. After division, the even register holds the remainder and the odd register the quotient. Now consider the two three address code sequences (a) and (b) in which the only difference is

the operator in the second statement. The shortest assembly sequence for (a) and (b) are given in(c). Ri stands for register i. L, ST and A stand for load, store and add respectively. The optimal choice for the register into which 'a' is to be loaded depends on what will ultimately happen to e.

t := a + b t := a + b

t := t \* c t := t + c

t := t / d t := t / d

(a) (b)

Two three address code sequences

L R1, a L R0, a

A R1, b A R0, b

M R0, c A R0, c

D R0, d SRDA R0,

ST R1, t      D R0, d

                  ST R1, t

(a)            (b)

## 8.6 CHOICE OF EVALUATION ORDER

The order in which computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others. Picking a best order is another difficult, NP-complete problem. Initially, we shall avoid the

problem by generating code for the three -address statements in the order in which they have been produced by the intermediate code generator.

## 8.7 APPROCHES TO CODE GENERATION

The most important criterion for a code generator is that it produce correct code. Correctness takes on special significance because of the number of special cases that code generator must face. Given the premium on correctness, designing a code generator so it can be easily implemented, tested, and maintained is an important design goal

**Reference Counting Garbage Collection**

The difficulty in garbage collection is not the actual process of collecting the garbage--it is the problem of finding the garbage in the first place. An object is considered to be garbage when no references to that object exist. But how can we tell when no references to an object exist? A simple expedient is to keep track in each object of the total number of references to that object. That is, we add a special field to each object called a reference count . The idea is that the reference count field is not accessible to the Java program. Instead, the reference count field is updated by the Java virtual machine itself.

Consider the statement

Object p = new Integer (57);

which creates a new instance of the Integer class. Only a single variable, p, refers to the object. Thus, its reference count should be one.

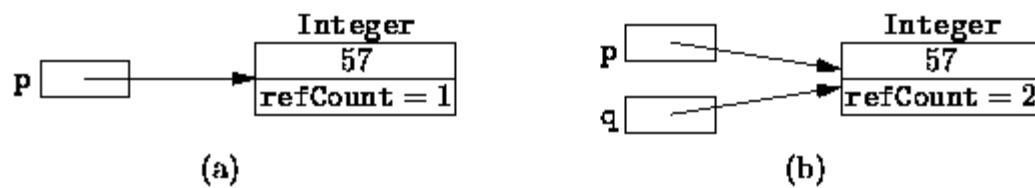


Figure: Objects with reference counters.

Now consider the following sequence of statements:

Object p = new Integer (57);

Object q = p;

This sequence creates a single Integer instance. Both p and q refer to the same object. Therefore, its reference count should be two.

## Shri Vishnu Engineering College For Women

In general, every time one reference variable is assigned to another, it may be necessary to update several reference counts. Suppose p and q are both reference variables. The assignment

```
p = q;
```

would be implemented by the Java virtual machine as follows:

```
if (p != q)
{
    if (p != null)
        --p.refCount;
    p = q;
    if (p != null)
        ++p.refCount;
}
```

For example suppose p and q are initialized as follows:

```
Object p = new Integer (57);
```

```
Object q = new Integer (99);
```

As shown in Figure □ (a), two Integer objects are created, each with a reference count of one. Now, suppose we assign q to p using the code sequence given above. Figure □ (b) shows that after the assignment, both p and q refer to the same object--its reference count is two. And the reference count on Integer(57) has gone to zero which indicates that it is garbage.

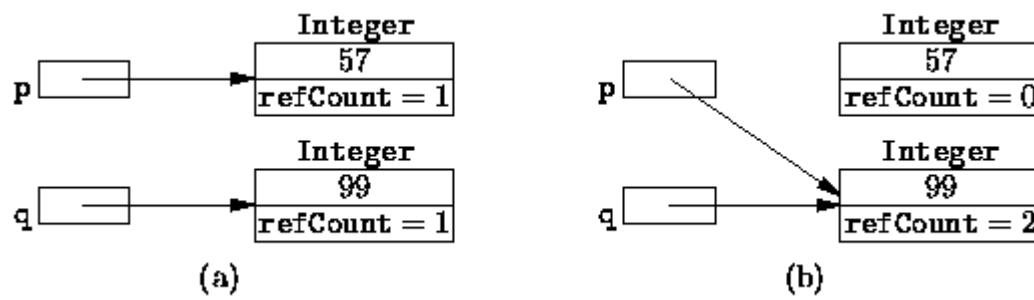


Figure: Reference counts before and after the assignment `p = q`.

The costs of using reference counts are twofold: First, every object requires the special reference count field. Typically, this means an extra word of storage must be allocated in each object. Second, every time one reference is assigned to another, the reference counts

must be adjusted as above. This increases significantly the time taken by assignment statements.

The advantage of using reference counts is that garbage is easily identified. When it becomes necessary to reclaim the storage from unused objects, the garbage collector needs only to examine the reference count fields of all the objects that have been created by the program. If the reference count is zero, the object is garbage.

It is not necessary to wait until there is insufficient memory before initiating the garbage collection process. We can reclaim memory used by an object immediately when its reference goes to zero. Consider what happens if we implement the Java assignment  $p = q$  in the Java virtual machine as follows:

```
if (p != q)
{
    if (p != null)
        if (--p.refCount == 0)
            heap.release (p);

    p = q;
    if (p != null)
        ++p.refCount;
}
```

Notice that the release method is invoked immediately when the reference count of an object goes to zero, i.e., when it becomes garbage. In this way, garbage may be collected incrementally as it is created.

**TEXT BOOKS:**

1. Compilers, Principles Techniques and Tools- Alfred V Aho, Monical S Lam, Ravi Sethi, Jeffrey D. Ullman, 2<sup>nd</sup> ed, Pearson, 2007.
2. Principles of compiler design, V. Raghavan, 2<sup>nd</sup> ed, TMH, 2011.
3. Principles of compiler design, 2<sup>nd</sup> ed, Nandini Prasad, Elsevier

**REFERENCE BOOKS:**

1. <http://www.nptel.iitm.ac.in/downloads/106108052/>
2. Compiler construction, Principles and Practice, Kenneth C Louden, CENGAGE
3. Implementations of Compiler, A new approach to Compilers including the algebraic methods, Yunlinsu, SPRINGER

## UNIT -1

### 1.1 OVERVIEW OF LANGUAGE PROCESSING SYSTEM

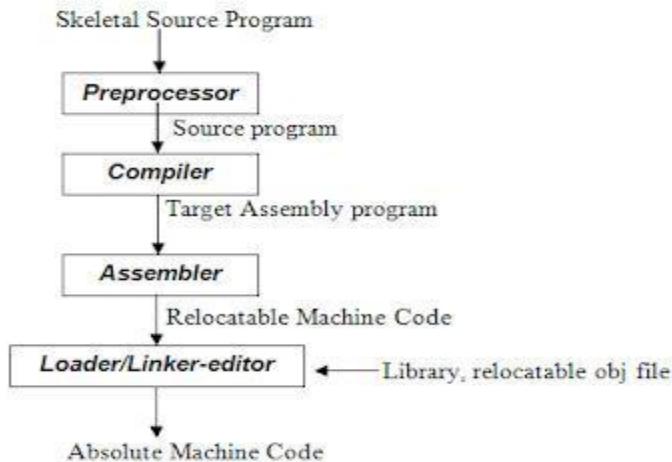


Fig 1.1 Language -processing System

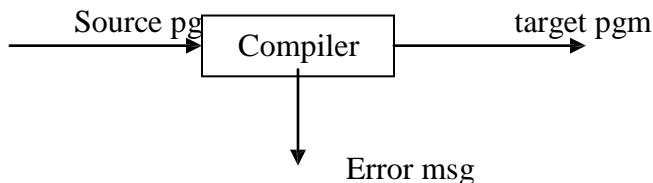
### 1.2 Preprocessor

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

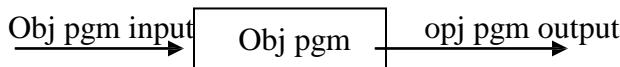
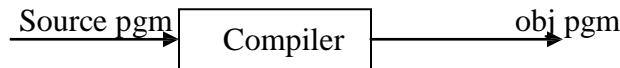
1. *Macro processing*: A preprocessor may allow a user to define macros that are short hands for longer constructs.
2. *File inclusion*: A preprocessor may include header files into the program text.
3. *Rational preprocessor*: these preprocessors augment older languages with more modern flow-of-control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

### 1.3 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.

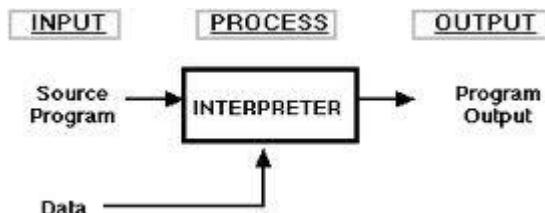


Executing a program written in HLL programming language is basically of two parts. the source program must first be compiled/translated into an object program. Then the results object program is loaded into memory and executed.



**1.4 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 assembler were written to automate the translation of assembly language into machine language. The input to an assembler program is called source program, the output is a machine language translation (object program).

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



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

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

#### *Advantages:*

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

***Disadvantages:***

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

**2 *Loader and Link-editor:***

Once the assembler procedures 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 execute. 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 over come this 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.

## **1.6 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. Beside program translation, the translator performs another very important role, the error-detection. Any violation of d HLL specification would be detected and reported to the programmers. Important role of 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.

## **1.7 TYPE OF TRANSLATORS:-**

- INTERPRETOR
- COMPILER
- PREPROSSESSOR

## 1.9 STRUCTURE OF THE COMPILER DESIGN

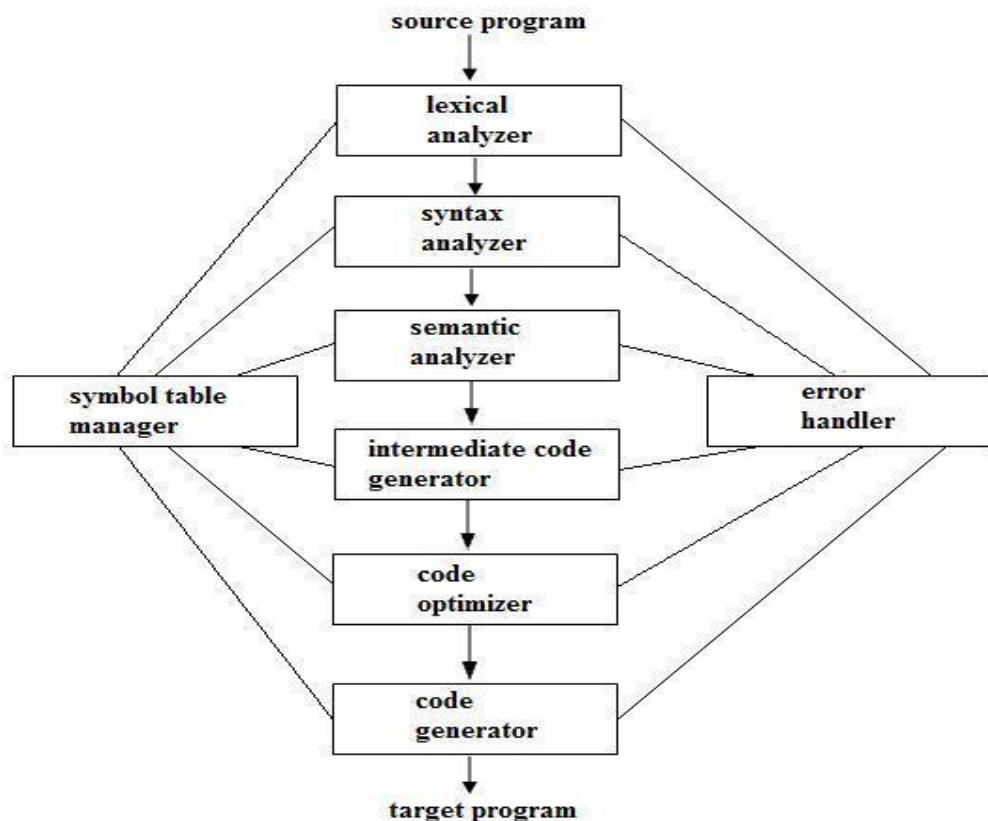
**Phases of a compiler:** A compiler operates in phases. A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation. The phases of a compiler are shown in below

There are two phases of compilation.

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

Compilation process is partitioned into

### PHASES OF A COMPILER



**Fig 1.5 Phases of a compiler**

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**.

#### 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 program and records essential information about 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 same job as the original, but in a way that saves time and / or spaces.

### 1, Local Optimization:-

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

If  $A > B$  goto L2

Goto L3

**L2 :**

This can be replaced by a single statement

If  $A < B$  goto L3

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

$A := B + C + D$

$E := B + C + F$

Might be evaluated as

$T1 := B + C$

$A := T1 + D$

$E := T1 + F$

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

### 2, 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 :-

Cg 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 :-

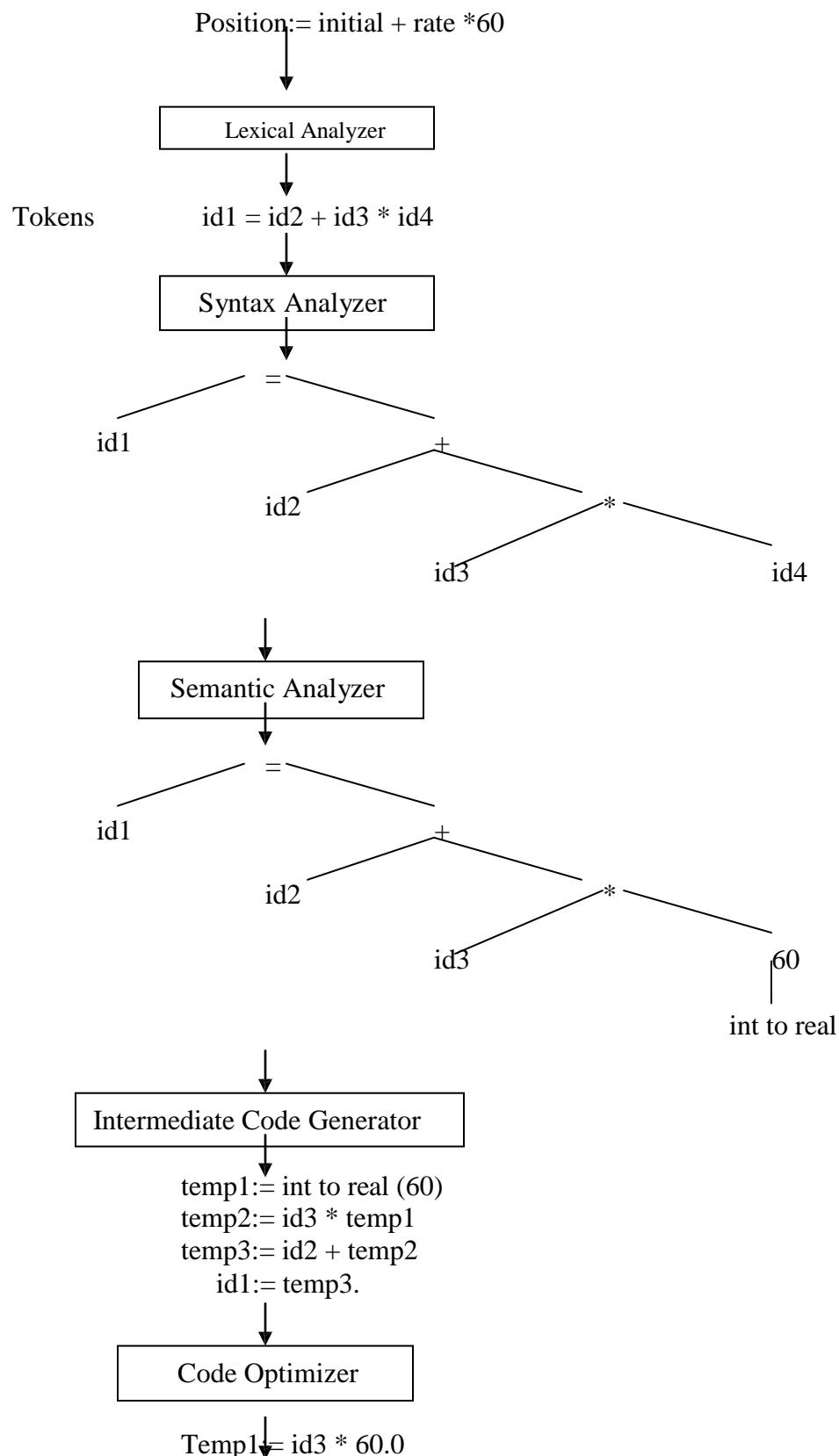
A compiler needs to collect information about all the data objects that appear in the source program. The information about data objects is collected by the early phases of the compiler-lexical and syntactic analyzers. The data structure used to record this information is called as Symbol Table.

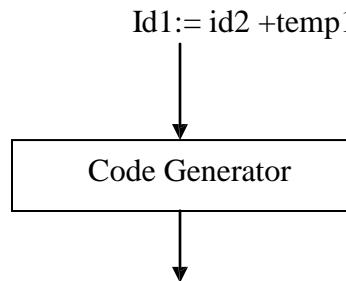
## Error Handling :-

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:





MOVF id3, r2  
MULF \*60.0, r2  
MOVF id2, r2  
ADDF r2, r1  
MOVF r1, id1

### **1.10 TOKEN**

LA reads the source program one character at a time, carving the source program into a sequence of automatic units called 'Tokens'.

- 1, Type of the token.
- 2, Value of the token.

Type : variable, operator, keyword, constant

Value : Name of variable, current variable (or) pointer to symbol table.

**If the symbols given in the standard format the LA accepts and produces token as output.** Each token is a sub-string of the program that is to be treated as a single unit. Token are two types.

- 1, Specific strings such as IF (or) semicolon.
- 2, Classes of string such as identifiers, label, constants.

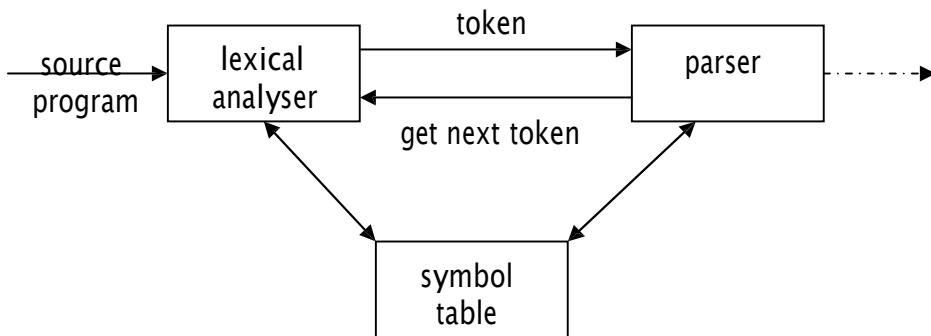
## UNIT II- LEXICAL ANALYSIS

### **LEXICAL ANALYSIS**

Lexical analysis is the process of converting a sequence of characters into a sequence of tokens. A program or function which performs lexical analysis is called a lexical analyzer or scanner. A lexer often exists as a single function which is called by a parser or another function.

### **THE ROLE OF THE LEXICAL ANALYZER**

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



- Upon receiving a “get next token” command from the parser, the lexical analyzer reads input characters until it can identify the next token.

### **ISSUES OF LEXICAL ANALYZER**

There are three issues in lexical analysis:

- To make the design simpler.
- To improve the efficiency of the compiler.
- To enhance the computer portability.

### **TOKENS**

A token is a string of characters, categorized according to the rules as a symbol (e.g., IDENTIFIER, NUMBER, COMMA). The process of forming tokens from an input stream of characters is called **tokenization**.

A token can look like anything that is useful for processing an input text stream or text file. Consider this expression in the C programming language: sum=3+2;

Lexeme	Token type
sum	Identifier
=	Assignment operator
3	Number
+	Addition operator
2	Number
;	End of statement

## **LEXEME:**

Collection or group of characters forming tokens is called Lexeme.

## **PATTERN:**

A pattern is a description of the form that the lexemes of a token may take.

In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.

## **Attributes for Tokens**

Some tokens have attributes that can be passed back to the parser. The lexical analyzer collects information about tokens into their associated attributes. The attributes influence the translation of tokens.

- i) Constant : value of the constant
- ii) Identifiers: pointer to the corresponding symbol table entry.

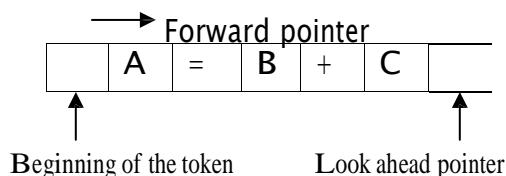
## **ERROR RECOVERY STRATEGIES IN LEXICAL ANALYSIS:**

The following are the error-recovery actions in lexical analysis:

- 1) Deleting an extraneous character.
- 2) Inserting a missing character.
- 3) Replacing an incorrect character by a correct character.
- 4) Transforming two adjacent characters.
- 5) **Panic mode recovery:** Deletion of successive characters from the token until error is resolved.

## **INPUT BUFFERING**

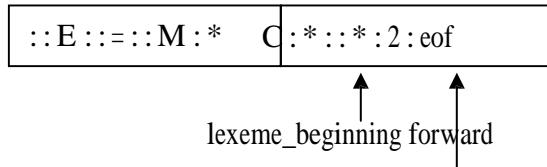
We often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme. As characters are read from left to right, each character is stored in the buffer to form a meaningful token as shown below:



We introduce a two-buffer scheme that handles large look aheads safely. We then consider an improvement involving "sentinels" that saves time checking for the ends of buffers.

## BUFFER PAIRS

- A buffer is divided into two N-character halves, as shown below



- Each buffer is of the same size N, and N is usually the number of characters on one disk block. E.g., 1024 or 4096 bytes.
- Using one system read command we can read N characters into a buffer.
- If fewer than N characters remain in the input file, then a special character, represented by **eof**, marks the end of the source file.
- Two pointers to the input are maintained:
  1. Pointer **lexeme\_beginning**, marks the beginning of the current lexeme, whose extent we are attempting to determine.
  2. Pointer **forward** scans ahead until a pattern match is found.  
Once the next lexeme is determined, forward is set to the character at its right end.
- The string of characters between the two pointers is the current lexeme.  
After the lexeme is recorded as an attribute value of a token returned to the parser, lexeme\_beginning is set to the character immediately after the lexeme just found.

### Advancing forward pointer:

Advancing forward pointer requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer. If the end of second buffer is reached, we must again reload the first buffer with input and the pointer wraps to the beginning of the buffer.

### Code to advance forward pointer:

```

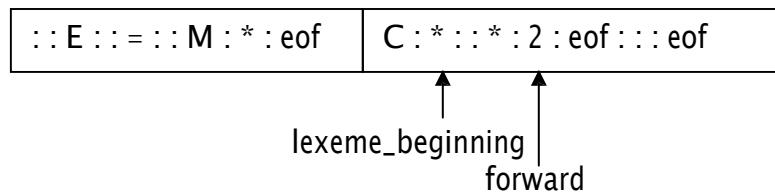
if forward at end of first half then begin
  reload second half;
  forward := forward + 1
end
else if forward at end of second half then begin
  reload second half;
  move forward to beginning of first half
end
else forward := forward + 1;

```

## SENTINELS

- For each character read, we make two tests: one for the end of the buffer, and one to determine what character is read. We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a sentinel character at the end.
- The sentinel is a special character that cannot be part of the source program, and a natural choice is the character eof.

- The sentinel arrangement is as shown below:



Note that eof retains its use as a marker for the end of the entire input. Any eof that appears other than at the end of a buffer means that the input is at an end.

### **Code to advance forward pointer:**

```

forward := forward + 1;
if forward ↑ = eof then begin
    if forward at end of first half then begin
        reload second half;
        forward := forward + 1
    end
    else if forward at end of second half then begin
        reload first half;
        move forward to beginning of first half
    end
else /* eof within a buffer signifying end of input */
    terminate lexical analysis
end

```

## **SPECIFICATION OF TOKENS**

There are 3 specifications of tokens:

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

### **Strings and Languages**

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

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

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

### **Operations on strings**

The following string-related terms are commonly used:

1. A **prefix** of string  $s$  is any string obtained by removing zero or more symbols from the end of string  $s$ .

For example, ban is a prefix of banana.

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

### Operations on languages:

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

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

The following example shows the operations on strings: Let

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

1. Union :  $L \cup S = \{0,1,a,b,c\}$
2. Concatenation :  $L \cdot S = \{0a,1a,0b,1b,0c,1c\}$
3. Kleene closure :  $L^* = \{\epsilon, 0, 1, 00, \dots\}$
4. Positive closure :  $L^+ = \{0, 1, 00, \dots\}$

### Regular Expressions

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

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

1.  $\epsilon$  is a regular expression, and  $L(\epsilon)$  is  $\{\epsilon\}$ , that is, the language whose sole member is the empty string.
2. If 'a' is a symbol in  $\Sigma$ , then 'a' is a regular expression, and  $L(a) = \{a\}$ , that is, the language with one string, of length one, with 'a' in its one position.
3. Suppose r and s are regular expressions denoting the languages  $L(r)$  and  $L(s)$ . Then,
  - a)  $(r)(s)$  is a regular expression denoting the language  $L(r)L(s)$ .
  - b)  $(r)^*$  is a regular expression denoting  $(L(r))^*$ .
  - c)  $(r)$  is a regular expression denoting  $L(r)$ .

4. The unary operator \* has highest precedence and is left associative.
5. Concatenation has second highest precedence and is left associative.
6. | has lowest precedence and is left associative.

## Regular set

A language that can be defined by a regular expression is called a regular set.

If two regular expressions r and s denote the same regular set, we say they are equivalent and Write  $r = s$ .

There are a number of algebraic laws for regular expressions that can be used to manipulate into equivalent forms.

For instance,  $r|s = s|r$  is commutative;  $r|(s|t) = (r|s)|t$  is associative.

## Regular Definitions

Giving names to regular expressions is referred to as a Regular definition. If  $\Sigma$  is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form

$$\begin{aligned}d_1 &\rightarrow r_1 \\d_2 &\rightarrow r_2 \\ \dots \\d_n &\rightarrow r_n\end{aligned}$$

1. Each  $d_i$  is a distinct name.
2. Each  $r_i$  is a regular expression over the alphabet  $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$ .

Example: Identifiers is the set of strings of letters and digits beginning with a letter. Regular definition for this set:

letter  $\rightarrow A | B | \dots | Z | a | b | \dots | z |$

digit  $\rightarrow 0 | 1 | \dots | 9$

id  $\rightarrow$  letter ( letter | digit ) \*

## Shorthands

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

### 1. One or more instances (+):

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

## 2. Zero or one instance ( ?):

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

## 3. Character Classes:

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

## Non-regular Set

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

## RECOGNITION OF TOKENS

Consider the following grammar fragment:

$\text{stmt} \rightarrow \text{if expr then stmt}$   
           |  $\text{if expr then stmt else stmt}$   
           |  $\epsilon$

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

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

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

if	$\rightarrow$	if
then	$\rightarrow$	then
else	$\rightarrow$	else
relop	$\rightarrow$	$<= = > >=$
id	$\rightarrow$	letter(letter digit)*
num	$\rightarrow$	digit+ (.digit+)?(E(+ -)?digit+)?

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

## Transition diagrams

It is a diagrammatic representation to depict the action that will take place when a lexical analyzer is called by the parser to get the next token. It is used to keep track of information about the characters that are seen as the forward pointer scans the input.

## Transition diagram for relational operators

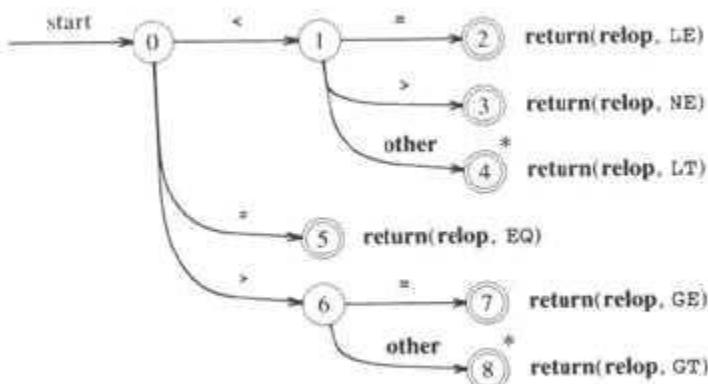
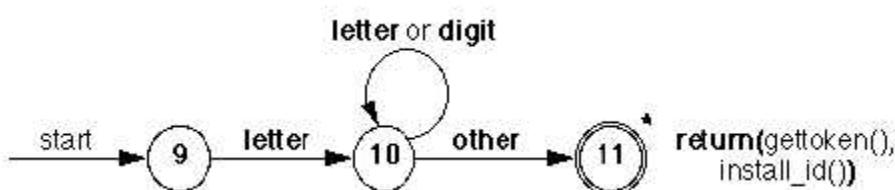


Fig. 3.12. Transition diagram for relational operators.

## Transition diagram for identifiers and keywords



## A LANGUAGE FOR SPECIFYING LEXICAL ANALYZER

There is a wide range of tools for constructing lexical analyzers.

- Lex
- YACC

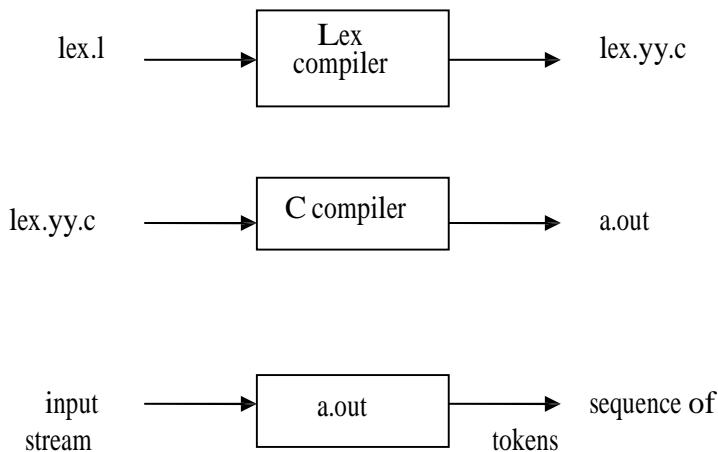
### LEX

Lex is a computer program that generates lexical analyzers. Lex is commonly used with the yacc parser generator.

### Creating a lexical analyzer

- First, a specification of a lexical analyzer is prepared by creating a program lex.l in the Lex language. Then, lex.l is run through the Lex compiler to produce a C program lex.yy.c.

- Finally, lex.yy.c is run through the C compiler to produce an object program a.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.



## Lex Specification

A Lex program consists of three parts:

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

**Definitions** include declarations of variables, constants, and regular definitions

**Rules** are statements of the form p1

```
{action1}
p2 {action2}
...
pn {actionn}
```

where  $p_i$  is regular expression and  $action_i$  describes what action the lexical analyzer should take when pattern  $p_i$  matches a lexeme. Actions are written in C code.

- User subroutines** are auxiliary procedures needed by the actions. These can be compiled separately and loaded with the lexical analyzer.

## YACC- YET ANOTHER COMPILER-COMPILER

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

## **FINITE AUTOMATA**

Finite Automata is one of the mathematical models that consist of a number of states and edges. It is a transition diagram that recognizes a regular expression or grammar.

### **Types of Finite Automata**

There are two types of Finite Automata:

Non-deterministic Finite Automata (NFA)

Deterministic Finite Automata (DFA)

### **Non-deterministic Finite Automata**

NFA is a mathematical model that consists of five tuples denoted by

$$M = \{ Q_n, \Sigma, \delta, q_0, f_n \}$$

$Q_n$  – finite set of states

$\Sigma$  – finite set of input symbols

$\delta$  – transition function that maps state-symbol pairs to set of states

$q_0$  – starting state

$f_n$  – final state

### **Deterministic Finite Automata**

DFA is a special case of a NFA in which i) no state has an  $\epsilon$ -transition.

ii) there is at most one transition from each state on any input.

DFA has five tuples denoted by

$$M = \{ Q_d, \Sigma, \delta, q_0, f_d \}$$

$Q_d$  – finite set of states

$\Sigma$  – finite set of input symbols

$\delta$  – transition function that maps state-symbol pairs to set of states

$q_0$  – starting state

$f_d$  – final state

### **Converting a Regular Expression into a Deterministic Finite Automaton**

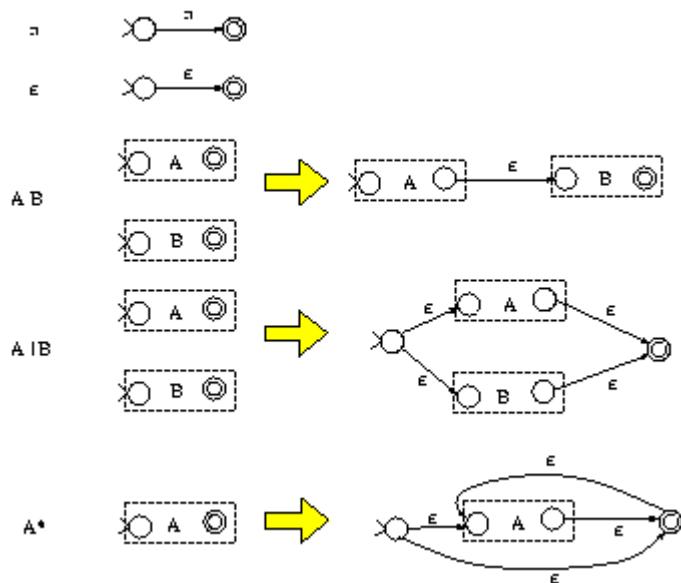
The task of a scanner generator, such as flex, is to generate the transition tables or to synthesize the scanner program given a scanner specification (in the form of a set of REs). So it needs to convert a RE into a DFA. This is accomplished in two steps: first it converts

a RE into a non-deterministic finite automaton (NFA) and then it converts the NFA into a DFA.

A NFA is similar to a DFA but it also permits multiple transitions over the same character and transitions over  $\epsilon$ . The first type indicates that, when reading the common character associated with these transitions, we have more than one choice; the NFA succeeds if at least one of these choices succeeds. The  $\epsilon$  transition doesn't consume any input characters, so you may jump to another state for free.

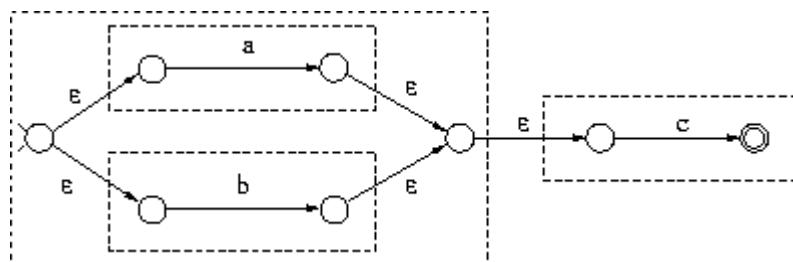
Clearly DFAs are a subset of NFAs. But it turns out that DFAs and NFAs have the same expressive power. The problem is that when converting a NFA to a DFA we may get an exponential blowup in the number of states.

We will first learn how to convert a RE into a NFA. This is the easy part. There are only 5 rules, one for each type of RE:



The algorithm constructs NFAs with only one final state. For example, the third rule indicates that, to construct the NFA for the RE  $AB$ , we construct the NFAs for  $A$  and  $B$  which are represented as two boxes with one start and one final state for each box. Then the NFA for  $AB$  is constructed by connecting the final state of  $A$  to the start state of  $B$  using an empty transition.

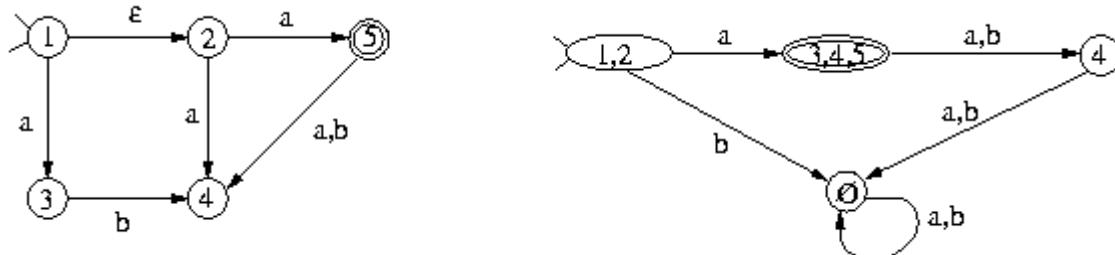
For example, the RE  $(a|b)c$  is mapped to the following NFA:



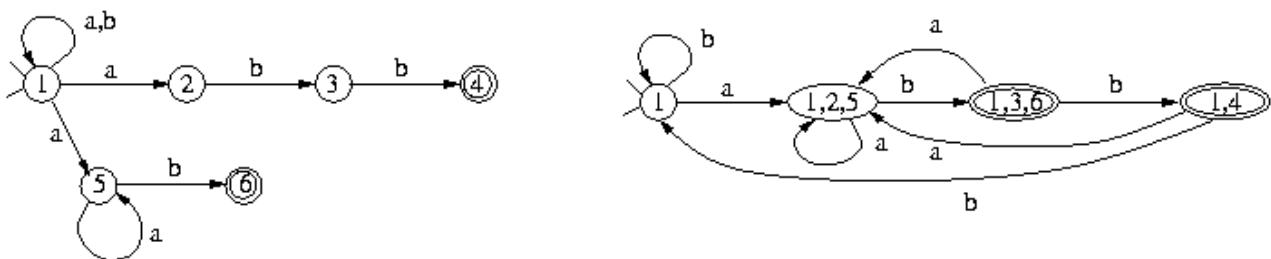
The next step is to convert a NFA to a DFA (called *subset construction*). Suppose that you assign a number to each NFA state. The DFA states generated by subset construction have sets of numbers, instead of just one number. For example, a DFA state may have been assigned the set  $\{5,6,8\}$ . This indicates that arriving to the state labeled  $\{5,6,8\}$  in the DFA

is the same as arriving to the state 5, the state 6, or the state 8 in the NFA when parsing the same input. (Recall that a particular input sequence when parsed by a DFA, leads to a unique state, while when parsed by a NFA it may lead to multiple states.)

First we need to handle transitions that lead to other states for free (without consuming any input). These are the  $\epsilon$ -transitions. We define the *closure* of a NFA node as the set of all the nodes reachable by this node using zero, one, or more  $\epsilon$ -transitions. For example, The closure of node 1 in the left figure below

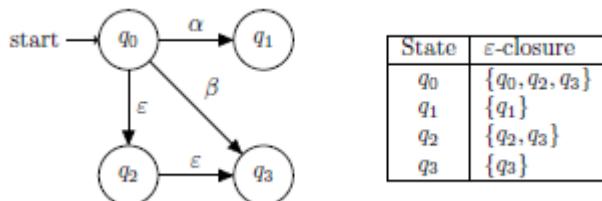


is the set  $\{1,2\}$ . The start state of the constructed DFA is labeled by the closure of the NFA start state. For every DFA state labeled by some set  $\{s_1, \dots, s_n\}$  and for every character  $c$  in the language alphabet, you find all the states reachable by  $s_1, s_2, \dots$ , or  $s_n$  using  $c$  arrows and you union together the closures of these nodes. If this set is not the label of any other node in the DFA constructed so far, you create a new DFA node with this label. For example, node  $\{1,2\}$  in the DFA above has an arrow to a  $\{3,4,5\}$  for the character  $a$  since the NFA node 3 can be reached by 1 on  $a$  and nodes 4 and 5 can be reached by 2. The  $b$  arrow for node  $\{1,2\}$  goes to the error node which is associated with an empty set of NFA nodes. The following NFA recognizes  $(a|b)^*(abb|a^+b)$ , even though it wasn't constructed with the 5 RE-to-NFA rules. It has the following DFA:

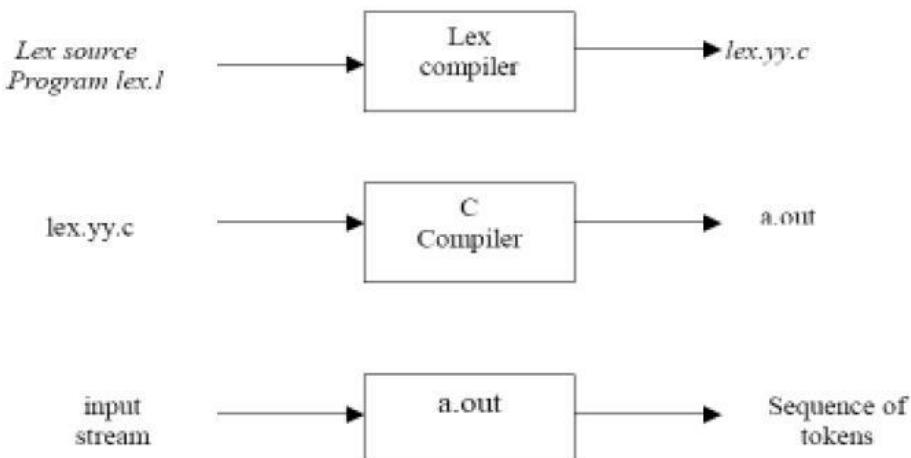


## Converting NFAs to DFAs

To convert an NFA to a DFA, we must and a way to remove all "ε"-transitions and to ensure that there is one transition per symbol in each state. We do this by constructing a DFA in which each state corresponds to a set of some states from the NFA. In the DFA, transitions from a state  $S$  by some symbol go to the state  $S'$  that consists of all the possible NFA-states that could be reached by from some NFA state  $q$  contained in the present DFA state  $S$ . The resulting DFA "simulates" the given NFA in the sense that a single DFA-transition represents many simultaneous NFA-transitions. The first concept we need is the "E-closure pronounced 'epsilon closure'". The "E-closure" of an NFA state  $q$  is the set containing  $q$  along with all states in the automaton that are reachable by any number of "E-transitions from  $q$ ". In the following automaton, the "E-closures" are given in the table to the right:



Likewise, we can do the " $\epsilon$ -closure" of a set of states to be the states reachable by " $\epsilon$ " - transitions from its members. In other words, this is the union of the " $\epsilon$ " -closures of its elements. To convert our NFA to its DFA counterpart, we begin by taking the " $\epsilon$ " -closure of the start state  $q$  of our NFA and constructing a new start state  $S$ . in our DFA corresponding to that " $\epsilon$ " -closure. Next, for each symbol in our alphabet, we record the set of NFA states that we can reach from  $S$  on that symbol. For each such set, we make a DFA state corresponding to its " $\epsilon$ -closure", taking care to do this only once for each set. In the case two sets are equal, we simply reuse the existing DFA state that we already constructed. This process is then repeated for each of the new DFA states (that is, set of NFA states) until we run out of DFA states to process. Finally, every DFA state whose corresponding set of NFA states contains an accepting state is itself marked as an accepting state.



## 2.17 Lex specifications:

A Lex program (the `.l` file ) consists of three parts:

```

declarations
%%
translation rules
%%
auxiliary procedures

```

1. The **declarations** section includes declarations of variables, manifest constants (A manifest constant is an identifier that is declared to represent a constant e.g. `#define PIE 3.14`), and regular definitions.
2. The **translation rules** of a Lex program are statements of the form :

$p1 \quad \{action\}$

$p_2$	{action 2}
$p_3$	{action 3}
...	...
...	...

where each  $p$  is a regular expression and each *action* is a program fragment describing what action the lexical analyzer should take when a pattern  $p$  matches a lexeme. In Lex the actions are written in C.

3. The third section holds whatever *auxiliary procedures* are needed by the *actions*. Alternatively these procedures can be compiled separately and loaded with the lexical analyzer.

## **UNIT -3**

### **SYNTAX ANALYSIS**

#### **3.1 ROLE OF THE PARSER**

Parser obtains a string of tokens from the lexical analyzer and verifies that it can be generated by the language for the source program. The parser should report any syntax errors in an intelligible fashion. The two types of parsers employed are:

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

Therefore there are two types of parsing methods— top-down parsing and bottom-up parsing

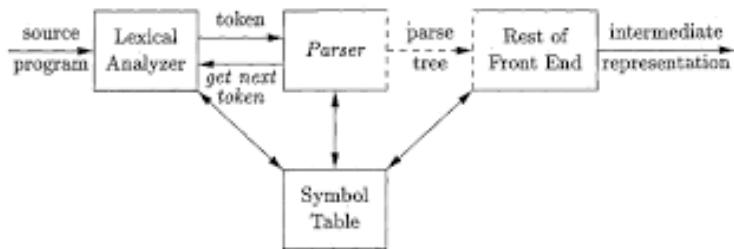


Figure 4.1: Position of parser in compiler model

#### **3.2 TOP-DOWN PARSING**

A program that performs syntax analysis is called a parser. A syntax analyzer takes tokens as input and output error message if the program syntax is wrong. The parser uses symbol-look-ahead and an approach called top-down parsing without backtracking. Top-down parsers check to see if a string can be generated by a grammar by creating a parse tree starting from the initial symbol and working down. Bottom-up parsers, however, check to see if a string can be generated from a grammar by creating a parse tree from the leaves, and working up. Early parser generators such as YACC creates bottom-up parsers whereas many of Java parser generators such as JavaCC create top-down parsers.

#### **3.3 RECURSIVE DESCENT PARSING**

Typically, top-down parsers are implemented as a set of recursive functions that descent through a parse tree for a string. This approach is known as recursive descent parsing, also known as LL(k) parsing where the first L stands for left-to-right, the second L stands for

leftmost-derivation, and  $k$  indicates  $k$ -symbol lookahead. Therefore, a parser using the single symbol look-ahead method and top-down parsing without backtracking is called LL(1) parser. In the following sections, we will also use an extended BNF notation in which some regulation expression operators are to be incorporated.

A syntax expression defines sentences of the form , or . A syntax of the form defines sentences that consist of a sentence of the form followed by a sentence of the form followed by a sentence of the form . A syntax of the form defines zero or one occurrence of the form .

A syntax of the form defines zero or more occurrences of the form .

A usual implementation of an LL(1) parser is:

- initialize its data structures,
- get the lookahead token by calling scanner routines, and
- call the routine that implements the start symbol.

Here is an example.

```
proc syntaxAnalysis()  
begin  
    initialize(); // initialize global data and structures  
    nextToken(); // get the lookahead token  
    program(); // parser routine that implements the start symbol  
end;
```

### 3.4 FIRST AND FOLLOW

To compute  $\text{FIRST}(X)$  for all grammar symbols  $X$ , apply the following rules until no more terminals or  $e$  can be added to any  $\text{FIRST}$  set.

1. If  $X$  is terminal, then  $\text{FIRST}(X)$  is  $\{X\}$ .
2. If  $X \rightarrow e$  is a production, then add  $e$  to  $\text{FIRST}(X)$ .
3. If  $X$  is nonterminal 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  $e$  is in all of  $\text{FIRST}(Y_1), \dots, \text{FIRST}(Y_{i-1})$  that is,  
 $Y_1 \dots Y_{i-1} =^* e$ . If  $e$  is in  $\text{FIRST}(Y_j)$  for all  $j=1, 2, \dots, k$ , then add  $e$  to  $\text{FIRST}(X)$ . For example, everything in  $\text{FIRST}(Y_j)$  is surely in  $\text{FIRST}(X)$ . If  $y_1$  does not derive  $e$ , then we add nothing more to  $\text{FIRST}(X)$ , but if  $Y_1 =^* e$ , then we add  $\text{FIRST}(Y_2)$  and so on.

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

1. Place \$ in FOLLOW(S), where S is the start symbol and \$ in the input right endmarker.
2. If there is a production A=>aBs where FIRST(s) except e is placed in FOLLOW(B).
3. If there is a production A->aB or a production A->aBs where FIRST(s) contains e, then everything in FOLLOW(A) is in FOLLOW(B).

Consider the following example to understand the concept of First and Follow. Find the first and follow of all nonterminals in the Grammar-

E -> TE'

E' -> +TE'|e

T -> FT'

T' -> \*FT'|e

F -> (E)|id

Then:

FIRST(E)=FIRST(T)=FIRST(F)={(,id}

FIRST(E')={+,e}

FIRST(T')={\*,e}

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

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

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

For example, id and left parenthesis are added to FIRST(F) by rule 3 in definition of FIRST with i=1 in each case, since FIRST(id)=(id) and FIRST('(')={()} by rule 1. Then by rule 3 with i=1, the production T -> FT' implies that id and left parenthesis belong to FIRST(T) also.

To compute FOLLOW, we put \$ in FOLLOW(E) by rule 1 for FOLLOW. By rule 2 applied to production F-> (E), right parenthesis is also in FOLLOW(E). By rule 3 applied to production E-> TE', \$ and right parenthesis are in FOLLOW(E').

### **3.5 CONSTRUCTION OF PREDICTIVE PARSING TABLES**

For any grammar G, the following algorithm can be used to construct the predictive parsing table. The algorithm is

Input : Grammar G

Output : Parsing table M

Method

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

### **3.6.LL(1) GRAMMAR**

The above algorithm can be applied to any grammar G to produce a parsing table M. For some Grammars, for example if G is left recursive or ambiguous, then M will have at least one multiply-defined entry. A grammar whose parsing table has no multiply defined entries is said to be LL(1). It can be shown that the above algorithm can be used to produce for every LL(1) grammar G a parsing table M that parses all and only the sentences of G. LL(1) grammars have several distinctive properties. No ambiguous or left recursive grammar can be LL(1). There remains a question of what should be done in case of multiply defined entries. One easy solution is to eliminate all left recursion and left factoring, hoping to produce a grammar which will produce no multiply defined entries in the parse tables. Unfortunately there are some grammars which will give an LL(1) grammar after any kind of alteration. In general, there are no universal rules to convert multiply defined entries into single valued entries without affecting the language recognized by the parser.

The main difficulty in using predictive parsing is in writing a grammar for the source language such that a predictive parser can be constructed from the grammar. Although left recursion elimination and left factoring are easy to do, they make the resulting grammar hard to read and difficult to use the translation purposes. To alleviate some of this difficulty, a common organization for a parser in a compiler is to use a predictive parser for control

constructs and to use operator precedence for expressions. However, if an LR parser generator is available, one can get all the benefits of predictive parsing and operator precedence automatically.

### 3.7. ERROR RECOVERY IN PREDICTIVE PARSING

The stack of a nonrecursive predictive parser makes explicit the terminals and nonterminals that the parser hopes to match with the remainder of the input. We shall therefore refer to symbols on the parser stack in the following discussion. An error is detected during predictive parsing when the terminal on top of the stack does not match the next input symbol or when nonterminal A is on top of the stack, a is the next input symbol, and the parsing table entry  $M[A,a]$  is empty.

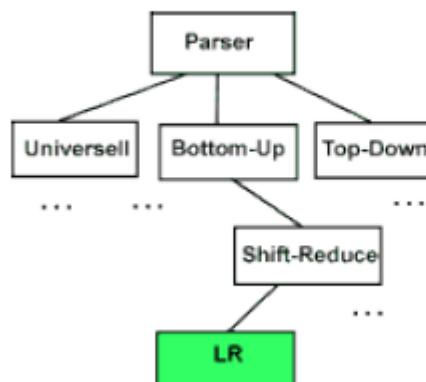
Panic-mode error recovery is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens appears. Its effectiveness depends on the choice of synchronizing set. The sets should be chosen so that the parser recovers quickly from errors that are likely to occur in practice. Some heuristics are as follows

- ⊕ As a starting point, we can place all symbols in  $\text{FOLLOW}(A)$  into the synchronizing set for nonterminal A. If we skip tokens until an element of  $\text{FOLLOW}(A)$  is seen and pop A from the stack, it is likely that parsing can continue.
- ⊕ It is not enough to use  $\text{FOLLOW}(A)$  as the synchronizing set for A. For example, if semicolons terminate statements, as in C, then keywords that begin statements may not appear in the FOLLOW set of the nonterminal generating expressions. A missing semicolon after an assignment may therefore result in the keyword beginning the next statement being skipped. Often, there is a hierarchical structure on constructs in a language; e.g., expressions appear within statements, which appear within blocks, and so on. We can add to the synchronizing set of a lower construct the symbols that begin higher constructs. For example, we might add keywords that begin statements to the synchronizing sets for the nonterminals generating expressions.
- ⊕ If we add symbols in  $\text{FIRST}(A)$  to the synchronizing set for nonterminal A, then it may be possible to resume parsing according to A if a symbol in  $\text{FIRST}(A)$  appears in the input.

- ⊕ If a nonterminal can generate the empty string, then the production deriving e can be used as a default. Doing so may postpone some error detection, but cannot cause an error to be missed. This approach reduces the number of nonterminals that have to be considered during error recovery.
- ⊕ If a terminal on top of the stack cannot be matched, a simple idea is to pop the terminal, issue a message saying that the terminal was inserted, and continue parsing. In effect, this approach takes the synchronizing set of a token to consist of all other tokens.

### 3.8 LR PARSING INTRODUCTION

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



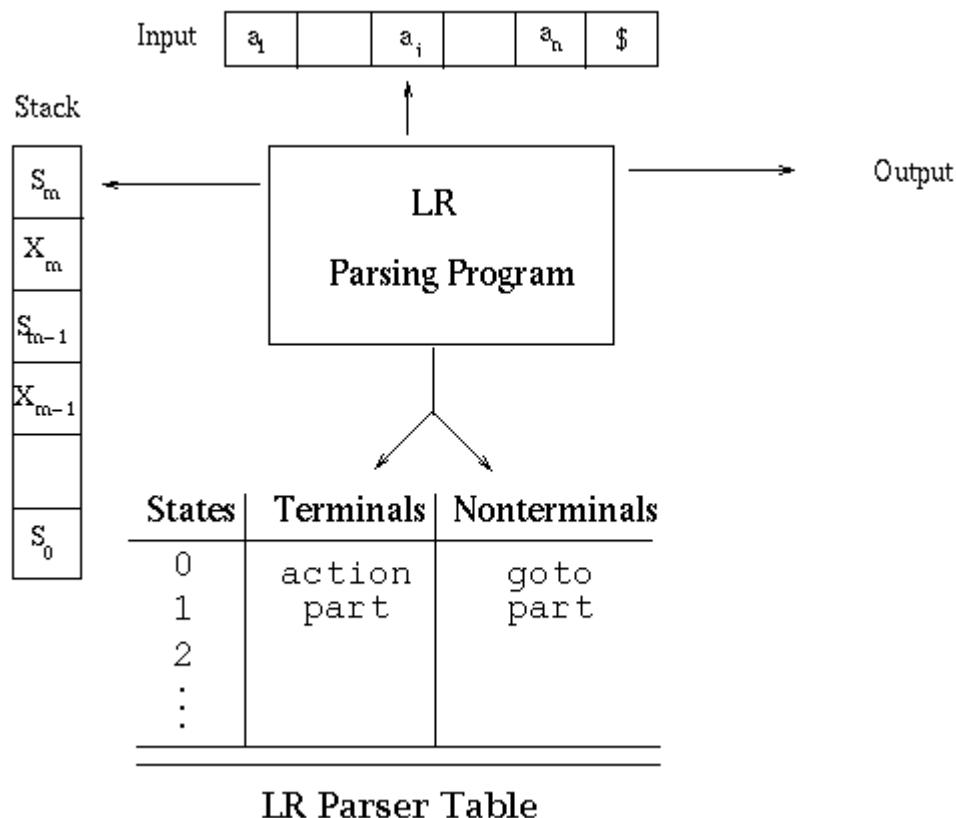
### WHY LR PARSING:

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

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

## MODELS OF LR PARSERS

The schematic form of an LR parser is shown below.



The program uses a stack to store a string of the form  $s_0X_1s_1X_2\dots X_ms_m$  where  $s_m$  is on top. Each  $X_i$  is a grammar symbol and each  $s_i$  is a symbol representing a state. Each state symbol summarizes the information contained in the stack below it. The combination of the state symbol on top of the stack and the current input symbol are used to index the parsing table and determine the shift/reduce parsing decision. The parsing table consists of two parts: a parsing action function *action* and a goto function *goto*. The program driving the LR parser behaves as follows: It determines  $s_m$  the state currently on top of the stack and  $a_i$  the current input symbol. It then consults  $\text{action}[s_m, a_i]$ , which can have one of four values:

- shift  $s$ , where  $s$  is a state
- reduce by a grammar production  $A \rightarrow b$

- accept
- error

The function goto takes a state and grammar symbol as arguments and produces a state.

For a parsing table constructed for a grammar G, the goto table is the transition function of a deterministic finite automaton that recognizes the viable prefixes of G. Recall that the viable prefixes of G are those prefixes of right-sentential forms that can appear on the stack of a shift-reduce parser because they do not extend past the rightmost handle.

A configuration of an LR parser is a pair whose first component is the stack contents and whose second component is the unexpanded input:

$(s_0 X_1 s_1 X_2 s_2 \dots X_m s_m, a_i a_{i+1} \dots a_n \$)$

This configuration represents the right-sentential form

$X_1 X_2 \dots X_m a_i a_{i+1} \dots a_n$

in essentially the same way a shift-reduce parser would; only the presence of the states on the stack is new. Recall the sample parse we did (see Example 1: Sample bottom-up parse) in which we assembled the right-sentential form by concatenating the remainder of the input buffer to the top of the stack. The next move of the parser is determined by reading  $a_i$  and  $s_m$ , and consulting the parsing action table entry  $\text{action}[s_m, a_i]$ . Note that we are just looking at the state here and no symbol below it. We'll see how this actually works later.

The configurations resulting after each of the four types of move are as follows:

If  $\text{action}[s_m, a_i] = \text{shift } s$ , the parser executes a shift move entering the configuration

$(s_0 X_1 s_1 X_2 s_2 \dots X_m s_m a_i s, a_{i+1} \dots a_n \$)$

Here the parser has shifted both the current input symbol  $a_i$  and the next symbol.

If  $\text{action}[s_m, a_i] = \text{reduce } A \rightarrow b$ , then the parser executes a reduce move, entering the configuration,

$(s_0 X_1 s_1 X_2 s_2 \dots X_m s_m A s, a_{i+1} \dots a_n \$)$

where  $s = \text{goto}[s_m, A]$  and  $r$  is the length of  $b$ , the right side of the production. The parser first popped  $2r$  symbols off the stack ( $r$  state symbols and  $r$  grammar symbols), exposing state  $s_m$ . The parser then pushed both  $A$ , the left side of the production, and  $s$ , the entry for  $\text{goto}[s_m, A]$ , onto the stack. The current input symbol is not changed in a reduce move.

The output of an LR parser is generated after a reduce move by executing the semantic action associated with the reducing production. For example, we might just print out the production

reduced.

If  $\text{action}[\text{sm}, \text{ai}] = \text{accept}$ , parsing is completed.

### 3.9 SHIFT REDUCE PARSING

A shift-reduce parser uses a parse stack which (conceptually) contains grammar symbols.

During the operation of the parser, symbols from the input are shifted onto the stack. If a prefix of the symbols on top of the stack matches the RHS of a grammar rule which is the correct rule to use within the current context, then the parser reduces the RHS of the rule to its LHS, replacing the RHS symbols on top of the stack with the nonterminal occurring on the LHS of the rule. This shift-reduce process continues until the parser terminates, reporting either success or failure. It terminates with success when the input is legal and is accepted by the parser. It terminates with failure if an error is detected in the input. The parser is nothing but a stack automaton which may be in one of several discrete states. A state is usually represented simply as an integer. In reality, the parse stack contains states, rather than grammar symbols. However, since each state corresponds to a unique grammar symbol, the state stack can be mapped onto the grammar symbol stack mentioned earlier.

The operation of the parser is controlled by a couple of tables:

#### ACTION TABLE

The action table is a table with rows indexed by states and columns indexed by terminal symbols. When the parser is in some state  $s$  and the current lookahead terminal is  $t$ , the action taken by the parser depends on the contents of  $\text{action}[s][t]$ , which can contain four different kinds of entries:

*Shift s'*

*Shift state s' onto the parse stack.*

*Reduce r*

*Reduce by rule r. This is explained in more detail below.*

*Accept*

*Terminate the parse with success, accepting the input.*

*Error*

Signal a parse error

#### GOTO TABLE

The goto table is a table with rows indexed by states and columns indexed by nonterminal

symbols. When the parser is in state  $s$  immediately after reducing by rule  $N$ , then the next state to enter is given by  $\text{goto}[s][N]$ .

The current state of a shift-reduce parser is the state on top of the state stack. The detailed operation of such a parser is as follows:

1. Initialize the parse stack to contain a single state  $s_0$ , where  $s_0$  is the distinguished initial state of the parser.

2. Use the state  $s$  on top of the parse stack and the current lookahead  $t$  to consult the action table entry  $\text{action}[s][t]$ :

- If the action table entry is shift  $s'$  then push state  $s'$  onto the stack and advance the input so that the lookahead is set to the next token.

- If the action table entry is reduce  $r$  and rule  $r$  has  $m$  symbols in its RHS, then pop  $m$  symbols off the parse stack. Let  $s'$  be the state now revealed on top of the parse stack and  $N$  be the LHS nonterminal for rule  $r$ . Then consult the goto table and push the state given by  $\text{goto}[s'][N]$  onto the stack. The lookahead token is not changed by this step.

- If the action table entry is accept, then terminate the parse with success.
- If the action table entry is error, then signal an error.

3. Repeat step (2) until the parser terminates.

For example, consider the following simple grammar

0)  $\$S: \text{stmt} <\text{EOF}>$

1)  $\text{stmt}: \text{ID} ':=' \text{expr}$

2)  $\text{expr}: \text{expr} '+' \text{ID}$

3)  $\text{expr}: \text{expr} '-' \text{ID}$

4)  $\text{expr}: \text{ID}$

which describes assignment statements like  $a := b + c - d$ . (Rule 0 is a special augmenting production added to the grammar).

One possible set of shift-reduce parsing tables is shown below (sn denotes shift n, rn denotes reduce n, acc denotes accept and blank entries denote error entries):

Parser Tables

Parser Tables

		Action Table				Goto Table		
		ID	' := '	' + '	' - '	<EOF>	stmt	expr
0	s1						g2	
1		s3						
2					s4			
3	s5	I						g6
4	acc	acc	acc	acc	acc			
5	r4	r4	r4	r4	r4			
6	r1	r1	s7	s8	r1			
7	s9							
8	s10							
9	r2	r2	r2	r2	r2			
10	r3	r3	r3	r3	r3			

A trace of the parser on the input  $a := b + c - d$  is shown below:

Stack	Remaining Input	Action
0/\$\$	$a := b + c - d$	s1
0/\$\$ 1/a	$= b + c - d$	s3
0/\$\$ 1/a 3/	$= b + c - d$	s5
0/\$\$ 1/a 3/	$= 5/b + c - d$	r4
0/\$\$ 1/a 3/	$= + c - d$	g6 on expr
0/\$\$ 1/a 3/	$= 6/expr + c - d$	s7
0/\$\$ 1/a 3/	$= 6/expr 7/+ c - d$	s9
0/\$\$ 1/a 3/	$= 6/expr 7/+ 9/c - d$	r2
0/\$\$ 1/a 3/	$= - d$	g6 on expr
0/\$\$ 1/a 3/	$= 6/expr - d$	s8
0/\$\$ 1/a 3/	$= 6/expr 8/- d$	s10
0/\$\$ 1/a 3/	$= 6/expr 8/- 10/d <EOF>$	r3
0/\$\$ 1/a 3/	$= <EOF>$	g6 on expr
0/\$\$ 1/a 3/	$= 6/expr <EOF>$	r1
0/\$\$	$<EOF>$	g2 on stmt
0/\$\$ 2/stmt	$<EOF>$	s4
0/\$\$ 2/stmt 4/	$<EOF>$	accept

Each stack entry is shown as a state number followed by the symbol which caused the transition to that state.

### 3.10 SLR PARSER

An  $LR(0)$  item (or just *item*) of a grammar  $G$  is a production of  $G$  with a dot at some position of the right side indicating how much of a production we have seen up to a given point.

For example, for the production  $E \rightarrow E + T$  we would have the following items:

[ $E \rightarrow .E + T$ ]

[ $E \rightarrow E. + T$ ]

[ $E \rightarrow E + .T$ ]

[ $E \rightarrow E + T.$ ]

Stack	State	Comments
Empty	$[E' \rightarrow .E]$	can't go anywhere from here
	e-transition	so we follow an e-transition
Empty	$[F \rightarrow .(E)]$	now we can shift the (
(	$[F \rightarrow (.E)]$	building the handle (E); This state says: "I have ( on the stack and expect the input to give me tokens that can eventually be reduced to give me the rest of the handle, E.)"

## CONSTRUCTING THE SLR PARSING TABLE

To construct the parser table we must convert our NFA into a DFA. The states in the LR table will be the e-closures of the states corresponding to the items SO...the process of creating the LR state table parallels the process of constructing an equivalent DFA from a machine with e-transitions. Been there, done that - this is essentially the subset construction algorithm so we are in familiar territory here.

We need two operations: closure()

and goto().

closure()

If I is a set of items for a grammar  $G$ , then closure(I) is the set of items constructed from I by the two rules: Initially every item in I is added to closure(I)

If  $A \rightarrow a.Bb$  is in closure(I), and  $B \rightarrow g$  is a production, then add the initial item  $[B \rightarrow .g]$  to I, if it is not already there. Apply this rule until no more new items can be added to closure(I).

From our grammar above, if I is the set of one item  $\{[E' \rightarrow .E]\}$ , then closure(I) contains:

I<sub>0</sub>:  $E' \rightarrow .E$

$E \rightarrow .E + T$

$E \rightarrow .T$

$T \rightarrow .T * F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

goto()

goto(I, X), where I is a set of items and X is a grammar symbol, is defined to be the closure of the set of all items  $[A \rightarrow aX.b]$  such that  $[A \rightarrow a.Xb]$  is in I. The idea here is fairly intuitive: if I is the set of items that are valid for some viable prefix g, then goto(I, X) is the set of items that are valid for the viable prefix gX.

## SETS-OF-ITEMS-CONSTRUCTION

To construct the canonical collection of sets of LR(0) items for augmented grammar  $G'$ .

*procedure items( $G'$ )*

*begin*

Department of CSE

UNIT III

```
C := {closure({[S' -> .S]})};  
repeat  
for each set of items in C and each grammar symbol X  
such that goto(I, X) is not empty and not in C do  
add goto(I, X) to C;  
until no more sets of items can be added to C  
end;
```

### **ALGORITHM FOR CONSTRUCTING AN SLR PARSING TABLE**

**Input:** augmented grammar  $G'$

**Output:** SLR parsing table functions action and goto for  $G'$

**Method:**

*Construct  $C = \{I_0, I_1, \dots, I_n\}$  the collection of sets of LR(0) items for  $G'$ .*

*State  $i$  is constructed from  $I_i$ :*

*if  $[A \rightarrow a.ab]$  is in  $I_i$  and  $\text{goto}(I_i, a) = I_j$ , then set  $\text{action}[i, a]$  to "shift  $j$ ". Here  $a$  must be a terminal.*

*if  $[A \rightarrow a.]$  is in  $I_i$ , then set  $\text{action}[i, a]$  to "reduce  $A \rightarrow a$ " for all  $a$  in  $\text{FOLLOW}(A)$ . Here  $A$  may not be  $S'$ .*

*if  $[S' \rightarrow S.]$  is in  $I_i$ , then set  $\text{action}[i, \$]$  to "accept"*

If any conflicting actions are generated by these rules, the grammar is not SLR(1) and the algorithm fails to produce a parser. The goto transitions for state  $i$  are constructed for all nonterminals  $A$  using the rule: If  $\text{goto}(I_i, A) = I_j$ , then  $\text{goto}[i, A] = j$ .

All entries not defined by rules 2 and 3 are made "error".

The initial state of the parser is the one constructed from the set of items containing  $[S' \rightarrow .S]$ .

Let's work an example to get a feel for what is going on,

An Example

- (1)  $E \rightarrow E * B$
- (2)  $E \rightarrow E + B$
- (3)  $E \rightarrow B$
- (4)  $B \rightarrow 0$
- (5)  $B \rightarrow 1$

The Action and Goto Table The two LR(0) parsing tables for this grammar look as follows:

	action					goto	
state	*	+	0	1	\$	E	B
0			s1	s2		3	4
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	s5	s6			acc		
4	r3	r3	r3	r3	r3		
5			s1	s2		7	
6			s1	s2		8	
7	r1	r1	r1	r1	r1		
8	r2	r2	r2	r2	r2		

### 3.11 LALR PARSER:

We begin with two observations. First, some of the states generated for LR(1) parsing have the same set of core (or first) components and differ only in their second component, the lookahead symbol. Our intuition is that we should be able to merge these states and reduce the number of states we have, getting close to the number of states that would be generated for LR(0) parsing. This observation suggests a hybrid approach: We can construct the canonical LR(1) sets of items and then look for sets of items having the same core. We merge these sets with common cores into one set of items. The merging of states with common cores can never produce a shift/reduce conflict that was not present in one of the original states because shift actions depend only on the core, not the lookahead. But it is possible for the merger to produce a reduce/reduce conflict.

Our second observation is that we are really only interested in the lookahead symbol in places where there is a problem. So our next thought is to take the LR(0) set of items and add lookahead only where they are needed. This leads to a more efficient, but much more complicated method.

## ALGORITHM FOR EASY CONSTRUCTION OF AN LALR TABLE

Input:  $G'$

Output: LALR parsing table functions with action and goto for  $G'$ .

Method:

1. Construct  $C = \{I_0, I_1, \dots, I_n\}$  the collection of sets of LR(1) items for  $G'$ .
2. For each core present among the set of LR(1) items, find all sets having that core and replace these sets by the union.
3. Let  $C' = \{J_0, J_1, \dots, J_m\}$  be the resulting sets of LR(1) items. The parsing actions for state  $i$  are constructed from  $J_i$  in the same manner as in the construction of the canonical LR parsing table.
4. If there is a conflict, the grammar is not LALR(1) and the algorithm fails.
5. The goto table is constructed as follows: If  $J$  is the union of one or more sets of LR(1) items, that is,  $J = I_0 \cup I_1 \cup \dots \cup I_k$ , then the cores of  $\text{goto}(I_0, X)$ ,  $\text{goto}(I_1, X)$ , ...,  $\text{goto}(I_k, X)$  are the same, since  $I_0, I_1, \dots, I_k$  all have the same core. Let  $K$  be the union of all sets of items having the same core as  $\text{goto}(I_1, X)$ .
6. Then  $\text{goto}(J, X) = K$ .

Consider the above example,

$I_3$  &  $I_6$  can be replaced by their union

$I_{36}: C \rightarrow c.C, c/d/\$$

$C \rightarrow .Cc, C/D/\$$

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

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

$I_{89}: C \rightarrow Cc., c/d/\$$

Parsing Table

state	c	d	\$	S	C
0	S36	S47		1	2
1			Accept		
2	S36	S47			5

36	S36	S47			89
47	R3	R3			
5			R1		
89	R2	R2	R2		

## HANDLING ERRORS

The LALR parser may continue to do reductions after the LR parser would have spotted an error, but the LALR parser will never do a shift after the point the LR parser would have discovered the error and will eventually find the error.

## 3.12 LR ERROR RECOVERY

An LR parser will detect an error when it consults the parsing action table and find a blank or error entry. Errors are never detected by consulting the goto table. An LR parser will detect an error as soon as there is no valid continuation for the portion of the input thus far scanned. A canonical LR parser will not make even a single reduction before announcing the error. SLR and LALR parsers may make several reductions before detecting an error, but they will never shift an erroneous input symbol onto the stack.

### 3.12.1 PANIC-MODE ERROR RECOVERY

We can implement panic-mode error recovery by scanning down the stack until a state  $s$  with a goto on a particular nonterminal  $A$  is found. Zero or more input symbols are then discarded until a symbol  $a$  is found that can legitimately follow  $A$ . The situation might exist where there is more than one choice for the nonterminal  $A$ . Normally these would be nonterminals representing major program pieces, e.g. an expression, a statement, or a block. For example, if  $A$  is the nonterminal `stmt`,  $a$  might be `semicolon` or `}`, which marks the end of a statement sequence. This method of error recovery attempts to eliminate the phrase containing the syntactic error. The parser determines that a string derivable from  $A$  contains an error. Part of that string has already been processed, and the result of this processing is a sequence of states on top of the stack. The remainder of the string is still in the input, and the parser attempts to skip over the remainder of this string by looking for a symbol on the input that can legitimately follow  $A$ . By removing states from the stack, skipping over the input, and pushing  $\text{GOTO}(s, A)$  on the stack, the parser pretends that it has found an instance of  $A$  and resumes normal parsing.

### **3.12.2 PHRASE-LEVEL RECOVERY**

Phrase-level recovery is implemented by examining each error entry in the LR action table and deciding on the basis of language usage the most likely programmer error that would give rise to that error. An appropriate recovery procedure can then be constructed; presumably the top of the stack and/or first input symbol would be modified in a way deemed appropriate for each error entry. In designing specific error-handling routines for an LR parser, we can fill in each blank entry in the action field with a pointer to an error routine that will take the appropriate action selected by the compiler designer.

The actions may include insertion or deletion of symbols from the stack or the input or both, or alteration and transposition of input symbols. We must make our choices so that the LR parser will not get into an infinite loop. A safe strategy will assure that at least one input symbol will be removed or shifted eventually, or that the stack will eventually shrink if the end of the input has been reached. Popping a stack state that covers a non terminal should be avoided, because this modification eliminates from the stack a construct that has already been successfully parsed.

## UNIT IV- SYNTAX DIRECTED TRANSLATION & RUN TIME ENVIRONMENT

### SEMANTIC ANALYSIS

- Semantic Analysis computes additional information related to the meaning of the program once the syntactic structure is known.
- In typed languages as C, semantic analysis involves adding information to the symbol table and performing type checking.
- The information to be computed is beyond the capabilities of standard parsing techniques, therefore it is not regarded as syntax.
- As for Lexical and Syntax analysis, also for Semantic Analysis we need both a Representation Formalism and an Implementation Mechanism.
- As representation formalism this lecture illustrates what are called Syntax Directed Translations.

### SYNTAX DIRECTED TRANSLATION

- The Principle of Syntax Directed Translation states that the meaning of an input sentence is related to its syntactic structure, i.e., to its Parse-Tree.
- By Syntax Directed Translations we indicate those formalisms for specifying translations for programming language constructs guided by context-free grammars.
  - We associate Attributes to the grammar symbols representing the language constructs.
  - Values for attributes are computed by Semantic Rules associated with grammar productions.
- Evaluation of Semantic Rules may:
  - Generate Code;
  - Insert information into the Symbol Table;
  - Perform Semantic Check;
  - Issue error messages;
  - etc.

There are two notations for attaching semantic rules:

1. **Syntax Directed Definitions.** High-level specification hiding many implementation details (also called **Attribute Grammars**).
2. **Translation Schemes.** More implementation oriented: Indicate the order in which semantic rules are to be evaluated.

### **Syntax Directed Definitions**

- **Syntax Directed Definitions** are a generalization of context-free grammars in which:
  1. Grammar symbols have an associated set of **Attributes**;
  2. Productions are associated with **Semantic Rules** for computing the values of attributes.
    - Such formalism generates **Annotated Parse-Trees** where each node of the tree is a record with a field for each attribute (e.g.,  $X.a$  indicates the attribute  $a$  of the grammar symbol  $X$ ).
    - The value of an attribute of a grammar symbol at a given parse-tree node is defined by a semantic rule associated with the production used at that node.

We distinguish between two kinds of attributes:

1. **Synthesized Attributes.** They are computed from the values of the attributes of the children nodes.
2. **Inherited Attributes.** They are computed from the values of the attributes of both the siblings and the parent nodes

### **Syntax Directed Definitions: An Example**

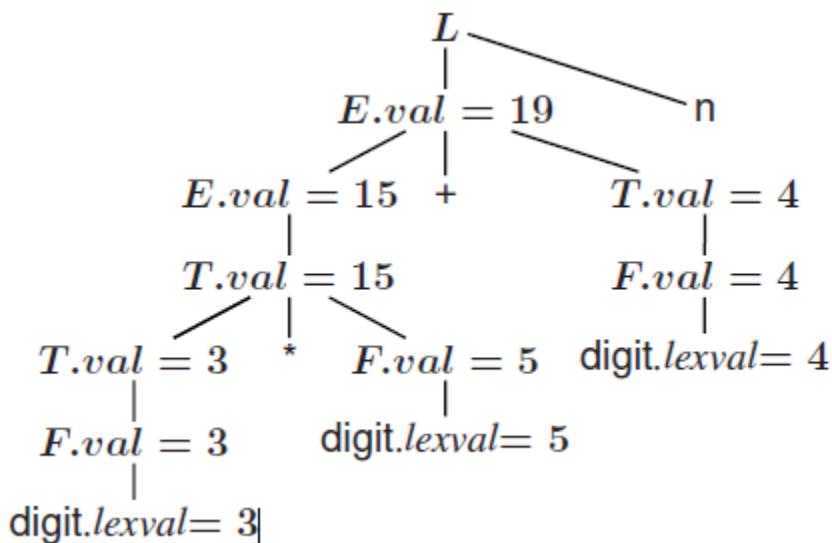
- **Example.** Let us consider the Grammar for arithmetic expressions. The Syntax Directed Definition associates to each non terminal a synthesized attribute called  $val$ .

PRODUCTION	SEMANTIC RULE
$L \rightarrow E \text{n}$	$print(E.\text{val})$
$E \rightarrow E_1 + T$	$E.\text{val} := E_1.\text{val} + T.\text{val}$
$E \rightarrow T$	$E.\text{val} := T.\text{val}$
$T \rightarrow T_1 * F$	$T.\text{val} := T_1.\text{val} * F.\text{val}$
$T \rightarrow F$	$T.\text{val} := F.\text{val}$
$F \rightarrow (E)$	$F.\text{val} := E.\text{val}$
$F \rightarrow \text{digit}$	$F.\text{val} := \text{digit}.lexval$

### S-ATTRIBUTED DEFINITIONS

**Definition.** An **S-Attributed Definition** is a Syntax Directed Definition that uses only synthesized attributes.

- **Evaluation Order.** Semantic rules in a S-Attributed Definition can be evaluated by a bottom-up, or PostOrder, traversal of the parse-tree.
- **Example.** The above arithmetic grammar is an example of an S-Attributed Definition. The annotated parse-tree for the input  $3*5+4\text{n}$  is:



## L-attributed definition

**Definition:** A SDD its *L-attributed* if each inherited attribute of  $X_i$  in the RHS of  $A ! X_1 : \dots : X_n$  depends only on

1. attributes of  $X_1; X_2; \dots; X_{i-1}$  (symbols to the left of  $X_i$  in the RHS)
2. inherited attributes of  $A$ .

## Restrictions for translation schemes:

1. Inherited attribute of  $X_i$  must be computed by an action before  $X_i$ .
2. An action must not refer to synthesized attribute of any symbol to the right of that action.
3. Synthesized attribute for  $A$  can only be computed after all attributes it references have been completed (usually at end of RHS).

## SYMBOL TABLES

A symbol table is a major data structure used in a compiler. Associates attributes with identifiers used in a program. For instance, a type attribute is usually associated with each identifier. A symbol table is a necessary component Definition (declaration) of identifiers appears once in a program. Use of identifiers may appear in many places of the program text. Identifiers and attributes are entered by the analysis phases. When processing a definition (declaration) of an identifier. In simple languages with only global variables and implicit declarations. The scanner can enter an identifier into a symbol table if it is not already there.

In block-structured languages with scopes and explicit declarations:

- The parser and/or semantic analyzer enter identifiers and corresponding attributes
- Symbol table information is used by the analysis and synthesis phases
- To verify that used identifiers have been defined (declared)
- To verify that expressions and assignments are semantically correct – type checking
- To generate intermediate or target code

### ✓ Symbol Table Interface

The basic operations defined on a symbol table include:

- allocate – to allocate a new empty symbol table
- free – to remove all entries and free the storage of a symbol table
- insert – to insert a name in a symbol table and return a pointer to its entry

- lookup – to search for a name and return a pointer to its entry
- set\_attribute – to associate an attribute with a given entry
- get\_attribute – to get an attribute associated with a given entry

Other operations can be added depending on requirement. For example, a delete operation removes a name previously inserted. Some identifiers become invisible (out of scope) after exiting a block.

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

#### Basic Implementation Techniques

- First consideration is how to insert and lookup names
- Variety of implementation techniques
- Unordered List
- Simplest to implement
- Implemented as an array or a linked list
- Linked list can grow dynamically – alleviates problem of a fixed size array
- Insertion is fast  $O(1)$ , but lookup is slow for large tables –  $O(n)$  on average
- Ordered List
  - If an array is sorted, it can be searched using binary search –  $O(\log_2 n)$
  - Insertion into a sorted array is expensive –  $O(n)$  on average
  - Useful when set of names is known in advance – table of reserved words
  - Binary Search Tree
  - Can grow dynamically
- Insertion and lookup are  $O(\log_2 n)$  on average

## RUNTIME ENVIRONMENT

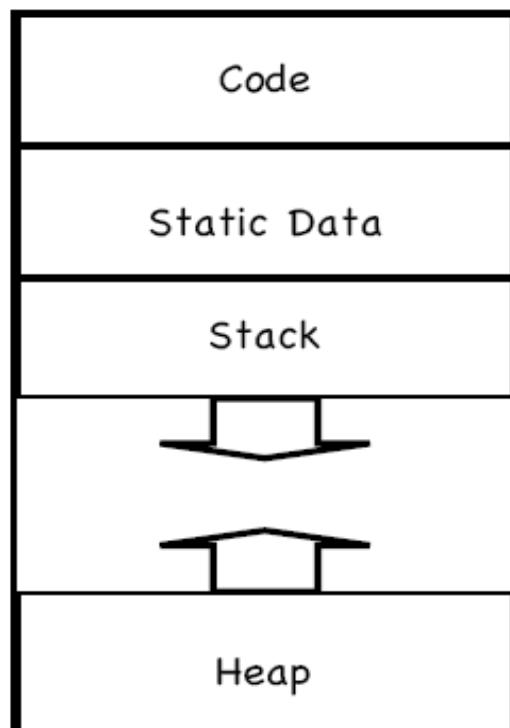
- Runtime organization of different storage locations
- Representation of scopes and extents during program execution.
- Components of executing program reside in blocks of memory (supplied by OS).
- Three kinds of entities that need to be managed at runtime:
  - Generated code for various procedures and programs.
- forms text or code segment of your program: size known at compile time.
  - Data objects:
- Global variables/constants: size known at compile time
- Variables declared within procedures/blocks: size known
- Variables created dynamically: size unknown.
  - Stack to keep track of procedure
- activations. Subdivide memory conceptually into code and data areas:
  - Code:

Program • instructions

- Stack: Manage activation of procedures at runtime.
- Heap: holds variables created dynamically

## STORAGE ORGANIZATION

1. *Fixed-size objects can be placed in predefined locations.*



2. Run-time stack and heap The STACK is used to store:

- Procedure activations.
- The status of the machine just before calling a procedure, so that the status can be restored when the called procedure returns.
- The HEAP stores data allocated under program control (e.g. by malloc() in C). Activation records

Any information needed for a single activation of a procedure is stored in the ACTIVATION RECORD (sometimes called the STACK FRAME). Today, we'll assume the stack grows DOWNWARD, as on, e.g., the Intel architecture. The activation record gets pushed for each procedure call and popped for each procedure return.

## STATIC ALLOCATION

Statically allocated names are bound to storage at compile time. Storage bindings of statically allocated names never change, so even if a name is local to a procedure, its name is always bound to the same storage. The compiler uses the type of a name (retrieved from the symbol table) to determine storage size required. The required number of bytes (possibly aligned) is set aside for the name. The address of the storage is fixed at compile time.

Limitations:

- The size required must be known at compile time.
- Recursive procedures cannot be implemented as all locals are statically allocated.
- No data structure can be created dynamically as all data's static.

```

float f(int k)
{
    float c[10],b;
    b = c[k]*3.14;
    return b;
}

```

Return value	offset = 0
Parameter k	offset = 4
Local c[10]	offset = 8
Local b	offset = 48

### ❖ Stack-dynamic allocation

- ✓ Storage is organized as a stack.
- ✓ Activation records are pushed and popped.
- ✓ Locals and parameters are contained in the activation records for the call.
- ✓ This means locals are bound to fresh storage on every call.
- ✓ If we have a stack growing downwards, we just need a stack\_top pointer.
- ✓ To allocate a new activation record, we just increase stack\_top.
- ✓ To deallocate an existing activation record, we just decrease stack\_top.

### ❖ Address generation in stack allocation

The position of the activation record on the stack cannot be determined statically. Therefore the compiler must generate addresses RELATIVE to the activation record. If we have a downward-growing stack and a stack\_top pointer, we generate addresses of the form stack\_top + offset

## HEAP ALLOCATION

Some languages do not have tree-structured allocations. In these cases, activations have to be allocated on the heap. This allows strange situations, like callee activations that live longer than their callers' activations. This is not common. Heap is used for allocating space for objects created at run time. For example: nodes of dynamic data structures such as linked lists and trees.

Dynamic memory allocation and deallocation based on the requirements of the program `malloc()` and `free()` in C programs

`new()` and `delete()` in C++ programs

`new()` and garbage collection in Java programs

Allocation and deallocation may be *completely manual* (C/C++), *semi-automatic* (Java), or *fully automatic* (Lisp).

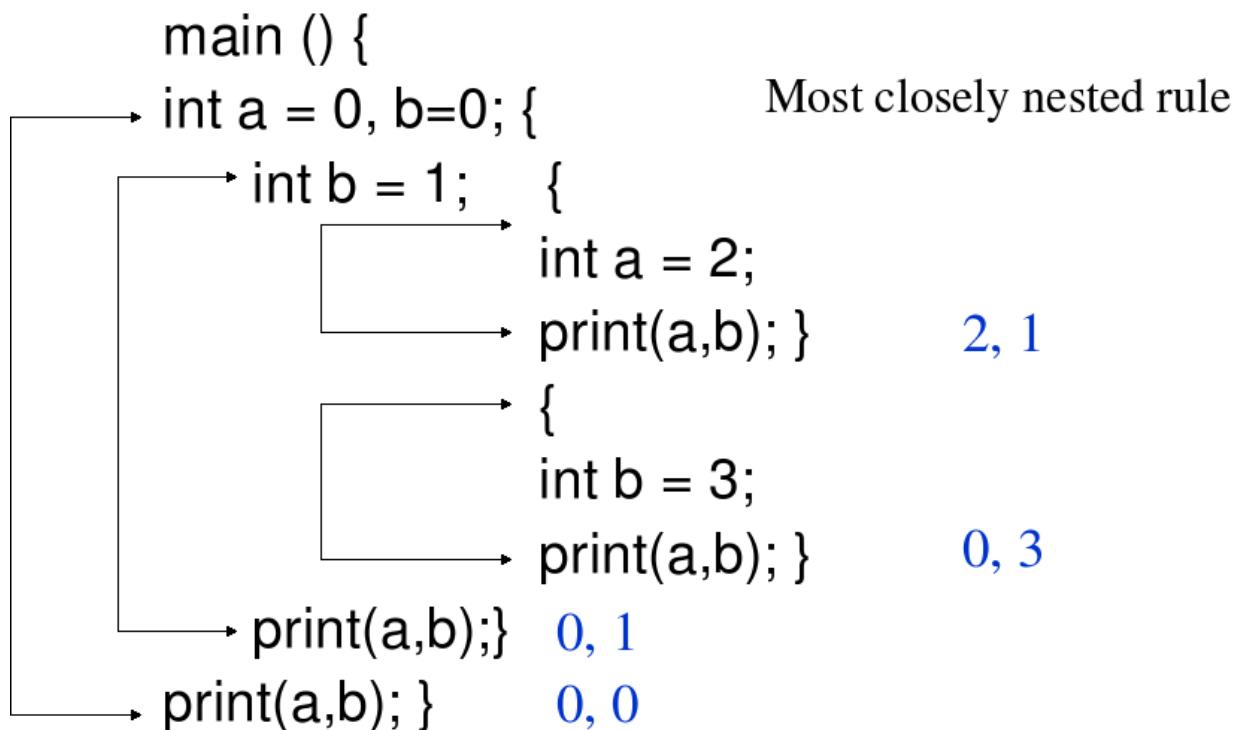
## PARAMETERS PASSING

A language has first-class functions if functions can be declared within any scope passed as arguments to other functions returned as results of functions. In a language with first-class functions and static scope, a function value is generally represented by a closure, a pair consisting of a pointer to function code and a pointer to an activation record. Passing functions as arguments is very useful in structuring of systems using upcalls.

An example:

```
main()
{
    int
    x =
    4;
    int f
    (int
    y) {
    return
    n
    x*y;
}
int g (int →int h){
    int x = 7;
```

```
return h(3) + x;
}
```



### Call-by-Value

The actual parameters are evaluated and their r-values are passed to the called procedure

A procedure called by value can affect its caller either through nonlocal names or through pointers.

Parameters in C are always passed by value. Array is unusual, what is passed by value is a pointer.

Pascal uses pass by value by default, but var parameters are passed by reference.

### Call-by-Reference

Also known as call-by-address or call-by-location. The caller passes to the called procedure the l-value of the parameter.

If the parameter is an expression, then the expression is evaluated in a new location, and the address of the new location is passed.

Parameters in Fortran are passed by reference an old implementation bug in Fortran

```
func(a,b) { a = b };  
call func(3,4); print(3);
```

### **Copy-Restore**

A hybrid between call-by-value and call-by reference.

The actual parameters are evaluated and their r-values are passed as in call- by-value. In addition, l values are determined before the call.

When control returns, the current r-values of the formal parameters are copied back into the l-values of the actual parameters.

### **Call-by-Name**

The actual parameters literally substituted for the formals. This is like a macro-expansion or in-line expansion Call-by-name is not used in practice. However, the conceptually related technique of in-line expansion is commonly used. In-lining may be one of the most effective optimization transformations if they are guided by execution profiles.

## UNIT V - CODE OPTIMIZATION AND CODE GENERATION

### 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.

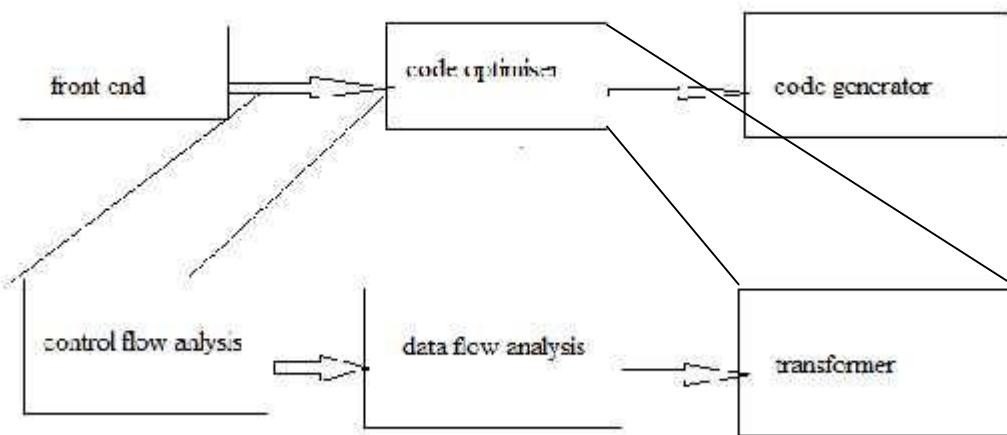
#### **Machine dependant optimizations:**

- 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.

- 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<sub>4</sub>: =4\*i is eliminated as its computation is already in t<sub>1</sub>. 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

**Get useful study materials from [www.rejinpaul.com](http://www.rejinpaul.com)**

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  
 $a=1.570$  thereby 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 an expensive operation by a cheaper one, such as a multiplication by an addition.

➤ **Code Motion:**

- An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed ( a loop-invariant computation) and places the expression before the loop. Note that the notion “before the loop” assumes the existence of an entry for the loop. For example, evaluation of limit-2 is a loop-invariant computation in the following while-statement:

```
while (i <= limit-2) /* statement does not change limit*/
```

Code motion will result in the equivalent of

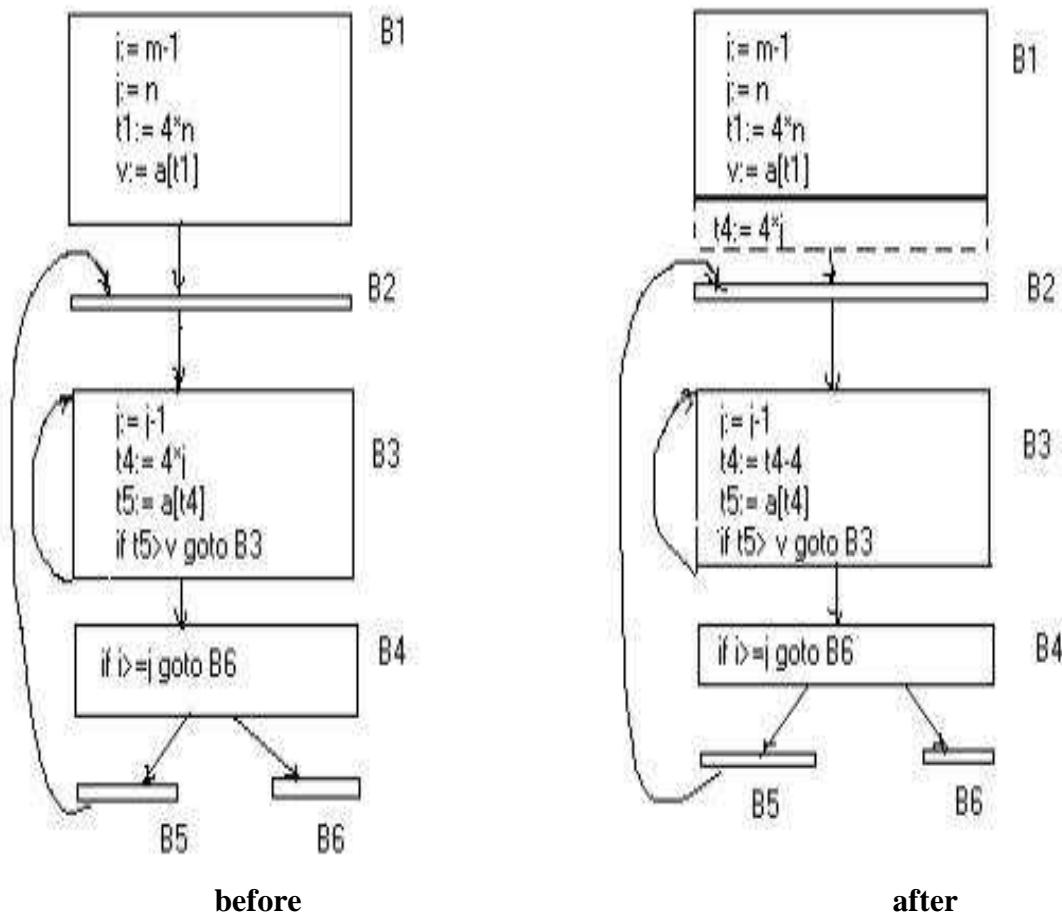
```
t= limit-2;
while (i<=t) /* statement does not change limit or t */
```

➤ **Induction Variables :**

- Loops are usually processed inside out. For example consider the loop around B3.
- Note that the values of  $j$  and  $t_4$  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
```

➤ **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  $\leq$ ,  $\geq$ ,  $<$ ,  $>$ ,  $+$  and  $=$  sometimes generate unexpected common sub expressions.
- Associative laws may also be applied to expose common sub expressions. For example, if the source code has the assignments

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

the following intermediate code may be generated:

$a := b + c$   
 $t := c + d$   
 $e := t + b$

- Example:

$x := x + 0$  can be removed

$x := y^{**} 2$  can be replaced by a cheaper statement  $x := y * y$

- The compiler writer should examine the language carefully to determine 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

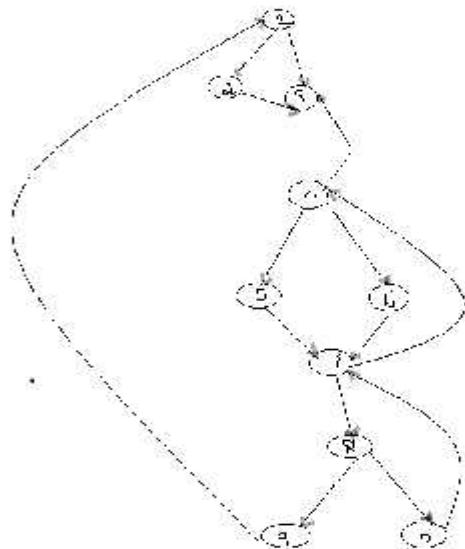
A graph representation of three-address statements, called a **flow graph**, is useful for understanding code-generation algorithms, even if the graph is not explicitly constructed by a code-generation algorithm. Nodes in the flow graph represent computations, and the edges represent the flow of control.

### Dominators:

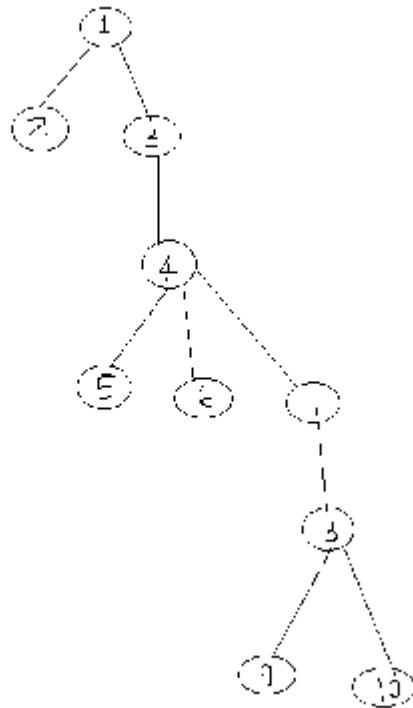
In a flow graph, a node  $d$  dominates node  $n$ , if every path from initial node of the flow graph to  $n$  goes through  $d$ . This will be denoted by  $d \text{ dom } n$ . Every initial node dominates all the remaining nodes in the flow graph and the entry of a loop dominates all nodes in the loop. 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 going through the other.
- \*node 7 dominates 7,8 ,9 and 10.
- \*node 8 dominates 8,9 and 10.
- \*node 9 and 10 dominates only themselves.



- The way of presenting dominator information is in a tree, called the dominator tree in which the initial node is the root.
- The parent of each other node is its immediate dominator.
- Each node  $d$  dominates only its descendants in the tree.
- The existence of dominator tree follows from a property of dominators; each node has a unique immediate dominator in that is the last dominator of  $n$  on any path from the initial node to  $n$ .
- In terms of the  $\text{dom}$  relation, the immediate dominator  $m$  has the property is  $d = !n$  and  $d \text{ dom } n$ , then  $d \text{ dom } m$ .



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

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

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

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

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

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

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

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

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

### Natural Loop:

- One application of dominator information is in determining the loops of a flow graph suitable for improvement.
- The properties of loops are
  - ✓ A loop must have a single entry point, called the header. This entry point-dominates all nodes in the loop, or it would not be the sole entry to the loop.
  - ✓ There must be at least one way to iterate the loop(i.e.)at least one path back to the header.
- One way to find all the loops in a flow graph is to search for edges in the flow graph whose heads dominate their tails. If  $a \rightarrow b$  is an edge, b is the head and a is the tail. These types of edges are called as back edges.
- ✓ Example:

In the above graph,

$7 \rightarrow 4$	4 DOM 7
$10 \rightarrow 7$	7 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  $m^*d$  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;

```

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

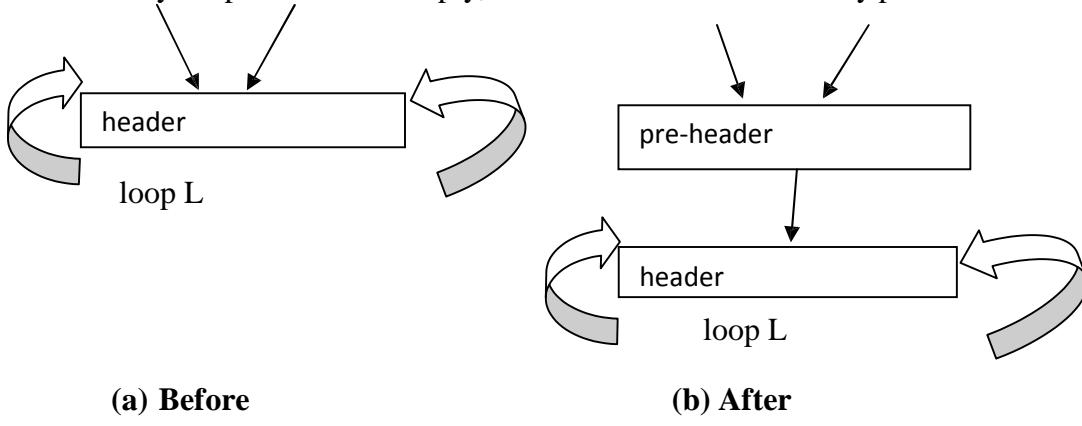
```

### 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 disjointed or one is entirely contained in the other. Thus, neglecting loops with the same header for the moment, we have a natural notion of inner loop: one that contains no other loop.
- When two natural loops have the same header, but neither is nested within the other, they are combined and treated as a single loop.

### Pre-Headers:

- Several transformations require us to move statements “before the header”. Therefore begin treatment of a loop L by creating a new block, called the preheader.
- The pre-header has only the header as successor, and all edges which formerly entered the header of L from outside L instead enter the pre-header.
- Edges from inside loop L to the header are not changed.
- Initially the pre-header is empty, but transformations on L may place statements in it.



### Reducible flow graphs:

- Reducible flow graphs are special flow graphs, for which several code optimization transformations are especially easy to perform, loops are unambiguously defined, dominators can be easily calculated, data flow analysis problems can also be solved efficiently.
- Exclusive use of structured flow-of-control statements such as if-then-else, while-do, continue, and break statements produces programs whose flow graphs are always reducible.

- The most important properties of reducible flow graphs are that there are no jumps into the middle of loops from outside; the only entry to a loop is through its header.
- **Definition:**  
A flow graph G is reducible if and only if we can partition the edges into two disjoint groups, *forward* edges and *back* edges, with the following properties.
  - ✓ The forward edges from an acyclic graph in which every node can be reached from initial node of G.
  - ✓ The back edges consist only of edges where heads dominate theirs 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.

## PEEPHOLE OPTIMIZATION

- A statement-by-statement code-generations strategy often produce target code that contains redundant instructions and suboptimal constructs .The quality of such target code can be improved by applying “optimizing” transformations to the target program.
- A simple but effective technique for improving the target code is peephole optimization, a method for trying to improving the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence, whenever possible.
- The peephole is a small, moving window on the target program. The code in the peephole need not contiguous, although some implementations do require this.it is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.
- We shall give the following examples of program transformations that are characteristic of peephole optimizations:
  - ✓ Redundant-instructions elimination
  - ✓ Flow-of-control optimizations
  - ✓ Algebraic simplifications
  - ✓ Use of machine idioms
  - ✓ Unreachable Code

**Redundant Loads And Stores:**

If we see the instructions sequence

(1) MOV R<sub>0</sub>,a

(2) MOV a,R<sub>0</sub>

we can delete instructions (2) because whenever (2) is executed, (1) will ensure that the value of **a** is already in register R<sub>0</sub>. If (2) had a label we could not be sure that (1) was always executed immediately before (2) and so we could not remove (2).

**Unreachable Code:**

- Another opportunity for peephole optimizations is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable **debug** is 1. In C, the source code might look like:

```
#define debug 0
```

```
....
```

```
If ( debug ) {
```

```
    Print debugging information
```

```
}
```

- In the intermediate representations the if-statement may be translated as:

```
If debug =1 goto L2
```

```
goto L2
```

```
L1: print debugging information
```

```
L2: .....(a)
```

- One obvious peephole optimization is to eliminate jumps over jumps . Thus no matter what the value of **debug**; (a) can be replaced by:

```
If debug ≠1 goto L2
```

```
Print debugging information
```

```
L2: .....(b)
```

- As the argument of the statement of (b) evaluates to a constant **true** it can be replaced by

If debug ≠ 0 goto L2

Print debugging information

L2: .....(c)

- As the argument of the first statement of (c) evaluates to a constant true, it can be replaced by goto L2. Then all the statements that print debugging aids are manifestly unreachable and can be eliminated one at a time.

### **Flows-Of-Control Optimizations:**

- The unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the jump sequence

goto L1

....

L1: gotoL2

by the sequence

goto L2

....

L1: goto L2

- If there are now no jumps to L1, then it may be possible to eliminate the statement L1:goto L2 provided it is preceded by an unconditional jump .Similarly, the sequence

if a < b goto L1

....

L1: goto L2

can be replaced by

If a < b goto L2

....

L1: goto L2

- Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto. Then the sequence

goto L1

.....

L1: if a < b goto L2

L3: .....(1)

- May be replaced by

If a < b goto L2

goto L3

.....

L3: .....(2)

- While the number of instructions in (1) and (2) is the same, we sometimes skip the unconditional jump in (2), but never in (1). Thus (2) is superior to (1) in execution time

### **Algebraic Simplification:**

- There is no end to the amount of algebraic simplification that can be attempted through peephole optimization. Only a few algebraic identities occur frequently enough that it is worth considering implementing them. For example, statements such as

$x := x + 0$

Or

$x := x * 1$

- Are often produced by straightforward intermediate code-generation algorithms, and they can be eliminated easily through peephole optimization.

### **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.

$$X^2 \rightarrow X*X$$

### **Use of Machine Idioms:**

- The target machine may have hardware instructions to implement certain specific operations efficiently. For example, some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value.
- The use of these modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like  $i := i + 1$ .

i:=i+1 → i++  
i:=i-1 → i- -

## INTRODUCTION TO GLOBAL DATAFLOW ANALYSIS

- In order to do code optimization and a good job of code generation , compiler needs to collect information about the program as a whole and to distribute this information to each block in the flow graph.
- A compiler could take advantage of “reaching definitions” , such as knowing where a variable like *debug* was last defined before reaching a given block, in order to perform transformations are just a few examples of data-flow information that an optimizing compiler collects by a process known as data-flow analysis.
- Data-flow information can be collected by setting up and solving systems of equations of the form :

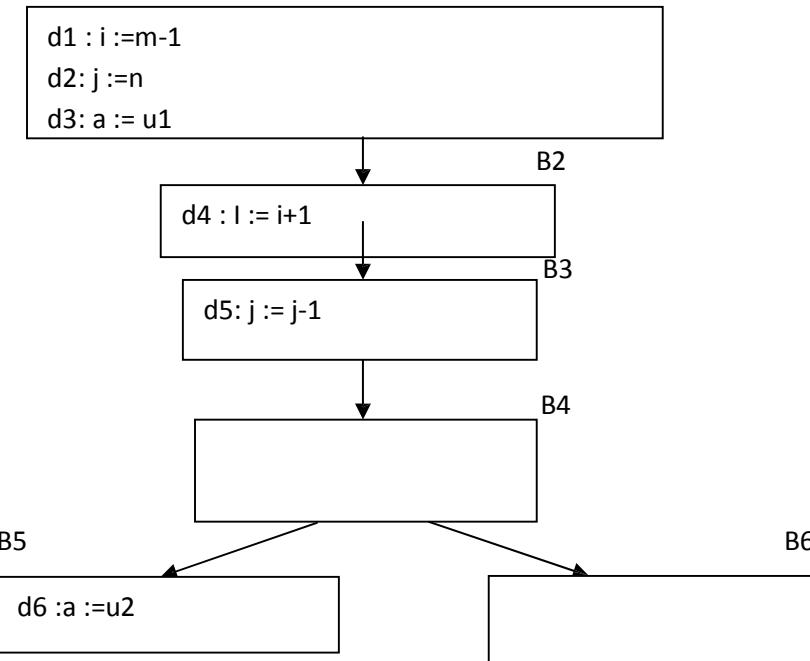
$$\text{out } [S] = \text{gen } [S] \cup (\text{in } [S] - \text{kill } [S])$$

This equation can be read as “ the information at the end of a statement is either generated within the statement , or enters at the beginning and is not killed as control flows through the statement.”

- The details of how data-flow equations are set and solved depend on three factors.
- ✓ The notions of generating and killing depend on the desired information, i.e., on the data flow analysis problem to be solved. Moreover, for some problems, instead of proceeding along with flow of control and defining out[s] in terms of in[s], we need to proceed backwards and define in[s] in terms of out[s].
- ✓ Since data flows along control paths, data-flow analysis is affected by the constructs in a program. In fact, when we write out[s] we implicitly assume that there is unique end point where control leaves the statement; in general, equations are set up at the level of basic blocks rather than statements, because blocks do have unique end points.
- ✓ There are subtleties that go along with such statements as procedure calls, assignments through pointer variables, and even assignments to array variables.

### Points and Paths:

- Within a basic block, we talk of the point between two adjacent statements, as well as the point before the first statement and after the last. Thus, block B1 has four points: one before any of the assignments and one after each of the three assignments.



- Now let us take a global view and consider all the points in all the blocks. A path from  $p_1$  to  $p_n$  is a sequence of points  $p_1, p_2, \dots, p_n$  such that for each  $i$  between 1 and  $n-1$ , either
- $p_i$  is the point immediately preceding a statement and  $p_{i+1}$  is the point immediately following that statement in the same block, or
- $p_i$  is the end of some block and  $p_{i+1}$  is the beginning of a successor block.

#### Reaching definitions:

- A definition of variable  $x$  is a statement that assigns, or may assign, a value to  $x$ . The most common forms of definition are assignments to  $x$  and statements that read a value from an i/o device and store it in  $x$ .
- These statements certainly define a value for  $x$ , and they are referred to as **unambiguous** definitions of  $x$ . There are certain kinds of statements that may define a value for  $x$ ; they are called **ambiguous** definitions. The most usual forms of **ambiguous** definitions of  $x$  are:
  - A call of a procedure with  $x$  as a parameter or a procedure that can access  $x$  because  $x$  is in the scope of the procedure.
  - An assignment through a pointer that could refer to  $x$ . For example, the assignment  $*q := y$  is a definition of  $x$  if it is possible that  $q$  points to  $x$ . we must assume that an assignment through a pointer is a definition of every variable.
- We say a definition  $d$  reaches a point  $p$  if there is a path from the point immediately following  $d$  to  $p$ , such that  $d$  is not “killed” along that path. Thus a point can be reached

by an unambiguous definition and an ambiguous definition of the same variable appearing later along one path.

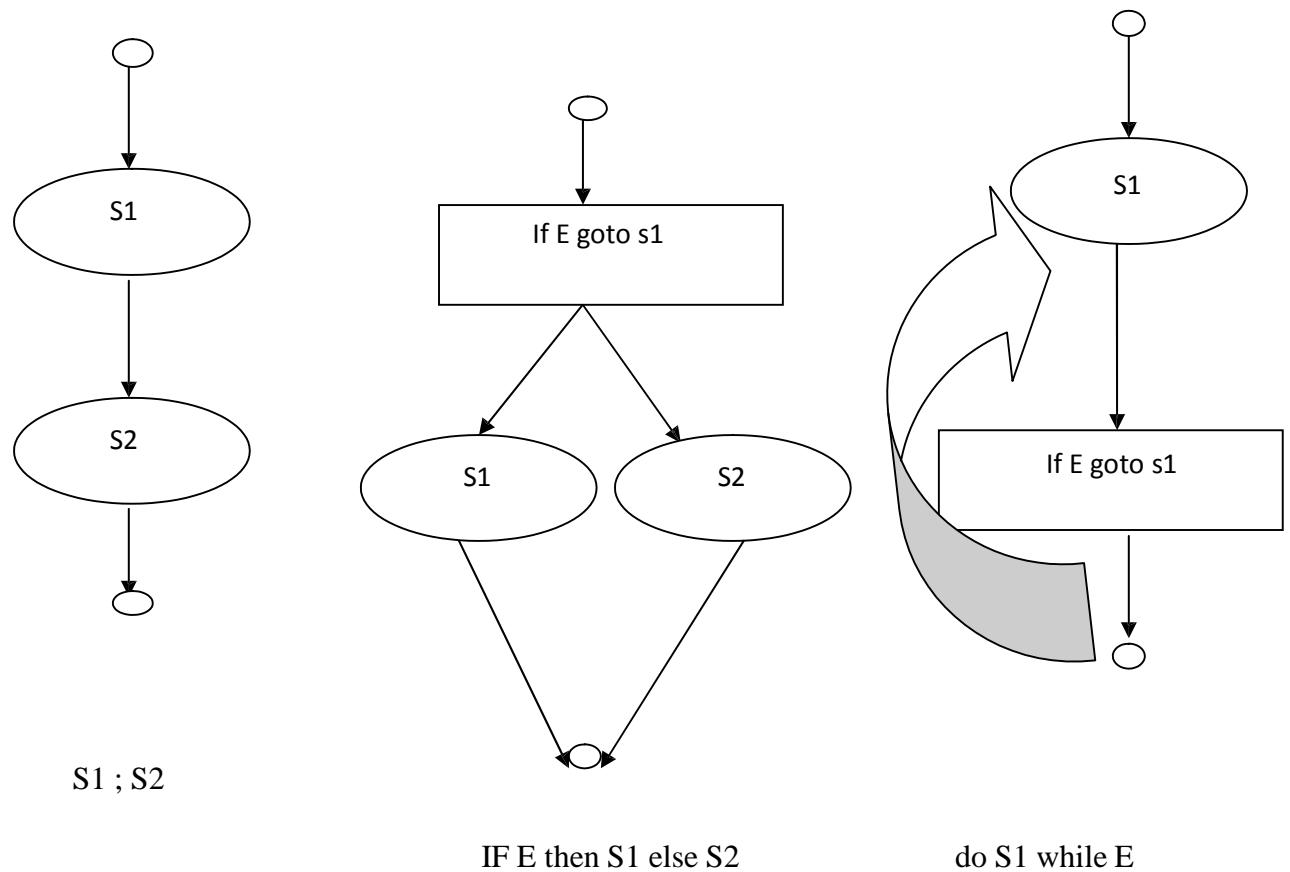
### Data-flow analysis of structured programs:

- Flow graphs for control flow constructs such as do-while statements have a useful property: there is a single beginning point at which control enters and a single end point that control leaves from when execution of the statement is over. We exploit this property when we talk of the definitions reaching the beginning and the end of statements with the following syntax.

$S \rightarrow id := E | S; S | \text{if } E \text{ then } S \text{ else } S | \text{do } S \text{ while } E$

$E \rightarrow id + id | id$

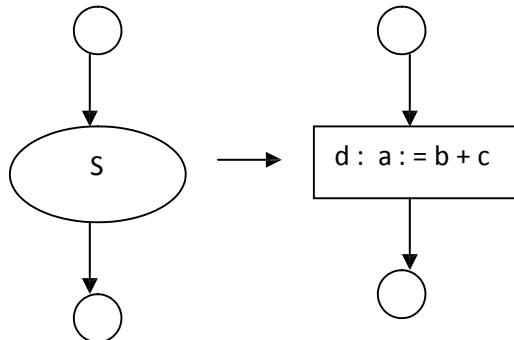
- Expressions in this language are similar to those in the intermediate code, but the flow graphs for statements have restricted forms.



- We define a portion of a flow graph called a *region* to be a set of nodes N that includes a header, which dominates all other nodes in the region. All edges between nodes in N are in the region, except for some that enter the header.
- The portion of flow graph corresponding to a statement S is a region that obeys the further restriction that control can flow to just one outside block when it leaves the region.

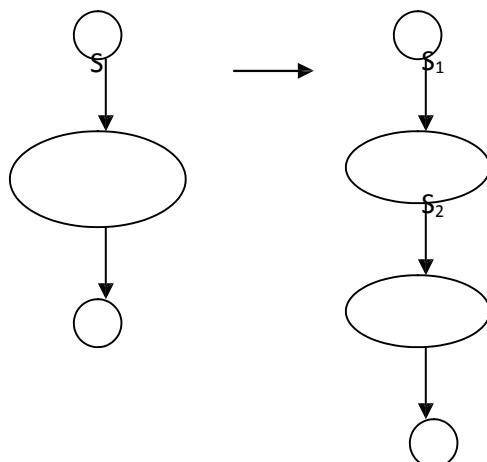
- We say that the beginning points of the dummy blocks at the entry and exit of a statement's region are the beginning and end points, respectively, of the statement. The equations are inductive, or syntax-directed, definition of the sets  $\text{in}[S]$ ,  $\text{out}[S]$ ,  $\text{gen}[S]$ , and  $\text{kill}[S]$  for all statements  $S$ .
- **gen[S] is the set of definitions “generated” by S while kill[S] is the set of definitions that never reach the end of S.**
- Consider the following data-flow equations for reaching definitions :

i )



$$\begin{aligned}\text{gen } [S] &= \{ d \} \\ \text{kill } [S] &= D_a - \{ d \} \\ \text{out } [S] &= \text{gen } [S] \cup (\text{in } [S] - \text{kill } [S])\end{aligned}$$

- Observe the rules for a single assignment of variable  $a$ . Surely that assignment is a definition of  $a$ , say  $d$ . Thus  
 $\text{Gen}[S]=\{d\}$
- On the other hand,  $d$  “kills” all other definitions of  $a$ , so we write  
 $\text{Kill}[S] = D_a - \{d\}$   
Where,  $D_a$  is the set of all definitions in the program for variable  $a$ . ii )



$$\begin{aligned}\text{gen } [S] &= \text{gen } [S_2] \cup (\text{gen } [S_1] - \text{kill } [S_2]) \\ \text{Kill } [S] &= \text{kill } [S_2] \cup (\text{kill } [S_1] - \text{gen } [S_2])\end{aligned}$$

$$\begin{aligned}\text{in } [S_1] &= \text{in } [S] \\ \text{in } [S_2] &= \text{out } [S_1] \\ \text{out } [S] &= \text{out } [S_2]\end{aligned}$$

- Under what circumstances is definition d generated by  $S=S_1; S_2$ ? First of all, if it is generated by  $S_2$ , then it is surely generated by S. If d is generated by  $S_1$ , it will reach the end of S provided it is not killed by  $S_2$ . Thus, we write  
 $\text{gen}[S] = \text{gen}[S_2] \cup (\text{gen}[S_1] - \text{kill}[S_2])$
- Similar reasoning applies to the killing of a definition, so we have  
 $\text{Kill}[S] = \text{kill}[S_2] \cup (\text{kill}[S_1] - \text{gen}[S_2])$

### **Conservative estimation of data-flow information:**

- There is a subtle miscalculation in the rules for gen and kill. We have made the assumption that the conditional expression E in the if and do statements are “uninterpreted”; that is, there exists inputs to the program that make their branches go either way.
- We assume that any graph-theoretic path in the flow graph is also an execution path, i.e., a path that is executed when the program is run with least one possible input.
- When we compare the computed gen with the “true” gen we discover that the true gen is always a subset of the computed gen. On the other hand, the true kill is always a superset of the computed kill.
- These containments hold even after we consider the other rules. It is natural to wonder whether these differences between the true and computed gen and kill sets present a serious obstacle to data-flow analysis. The answer lies in the use intended for these data.
- Overestimating the set of definitions reaching a point does not seem serious; it merely stops us from doing an optimization that we could legitimately do. On the other hand, underestimating the set of definitions is a fatal error; it could lead us into making a change in the program that changes what the program computes. For the case of reaching definitions, then, we call a set of definitions safe or conservative if the estimate is a superset of the true set of reaching definitions. We call the estimate unsafe, if it is not necessarily a superset of the truth.
- Returning now to the implications of safety on the estimation of gen and kill for reaching definitions, note that our discrepancies, supersets for gen and subsets for kill are both in the safe direction. Intuitively, increasing gen adds to the set of definitions that can reach a point, and cannot prevent a definition from reaching a place that it truly reached. Decreasing kill can only increase the set of definitions reaching any given point.

### **Computation of in and out:**

- Many data-flow problems can be solved by synthesized translations similar to those used to compute gen and kill. It can be used, for example, to determine loop-invariant computations.
- However, there are other kinds of data-flow information, such as the reaching-definitions problem. It turns out that in is an inherited attribute, and out is a synthesized attribute depending on in. We intend that  $\text{in}[S]$  be the set of definitions reaching the beginning of

S, taking into account the flow of control throughout the entire program, including statements outside of S or within which S is nested.

- The set  $\text{out}[S]$  is defined similarly for the end of s. it is important to note the distinction between  $\text{out}[S]$  and  $\text{gen}[S]$ . The latter is the set of definitions that reach the end of S without following paths outside S.
- Assuming we know  $\text{in}[S]$  we compute  $\text{out}$  by equation, that is

$$\text{Out}[S] = \text{gen}[S] \cup (\text{in}[S] - \text{kill}[S])$$

- Considering cascade of two statements  $S_1; S_2$ , as in the second case. We start by observing  $\text{in}[S_1] = \text{in}[S]$ . Then, we recursively compute  $\text{out}[S_1]$ , which gives us  $\text{in}[S_2]$ , since a definition reaches the beginning of  $S_2$  if and only if it reaches the end of  $S_1$ . Now we can compute  $\text{out}[S_2]$ , and this set is equal to  $\text{out}[S]$ .
- Considering if-statement we have conservatively assumed that control can follow either branch, a definition reaches the beginning of  $S_1$  or  $S_2$  exactly when it reaches the beginning of S.

$$\text{In}[S_1] = \text{in}[S_2] = \text{in}[S]$$

- If a definition reaches the end of S if and only if it reaches the end of one or both sub statements; i.e,

$$\text{Out}[S] = \text{out}[S_1] \cup \text{out}[S_2]$$

### Representation of sets:

- Sets of definitions, such as  $\text{gen}[S]$  and  $\text{kill}[S]$ , can be represented compactly using bit vectors. We assign a number to each definition of interest in the flow graph. Then bit vector representing a set of definitions will have 1 in position I if and only if the definition numbered I is in the set.
- The number of definition statement can be taken as the index of statement in an array holding pointers to statements. However, not all definitions may be of interest during global data-flow analysis. Therefore the number of definitions of interest will typically be recorded in a separate table.
- A bit vector representation for sets also allows set operations to be implemented efficiently. The union and intersection of two sets can be implemented by logical or and logical and, respectively, basic operations in most systems-oriented programming languages. The difference  $A-B$  of sets A and B can be implemented by taking the complement of B and then using logical and to compute A

### **Local reaching definitions:**

- Space for data-flow information can be traded for time, by saving information only at certain points and, as needed, recomputing information at intervening points. Basic blocks are usually treated as a unit during global flow analysis, with attention restricted to only those points that are the beginnings of blocks.
- Since there are usually many more points than blocks, restricting our effort to blocks is a significant savings. When needed, the reaching definitions for all points in a block can be calculated from the reaching definitions for the beginning of a block.

### **Use-definition chains:**

- It is often convenient to store the reaching definition information as "use-definition chains" or "ud-chains", which are lists, for each use of a variable, of all the definitions that reaches that use. If a use of variable  $a$  in block  $B$  is preceded by no unambiguous definition of  $a$ , then ud-chain for that use of  $a$  is the set of definitions in  $\text{in}[B]$  that are definitions of  $a$ . In addition, if there are ambiguous definitions of  $a$ , then all of these for which no unambiguous definition of  $a$  lies between it and the use of  $a$  are on the ud-chain for this use of  $a$ .

### **Evaluation order:**

- The techniques for conserving space during attribute evaluation, also apply to the computation of data-flow information using specifications. Specifically, the only constraint on the evaluation order for the gen, kill, in and out sets for statements is that imposed by dependencies between these sets. Having chosen an evaluation order, we are free to release the space for a set after all uses of it have occurred.
- Earlier circular dependencies between attributes were not allowed, but we have seen that data-flow equations may have circular dependencies.

### **General control flow:**

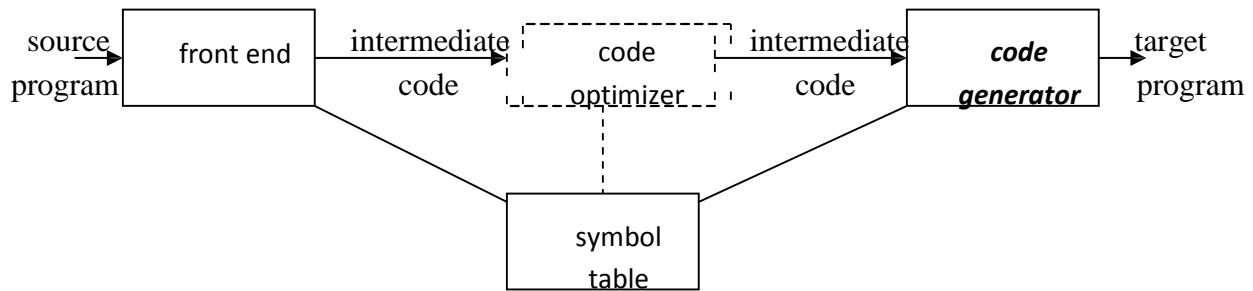
- Data-flow analysis must take all control paths into account. If the control paths are evident from the syntax, then data-flow equations can be set up and solved in a syntax-directed manner.
- When programs can contain goto statements or even the more disciplined break and continue statements, the approach we have taken must be modified to take the actual control paths into account.
- Several approaches may be taken. The iterative method works arbitrary flow graphs. Since the flow graphs obtained in the presence of break and continue statements are reducible, such constraints can be handled systematically using the interval-based methods

- However, the syntax-directed approach need not be abandoned when break and continue statements are allowed.

## 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.

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.

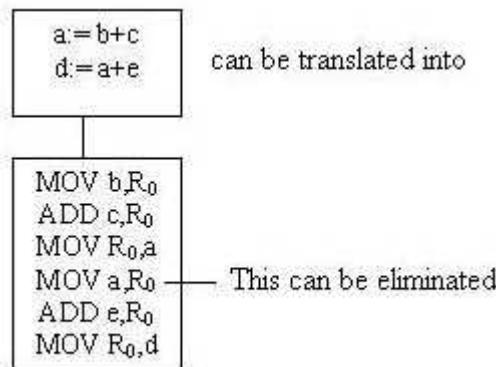
For example,

$j : \text{goto } i$  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
<i>absolute</i>	M	M	1
<i>register</i>	R	R	0
<i>indexed</i>	c(R)	$c + \text{contents}(R)$	1
<i>indirect register</i>	*R	$\text{contents}(R)$	0
<i>indirect indexed</i>	$*c(R)$	$\text{contents}(c + \text{contents}(R))$	1
<i>literal</i>	#c	c	1

- For example :  $\text{MOV } R_0, M$  stores contents of Register  $R_0$  into memory location  $M$  ;  
 $\text{MOV } 4(R_0), M$  stores the value  $\text{contents}(4+\text{contents}(R_0))$  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 :  $\text{MOV } R_0, R_1$  copies the contents of register  $R_0$  into  $R_1$ . 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)  $\text{MOV } b, R_0$   
 $\text{ADD } c, R_0$                           cost = 6  
 $\text{MOV } R_0, a$

ii)  $\text{MOV } b, a$   
 $\text{ADD } c, a$                           cost = 6

iii) Assuming  $R_0, R_1$  and  $R_2$  contain the addresses of  $a, b$ , and  $c$  :

$\text{MOV } *R_1, *R_0$   
 $\text{ADD } *R_2, *R_0$                           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:
  - Static allocation
  - 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:
  - Call,
  - Return,
  - Halt, and
  - Action, a placeholder for other statements.
- We assume that the run-time memory is divided into areas for:
  - Code
  - Static data
  - Stack

**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*

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)	t <sub>1</sub> := 4 * i
(4)	t <sub>2</sub> := a[t <sub>1</sub> ] /*compute a[i] */
(5)	t <sub>3</sub> := 4 * i
(6)	t <sub>4</sub> := b[t <sub>3</sub> ] /*compute b[i] */
(7)	t <sub>5</sub> := t <sub>2</sub> *t <sub>4</sub>
(8)	t <sub>6</sub> := prod+t <sub>5</sub>
(9)	prod := t <sub>6</sub>
(10)	t <sub>7</sub> := i+1
(11)	i := t <sub>7</sub>
(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:**

$$\begin{array}{ll}
 \text{a} := \text{b} + \text{c} & \text{a} := \text{b} + \text{c} \\
 \text{b} := \text{a} - \text{d} & \xrightarrow{\hspace{1cm}} \quad \text{b} := \text{a} - \text{d} \\
 \text{c} := \text{b} + \text{c} & \quad \quad \quad \text{c} := \text{b} + \text{c} \\
 \text{d} := \text{a} - \text{d} & \quad \quad \quad \text{d} := \text{b}
 \end{array}$$

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:

$$\begin{array}{l}
 t_1 := b + c \\
 t_2 := x + y
 \end{array}$$

We can interchange the two statements without affecting the value of the block if and only if neither  $x$  nor  $y$  is  $t_1$  and neither  $b$  nor  $c$  is  $t_2$ .

**2. Algebraic transformations:**

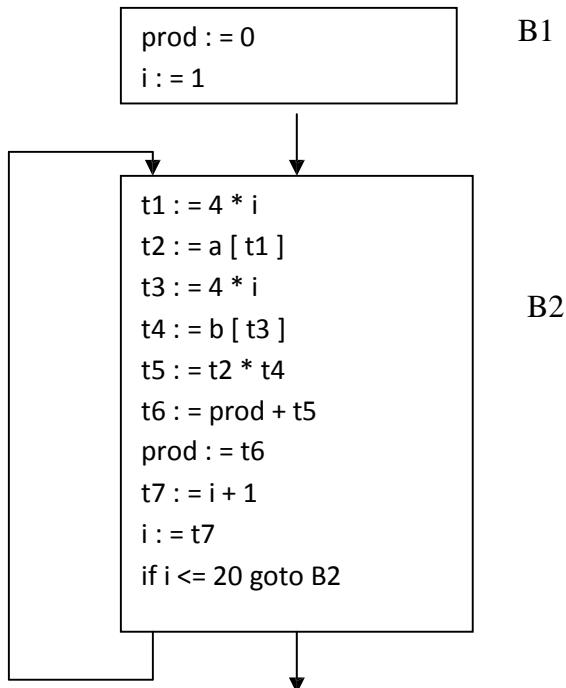
Algebraic transformations can be used to change the set of expressions computed by a basic block into an algebraically equivalent set.

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:



- $B_1$  is the *initial* node.  $B_2$  immediately follows  $B_1$ , so there is an edge from  $B_1$  to  $B_2$ . The target of jump from last statement of  $B_1$  is the first statement  $B_2$ , so there is an edge from  $B_1$  (last statement) to  $B_2$  (first statement).
- $B_1$  is the *predecessor* of  $B_2$ , and  $B_2$  is a *successor* of  $B_1$ .

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	Liveliness	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 R<sub>j</sub>, R<sub>i</sub>                      Cost = 1 // if R<sub>i</sub> contains b and R<sub>j</sub> contains c

(or)

ADD c, R<sub>i</sub>                      Cost = 2 // if c is in a memory location

(or)

MOV c, R<sub>j</sub>                      Cost = 3 // move c from memory to R<sub>j</sub> and add

ADD R<sub>j</sub>, R<sub>i</sub>

### 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.

### 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, R <sub>0</sub> SUB b, R <sub>0</sub>	R <sub>0</sub> contains t	t in R <sub>0</sub>
$u := a - c$	MOV a, R <sub>1</sub> SUB c, R <sub>1</sub>	R <sub>0</sub> contains t R <sub>1</sub> contains u	t in R <sub>0</sub> u in R <sub>1</sub>
$v := t + u$	ADD R <sub>1</sub> , R <sub>0</sub>	R <sub>0</sub> contains v R <sub>1</sub> contains u	u in R <sub>1</sub> v in R <sub>0</sub>
$d := v + u$	ADD R <sub>1</sub> , R <sub>0</sub> MOV R <sub>0</sub> , d	R <sub>0</sub> contains d	d in R <sub>0</sub> d in R <sub>0</sub> and memory

### 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
$\text{if } x < y \text{ goto } z$	CMP x, y CJ< z /* jump to z if condition code is negative */
$x := y + z$ $\text{if } x < 0 \text{ goto } z$	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( $y$ ).

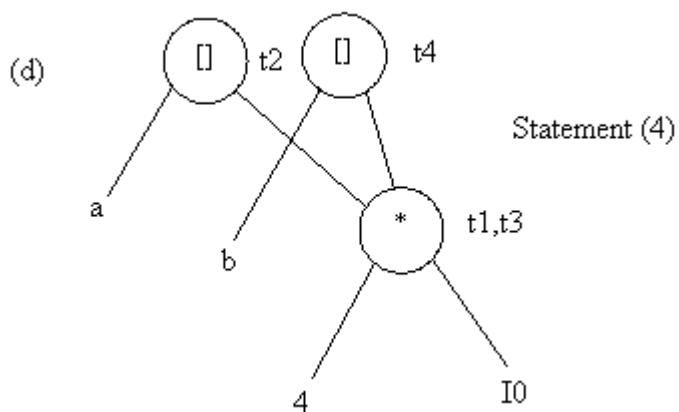
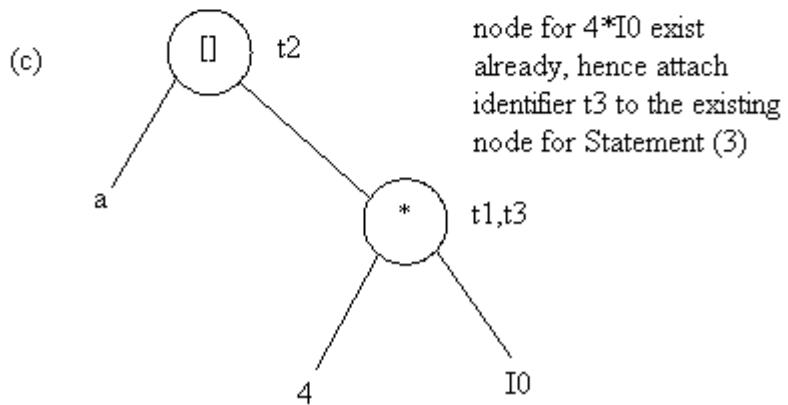
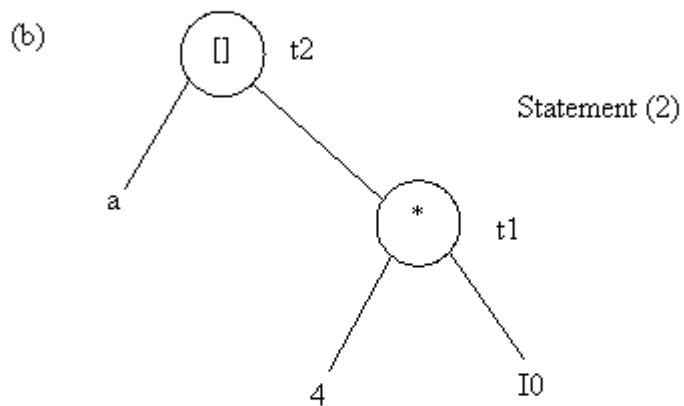
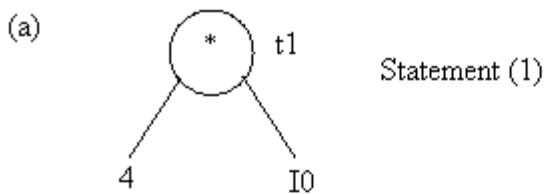
**Step 3:** Delete  $x$  from the list of identifiers for node( $x$ ). Append  $x$  to the list of attached

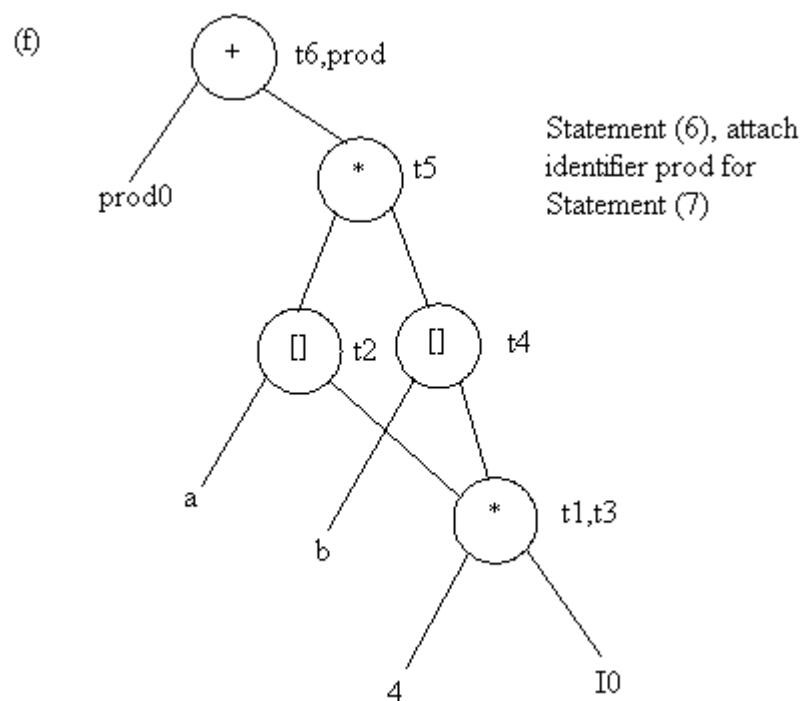
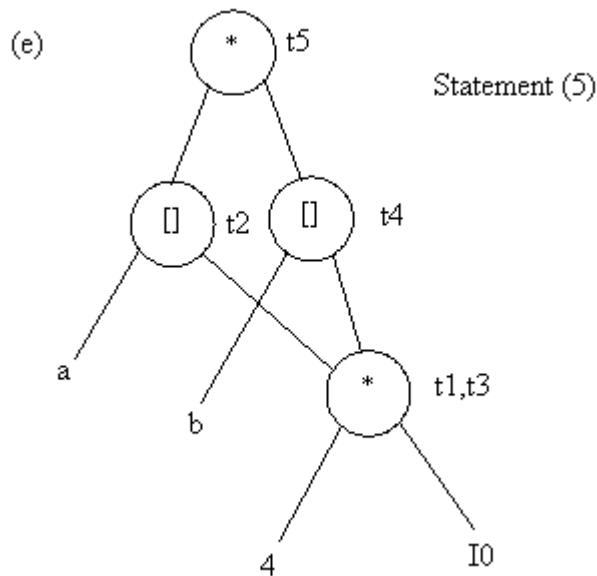
identifiers for the node  $n$  found in step 2 and set node( $x$ ) to  $n$ .

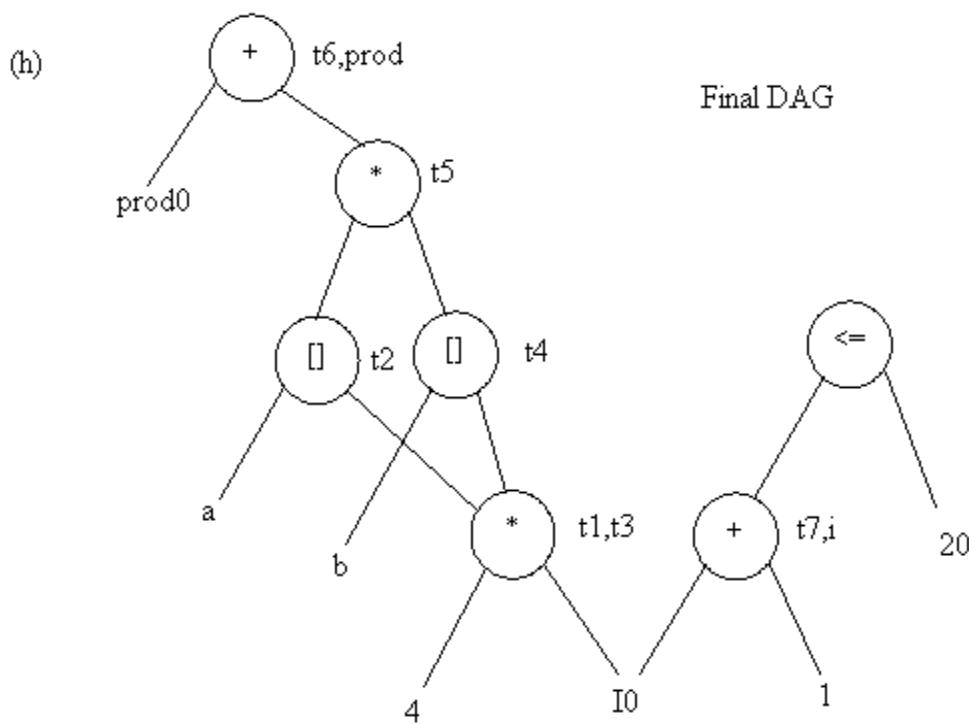
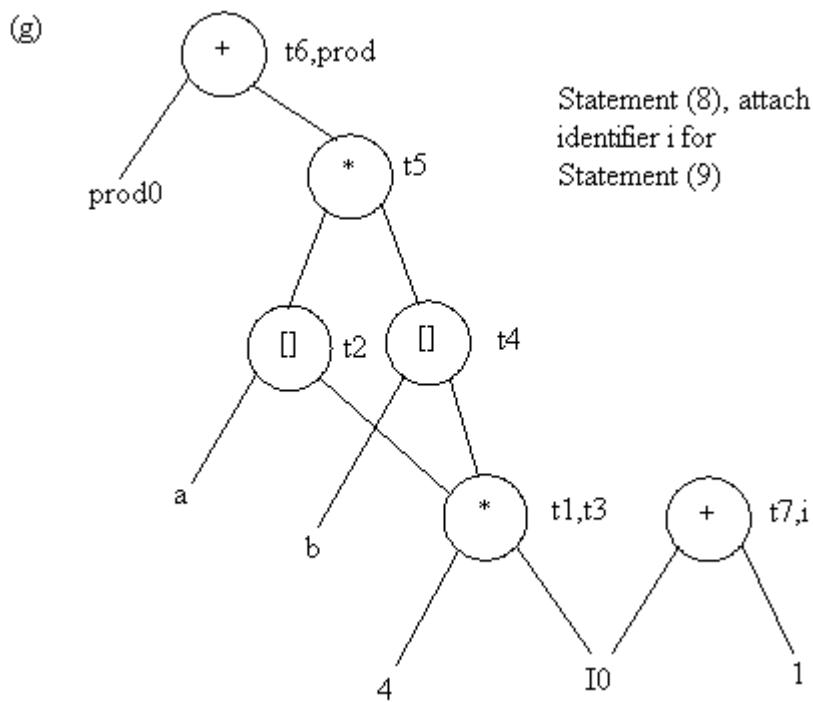
**Example:** Consider the block of three- address statements:

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







### 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.

## GENERATING CODE FROM DAGs

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 t<sub>1</sub> occurs immediately before t<sub>4</sub>.

```
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 R<sub>0</sub> , t<sub>1</sub>** and **MOV t<sub>1</sub> , R<sub>1</sub>** have been saved.

## A Heuristic ordering for Dags

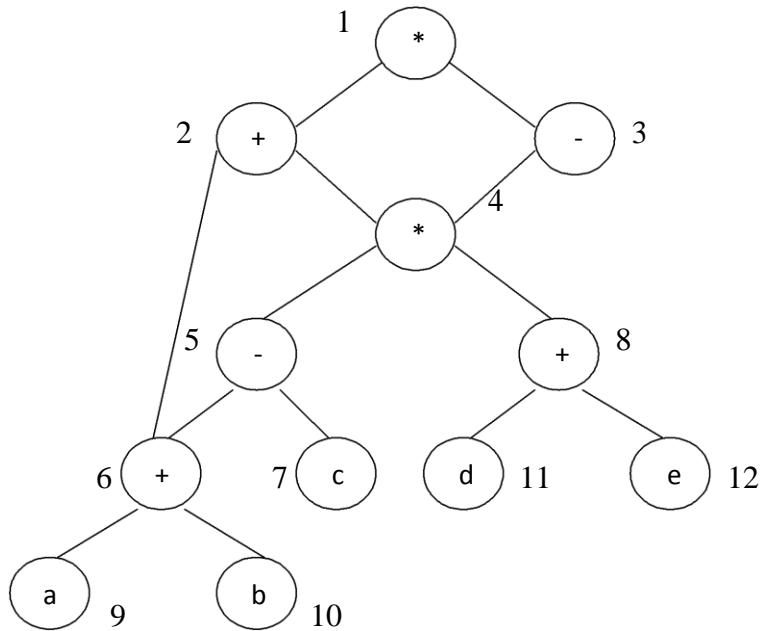
The heuristic ordering algorithm attempts to make the evaluation of a node immediately follow the evaluation of its leftmost argument.

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**  
**begin**
- 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.

**Code  
sequence:**

```
t8 := d + e  
t6 := a + b  
t5 := t6 - c  
t4 := t5 * t8  
t3 := t4 - e  
t2 := t6 + t4  
t1 := t2 * t3
```

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

## **IMPORTANT QUESTIONS & ANSWERS**

**Department of Computer Science and Engineering**

**SUBJECT CODE: CS8602**

**SUBJECT NAME: Compiler Design**

**Regulation: 2017**

**Semester and Year: 06/III**

**ANNA UNIVERSITY, CHENNAI-25  
SYLLABUS COPY  
REGULATION 2017**

<b>CS8602</b>	<b>COMPILER DESIGN</b>	<b>L T P C</b>	<b>3 0 0 3</b>
<b>1.</b>	<b>INTRODUCTION TO COMPILERS</b>		<b>5</b>
Translators – Compilation and Interpretation-Language Processors-The phases of Compiler-Errors encountered in Different Phases-The Grouping of Phases-Compiler Construction Tools-Programming Language Basics.			
<b>2.</b>	<b>LEXICAL ANALYSIS</b>		<b>9</b>
Need and Role of Lexical Analyzer-Lexical Errors-Expressing Tokens by Regular Expressions-Converting Regular Expression to DFA- Minimization of DFA-Language for Specifying Lexical Analyzers-LEX-Design of Lexical Analyzer for a sample Language.			
<b>3.</b>	<b>SYNTAX ANALYSIS</b>		<b>10</b>
Need and Role of the Parser-Context Free Grammars -Top Down Parsing -General Strategies-Recursive Descent Parser Predictive Parser-LL(1) Parser-Shift Reduce Parser-LR Parser-LR (0)Item-Construction of SLR Parsing Table -Introduction to LALR Parser - Error Handling and Recovery in Syntax Analyzer-YACC-Design of a syntax Analyzer for a Sample Language .			
<b>4.</b>	<b>SYNTAX DIRECTED TRANSLATION &amp; RUN TIME ENVIRONMENT</b>		<b>12</b>
Syntax directed Definitions-Construction of Syntax Tree-Bottom-up Evaluation of S-Attribute Definitions- Design of predictive translator - Type Systems-Specification of a simple type checker-Equivalence of Type Expressions-Type Conversions. RUN-TIME ENVIRONMENT: Source Language Issues-Storage Organization-Storage Allocation-Parameter Passing-Symbol Tables-Dynamic Storage Allocation-Storage Allocation in FORTAN.			
<b>5.</b>	<b>CODE OPTIMIZATION AND CODE GENERATION</b>		<b>9</b>
Principal Sources of Optimization-DAG- Optimization of Basic Blocks-Global Data Flow Analysis-Efficient Data Flow Algorithms-Issues in Design of a Code Generator - A Simple Code Generator Algorithm			
			<b>TOTAL : 45</b>

## TEXT BOOKS

1. Alfred V Aho, Monica S. Lam, Ravi Sethi and Jeffrey D Ullman, "Compilers – Principles, Techniques and Tools", 2nd Edition, Pearson Education, 2007.

## REFERENCES

1. Alfred V Aho, Monica S. Lam, Ravi Sethi and Jeffrey D Ullman, "Compilers – Principles, Techniques and Tools", 2nd Edition, Pearson Education, 2007.
2. Randy Allen, Ken Kennedy, "Optimizing Compilers for Modern Architectures: A Dependence-based Approach", Morgan Kaufmann Publishers, 2002.
3. Steven S. Muchnick, "Advanced Compiler Design and Implementation", Morgan Kaufmann Publishers - Elsevier Science, India, Indian Reprint 2003.
4. Keith D Cooper and Linda Torczon, "Engineering a Compiler", Morgan Kaufmann Publishers Elsevier Science, 2004.
5. Charles N. Fischer, Richard. J. LeBlanc, "Crafting a Compiler with C", Pearson Education, 2008.

## TABLE OF CONTENTS

<b>Sl.no</b>	<b>Topic</b>	<b>Page No</b>
1.	Aim & Objective of the subject	1
2.	Detailed Lesson Plan	2
<b>Unit I- Introduction To Compilers</b>		
3.	Part A	4
4.	Language Processor	8
5.	Software Tools used in Compilation	9
6.	Analysis Part of Compilation	10
7.	Phases Of Compiler	11
8.	Cousins Of Compiler	16
9.	The Grouping Of Phases	19
10.	Compiler Construction Tools	19
<b>Unit II- Lexical Analysis</b>		
11.	Part A	20
12.	Need and Role of Lexical Analyzer	23
13.	Input Buffering	25
14.	Expressing Tokens by Regular Expressions	27
15.	Language for Specifying Lexical Analyzer	32
16.	Regular Expression to DFA	34
<b>Unit III- Syntax Analysis</b>		
17.	Part A	37
18.	Need and Role of the Parser	41
19.	Various Terminologies in Parsing	43
20.	Context Free Grammars	44
21.	Writing a Grammar	47
22.	Recursive Descent Parser	50
23.	Predictive Parser	53
24.	Construction of Predictive Parsing Table	55
25.	Shift Reduce Parser	57
26.	LR Parser	60
27.	Construction of SLR Parsing Table	64
28.	Construction of LALR Parsing Table	67
29.	YACC-Design of a Syntax Analyzer	70
<b>Unit IV- Syntax Directed Translation &amp; Runtime Environment</b>		
30.	Part A	73
31.	Source Language Issues	76
32.	Storage Organization	77
33.	Storage Allocation	79
34.	Type Systems	83

35.	Specification of a Simple Type Checker	87
36.	Syntax Directed Definition	89
37.	Construction of Syntax Tree	95
38.	Parameter Parsing	96
39.	Design of a Predictive Translator	98
Unit V- Code Optimization and Code Generation		
40.	Part A	101
41.	Principal Sources of Optimization	104
42.	DAG	108
43.	Optimization of Basic Blocks	114
44.	Global Data Flow Analysis	116
46.	Issues in the Design of A Code Generator	123
47.	A simple Code Generator Algorithm	125
48.	Peephole Optimization	128
49.	Industrial / Practical Connectivity of the subject	131
50.	University Question papers	132

## AIM AND OBJECTIVE OF THE SUBJECT

The student should be made to

1. Learn the design principles of a Compiler.
2. Learn the various Parsing Techniques and different levels of translation.
3. Learn how to optimize and effectively generate machine codes.



## DETAILED LESSON PLAN

### Text Book

- Alfred V Aho, Monica S. Lam, Ravi Sethi and Jeffrey D Ullman, "Compilers – Principles, Techniques and Tools", 2nd Edition, Pearson Education, 2007.

### References

- Alfred V Aho, Ravi Sethi and Jeffrey D Ullman, "Compilers – Principles, Techniques and Tools", 2nd Edition, Pearson Education, 2007.
- Randy Allen, Ken Kennedy, "Optimizing Compilers for Modern Architectures: A Dependence-based Approach", Morgan Kaufmann Publishers, 2002.
- Steven S. Muchnick, "Advanced Compiler Design and Implementation", Morgan Kaufmann Publishers - Elsevier Science, India, Indian Reprint 2003.
- Keith D Cooper and Linda Torczon, "Engineering a Compiler", Morgan Kaufmann Publishers Elsevier Science, 2004.
- Charles N. Fischer, Richard. J. LeBlanc, "Crafting a Compiler with C", Pearson Education, 2008.

Sl. No	Unit	Topic / Portions to be Covered	Hours Required / Planned	Cumulative Hrs	Books Referred
1	I	Translators-Compilation and Interpretation-Language Processors	1	1	TB1
2		The Phases of Compiler	1	2	RB2
3		Errors encountered in Different Phases	1	3	RB2
4		The Grouping of Phases	1	4	RB2
5		Compiler Construction Tools- Programming Language Basics.	1	5	RB2 TB1
6	II	Need and Role of Lexical Analyzer	1	6	RB2
7		Lexical Errors	1	7	RB2
8		Expressing Tokens by Regular Expressions	1	8	RB2
9		Converting Regular Expression to DFA	1	9	TB1
10		Minimization of DFA	2	11	RB2
11		Language for Specifying Lexical Analyzers	2	13	RB2
12		LEX-Design of Lexical Analyzer for a sample Language	1	14	RB2
13	III	Need and Role of the Parser	1	15	RB2
14		Context Free Grammars	1	16	RB2
15		Top Down Parsing -General Strategies	1	17	RB2

<b>Sl. No</b>	<b>Unit</b>	<b>Topic / Portions to be Covered</b>	<b>Hours Require d / Planned</b>	<b>Cumulativ e Hrs</b>	<b>Books Referre d</b>
16	IV	Recursive Descent Parser Predictive Parser	1	18	RB2 RB2
17		LL(1) Parser- Shift Reduce Parser	1	19	RB2
18		LR Parser	1	20	RB2
19		LR (0)Item-Construction of SLR Parsing Table	1	21	RB2
20		Introduction to LALR Parser	1	22	RB2
21		Error Handling and Recovery in Syntax Analyzer	1	23	RB2
22		YACC-Design of a syntax Analyzer for a Sample Language	1	24	TB1
23	V	Syntax directed Definitions	1	25	RB2
24		Construction of Syntax Tree	1	26	RB2
25		Bottom-up Evaluation of S-Attribute Definitions	1	27	RB2
26		Design of predictive translator	1	28	RB2
27		Type Systems-Specification of a simple type checker	1	29	RB2
28		Equivalence of Type Expressions-Type Conversions	1	30	RB2
29		Source Language Issues-Storage Organization	1	31	RB2
30		Storage Allocation	1	32	RB2
31		Parameter Passing	1	33	RB2
32		Symbol Tables	1	34	RB2
33		Dynamic Storage Allocation	1	35	RB2
34		Storage Allocation in FORTAN	1	36	RB2
35		Principal Sources of Optimization	1	37	RB2
36		DAG	1	38	RB2
37		Optimization of Basic Blocks	1	39	RB2
38		Global Data Flow Analysis	2	41	RB2
40		Efficient Data Flow Algorithms	1	42	RB2
41		Issues in Design of a Code Generator	1	43	RB2
42		A Simple Code Generator Algorithm	2	45	RB2

## **UNIT I- INRODUCTION TO COMPILERS**

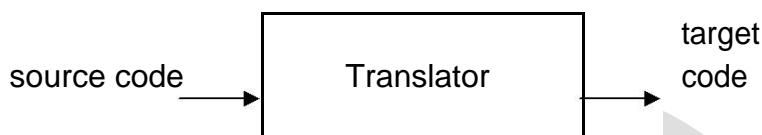
Translators-Compilation and Interpretation-Language Processors-The Phases of Compilers-Error encountered in different Phases-The Grouping of Phases-Compiler construction Tools-Programming Language Basics

### **PART-A**

#### **1. What is Translator and what are its types?**

Translator:

It is a program that translates one language to another.



Types of Translator:

1. Interpreter
2. Compiler
3. Assembler.

#### **2. What is an Interpreter?**

[Apr/May 2011]

Interpreter:

It is one of the translators that translate high level language to low level language.



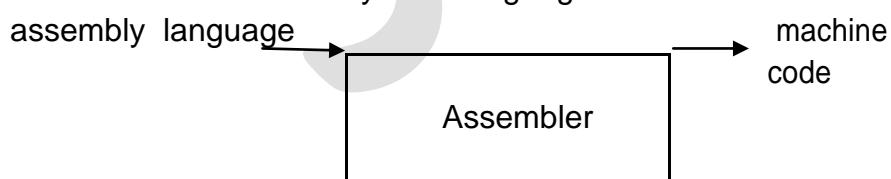
During execution, it checks line by line for errors.

Example: Basic, Lower version of Pascal.

#### **3. What is an Assembler?**

Assembler:

It translates assembly level language to machine code.



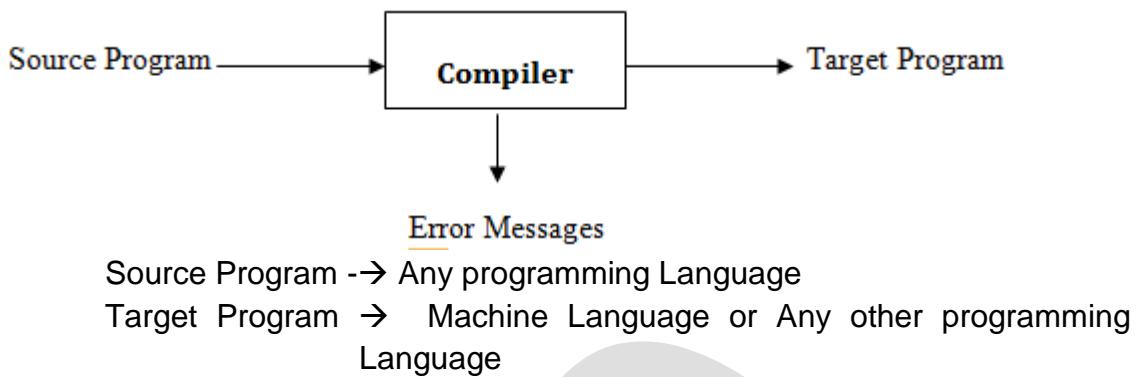
Example: Microprocessor 8085, 8086.

#### **4. What are the classifications of compilers?**

- Single Pass Compiler
- Multi Pass Compiler
- Load and go Compiler
- Debugging Compiler
- Optimizing Compiler

#### **5. What is Compiler?**

A Compiler is a program that reads a program written in one language and translates it into an equivalent program in another language – the target language.



## 6. What are the two parts of compilation?

[May /June 2016]

- i. Analysis
- ii. Synthesis

**Analysis :** Analysis part breaks up the source program into pieces and creates an intermediate representation of the source program.

**Synthesis:** Synthesis part constructs the desired target program from the intermediate representation.

## 7. What are the software tools used to perform analysis?

Structure Editor

- 1) Pretty Printer
- 2) Static Checker
- 3) Interpreters

## 8. What are the three phases of analysis phase of compilation?

- 1) Linear analysis
- 2) Hierarchical Analysis
- 3) Semantic Analysis

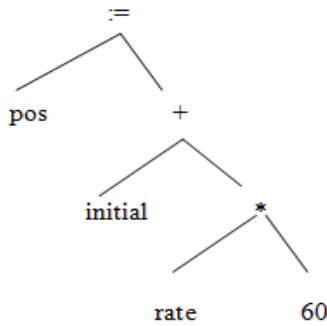
Linear Analysis : The stream of characters are read from left to right and grouped into tokens that are sequences of characters having a collective meaning. It is called as lexical analysis or scanning

Hierarchical Analysis : The characters or tokens are grouped hierarchically into nested collections with collective meaning. It is called as parsing or syntax analysis.

Semantic Analysis : In this analysis certain checks are made to ensure that the components of a program fit together meaningfully.

## 9. What is syntax Tree?

A syntax tree is a compressed representation of the parse tree in which the operator appears as the interior nodes and the operands are the children of the node for that operator



## 10. What is symbol Table?

[Nov/Dec 2016]

A Symbol table is a data structure containing a record for each identifier with fields for the attributes of the identifier. This allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly.

## 11. What are the properties of intermediate form?

- 1) Each three address instruction has at most one operator in addition to the assignment
- 2) The compiler must generate a temporary name to hold the value computed by each instruction.
- 3) Some three address instructions have fewer than three operands.

**Example:**

```

temp1=inttoreal(60)
temp2=id3*temp1
temp3=id2+temp2
id1=temp3
  
```

## 12. What are the functions of Preprocessor?

- **Macroprocessing :**

A preprocessor may allow a user to define macros that are shorthand for language constructs

- **File Inclusion :**

A preprocessors include the header files into the program text

- **Rational Preprocessor**

These processors augment older languages with more modern flow of control and data structuring facilities.

- **Language Extensions:**

The processors attempt to add capabilities to the language by what amounts to build in macros.

For eg) Equel is a database query language embedded in C. Statements beginning with ## are taken by the preprocessor to be data base access statements.

**13. What is front end and back end of a compiler?**

[Nov/Dec 2013]

**Front end of a compiler includes,**

- Lexical Analysis
- Syntactic Analysis
- The creation of the symbol Table
- Semantic Analysis
- Intermediate code generation

These phases depend on the source languages and are independent of target machine.

**Back end of a compiler includes**

- Code Optimization
- Code Generation

These phases depend on the target machine and do not independent of the source language.

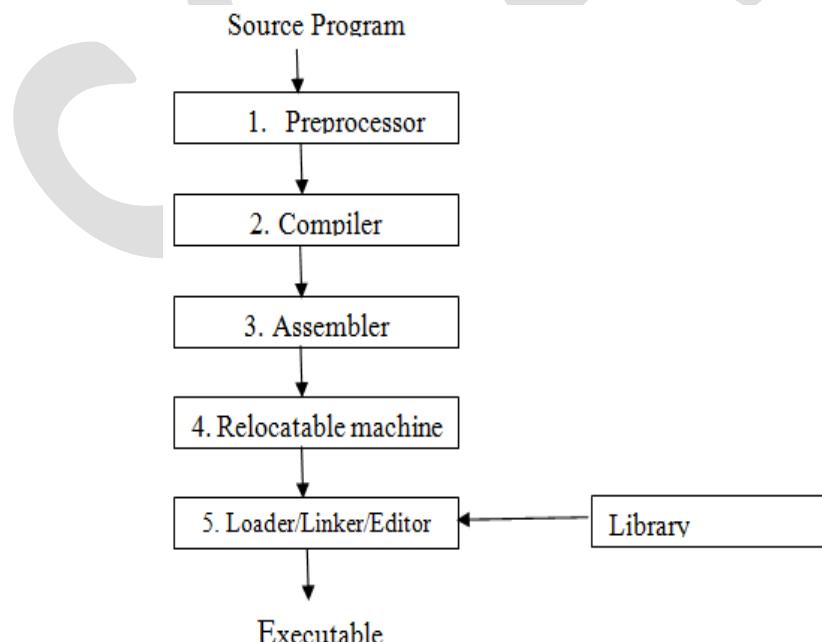
**14. List the various compiler construction tools.**

[Nov/Dec 2016]

- a. Parser Generator
- b. Scanner Generator
- c. Syntax directed translated engine
- d. Automatic code Generator
- e. Data flow engines.

**15. What is a language processing system?**

[May /June 2016]



**16. Write a regular Definition to represent date in the following format :JAN-5<sup>th</sup> 2014.(May2015)**

General RD Format for mm-dd-yyyy is: [1-12]-[1-31]-[0-9][0-9][0-9][0-9]  
 RD for JAN-5<sup>th</sup>-2014 is :[5]-[1]-[2][0][1][4]

## PART-B

### 1. a. LANGUAGE PROCESSOR

- ❖ Define the following terms: Compiler, Translator, Interpreter and differentiate between them.(6) [May/June 2014]

#### **Translator:**

It is a program that translates one language to another.



#### **Types of Translator:**

1. Interpreter
2. Compiler
3. Assembler

#### **1. Interpreter:**

It is one of the translators that translate high level language to low level language.

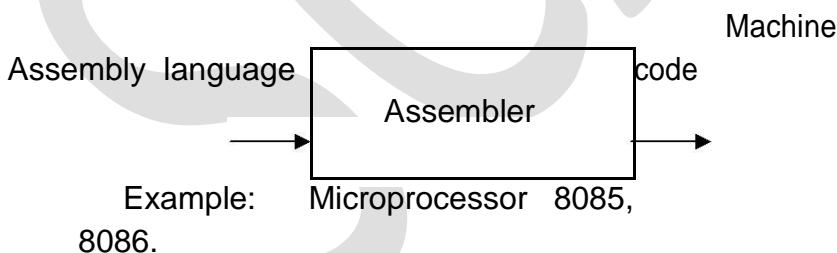


During execution, it checks line by line for errors.

Example: Basic, Lower version of Pascal.

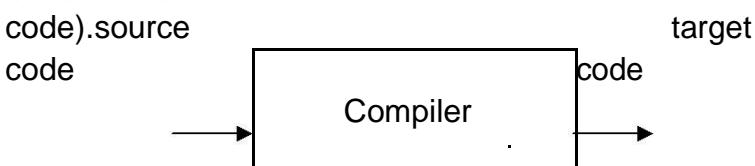
#### **2. Assembler:**

It translates assembly level language to machine code.



#### **3. Compiler:**

It is a program that translates one language(source code) to another language (target code).source code



It executes the whole program and then displays the errors.

Example: C, C++, COBOL, higher version of Pascal.

### Difference between compiler and Interpreter.

<b>COMPILER</b>	<b>INTERPRETER</b>
It is a translator that translates high level to low level language	It is a translator that translates high level to low level language
It displays the errors after the whole program is executed	It checks line by line for errors.
Examples: Basic, lower version of Pascal.	Examples: C, C++, Cobol, higher version of Pascal.

### 1.b. SOFTWARE TOOLS USED IN COMPIRATION

- ❖ Describe the software tools that are used for analysis part of compilation. [Apr / May 2009]

**1) Structure editor:**

- Takes as input a sequence of commands to build a source program.
- The structure editor not only performs the text-creation and modification functions of an ordinary text editor, but it also analyzes the program text, putting an appropriate hierarchical structure on the source program.
- For example, it can supply key words automatically - while .... do and begin..... end.

**2) Pretty printers :**

- A pretty printer analyzes a program and prints it in such a way that the structure of the program becomes clearly visible.
- For example, comments may appear in a special font.

**3) Static checkers :**

- A static checker reads a program, analyzes it, and attempts to discover potential bugs without running the program.
- For example, a static checker may detect that parts of the source program can never be executed.

**4) Interpreters :**

- Translates from high level language ( BASIC, FORTRAN, etc..) into machine language.
- An interpreter might build a syntax tree and then carry out the operations at the nodes as it walks the tree.
- Interpreters are frequently used to execute command language since each operator executed in a command language is usually an invocation of a complex routine such as an editor or compiler.

## **2. ANALYSIS PART OF COMPIRATION**

- ❖ Explain the analysis part of compilation with an example. [Apr / May 2009]

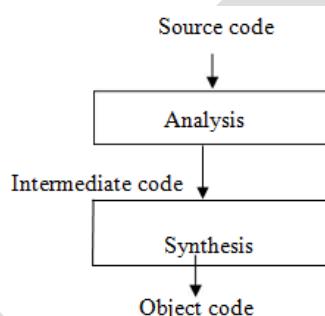
### **Parts of Compilation**

There are 2 parts to compilation:

1. Analysis
2. Synthesis

**Analysis** part breaks down the source program into constituent pieces and creates an intermediate representation of the source program.

**Synthesis** part constructs the desired target program from the intermediate representation.



### **Analysis of the Source Program**

Analysis consists of 3 phases:

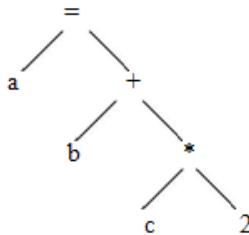
#### **Linear/Lexical Analysis :**

- It is also called scanning. It is the process of reading the characters from left to right and grouping into tokens having a collective meaning.
- For example, in the assignment statement  $a=b+c^2$ , the characters would be grouped into the following tokens:

- i) The identifier1 'a'
- ii) The assignment symbol (=)
- iii) The identifier2 'b'
- iv) The plus sign (+)
- v) The identifier3 'c'
- vi) The multiplication sign (\*)
- vii) The constant '2'

#### **Syntax Analysis :**

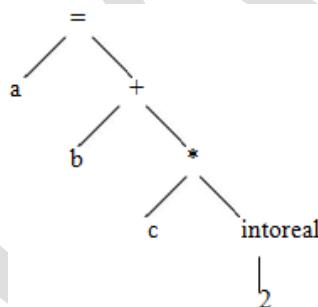
- It is called parsing or hierarchical analysis. It involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output.
- They are represented using a syntax tree as shown below:



- A **syntax tree** is the tree generated as a result of syntax analysis in which the interior nodes are the operators and the leaf nodes are the operands.
- This analysis shows an error when the syntax is incorrect.

#### Semantic Analysis:

- It checks the source programs for semantic errors and gathers type information for the subsequent code generation phase. It uses the syntax tree to identify the operators and operands of statements.
- An important component of semantic analysis is **type checking**. Here the compiler checks that each operator has operands that are permitted by the source language specification.
- For example if a b and c are real , then semantic analysis phase invokes the error recovery routine to convert integer 2 into real.



### 3. THE PHASES OF COMPILER

- ❖ Explain the Various phases of compiler and trace the program segment pos=initial+rate \* 60 for all the phases.(10)[April/May ,Nov 2011,] [May /June 2016]
- ❖ Explain the different phases of compiler in detail.(12) . [Nov/Dec 2013]
- ❖ Explain in detail the process of compilation.Illustrate the output of each phase of the compilation for the input  $a=(b+c)*(b+c)*2(16)$  . [May/June 2014]
- ❖ What are the various phases of compiler?Explain each phase in detail.(16) [May/June 2012] [Nov/Dec 2016].

### PHASES OF COMPILER

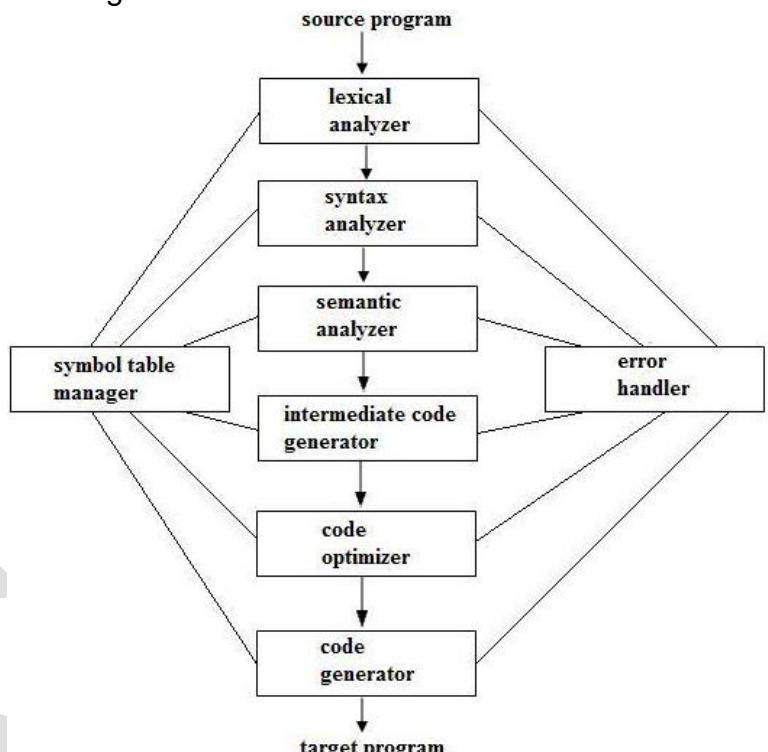
A Compiler operates in phases, each of which transforms the source program from one representation into another. The following are the phases of the compiler:

### Main phases:

- 1) Lexical analysis
- 2) Syntax analysis
- 3) Semantic analysis
- 4) Intermediate code generation
- 5) Code optimization
- 6) Code generation

### Sub-Phases:

- 1) Symbol table management
- 2) Error handling



### Lexical Analysis:

- It is the first phase of the compiler. It gets input from the source program and produces tokens as output.
- It reads the characters one by one, starting from left to right and forms the tokens.
- **Token** : It represents a logically cohesive sequence of characters such as keywords, operators, identifiers, special symbols etc.

Example: a +b =20

Here, a,b,+,=,20 are all separate tokens.

Group of characters forming a token is called the **Lexeme**.

The lexical analyser not only generates a token but also enters the lexeme into the symbol table if it is not already there.

### Syntax Analysis:

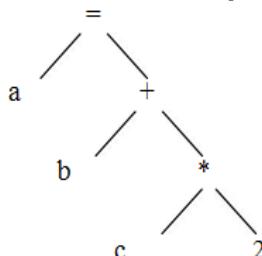
- It is the second phase of the compiler. It is also known as parser.
- It gets the token stream as input from the lexical analyser of the

compiler and generates syntax tree as the output.

- Syntax tree:

It is a tree in which interior nodes are operators and exterior nodes are operands.

□ Example: For  $a=b+c*2$ , syntax tree is



### Semantic Analysis:

- It is the third phase of the compiler.
- It gets input from the syntax analysis as parse tree and checks whether the given syntax is correct or not.
- It performs type conversion of all the data types into real data types.

### Intermediate Code Generation:

- It is the fourth phase of the compiler.
- It gets input from the semantic analysis and converts the input into output as intermediate code such as three-address code.
- Each three address instruction has at most one operator in addition to the assignment
- The compiler must generate a temporary name to hold the value computed by each instruction.
- Some three address instructions have fewer than three operands.
- The three -address code consists of a sequence of instructions, each of which has atmost three operands.

Example:  $t1=t2+t3$

### Code Optimization:

- It is the fifth phase of the compiler.
- It gets the intermediate code as input and produces optimized intermediate code as output.
- This phase reduces the redundant code and attempts to improve the intermediate code so that faster-running machine code will result.
- During the code optimization, the result of the program is not affected.
- To improve the code generation, the optimization involves
  - Deduction and removal of dead code (unreachable code).
  - Calculation of constants in expressions and terms.
  - Collapsing of repeated expression into temporary string.
  - Loop unrolling.
  - Moving code outside the loop.

- Removal of unwanted temporary variables.

### **Code Generation:**

- It is the final phase of the compiler.
- It gets input from code optimization phase and produces the target code or object code as result.
- Intermediate instructions are translated into a sequence of machine instructions that perform the same task.
- The code generation involves
  - allocation of register and memory
  - generation of correct references
  - generation of correct data types
  - generation of missing code

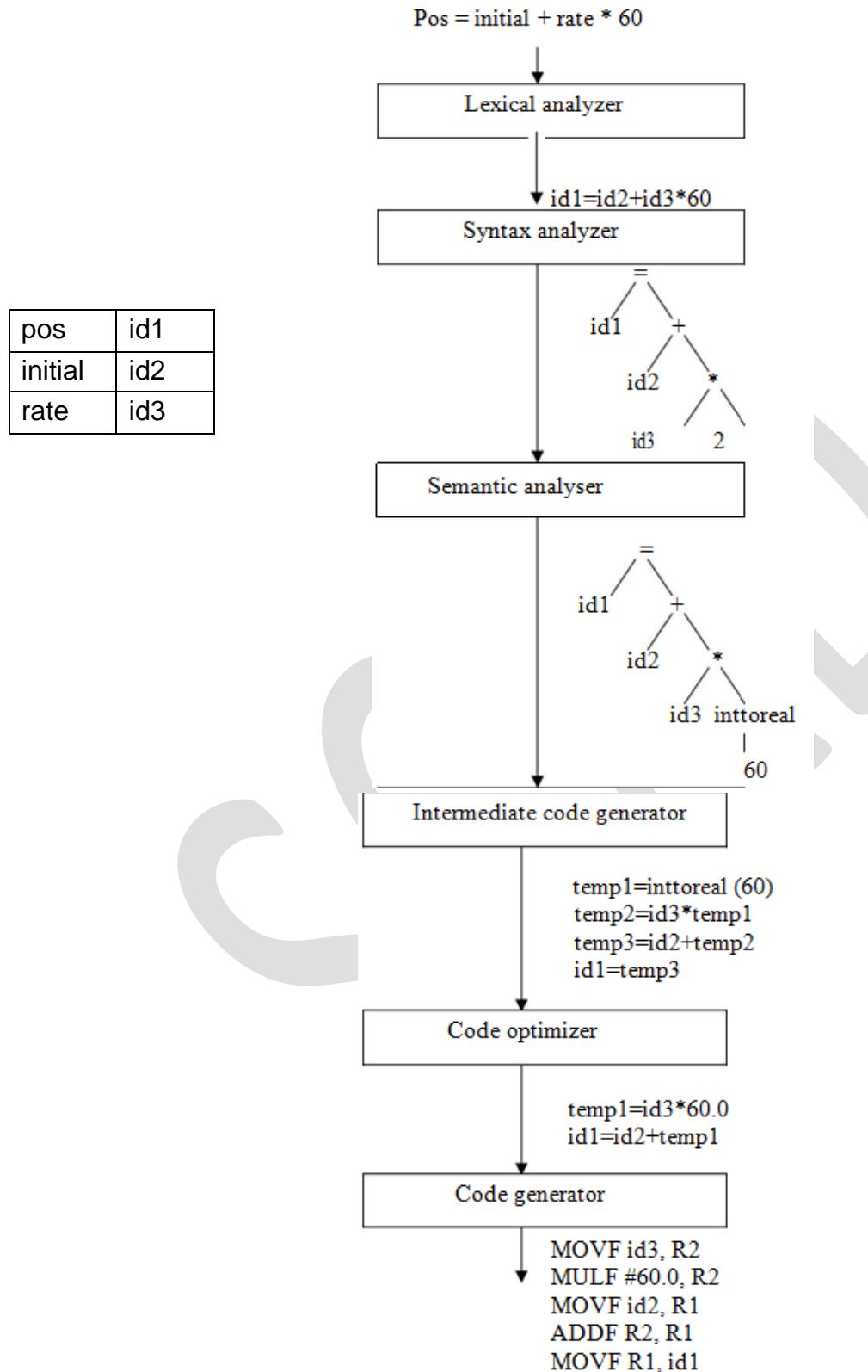
### **Symbol Table Management:**

- Symbol table is used to store all the information about identifiers used in the program.
- It is a data structure containing a record for each identifier, with fields for the attributes of the identifier.
- It allows to find the record for each identifier quickly and to store or retrieve data from that record.
- Whenever an identifier is detected in any of the phases, it is stored in the symbol table.

### **Error Handling:**

- Each phase can encounter errors. After detecting an error, a phase must handle the error so that compilation can proceed.
- In lexical analysis, errors occur in separation of tokens.
- In syntax analysis, errors occur during construction of syntax tree.
- In semantic analysis, errors occur when the compiler detects constructs with right syntactic structure but no meaning and during type conversion.
- In code optimization, errors occur when the result is affected by the optimization.
- In code generation, it shows error when code is missing etc.

To illustrate the translation of source code through each phase, consider the statement `pos=initial+rate*60`. The figure shows the representation of this statement after each phase:



### 3 b) Explain the various Error Encountered in Different Phases of compiler.

[May , Nov 2016]

#### Error Encountered in Different Phases

Each phase can encounter errors. After detecting an error, a phase must somehow deal with the error, so that compilation can proceed.

A program may have the following kinds of errors at various stages:

#### Lexical Errors

It includes incorrect or misspelled name of some identifier i.e., identifiers typed incorrectly.

#### Syntactical Errors

It includes missing semicolon or unbalanced parenthesis. Syntactic errors are handled by syntax analyzer (parser).

When an error is detected, it must be handled by parser to enable the parsing of the rest of the input. In general, errors may be expected at various stages of compilation but most of the errors are syntactic errors and hence the parser should be able to detect and report those errors in the program.

**The goals of error handler in parser are:**

- Report the presence of errors clearly and accurately.
- Recover from each error quickly enough to detect subsequent errors.
- Add minimal overhead to the processing of correcting programs.

There are four common error-recovery strategies that can be implemented in the parser to deal with errors in the code.

- Panic mode.
- Statement level.
- Error productions.
- Global correction.

#### Semantical Errors

These errors are a result of incompatible value assignment. The semantic errors that the semantic analyzer is expected to recognize are:

- Type mismatch.
- Undeclared variable.
- Reserved identifier misuse.
- Multiple declaration of variable in a scope.
- Accessing an out of scope variable.
- Actual and formal parameter mismatch.

#### Logical errors

These errors occur due to not reachable code-infinite loop.

## 4. COUSINS OF COMPILER

- ❖ Discuss the cousins of compiler.(4)

[Nov/Dec 2013]

#### COUSINS OF COMPILER

1. Preprocessor

2. Assembler
3. Loader and Link-editor

### **Preprocessor**

A preprocessor is a program that processes its input data to produce output that is used as input to another program. The output is said to be a preprocessed form of the input data, which is often used by some subsequent programs like compilers.

They may perform the following functions :

1. Macro processing
2. File Inclusion
3. Rational Preprocessors
4. Language extension

#### **1. Macro processing:**

A macro is a rule or pattern that specifies how a certain input sequence should be mapped to an output sequence according to a defined procedure. The mapping process that instantiates a macro into a specific output sequence is known as macro expansion.

#### **2. File Inclusion:**

Preprocessor includes header files into the program text. When the preprocessor finds an #include directive it replaces it by the entire content of the specified file.

#### **3. Rational Preprocessors:**

These processors change older languages with more modern flow-of-control and data-structuring facilities.

#### **4. Language extension :**

These processors attempt to add capabilities to the language by what amounts to built-in macros. For example, the language Equel is a database query language embedded in C.

### **ASSEMBLER**

Assembler creates object code by translating assembly instruction mnemonics into machine code. There are two types of assemblers:

- One-pass assemblers go through the source code once and assume that all symbols will be defined before any instruction that references them.
- Two-pass assemblers create a table with all symbols and their values in the first pass, and then use the table in a second pass to generate code.

#### **Two Pass Assembly :**

Consider the Assembly code

    Mov a R1

ADD #2 R1

MOV R1 b        Thus it computes b=a+2

The simplest form of assembler makes two passes over the input. A pass consists of reading an input file once.

In the first pass all the identifiers that denote storage locations are found and stored in a symbol table.

For eg) IDENTIFIER                    ADDRESS

a	0
b	4

In the second pass the assembler scans the input again. This time it translates each operation code into the sequence of bits representing that operation in machine language and it translates each identifier representing a location into the address given for that identifier in the symbol table.

The output of the second pass is relocatable machine code that it can be loaded starting at any location L in memory.

The following is machine code into which the assembly instructions might be translated

```
0001 01 00 00000000*
 0011 01 10 00000010
 0010 01 00 00000100*
```

The first four bits are the instruction code

```
0001 - Load
0010 - Store 0011 - Add
```

By load and store we mean moves from memory into a register and vice versa.

The next two bits designate a register and 01 refers to register 1

The two bits after that represent a “tag” represent the addressing mode.

00 → ordinary addressing mode

10 → immediate addressing mode

The last eight bits refers to a memory address.

### **Linker and Loader**

A **linker** or **link editor** is a program that takes one or more objects generated by a compiler and combines them into a single executable program.

Three tasks of the linker are :

1. Searches the program to find library routines used by program, e.g. printf(), math routines.
2. Determines the memory locations that code from each module will occupy and relocates its instructions by adjusting absolute references
3. Resolves references among files.

A **loader** is the part of an operating system that is responsible for loading programs in memory, one of the essential stages in the process of starting a program.

Suppose that the address space containing the data is to be loaded at starting at location L. Then L must be added to the address of the instruction.

Thus if L = 00001111 i.e 15 then a and b would be at locations 15 and 19 respectively and the instruction would appear as

```
0001 01 00 00001111
 0011 01 10 00000010
 0010 01 00 00010011
```

### **5.a. THE GROUPING OF PHASES**

- ❖ Explain the need for grouping of phases of compiler.

Nov/Dec 2014, 16, May/June 2014, 16

#### **GROUPING OF THE PHASES**

Compiler can be grouped into front and back ends:

**Front end:** analysis (machine independent)

These normally include

- lexical analysis
- syntactic analysis,
- the creation of the symbol table,
- semantic analysis
- the generation of intermediate code.

It also includes error handling that goes along with each of these phases.

**Back end:** synthesis (machine dependent)

It includes

- code optimization phase
- code generation phase

It also include along with the necessary error handling and symbol table operations.

#### **Compiler passes**

A collection of phases is done only once (single pass) or multiple times (multi pass)

- Single pass: usually requires everything to be defined before being used in source program.
- Multi pass: compiler may have to keep entire program representation in memory.

Several phases can be grouped into one single pass and the activities of these phases are interleaved during the pass. For example, lexical analysis, syntax analysis, semantic analysis and intermediate code generation might be grouped into one pass.

### **5. b. COMPILER CONSTRUCTION TOOLS.**

- ❖ State the compiler construction tools. Explain them.(6).[NOV/DEC 2011].
- ❖ Write short notes on compiler construction Tools.(8)[Nov/Dec 2014, 2016].
- ❖ Mention any four compiler construction tools with their benefits and drawbacks.(8) [Apr/May 2015]
- ❖ Briefly explain the compiler construction Tools. (8) [May /June 2012]

#### **COMPILER CONSTRUCTION TOOLS**

These are specialized tools that have been developed for helping implement various phases of a compiler. The following are the compiler

construction tools:

**1. Parser Generators:**

- These produce syntax analyzers, normally from input that is based on a context-free grammar.
- It consumes a large fraction of the running time of a compiler.
- Example-YACC (Yet Another Compiler-Compiler).

**2. Scanner Generator:**

- These generate lexical analyzers, normally from a specification based on regular expressions.
- The basic organization of lexical analyzers is based on finite automation.

**3. Syntax-Directed Translation:**

- These produce routines that walk the parse tree and as a result generate intermediate code.
- Each translation is defined in terms of translations at its neighbor nodes in the tree.

**4. Automatic Code Generators:**

- It takes a collection of rules to translate intermediate language into machine language. The rules must include sufficient details to handle different possible access methods for data.

**5. Data-Flow Engines:**

- It does code optimization using data-flow analysis, that is, the gathering of information about how values are transmitted from one part of a program to each other part.

## **UNIT II- LEXICAL ANALYSIS**

Need and Role of Lexical Analyzer – Lexical Errors-Expressing Tokens by Regular Expressions. Converting Regular Expressions to DFA-Minimization of DFA-Language for specifying Lexical Analyzers-LEX-Design of Lexical Analyzer for a sample Language.

### **PART-A**

**1. What are the reasons for separating the analysis phase of compilation into lexical analysis and parsing? (or)**

**What are the issues of lexical analysis?**

**[May /June 2016]**

1. The separation of lexical analysis from syntax analysis often allows us to simplify one or the other of the phases.
2. Compiler Efficiency is improved.
  - A separate lexical analyzer allows to construct a specialized and potentially more efficient for the task.
  - Specialized buffering techniques for reading input characters and processing tokens can significantly speed up the performance of a compiler

3. Compiler portability is enhanced.  
Input alphabet peculiarities and other device specific anomalies can be restricted to the lexical analyzer.

## 2. Define Token, Pattern and Lexeme? [May/June 2016] [Nov/Dec 2016]

- The set of string is described by a rule called a pattern associated with the token
- The pattern is said to match each string in the set
- A lexeme is a sequence of characters in the source program that is matched by the pattern for a token
- For ex) const pi=3.1416  
pi--> lexeme for the token identifier

## 3.What is panic mode error recovery? [Apr/May 2015, May/Jun 2011]

Successive characters are deleted from the remaining input until the lexical analyzer can find a well defined token.

➤ Other possible error recovery actions are

- Deleting an extraneous characters
- Inserting a missing character
- Replacing an incorrect character by a correct character
- Transposing two adjacent characters.

## 4. Define the following terms.(a)Alphabet (b) String (3) Language

- The term alphabet or character class denotes any finite set of symbols
- A String over some alphabet is a finite sequence of symbols drawn from that alphabet
- The length of a string s ,|s| is the number of occurrences of symbols in s
- The language denotes any set of strings over some fixed alphabet. Abstract languages like  $\emptyset$  the empty set or  $\{\epsilon\}$  the set containing only the empty string are languages.
- If x and y are strings then the concatenation of x and y written xy is the string formed by appending y to x  
For eg) if x=dog , y= house  
Then xy=doghouse

## 5.Define regular definition

[Apr/May 2015]

If  $\Sigma$  is an alphabet of basic symbols then a regular definition is a sequence of definition of the form

d1-->r1

d2-->r2

...

dn-->rn

where each  $d_i$  is a distinct name

each  $r_i$  is a regular expression over the symbols in  $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$

## 6. Give the regular definition for identifier.

[Nov/Dec 2013]

letter --> A|B|....|Z|a|b|c...|z

digit  $\rightarrow 0|1|2\dots|9$   
 id  $\rightarrow \text{letter}(\text{letter}|\text{digit})^*$

### 7. Give the regular definition for unsigned numbers.

digit  $\rightarrow 0|1|2\dots|9$   
 digit  $\rightarrow \text{digit digit}^*$   
 optional\_fraction  $\rightarrow .\text{digits}|\epsilon$   
 optional\_exponent  $\rightarrow (\text{E} (+|-)\epsilon)\text{digits}|\epsilon$   
 num  $\rightarrow \text{digits optional_fraction optional_exponent.}$

### 8. What is Finite Automata?

There are two types of Finite Automata :

- Non-deterministic Finite Automata (NFA)
- Deterministic Finite Automata (DFA)

#### Non-deterministic Finite Automata

NFA is a mathematical model that consists of five tuples denoted by

$$M = \{Q_n, \Sigma, \delta, q_0, f_n\}$$

Q

n – finite set of states

$\Sigma$  – finite set of input symbols

– transition function that maps state-symbol pairs to set of

$\delta$  states

$q_0$  – starting state

$f_n$  – final state

#### Deterministic Finite Automata

DFA is a special case of a NFA in which

- i) no state has an  $\epsilon$ -transition.
- ii) there is at most one transition from each state on any input.

DFA has five tuples denoted by

$$M = \{Q_d, \Sigma, \delta, q_0, f_d\}$$

$Q_d$  – finite set of states

$\Sigma$  – finite set of input symbols

$\delta$  – transition function that maps state-symbol pairs to set of states  
 starting

$q_0$  – state

$f_d$  – final state

### 9. What is Regular Expression? Or List the rules that form the BASIS. [Nov/Dec 2016]

1.  $\epsilon$  is a regular expression, and  $L(\epsilon)$  is  $\{\epsilon\}$ , that is, the language whose sole member is the empty string.
2. If 'a' is a symbol in  $\Sigma$ , then 'a' is a regular expression, and  $L(a) = \{a\}$ , that is, the language with one string, of length one, with 'a' in its one position.

**3.** Suppose r and s are regular expressions denoting the languages L(r) and L(s). Then,

- a)  $(r)|(s)$  is a regular expression denoting the language  $L(r) \cup L(s)$ .
- b)  $(r)(s)$  is a regular expression denoting the language  $L(r)L(s)$ .
- c)  $(r)^*$  is a regular expression denoting  $(L(r))^*$ .
- d)  $(r)$  is a regular expression denoting  $L(r)$ .

#### **10. Define Regular Set and Non regular set?**

**Regular Set:**

A language that can be defined by a regular expression is called a regular set. If two regular expressions r and s denote the same regular set, we say they are equivalent and write  $r = s$ .

**Non Regular Set:**

A language that cannot be described by regular expression is called a non regular set.

#### **11. Construct Regular expression for the language**

$$L = \{w \in \{a,b\}^*/w \text{ ends in } abb\}$$

$$\text{Ans: } \{a/b\}^*abb.$$

#### **12. Define concrete and abstract syntax with example. (May 2009)**

Abstract syntax tree is the tree in which node represents an operator and the children represents operands. Parse tree is called a concrete syntax tree, which shows how the start symbol of a grammar derives a string in the language. Abstract syntax tree, or simple syntax tree, differ from parse tree because superficial distinctions of form, unimportant for translation, do not appear in syntax tree.

#### **13. List the operations on languages. [May /June 2016]**

- 1. Union
- 2. Concatenation
- 3. Difference

#### **14. Write a grammar for branching statements. [May /June 2016]**

The Context free Grammar for if-else statement is,

$$\begin{aligned} S &\rightarrow iEtS \\ S &\rightarrow iEtSeS \\ S &\rightarrow a \\ E &\rightarrow b \end{aligned}$$

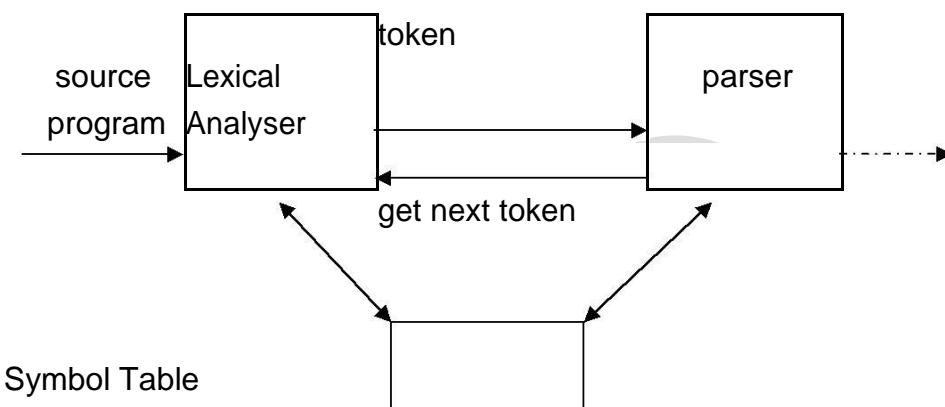
### **PART-B**

#### **1. NEED AND ROLE OF LEXICAL ANALYZER**

- ❖ What are the issues in the design of a lexical Analyzer.(4[May /June 2014, May /June 2012 ] [May /June 2016])
- ❖ Discuss the role of Lexical Analyzer in detail.(8).[Apr / May 2011] [Nov/Dec 2016].
- ❖ Describe the error recovery schemes in lexical phase of a compiler.[May 15]

## THE ROLE OF THE LEXICAL ANALYZER

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



- Upon receiving a “get next token” command from the parser, the lexical analyzer reads input characters until it can identify the next token.

## ISSUES OF LEXICAL ANALYZER

There are three issues in lexical analysis:

- To make the design simpler.
- To improve the efficiency of the compiler.
- To enhance the computer portability.

### Tokens

A token is a string of characters, categorized according to the rules as a symbol (e.g., IDENTIFIER, NUMBER, COMMA). The process of forming tokens from an input stream of characters is called **tokenization**.

A token can look like anything that is useful for processing an input text stream or text file. Consider this expression in the C programming language:  
sum=3+2;

### Lexeme:

A Lexeme is a sequence of characters in the source program that is matched by the token for a token . For example in the statement sum=2+3 ,

The substring sum is a lexeme for the token “identifier”

Lexeme	Token type	Informal Description of a Pattern
Sum	Identifier	Letter followed by letters and digits

=	Assignment operator	<,<=,>,>=,<>,<=
3	Number	Any numeric constant
+	Addition Operator	Arithmatic operators such as +,-,*,/
2	Number	Any numeric constant
	End of Statement	

**Pattern:**

The set of strings described by a rule called patterns.

In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.

**Attributes for Tokens**

Some tokens have attributes that can be passed back to the parser. The lexical analyzer collects information about tokens into their associated attributes. The attributes influence the translation of tokens.

- i) Constant : value of the constant
- ii) Identifiers: pointer to the corresponding symbol table entry.

**Error Recovery Strategies In Lexical Analysis:**

The following are the error-recovery actions in lexical analysis:

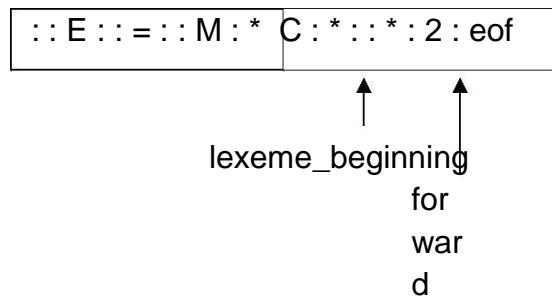
- 1) Deleting an extraneous character.
- 2) Inserting a missing character.
- 3) Replacing an incorrect character by a correct character
- 4) Transforming two adjacent characters.
- 5) Panic mode recovery: Deletion of successive characters from the token until error is resolved.

**2. INPUT BUFFERING**

- ❖ Describe input buffering techniques in detail.[Nov /Dec 2013]
- ❖ Explain briefly about input buffering in reading the source program for finding the tokens.[Nov/Dec 2011].
- ❖ Differentiate Lexeme Pattern and Token. (6)[May/Jun 2014]

**BUFFER PAIRS**

- A buffer is divided into two N-character halves, as shown below



- Each buffer is of the same size N, and N is usually the number of characters on one disk block. E.g., 1024 or 4096 bytes.
- Using one system read command we can read N characters into a buffer.
- If fewer than N characters remain in the input file, then a special character, represented by **eof**, marks the end of the source file.
- Two pointers to the input are maintained:
  1. Pointer **lexeme\_beginning**, marks the beginning of the current lexeme, whose extent we are attempting to determine.
  2. Pointer **forward** scans ahead until a pattern match is found. Once the next lexeme is determined, forward is set to the character at its right end.
- The string of characters between the two pointers is the current lexeme. After the lexeme is recorded as an attribute value of a token returned to the parser, lexeme \_beginning is set to the character immediately after the lexeme just found.

#### **Advancing forward pointer:**

Advancing forward pointer requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer. If the end of second buffer is reached, we must again reload the first buffer with input and the pointer wraps to the beginning of the buffer.

#### **Code to advance forward pointer:**

```

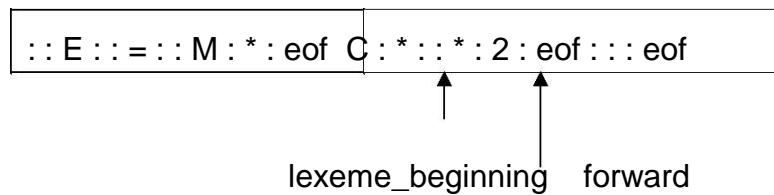
if forward at end of first half then
  begin reload second half;
  forward := forward + 1
end
else if forward at end of second half
  then begin reload second half;
  move forward to beginning of first half
end
else forward := forward + 1;
  
```

#### **Sentinels**

- For each character read, we make two tests: one for the end of the buffer,

and one to determine what character is read. We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a sentinel character at the end.

- The sentinel is a special character that cannot be part of the source program, and a natural choice is the character eof.
- The sentinel arrangement is as shown below:



Note that eof retains its use as a marker for the end of the entire input. Any eof that appears other than at the end of a buffer means that the input is at an end.

#### **Code to advance forward pointer:**

```

forward := forward +
1; if forward ↑ = eof
then begin
  if forward at end of first half then
    begin reload second half;
    forward := forward + 1
  end
  else if forward at end of second half
    then begin reload first half;
    move forward to beginning of
    first half end
  else /* eof within a buffer signifying end of
        input */ terminate lexical analysis
end

```

### **3. EXPRESSING TOKEN BY REGULAR EXPRESSIONS**

- ❖ Explain – Specification and Recognition of Tokens.(8) [Nov/Dec 2014].
- ❖ Discuss how FA is used to represent tokens and perform lexical analysis with example.(8). [Nov/Dec 2016]
- ❖ Draw the transition diagram for relational operator and unsigned numbers in Pascal.(8) [Apr/May 2011]
- ❖ Write notes on regular expressions.[May /June 2016]

### **SPECIFICATION OF TOKENS**

There are 3 specifications of tokens:

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

### Strings and Languages

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

A **string** over an alphabet is a finite sequence of symbols drawn from that alphabet.

A **language** is any countable set of strings over some fixed alphabet.

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

### Operations on strings

The following string-related terms are commonly used:

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

### Operations on languages:

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

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

The following example shows the operations on strings:

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

- |                  |  |
|------------------|--|
| 1. Union         | : $L \cup S = \{0,1,a,b,c\}$             |
| :                |  |
| 2. Concatenation | $L \cdot S = \{0a, 1a, 0b, 1b, 0c, 1c\}$ |
| Kleene           |  |
| 3. closure       | : $L^* = \{\epsilon, 0, 1, 00, \dots\}$  |

Positive

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

### Regular Expressions

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

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

1.  $\epsilon$  is a regular expression, and  $L(\epsilon)$  is  $\{\epsilon\}$ , that is, the language whose sole member is the empty string.
  2. If 'a' is a symbol in  $\Sigma$ , then 'a' is a regular expression, and  $L(a) = \{a\}$ , that is, the language with one string, of length one, with 'a' in its one position.
  3. Suppose  $r$  and  $s$  are regular expressions denoting the languages  $L(r)$  and  $L(s)$ . Then,
- a)  $(r)|(s)$  is a regular expression denoting the language  $L(r) \cup L(s)$ .
  - b)  $(r)(s)$  is a regular expression denoting the language  $L(r)L(s)$ .
  - c)  $(r)^*$  is a regular expression denoting  $(L(r))^*$ .
  - d)  $(r)$  is a regular expression denoting  $L(r)$ .

4. The unary operator  $*$  has highest precedence and is left associative.

5. Concatenation has second highest precedence and is left associative.

6.  $|$  has lowest precedence and is left associative.

### Regular set

A language that can be defined by a regular expression is called a regular set. If two regular expressions  $r$  and  $s$  denote the same regular set, we say they are equivalent and write  $r = s$ .

There are a number of algebraic laws for regular expressions that can be used to manipulate into equivalent forms.

For instance,  $r|s = s|r$  is commutative;  $r|(s|t) = (r|s)|t$  is associative.

### Regular Definitions

Giving names to regular expressions is referred to as a Regular definition. If  $\Sigma$  is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form

$$d_1 \rightarrow r_1$$

$$d_2 \rightarrow r_2$$

.....

$$d_n \rightarrow r_n$$

1. Each  $d_i$  is a distinct name.
2. Each  $r_i$  is a regular expression over the alphabet  $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$ .

**Example:** Identifiers is the set of strings of letters and digits beginning with a letter. Regular definition for this set:

$$\text{letter} \rightarrow A | B | \dots | Z | a | b | \dots | z$$

$$\text{digit} \rightarrow 0 | 1 | \dots | 9$$

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

### Shorthands

Certain constructs occur so frequently in regular expressions that it is

convenient to introduce notational shorthands for them.

### **1. One or more instances (+):**

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

### **2. Zero or one instance ( ?):**

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

### **3. Character Classes:**

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

## **Non-regular Set**

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

## **RECOGNITION OF TOKENS**

Consider the following grammar fragment:

$\text{stmt} \rightarrow \text{if expr then stmt}$

$| \text{if expr then stmt}$   
 $\quad | \text{else stmt } |\epsilon$

$\text{expr} \rightarrow \text{term relop term}$

$|\text{term}$

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

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

$\text{if } \rightarrow \text{If}$   
 $\text{then } \rightarrow \text{Then}$   
 $\text{else } \rightarrow \text{Else}$

relop → <|<=|=|>|>|=  
 id → letter(letter|digit)\*  
       digit+ (.digit+)?(E(+|-)  
 Num → )?digit+)??

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

### Transition diagrams

- ❖ As an intermediate step in the construction of a lexical Analyzer we first produce a stylized flowchart called a transition diagram.
- ❖ This Transition diagram are deterministic.
- ❖ One state is labeled as start state where control resides when we begin to recognize a token.
- ❖ On entering a state we read the next input character.
- ❖ If there is an edge from the current state whose label matches this character.
- ❖ We then go to the state pointed by the edge.Otherwise we indicate failure.
- ❖ Since keywords are sequence of letters ,they are exceptions to the rule that asequence of letters and digits starting with a letter is an identifier.
- ❖ When the accepting state is reached we execute some code to determine if the lexeme leading to the accepting state is a keyword or an identifier.
- ❖ The return statement next to the accepting state uses  
Gettoken() → to obtain the token.  
Install-id() → to obtain the attribute values to be returned.
- ❖ The symbol table is examined and if the lexeme is found there marked as a keyword install-id() returns 0.
- ❖ If the lexeme is found and is a variable install-id() returns a pointer to the symbol table entry.
- ❖ If the lexeme is not found in the symbol table it is installed as a variable and a pointer to the newly created entry is returned.

## Transition diagram for relational operators

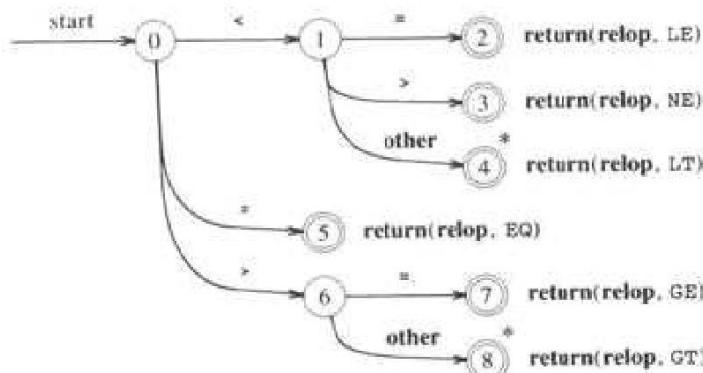
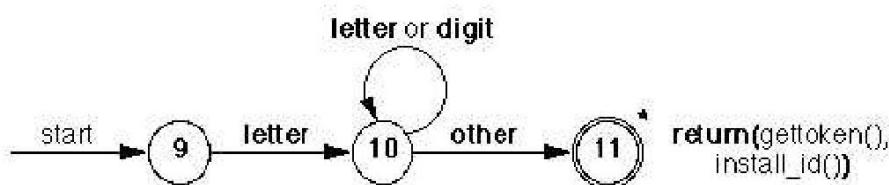


Fig. 3.12. Transition diagram for relational operators.

## Transition diagram for identifiers and keywords



## 4. A LANGUAGE FOR SPECIFYING LEXICAL ANALYZER

- ❖ Explain a language for specifying lexical Analyzer.(8).[Nov/Dec 2014]

There is a wide range of tools for constructing lexical analyzers.

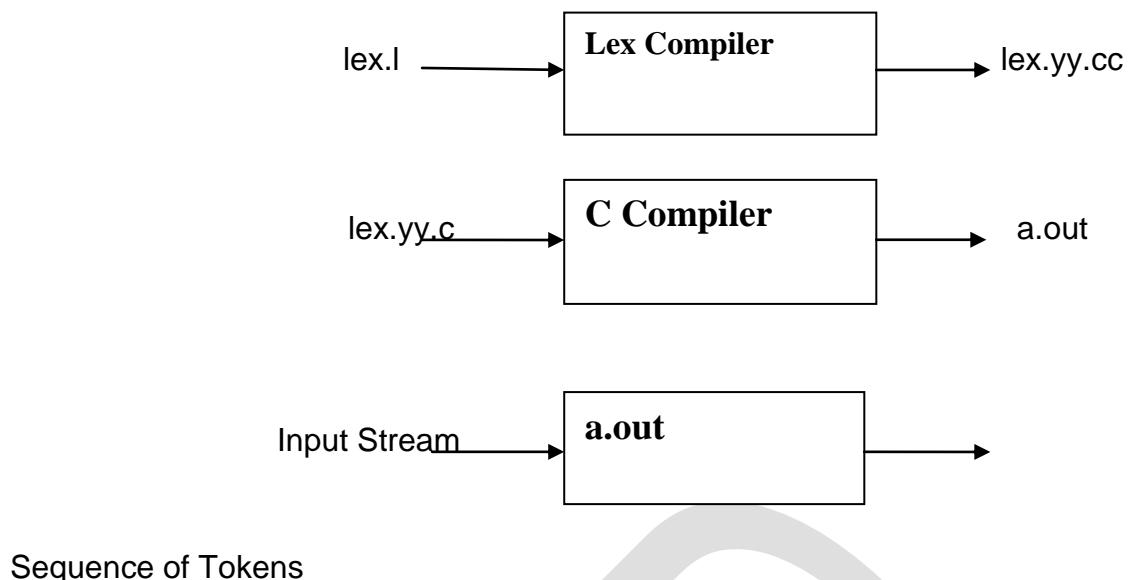
- Lex
- YACC

### LEX

Lex is a computer program that generates lexical analyzers. Lex is commonly used with the yacc parser generator.

### Creating a lexical analyzer

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



### **Lex Specification**

A Lex program consists of three parts:

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

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

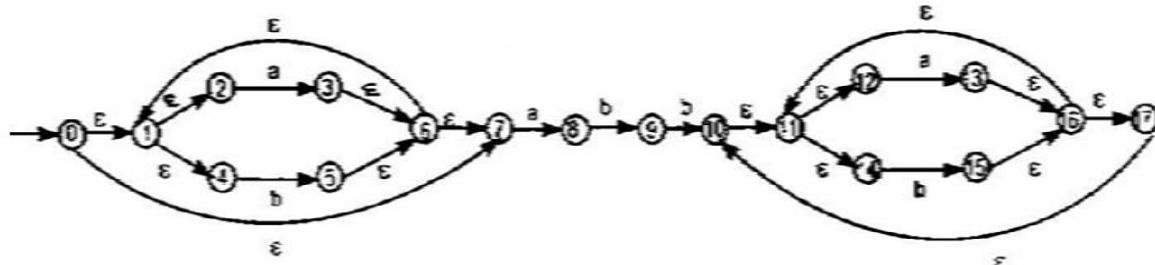
### **YACC- YET ANOTHER COMPILER-COMPILER**

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

## 5. REGULAR EXPRESSION TO DFA

**Construct a NFA using Thompson's construction algorithm for the regular expression  $(a|b)^*abb(a|b)^*$  and convert it into DFA. [May /June 2016] [Nov/Dec 2016].**

NFA:



The start state of DFA is E-closure (0)

$$\text{E-closure } (0) = \{0, 1, 2, 4, 7\} = A$$

The input symbol alphabet is {a, b}

$$\begin{aligned} \text{E-closure } (\text{move}(A, a)) &= \text{E-closure } (\text{move}(\{0, 1, 2, 4, 7\}, a)) \\ &= \text{E-closure } (3, 8) = \{1, 2, 3, 4, 6, 7, 8\} = B \end{aligned}$$

$$\text{DTrans } [A, a] = B$$

$$\begin{aligned} \text{E-closure } (\text{move}(A, b)) &= \text{E-closure } (\text{move}(\{0, 1, 2, 4, 7\}, b)) \\ &= \text{E-closure } (5) = \{1, 2, 4, 5, 6, 7\} = C \end{aligned}$$

$$\text{DTrans } [A, b] = C$$

$$\begin{aligned} \text{E-closure } (\text{move}(B, a)) &= \text{E-closure } (\text{move}(\{1, 2, 3, 4, 6, 7, 8\}, a)) \\ &= \text{E-closure } (3, 8) = \{1, 2, 3, 4, 6, 7, 8\} = B \end{aligned}$$

$$\text{DTrans } [B, a] = B$$

$$\begin{aligned} \text{E-closure } (\text{move}(B, b)) &= \text{E-closure } (\text{move}(\{1, 2, 3, 4, 6, 7, 8\}, b)) \\ &= \text{E-closure } (5, 9) = \{1, 2, 4, 5, 6, 7, 9\} = C \end{aligned}$$

$$\text{DTrans } [B, b] = C$$

$$\begin{aligned} \text{E-closure } (\text{move}(C, a)) &= \text{E-closure } (\text{move}(\{1, 2, 4, 5, 6, 7, 8\}, a)) \\ &= \text{E-closure } (3, 8) = B \end{aligned}$$

$$\text{DTrans } [C, a] = B$$

$$\begin{aligned} \text{E-closure } (\text{move}(C, b)) &= \text{E-closure } (\text{move}(\{1, 2, 4, 5, 6, 7, 8\}, b)) \\ &= \text{E-closure } (5) = C \end{aligned}$$

$$\text{DTrans } [C, b] = C$$

$$\begin{aligned} \text{E-closure } (\text{move}(D, a)) &= \text{E-closure } (\text{move}(\{1, 2, 4, 5, 6, 7, 9\}, a)) \\ &= \text{E-closure } (3, 8) = B \end{aligned}$$

$$\text{DTrans } [D, a] = B$$

$$\begin{aligned} \text{E-closure } (\text{move}(D, b)) &= \text{E-closure } (\text{move}(\{1, 2, 4, 5, 6, 7, 9\}, b)) \\ &= \text{E-closure } (5, 10) \\ &= \{1, 2, 4, 5, 6, 7, 10, 11, 12, 14, 17\} = E \end{aligned}$$

$$\text{DTrans } [D, b] = E$$

$$\begin{aligned} \text{E-closure } (\text{move}(E, a)) &= \text{E-closure } (\text{move}(\{1, 2, 4, 5, 6, 7, 10, 11, 12, 14, 17\}, a)) \\ &= \text{E-closure } (3, 8, 13) = \{1, 2, 3, 4, 6, 7, 8, 11, 12, 13, 14, 16, 17\} \\ &= F \end{aligned}$$

DTrans [E, a] = F

E-closure (move(E, b)) = E-closure (move({1, 2, 4, 5, 6, 7, 10, 11, 12, 14, 17}, b))  
= E-closure (5, 15) = {1, 2, 4, 5, 6, 7, 11, 12, 14, 15, 16, 17} = G

DTrans [E, b] = G

E-closure (move(F, a)) = E-closure (3, 8, 13) = F

DTrans [F, a] = F

E-closure (move(F, b)) = E-closure (5, 9, 15)  
= {1, 2, 4, 5, 6, 7, 9, 11, 12, 14, 15, 16, 17} = H

DTrans [F, b] = H

E-closure (move(G, a)) = E-closure (3, 8, 13) = F

DTrans [G, a] = F

E-closure (move(G, b)) = E-closure (5, 15) = G

DTrans [G, b] = G

E-closure (move(H, a)) = E-closure (3, 8, 13) = F

DTrans [H, a] = F

E-closure (move(H, b)) = E-closure (5, 10, 15)  
= {1, 2, 4, 5, 6, 7, 10, 11, 12, 14, 15, 16, 17} = I

DTrans [H, b] = I

E-closure (move(I, a)) = E-closure (3, 8, 13) = F

DTrans [I, a] = F

E-closure (move(I, b)) = E-closure (5, 15) = G

DTrans [I, b] = G

#### Transition table for DFA:

State	Input symbol	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E *	F	G
F *	F	H
G *	F	G
H *	F	I
I *	F	G

**5 b ) Write an algorithm for minimizing the number of states of a DFA. [Nov / Dec2016]**

Input : A DFA M with set o states S. Inputs £ .Transitions defined for all states and inputs,initial state s0 and set of final states F.

Output : A DFA M' accepting the same language as M and having as few states as possible .

One important result on finite automata, both theoretically and practically, is that for any regular language there is a unique DFA having the smallest number of states that accepts it. Let  $M = \langle Q, \Sigma, q_0, \delta, A \rangle$  be a DFA that accepts a language  $L$ . Then the following algorithm produces the DFA, denote it by  $M_1$ , that has the smallest number of states among the DFAs that accept  $L$ .

### **Minimization Algorithm for DFA**

**Construct a partition  $\Pi = \{ A, Q - A \}$  of the set of states  $Q$  ;**

$\Pi_{\text{new}} := \text{new\_partition}(\Pi)$  ;

**while** ( $\Pi_{\text{new}}$   $\neq$   $\Pi$ )

$\Pi := \Pi_{\text{new}}$  ;

$\Pi_{\text{new}} := \text{new\_partition}(\Pi)$

$\Pi_{\text{final}} := \Pi$ ;

**function new\_partition( $\Pi$ )**

**for each set  $S$  of  $\Pi$  do**

partition  $S$  into subsets such that two states  $p$  and  $q$  of  $S$  are in the same subset of  $S$

if and only if for each input symbol,  $p$  and  $q$  make a transition to (states of) the same set of  $\Pi$ .

The subsets thus formed are sets of the output partition in place of  $S$ .

If  $S$  is not partitioned in this process,  $S$  remains in the output partition.

**end**

Minimum DFA  $M_1$  is constructed from  $\Pi_{\text{final}}$  as follows:

- Select one state in each set of the partition  $\Pi_{\text{final}}$  as the representative for the set. These representatives are states of minimum DFA  $M_1$ .
  - Let  $p$  and  $q$  be representatives i.e. states of minimum DFA  $M_1$ . Let us also denote by  $p$  and  $q$  the sets of states of the original DFA  $M$  represented by  $p$  and  $q$ , respectively. Let  $s$  be a state in  $p$  and  $t$  a state in  $q$ . If a transition from  $s$  to  $t$  on symbol  $a$  exists in  $M$ , then the minimum DFA  $M_1$  has a transition from  $p$  to  $q$  on symbol  $a$ .
  - The start state of  $M_1$  is the representative which contains the start state of  $M$ .
  - The accepting states of  $M_1$  are representatives that are in  $A$ .
- Note that the sets of  $\Pi_{\text{final}}$  are either a subset of  $A$  or disjoint from  $A$ .

Remove from  $M_1$  the dead states and the states not reachable from the start state, if there are any. Any transitions to a dead state become undefined.

A state is a dead state if it is not an accepting state and has no out-going transitions except to itself.

## UNIT-III SYNTAX ANALYSIS

Need and Role of the Parser-Context free Grammars-Top Down Parsing-General Strategies-Recursive Descent Parser-Predictive Parser-LL(1) Parser-Shift Reduce Parser-LR Parser-LR(0) Item-Construction of SLR Parsing Table-Introduction to LALR Parser-Error Handling and Recovery in Syntax Analyzer.-YACC-Design of a syntax Analyzer for a sample Language.

### PART-A

#### **1.What are the three general types of parsers? Differentiate them.**

1. Universal parsing methods
2. Top down parsing
3. Bottom up parsing

- Universal parsing methods are too inefficient to use in production compilers and can parse any grammar
- Top down parsers build parse trees from the top (root) to the bottom (leaves).
- Bottom up parser build parse trees from the leaves and work up to the root.
- Top down and bottom up parser can parse only sub classes of Grammar such as LL Grammar and LR Grammar. These methods are efficient.

#### **2.What are the goals of error handler?**

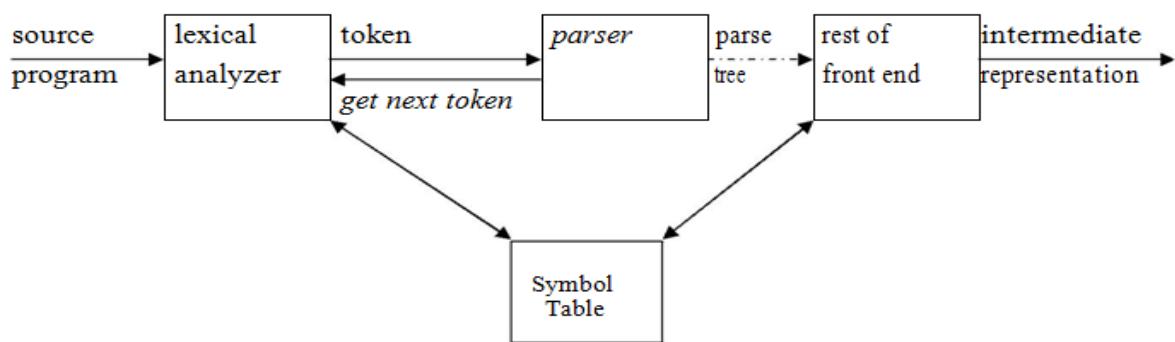
- It should report the presence of errors clearly and accurately
- It should recover from each error quickly enough to be able to detect subsequent errors
- It should not significantly slow down the processing of correct programs

#### **3.What is the Role of the Parser?**

The parser or syntactic analyzer obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language.

It reports any syntax errors in the program. It also recovers from commonly occurring errors so that it can continue processing its input.

#### ***Position of parser in compiler model***



The output of the parser is parse tree for the stream of tokens produced by the lexical Analyzer.

#### 4. What are the error recovery strategies in Parsing?

- Panic mode
- Phrase level
- Error productions
- Global corrections.

#### 5. What is meant by ambiguous grammar? [May/June 2012,2016] [Nov / Dec 2016]

A grammar that produces more than one parse tree, or more than one left most derivation or more than one right most derivation for some sentence is said to be ambiguous.

##### Example:

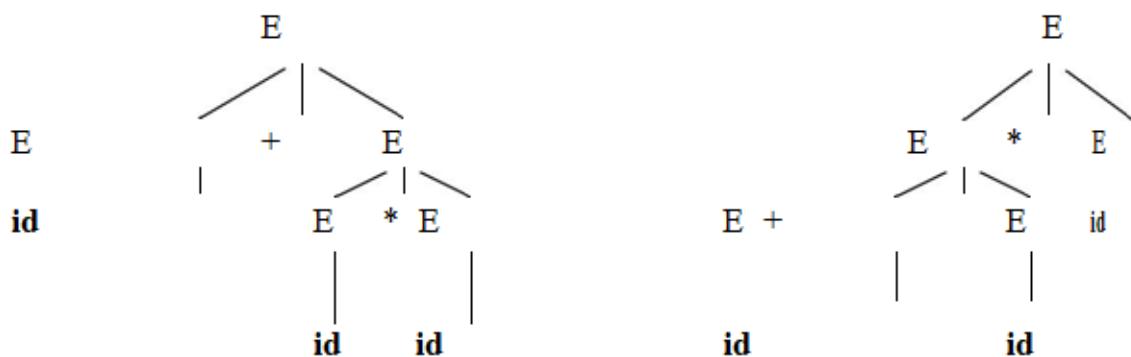
Grammar G :  $E \rightarrow E+E \mid E^*E \mid ( E ) \mid - E \mid id$  is ambiguous.

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

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

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

The two corresponding parse trees are :



#### 6. What is left recursive grammar? How will you eliminate the left recursive grammar and eliminate left recursion for the following grammar? [Nov/Dec 2013]

$E \rightarrow E + T \mid T$        $T \rightarrow T * F \mid F$        $F \rightarrow (E) \mid id$

##### Left Recursive Grammar:

A Grammar is left recursive if it has a non terminal A such that there is a derivation

$A \xrightarrow{+} A\alpha$  for some string  $\alpha$

##### Rule to eliminate Left recursive Grammar:

Left recursive pair of productions

$A \xrightarrow{+} A\alpha \mid \beta$

Would be replaced by the non left recursive productions

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' | \epsilon$$

without changing the set of strings derivable from A.

Non left recursive grammar:

$$E \rightarrow TE'$$

$$T \rightarrow FT'$$

$$F \rightarrow (E) | id$$

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

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

## 7. What is left factoring? [Nov/Dec 2011, Nov/Dec 2009]

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. The idea is that when it is not clear which of two alternative productions to expand a non terminal A.

If  $A \rightarrow \alpha\beta_1 | \alpha\beta_2$  are two A productions and the input begins with the non empty string derived from  $\alpha$ ,

Left factored the original productions become

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 | \beta_2$$

## 8. Define FIRST( $\alpha$ ) and FOLLOW(A)? What are the rules to compute FIRST(X) and FOLLOW(A)? [May /June 2016]

**FIRST( $\alpha$ ):**

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

**FOLLOW(A):**

FOLLOW(A for non terminal A to be the set of terminals a that can appear immediately to the right of A in some sentential form, that is, the set of terminals a such that there exists a derivation of the form  $S \Rightarrow^* \alpha A a \beta$  for some  $\alpha$  and  $\beta$

**Rules to compute FIRST(X):**

1. If X is a terminal then FIRST(X) is {X}
2. If  $X \rightarrow \epsilon$  is a production then add  $\epsilon$  to FIRST(X).
3. If X is a non terminal and  $X \rightarrow Y_1 Y_2 \dots Y_K$  is a production then place a in FIRST(X) if for some I, a is in FIRST( $Y_i$ )

**Rules to compute FOLLOW(A):**

1. Place \$ in FOLLOW(S) where S is the start symbol and \$ is the right end marker.
2. If there is a production  $A \rightarrow \alpha B \beta$  then everything in FIRST( $\beta$ ) except for  $\epsilon$  is places in FOLLOW(B).
3. If there is a production  $A \rightarrow \alpha B$  or a production  $A \rightarrow \alpha B \beta$  where FIRST( $\beta$ ) contains  $\epsilon$  then everything in FOLLOW(A) is in FOLLOW(B).

## 9. List the factors to be considered in Top down Parsing.

Top down parsing is an attempt to find a left most derivation for an input string.

Factors to be considered in Top down Parsing:

- Left recursive grammar can cause a top down parser to go into an infinite loop on writing procedure. So the Grammar should be non-left recursive for top-down parsing.
- Backtracking overhead may occur. Due to backtracking , it may reject some valid sentences. So the grammar should be left factored Grammar for top down parsing.

## 10. Define Handle and handle pruning? [Apr/May 2011, Nov/Dec 2011] [Nov/Dec 2016].

**Handle :**

A handle of a string is a sub string that matches the right side of a production and whose reduction to the non terminal on the left side of the production represents one step along the reverse of a right most derivation.

**Handle Pruning:**

A right most derivation in reverse can be obtained by “handle pruning”.If w is a sentence of the grammar then  $w = \gamma_n$  , where  $\gamma_n$  is the right sentential form of some right most derivation.

$$S = \gamma_1 \Rightarrow \gamma_2 \Rightarrow \gamma_3 \dots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = w$$

To reconstruct this derivation in reverse order we locate the handle  $\beta_n$  in  $\gamma_n$  and replace  $\beta_n$  by the left side of a production  $A_n \rightarrow \beta_n$  to obtain the (n-1)st right sentential form  $\gamma_{n-1}$ .

## 11.Define Viable prefixes with an example.

The set of prefixes of right sentential forms that can appear on the stack of a shift reduce parser are called viable prefixes. It is a prefix of a right sentential form that does not continue past the right end of the right most handle of that sentential form.

## 12. Differentiate sentence and Sentential Form.

Sentence	Sentential Form
If $S^* \Rightarrow w$ then the string w is called Sentence of G	If $S^* \Rightarrow \alpha$ then $\alpha$ is a sentential form of G
Sentence is a string of terminals. Sentence is a sentential form with no Non-Terminals.	Sentential form is a string of terminals and Non-Terminals.

### 13. Define LR(0) item or item and define the two classes of item.

#### LR(0) Item:

An LR(0) item (item for short) of a grammar G is a production of G with a dot at some position    of the right side Thus production  $A \rightarrow XYZ$  yields four items

$$\begin{aligned} A &\rightarrow .XYZ \\ A &\rightarrow X .YZ \\ A &\rightarrow XY .Z \\ A &\rightarrow XYZ . \end{aligned}$$

#### Two classes of Item:

All the sets of items can be divided into two classes of items

##### 1. Kernal items:

It includes the initial item  $S' \rightarrow .S$  and all items whose dots are not at the left end.

##### 2. Nonkernal items:

These have their dots at the left end.

### 14. Define closure operation and Goto operation?

#### Closure

If I is a set of items for Grammar G then closure(I) is the set of items constructed from I by the two rules.

1. Initially every item in I is added to closure(I)
2. If  $A \rightarrow \alpha . B\beta$  is in closure(I) and  $B \rightarrow \gamma$  is a production then add the item  $B \rightarrow .\gamma$  to I if it is not already there. This rule is applied until no more items can be added to closure(I).

#### Goto:

goto(I,X) is defined to be the closure of the set of all items  $[A \rightarrow \alpha X . \beta]$  such that  $[A \rightarrow \alpha . X\beta]$  is in I. Where I is the set of items and X is a grammar symbol.

### 15. Why SLR and LALR are more economical than Canonical LR Parser.

For a comparison of parser size ,the SLR and LALR tables for a grammar always have the same number of states and this number is typically several hundred states for a language like Pascal.The canonical LR table would typically have several thousand states for the same size language.Thus it is much easier and more economical to construct SLR and LALR tables than the Canonical LR tables.

## PART-B

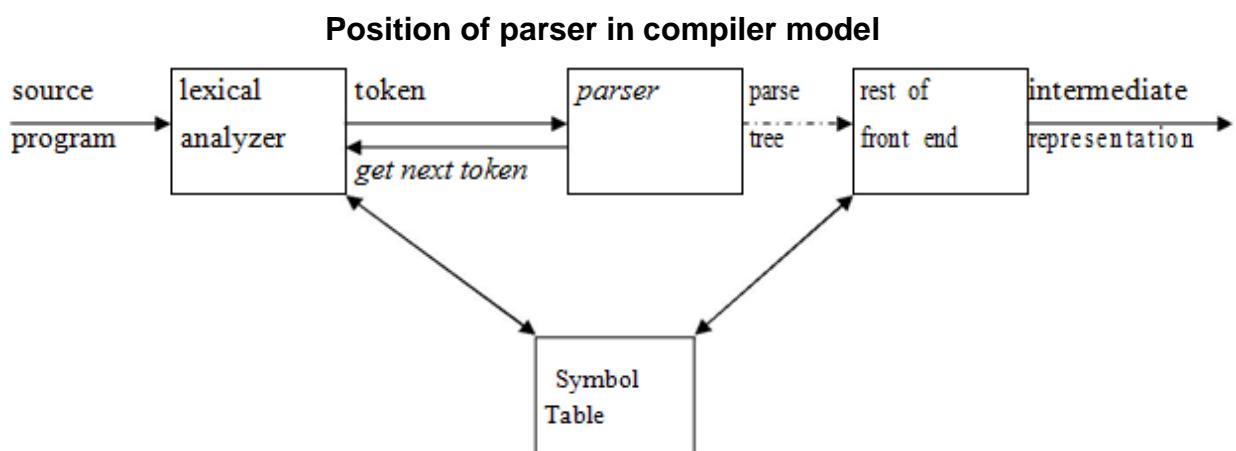
### 1.a.NEED AND ROLE OF THE PARSER

- ❖ Discuss the Role of the Parser in detail. [ Apr/May 2007, Apr/May 2015]
- ❖ Explain the error recovery strategies in syntax analysis.(8)[Apr/May 2011]

## THE ROLE OF PARSER

The parser or syntactic analyzer obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language.

It reports any syntax errors in the program. It also recovers from commonly occurring errors so that it can continue processing its input.



The output of the parser is parse tree for the stream of tokens produced by the lexical Analyzer.

### The Tasks of Parser

- Collecting informations about various tokens into the symbol table.
- Performing type checking.
- Semantic Analysis.

### Issues :

Parser cannot detect errors such as:

1. Variable re-declaration
2. Variable initialization before use.
3. Data type mismatch for an operation.

The above issues are handled by Semantic Analysis phase.

### Syntax error handling :

Programs can contain errors at many different levels. For example :

1. Lexical, such as misspelling a keyword.
2. Syntactic, such as an arithmetic expression with unbalanced parentheses.
3. Semantic, such as an operator applied to an incompatible operand.
4. Logical, such as an infinitely recursive call.

### The goals of error handler in a Parser:

1. It should report the presence of errors clearly and accurately.
2. It should recover from each error quickly enough to be able to detect subsequent errors.

3. It should not significantly slow down the processing of correct programs.

### **Error recovery strategies :**

The different strategies that a parse uses to recover from a syntactic error are:

1. Panic mode
2. Phrase level
3. Error productions
4. Global correction

#### **Panic mode recovery:**

On discovering an error, the parser discards input symbols one at a time until a synchronizing token is found. The synchronizing tokens are usually delimiters, such as semicolon or **end**. It has the advantage of simplicity and does not go into an infinite loop. When multiple errors in the same statement are rare, this method is quite useful.

#### **Phrase level recovery:**

On discovering an error, the parser performs local correction on the remaining input that allows it to continue. Example: Insert a missing semicolon or delete an extraneous semicolon etc.

#### **Error productions:**

The parser is constructed using augmented grammar with error productions. If an error production is used by the parser, appropriate error diagnostics can be generated to indicate the erroneous constructs recognized by the input.

#### **Global correction:**

Given an incorrect input string  $x$  and grammar  $G$ , certain algorithms can be used to find a parse tree for a string  $y$ , such that the number of insertions, deletions and changes of tokens is as small as possible. However, these methods are in general too costly in terms of time and space.

## **1. b. VARIOUS TERMINOLOGIES IN PARSING**

### **❖ Define the various Terminologies in Parsing.**

#### **a. Parsing:**

Parsing is the process of determining if a string of tokens can be generated by a Grammar. Parsing may be

- Top Down Parsing.
- Bottom up Parsing.

#### **b. Top Down Parsing:**

- Parsing method in which the construction of parse tree starts at the root and proceeds toward the leaves is called as top down parsing.
- Top Down Parsing method finds the left most derivation for a string of tokens.

#### **c. Bottom up Parsing:**

- Parsing method in which the construction of parse tree starts at the leaves and proceeds towards the root is called as Bottom up Parsing.
- Bottom Up Parsing method finds the right most derivation in reverse for a string of tokens.

#### **d. Recursive Descent Parsing:**

The general form of top down parsing called recursive descent parsing may involve backtracking that is making repeated scans of the input. It can be viewed as an attempt to construct a parse tree for the input string starting from the root and creating the nodes of the parse tree in preorder.

#### **e. Predictive Parsing:**

A special form of recursive descent Parsing in which the look ahead symbol unambiguously determines the procedures selected for each non terminal ,where no backtracking is required .It consist of stack, input buffer and finite control and tries to find the left most derivation.

#### **f. Shift Reduce Parsing:**

A general style of bottom up syntax analysis which attempts to construct a parse tree for an input string beginning at the leaves and working up towards the root.

#### **g. LR(k) Parsing :**

The “L” is for left to right scanning of the input ,the “R” for constructing a right most derivation in reverse and the k for the number of input symbols of look ahead that are used in making parsing decisions.

#### **h. LL(1) Grammar:**

A grammar whose parsing table has multiply defined entries is said to be LL(1) grammar.

L - Left most Derivation.

L - Input scanned from left to right.

1- One input symbol used as a look ahead symbol.

### **2.a.CONTEXT FREE GRAMMARS**

- ❖ **What is Context Free Grammar? Write short notes on derivations and ambiguous Grammar. [May /June 2016]**
- ❖ **Explain ambiguous grammar G : E->E+E|E\*E|(E)|-E|id for the sentence id+id\*id (8) [May /June 2014] [Nov/Dec 2016].**

### **CONTEXT-FREE GRAMMARS**

A Context-Free Grammar is a quadruple that consists of **terminals**, **non-terminals**, **start symbol** and **productions**.

**Terminals** : These are the basic symbols from which strings are formed.

**Non-Terminals** : These are the syntactic variables that denote a set of strings. These help to define the language generated by the grammar.

**Start Symbol** : One non-terminal in the grammar is denoted as the “Start-symbol” and the set of strings it denotes is the language defined by the grammar.

**Productions :** It specifies the manner in which terminals and non-terminals can be combined to form strings. Each production consists of a non-terminal, followed by an arrow, followed by a string of non-terminals and terminals.

**Example of context-free grammar:** The following grammar defines **simple arithmetic expressions**:

$expr \rightarrow expr \ op \ expr$

$expr \rightarrow ( \ expr )$

$expr \rightarrow -$

$expr$

$expr \rightarrow id$

$op \rightarrow + | - | * | /$

$op \rightarrow$

↑

In this grammar,

- id + - \* /** ↑ ( ) are terminals.
- expr , op** are non-terminals.
- expr** is the start symbol.
- Each line is a production.

### Derivations:

Two basic requirements for a grammar are :

1. To generate a valid string.
2. To recognize a valid string.

**Derivation** is a process that generates a valid string with the help of grammar by replacing the non-terminals on the left with the string on the right side of the production.

**Example :** Consider the following grammar for arithmetic expressions :

$E \rightarrow E+E \mid E^*E \mid ( E ) \mid -E \mid id$

To generate a valid string - (id+id) from the grammar the steps are

- $E \rightarrow -E$
- $E \rightarrow - (E)$
- $E \rightarrow - (E+E)$
- $E \rightarrow - (id+E)$
- $E \rightarrow - (id+id)$

In the above derivation,

- $E$  is the start symbol.
- (id+id) is the required sentence (only terminals).
- Strings such as  $E$ ,  $-E$ ,  $-(E)$ , . . . are called sentinel forms.

### Types of derivations:

The two types of derivation are:

1. Left most derivation
  2. Right most derivation.
- In leftmost derivations, the leftmost non-terminal in each sentinel is always

chosen first for replacement.

- In rightmost derivations, the rightmost non-terminal in each sentinel is always chosen first for replacement.

### **Example:**

Given grammar  $G : E \rightarrow E+E | E^*E | (E) | -E | id$

Sentence to be derived : – (id+id)

LEFTMOST DERIVATION

$$\begin{aligned} E &\rightarrow -E \\ E &\rightarrow -(E) \\ E &\rightarrow -(E+E) \\ E &\rightarrow -(id+E) \\ E &\rightarrow -(id+id) \end{aligned}$$

- 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 .The yield is the leaf nodes which are read from left to right.

### **Ambiguity:**

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

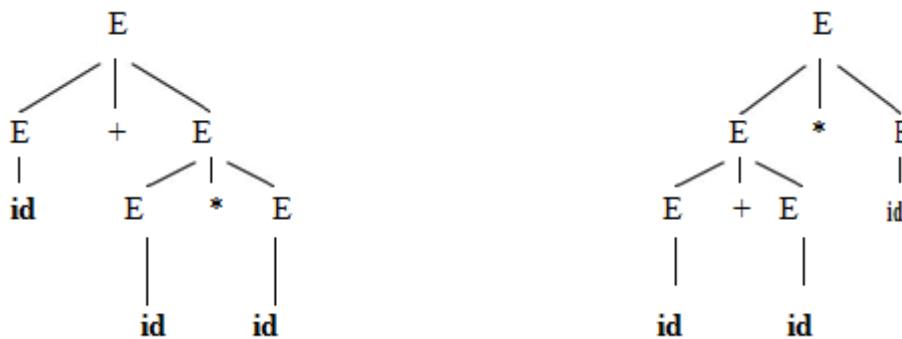
Example : Given grammar  $G : E \rightarrow E+E | E^*E | (E) | -E | id$

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

$$\begin{aligned} E &\rightarrow E+E \\ E &\rightarrow id + E \\ E &\rightarrow id + E * E \\ E &\rightarrow id + id * E \\ E &\rightarrow id + id * id \end{aligned}$$

$$\begin{aligned} E &\rightarrow E^* E \\ E &\rightarrow E + E * E \\ E &\rightarrow id + E * E \\ E &\rightarrow id + id * E \\ E &\rightarrow id + id * id \end{aligned}$$

The two corresponding parse trees are :



## 2.b WRITING A GRAMMAR

- ❖ Write down the algorithm to eliminate left recursion and left factoring and apply both to the following Grammar  $E \rightarrow E+T \mid T$ ,  $T \rightarrow T^*F \mid F$ ,  $F \rightarrow (E) \mid id$ .(8) [Apr/May 2015]
- ❖ Write down the method to eliminate the ambiguity of a Grammar.

### **WRITING A GRAMMAR**

There are four categories in writing a grammar :

1. Regular Expression Vs Context Free Grammar
2. Eliminating ambiguous grammar.
3. Eliminating left-recursion
4. Left-factoring.

Each parsing method can handle grammars only of a certain form hence, the initial grammar may have to be rewritten to make it parsable.

### **Reasons for using regular expressions to define the lexical syntax of a language.**

- The lexical rules of a language are simple and RE is used to describe them.
- Regular expressions provide a more concise and easier to understand notation for tokens than grammars.
- Efficient lexical analyzers can be constructed automatically from RE than from grammars.
- Separating the syntactic structure of a language into lexical and nonlexical parts provides a convenient way of modularizing the front end into two manageable-sized components

### **Regular Expressions vs. Context-Free Grammars:**

REGULAR EXPRESSION	CONTEXT-FREE GRAMMAR
It is used to describe the tokens of programming languages.	It consists of a quadruple where $S \rightarrow$ start symbol, $P \rightarrow$ production, $T \rightarrow$ terminal, $V \rightarrow$ variable or non-terminal.
It is used to check whether the given input is valid or not using <b>transition diagram</b> .	It is used to check whether the given input is valid or not using <b>derivation</b> .

The transition diagram has set of states and edges.	The context-free grammar has set of productions.
It has no start symbol	It has start symbol.
It is useful for describing the structure of lexical constructs such as identifiers, constants, keywords, and so forth.	It is useful in describing nested structures such as balanced parentheses, matching begin-end's and so on.

### Eliminating ambiguity:

Ambiguity of the grammar that produces more than one parse tree for leftmost or rightmost derivation can be eliminated by re-writing the grammar.

To eliminate ambiguity the general rule is

"Match each **else** with the closest previous unmatched **then**."

The idea is that a statement appearing between a **then** and an **else** must be matched.

i.e. It must not end with an unmatched **then** followed by any statement.

To eliminate ambiguity, the following grammar may be used:

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

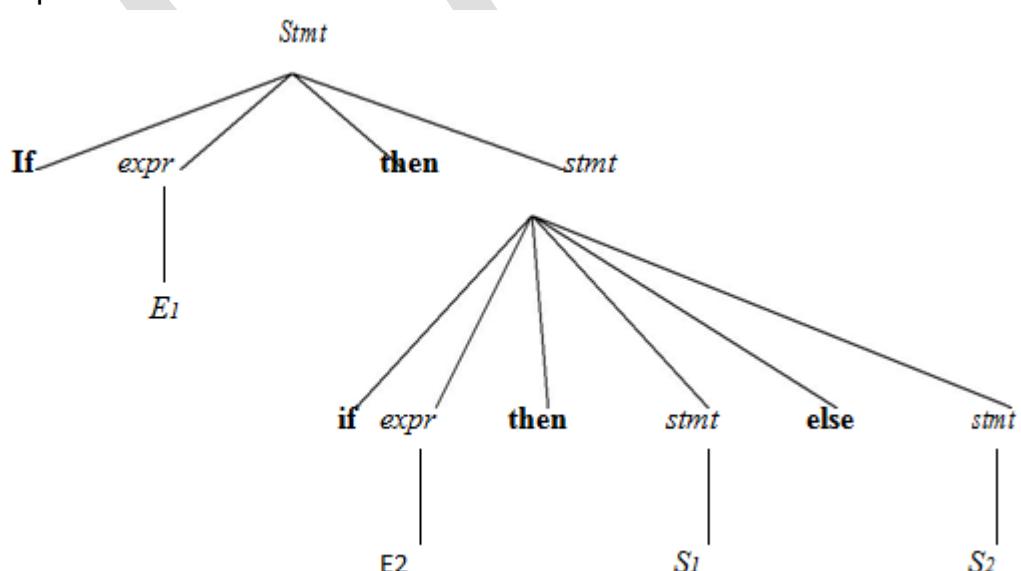
$matched\_stmt \rightarrow if \ expr \ then \ matched\_stmt \ else \ matched\_stmt \mid other$

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

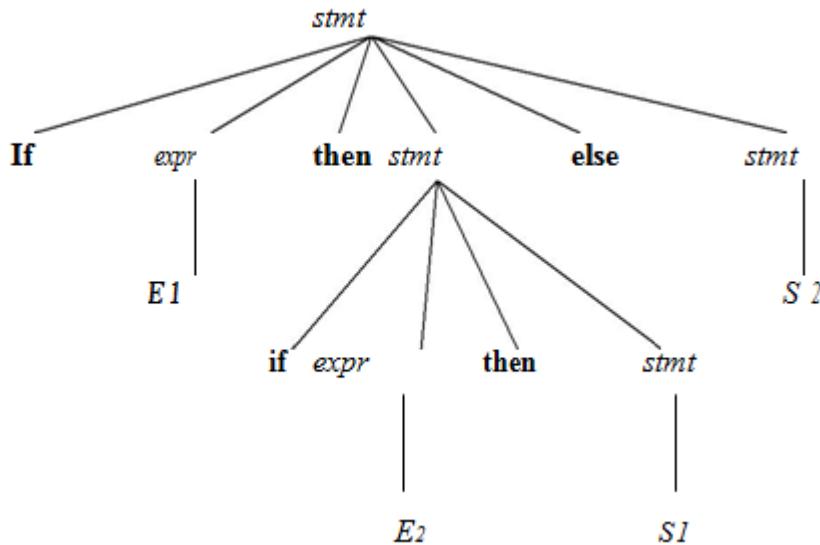
Consider this example, G:  $stmt \rightarrow if \ expr \ then \ stmt \mid if \ expr \ then \ stmt \ else \ stmt \mid other$

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.



### 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**

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \epsilon$$

without changing the set of strings derivable from  $A$

**Example :** Consider the following grammar for arithmetic expressions:

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

$$T \rightarrow T * F$$

$$\mid F$$

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

First eliminate the left recursion for  $E$  as

$$E \rightarrow TE'$$

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

Then eliminate for

as

$$T \rightarrow FT'$$

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

Thus the obtained grammar after eliminating left recursion is

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

$$\epsilon$$

$F \rightarrow (E) \mid id$

**Algorithm to eliminate left recursion:**

1. Arrange the non-terminals in some order A<sub>1</sub>, A<sub>2</sub> . . . A<sub>n</sub>.

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

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \epsilon$

$E \rightarrow b$

### 3.RECURSIVE DESCENT PARSER

- ❖ Describe the Recursive Descent Parser with an example. [Apr/May 2008]
- ❖ Construct parse tree for the input w= cad using top down parser. S->cAd    A->ab| a [Nov/Dec 2016].

#### **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 :

$S \rightarrow cAd$

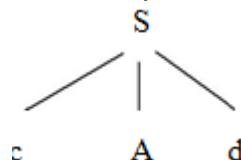
$A \rightarrow ab \mid a$

and the input string  $w=cad$ .

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

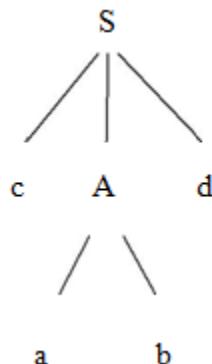
### Step1:

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.



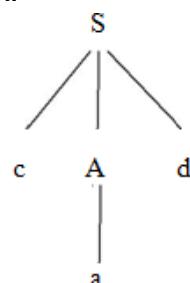
### Step3:

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.

### PREDICTIVE PARSER:

Eliminating left recursion from the grammar and left factoring the resulting grammar we can obtain a grammar that can be parsed by recursive descent parser

that needs no backtracking which is called as predictive parser.

Consider the grammar G :

$$S \rightarrow cAd$$

$$A \rightarrow ab \mid a \quad \text{and the input string}$$

$$w=cad.$$

The grammar non-left recursive and left factoring the grammar, the grammar is

$$S \rightarrow cAd$$

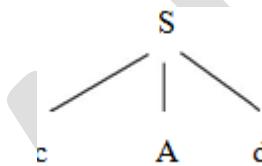
$$A \rightarrow aA'$$

$$A' \rightarrow b \mid \epsilon$$

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

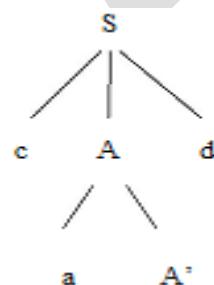
### Step1:

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.



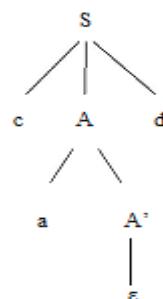
### Step2:

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  $A \rightarrow aA'$



### Step3:

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' and expand A' using  $A' \rightarrow \epsilon$ .



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

#### 4.a.PREDICTIVE PARSER

- ❖ Construct Predictive Parser with an example.[May/Jun 2012] [May /June 2016]

##### 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.

The table-driven predictive parser has an

- Input buffer
- Stack
- A parsing table
- An output stream

##### Input buffer:

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

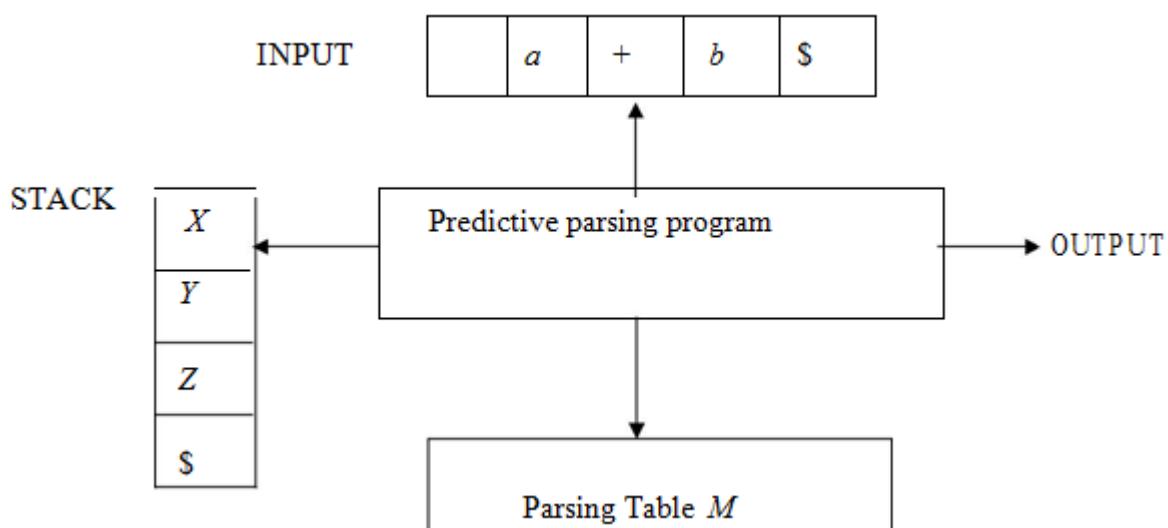
##### Stack:

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.

##### Non-recursive predictive parser



##### 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  $UVW$ .
4. If  $M[X, a] = \text{error}$ , the parser calls an error recovery routine.

### Algorithm for nonrecursive predictive parsing: [May /June 2016]

**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**

**until**  $X = \text{else error}()$

    \$ /\* stack is empty \*/

### 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	$F \rightarrow id$			$F \rightarrow (E)$		

Parsing the input string id + id \* id

STACK	INPUT	OUTPUT
\$E	id+id*id \$	
\$E'T	id+id*id \$	E → TE'
\$E'T'F	id+id*id \$	T → FT'
\$E'T'id	id+id*id \$	F → id
\$E'T'	+id*id \$	
\$E'	+id*id \$	T' → ε
\$E'T+	+id*id \$	E' → +TE'
\$E'T	id*id \$	
\$E'T'F	id*id \$	T → FT'
\$E'T'id	id*id \$	F → id
\$E'T'	*id \$	
\$E'T'F*	*id \$	T' → *FT'
\$E'T'F	id \$	
\$E'T'id	id \$	F → id
\$E'T'	\$	
\$E'	\$	T' → ε
\$	\$	E' → ε

#### 4 b. CONSTRUCTION OF PREDICTIVE PARSING TABLE

- ❖ Describe the algorithm for the construction of predictive Parsing table with an example.[Apr/May 2008] [May /June 2016]
- ❖ Construct parsing table for the grammar and find moves made by predictive parser on input id+id\*id and find FIRST and FOLLOW.

$E \rightarrow E+T \mid T \quad T \rightarrow T^*F \mid F \quad F \rightarrow (E) \mid id$  [Nov/Dec 2016].

##### Predictive parsing table construction:

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

1. FIRST
2. FOLLOW

##### FIRST:

If  $\alpha$  is a string of grammar symbols ,FIRST( $\alpha$ ) is the set of terminals that begin the string derivable from  $\alpha$ .If  $\alpha \Rightarrow \epsilon$  then  $\epsilon$  is also in FIRST( $\alpha$ ).

##### Rules for computing FIRST(X ):

1. If  $X$  is terminal, then FIRST( $X$ ) is  $\{X\}$ .
2. If  $X \rightarrow \epsilon$  is a production, then add  $\epsilon$  to FIRST( $X$ ).
3. If  $X$  is non- terminal and  $X \rightarrow a\alpha$  is a production then add  $a$  to 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 FIRST( $X$ ) if for some  $i$ ,  $a$  is in FIRST( $Y_i$ ), and  $\epsilon$  is in all of FIRST( $Y_1$ ),...,FIRST( $Y_k$ ).

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  $\text{FIRST}(X)$ .

### **FOLLOW :**

$\text{FOLLOW}(A)$  for non-terminal  $A$  is the set of terminals  $A$  that can appear immediately to the right of  $A$  in some sentential form . \*

That is the set of terminals  $a$  such that there exists a derivation of the form  $S \Rightarrow \alpha A a \beta$  for some  $\alpha$  and  $\beta$

### **Rules for computing FOLLOW(A) :**

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  $G$

**Output :** Parsing table  $M$

### **Method :**

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 :

$E \rightarrow E + T \mid T$

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

$F \rightarrow (E) \mid id$

After eliminating left-recursion the grammar is

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

#### **First( ) :**

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

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

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

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

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

#### **Follow()**

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

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

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

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

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

### **Applying the Algorithm for constructing the predictive Parsing Table.**

### **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	$F \rightarrow id$			$F \rightarrow (E)$		

4c.Explain LL(1) grammar for the sentence  $S \rightarrow iEtS \mid iEtSeS \mid a \quad E \rightarrow b$  (May /June 2016)

**Solution:**

The production rule for S needs to be left factored. After left factoring the grammar is,

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

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

$$E \rightarrow b$$

Compute FIRST and FOLLOW for S, S' and E.

$$\text{FIRST}(S) = (I, a)$$

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

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

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

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

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

The predictive parsing table is

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

### 5.a.SHIFT-REDUCE PARSING

- ❖ Describe the conflicts that may occur during shift Reduce Parsing [May/Jun 2012]
- ❖ Describe the shift reduce Parser with an example. [Apr/May 2008] [May /June 2016]

### **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**.

<b>REDUCTION (LEFTMOST)</b>	<b>RIGHTMOST DERIVATION</b>
<b>abbcde</b> ( $A \rightarrow b$ )	<b>S</b> $\rightarrow aABe$
<b>aAbcd</b>	$\rightarrow aAde$
e ( $A \rightarrow Abc$ )	$\rightarrow aAbcde$
aAde ( $B \rightarrow d$ )	$\rightarrow abbcde$
<b>aABe</b> ( $S \rightarrow aABe$ )	
S	

The reductions trace out the right-most derivation in reverse.

### **Handles:**

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

### **Example:**

Consider the grammar:

$$E \rightarrow E+E$$

$$E \rightarrow E^*E$$

$$E \rightarrow (E)$$

$$E \rightarrow id$$

And the input string  $id1+id2^*id3$

The rightmost derivation is :

$$\begin{aligned} E & \Rightarrow \underline{E+E} \\ & \Rightarrow E+\underline{E^*E} \\ & \Rightarrow E+E^*\underline{id3} \\ & \Rightarrow E+\underline{id2}^*id3 \\ & \Rightarrow id1+\underline{id2}^*id3 \end{aligned}$$

In the above derivation the underlined substrings are called **handles**.

### **Handle pruning:**

A rightmost derivation in reverse can be obtained by "**handle pruning**".

(i.e.) if  $w$  is a sentence or string of the grammar at hand, then  $w = \gamma_n$ , where  $\gamma_n$  is the  $n^{th}$  right-sentinel form of some rightmost derivation.

### **Actions in shift-reduce parser:**

- Shift – The next input symbol is shifted onto the top of the stack.

- Reduce – The parser replaces the handle within a stack with a non-terminal.
- Accept – The parser announces successful completion of parsing.
- Error – The parser discovers that a syntax error has occurred and calls an error recovery routine

### Stack implementation of shift-reduce parsing :

Stack	Input	Action
\$	id1+id2*id3 \$	Shift
\$ id1	+id2*id3 \$	reduce by E→id
\$ E	+id2*id3 \$	Shift
\$ E+	id2*id3 \$	Shift
\$ E+id2	*id3 \$	reduce by E→id
\$ E+E	*id3 \$	Shift
\$ E+E*	id3 \$	Shift
\$ E+E*id3	\$	reduce by E→id
\$ E+E*E	\$	reduce by E→ E *E
\$ E+E	\$	reduce by E→ E+E
\$ E	\$	accept

### Conflicts in shift-reduce parsing:

There are two conflicts that occur in shift shift-reduce parsing:

- Shift-reduce conflict:** The parser cannot decide whether to shift or to reduce.
- Reduce-reduce conflict:** The parser cannot decide which of several reductions to make.

#### 1. Shift-reduce conflict:

##### Example:

Consider the grammar:

$E \rightarrow E+E \mid E^*E \mid id$  and input id+id\*id

Stack	Input	Action	Stack	Input	Action
-------	-------	--------	-------	-------	--------

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

#### Reduce-reduce conflict:

Consider the grammar:

$$M \rightarrow R+R \mid R+c \mid R$$

$$R \rightarrow c$$

and input c+c

Stack	Input	Action	Stack	Input	Action
\$	c+c \$	Shift	\$	c+c \$	Shift
\$ c	+c \$	Reduce by R→c	\$ c	+c \$	Reduce by R→c
\$ R	+c \$	Shift	\$ R	+c \$	Shift
\$ R+	c \$	Shift	\$ R+	c \$	Shift
\$ R+c	\$	Reduce by R→c	\$ R+c	\$	Reduce by M→R+c
\$ R+R	\$	Reduce by M→R+R	\$ M	\$	
\$ M	\$				

### 5.b.LR PARSER

- ❖ Explain LR Parser with an example.[Apr/May 2009]

### LR PARSER

An efficient bottom-up syntax analysis technique that can be used to parse a large class of CFG is called LR( $k$ ) parsing. The 'L' is for left-to-right scanning of the input,

the 'R' for constructing a rightmost derivation in reverse, and the 'K' for the number of input symbols. When 'K' is omitted, it is assumed to be 1.

### **Advantages of LR parsing:**

- It recognizes virtually all programming language constructs for which CFG can be written.
- It is an efficient non-backtracking shift-reduce parsing method.
- A grammar that can be parsed using LR method is a proper superset of a grammar that can be parsed with predictive parser.
- It detects asyntactic error as soon as possible.

### **Drawbacks of LR method:**

It is too much of work to construct a LR parser by hand for a programming language grammar. A specialized tool, called a LR parser generator, is needed. Example: YACC.

### **Types of LR parsing method:**

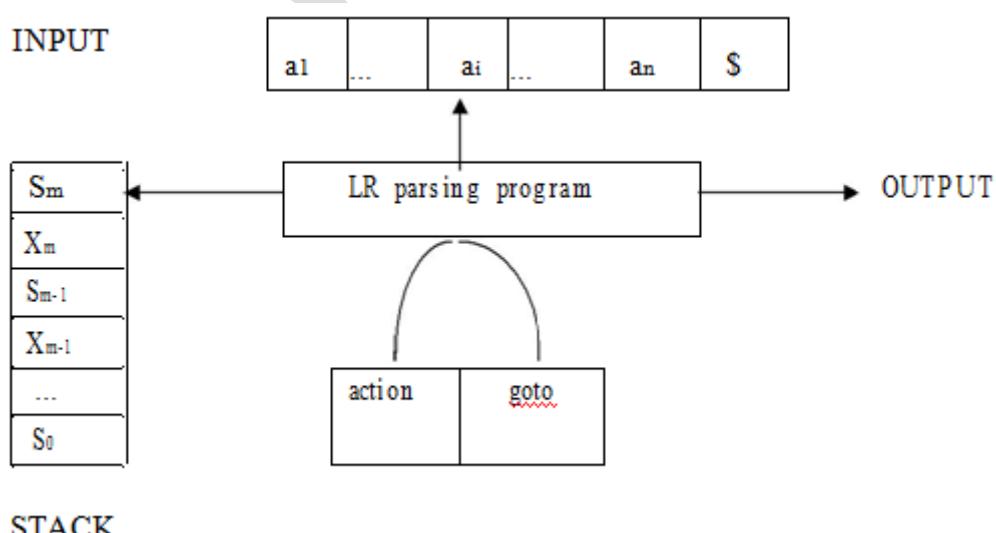
1. SLR- Simple LR
  - Easiest to implement, least powerful.
2. CLR- Canonical LR
  - Most powerful, most expensive.
3. LALR- Look -Ahead LR
  - Intermediate in size and cost between the other two methods.

### **The LR parsing algorithm:**

It consists of :

- An input
- An output
- A stack
- A driver program
- A parsing table that has two parts (*action* and *goto*).

The schematic form of an LR parser is as follows:



- The driver program is the same for all LR parser.

- The parsing program reads characters from an input buffer one at a time.
- The program uses a stack to store a string of the form  $s_0X_1s_1X_2s_2\dots X_m s_m$ , where  $s_m$  is on top. Each  $X_i$  is a grammar symbol and each  $s_i$  is a state.
- The parsing table consists of two parts : *action* and *goto* functions.

**Action** : The parsing program determines  $s_m$ , the state currently on top of stack, and  $a_i$ , the current input symbol. It then consults  $\text{action}[s_m, a_i]$  in the action table which can have one of four values :

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

**Goto** : The function *goto* takes a state and grammar symbol as arguments and produces a state.

A Configuration of an LR parser is a pair whose first component is the stack contents and whose second component is the unexpended input.

$$(s_0X_1s_1X_2s_2\dots X_m s_m, a_i a_{i+1} \dots a_n \$)$$

This configuration represents the right – sentential form.

$$X_1 X_2 \dots X_m a_i a_{i+1} \dots a_n.$$

The next move of the parser is determined by reading  $a_i$ , the current input symbol and  $s_m$ , the state on the top of the stack and then consulting the parsing action table entry  $\text{action}[s_m, a_i]$ .

The configurations resulting after each of the four types of move are as follows.

1. If  $\text{action}[s_m, a_i] = \text{shift } s$ , the parser executes a shift move entering the configuration  
 $(s_0X_1s_1X_2\dots X_m s_m a_i s, a_{i+1} \dots a_n \$).$

Here the parser has shifted both the current input symbol  $a_i$  and the next state  $s$  which is given in  $\text{action}[s_m, a_i]$  onto the stack  $a_{i+1}$  becomes the current input symbol.

2. If  $\text{action}[s_m, a_i] = \text{reduce } A \rightarrow \beta$  then the parser executes a reduce move entering the configuration

$$(s_0X_1s_1X_2\dots X_{m-r} s_{m-r} A s, a_i a_{i+1} \dots a_n \$)$$

Where  $s = \text{goto}[s_{m-r}, A]$  and  $r$  is the length of  $\beta$ , the right side of production. Here the parser popped  $2r$  symbols off the stack ( $r$  state symbols and  $r$  Grammar symbols), exposing state  $s_{m-r}$ . The parser then pushed both  $A$ , the left side of the production and  $s$ , the entry for  $\text{goto}[s_{m-r}, A]$  onto the stack.

3. If  $\text{action}[s_m, a_i] = \text{accept}$ , then Parsing is completed.
4. If  $\text{action}[s_m, a_i] = \text{error}$ , the Parser has discovered an error and calls an error recovery routine.

### LR Parsing algorithm:

**Input:** An input string  $w$  and an LR parsing table with functions *action* and *goto* for grammar  $G$ .

**Output:** If  $w$  is in  $L(G)$ , a bottom-up-parse for  $w$ ; otherwise, an error indication.

**Method:** Initially, the parser has  $s_0$  on its stack, where  $s_0$  is the initial state, and  $w\$$  in the input buffer. The parser then executes the following program :

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

**repeat forever begin**

let  $s$  be the state on top of the stack and  $a$  the symbol pointed to by  $ip$ ;

**if**  $action[s, a] = \text{shift } s'$  **then begin** push  $a$  then  $s'$  on top of the stack; advance  $ip$  to the next input symbol

**end**

**else if**  $action[s, a] = \text{reduce } A \rightarrow \beta$  **then begin** pop  $2^* |\beta|$  symbols off the stack;

let  $s'$  be the state now on top of the stack; push  $A$  then  $goto[s', A]$  on top of the stack; output the production  $A \rightarrow \beta$

**end**

**else if**  $action[s, a] = \text{accept}$  **then return**

**else error( )**

**end**

### Stack implementation:

Check whether the input  $\text{id} + \text{id} * \text{id}$  is valid or not.

STACK	INPUT	ACTION
0	$\text{id} + \text{id} * \text{id} \$$	GOTO ( I0 , id ) = s5 ; <b>shift</b>
0 id 5	$+ \text{id} * \text{id} \$$	GOTO ( I5 , + ) = r6 ; <b>reduce</b> by $F \rightarrow \text{id}$
0 F 3	$+ \text{id} * \text{id} \$$	GOTO ( I0 , F ) = 3 GOTO ( I3 , + ) = r4 ; <b>reduce</b> by $T \rightarrow F$
0 T 2	$+ \text{id} * \text{id} \$$	GOTO ( I0 , T ) = 2 GOTO ( I2 , + ) = r2 ; <b>reduce</b> by $E \rightarrow T$
0 E 1	$+ \text{id} * \text{id} \$$	GOTO ( I0 , E ) = 1 GOTO ( I1 , + ) = s6 ; <b>shift</b>
0 E 1 + 6	$\text{id} * \text{id} \$$	GOTO ( I6 , id ) = s5 ; <b>shift</b>
0 E 1 + 6 id 5	$* \text{id} \$$	GOTO ( I5 , * ) = r6 ; <b>reduce</b> by $F \rightarrow \text{id}$
0 E 1 + 6 F 3	$* \text{id} \$$	GOTO ( I6 , F ) = 3 GOTO ( I3 , * ) = r4 ; <b>reduce</b> by $T \rightarrow F$
0 E 1 + 6 T 9	$* \text{id} \$$	GOTO ( I6 , T ) = 9 GOTO ( I9 , * ) = s7 ; <b>shift</b>
0 E 1 + 6 T 9 * 7	$\text{id} \$$	GOTO ( I7 , id ) = s5 ; <b>shift</b>
0 E 1 + 6 T 9 * 7 id 5	$\$$	GOTO ( I5 , \$ ) = r6 ; <b>reduce</b> by $F \rightarrow \text{id}$
0 E 1 + 6 T 9 * 7 F 10	$\$$	GOTO ( I7 , F ) = 10 GOTO ( I10 , \$ ) = r3 ; <b>reduce</b> by $T \rightarrow T * F$

0 E 1+ 6 T 9	\$	GOTO ( I6 , T)=9 GOTO(I9,\$)=r1 ; reduce by E→E+T
--------------	----	--

### 6.a.CONSTRUCTION OF SLR PARSING TABLE

❖ Construct SLR Parsing Table for the following Grammar.

$E \rightarrow E + T \mid T \quad T \rightarrow T * F \mid F \quad F \rightarrow (E) \mid id$  and show the actions for the input string  $id1 * id2 + id3$  [Nov/Dec 2011,2016,Apr/May 2011]

#### CONSTRUCTING SLR(1) PARSING TABLE:

To perform SLR parsing, take grammar as input and do the following:

1. Find LR(0) items.
2. Completing the closure.
3. Compute  $goto(I,X)$ , where, I is set of items and X is grammar symbol.

#### LR(0) items:

An  $LR(0)$  item of a grammar G is a production of G with a dot at some position of the right side. For example, production  $A \rightarrow XYZ$  yields the four items :

$A \rightarrow . XYZ$   
 $A \rightarrow X . YZ$   
 $A \rightarrow XY . Z$   
 $A \rightarrow XYZ .$

#### Closure operation:

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

- \{ Initially, every item in I is added to closure(I).
- \{ If  $A \rightarrow \alpha . B\beta$  is in closure(I) and  $B \rightarrow \gamma$  is a production, then add the item  $B \rightarrow . \gamma$  to I , if it is not already there. We apply this rule until no more new items can be added to closure(I).

#### Goto operation:

$goto(I, X)$  is defined to be the closure of the set of all items  $[A \rightarrow \alpha X . \beta]$  such that  $[A \rightarrow \alpha . X\beta]$  is in I.

Steps to construct SLR parsing table for grammar G are:

1. Augment G and produce  $G'$
2. Construct the canonical collection of set of items C for  $G'$
3. Construct the parsing action function *action* and *goto* using the following algorithm that requires FOLLOW(A) for each non-terminal of grammar.

#### The Sets of Items Construction:

```
procedure items(G');
begin
  C= {closure({[S'→ . S]})};
  repeat
```

```

for each set of items I in C and each Grammar symbol X
    Such that goto(I,X) is not empty and not in C do
        Add goto(I,X) to C
    until no more sets of items can be added to C
end

```

### **Algorithm for construction of SLR parsing table:**

**Input :** An augmented grammar  $G'$

**Output :** The SLR parsing table functions *action* and *goto* for  $G'$

**Method :**

1. Construct  $C = \{I_0, I_1, \dots, I_n\}$ , the collection of sets of LR(0) items for  $G'$ .
2. State  $i$  is constructed from  $I_i$ . The parsing functions for state  $i$  are determined as follows:
  - (a) If  $[A \rightarrow \alpha \cdot a\beta]$  is in  $I_i$  and  $\text{goto}(I_i, a) = I_j$ , then set  $\text{action}[i, a]$  to "shift j". Here  $a$  must be terminal.
  - (b) If  $[A \rightarrow \alpha \cdot]$  is in  $I_i$ , then set  $\text{action}[i, a]$  to "reduce  $A \rightarrow \alpha$ " for all  $a$  in  $\text{FOLLOW}(A)$ .
  - (c) If  $[S' \rightarrow S \cdot]$  is in  $I_i$ , then set  $\text{action}[i, \$]$  to "accept".

If any conflicting actions are generated by the above rules, we say grammar is not SLR(1).

3. The *goto* transitions for state  $i$  are constructed for all non-terminals  $A$  using the rule: If  $\text{goto}(I_i, A) = I_j$ , then  $\text{goto}[i, A] = j$ .
- $\Sigma$  All entries not defined by rules (2) and (3) are made "error"  
 $\Sigma$  The initial state of the parser is the one constructed from the set of items containing  $[S' \rightarrow S \cdot]$ .

### **Example for SLR parsing:**

**Find the SLR parsing table for the given grammar  $E \rightarrow E+E \mid E^*E \mid (E) \mid id$ .**

Given grammar:

1.  $E \rightarrow E+E$
2.  $E \rightarrow E^*E$
3.  $E \rightarrow (E)$
4.  $E \rightarrow id$

Augmented grammar:

$E \rightarrow E$   
 $E \rightarrow E+E$   
 $E \rightarrow E^*E$   
 $E \rightarrow (E)$   
 $E \rightarrow id$   
 $I_0: E \rightarrow E$   
 E  $\rightarrow E+E$   
 E  $\rightarrow E^*E$   
 E  $\rightarrow (E)$

```

E->.id
I1: goto(I0, E)
    E->E.
    E->E.+E
    E->E.*E
I2: goto(I0, ())
    E->(.E)
    E->.E+E
    E->.E*E
    E->.(E)
    E->.id
I3: goto(I0, id)
E->id.
I4: goto(I1, +)
    E->E+.E
    E->.E+E
    E->.E*E
    E->.(E)
    E->.id
I5: goto(I1, *)
    E->E*.E
    E->.E+E
    E->.E*E
    E->.(E)
    E->.id
I6: goto(I2, E)
    E->(E.)
    E->E.+E
    E->E.*E
I7: goto(I4, E)
    E->E+E.
    E->E.+E
    E->E.*E
I8: goto(I5, E)
    E->E*E.
    E->E.+E
    E->E.*E
goto(I2, ())=I2
goto(I2, id)=I3
goto(I4, ())=I2
goto(I4, id)=I3
goto(I5, ())=I2
goto(I5, id)=I3

```

I9: goto(I6, ))

E->(E).

goto(I6, +)=I4

goto(I6, \*)=I5

goto(I7, +)=I4

goto(I7, \*)=I5

goto(I8, +)=I4

goto(I8, \*)=I5

First(E) = {(), id}

Follow(E)={+, \*, ), \$}

**SLR parsing table:**

States	Action						Goto
	+	*	(	)	id	\$	
0			S2		S3		1
1	S4	S5				Acc	
2			S2		S3		6
3	r4	r4		r4		r4	
4			S2		S3		7
5			S2		S3		8
6	S4	S5		S9			
7	S4, r1	S5, r1		r1		r1	
8	S4, r2	S5, r2		r2		r2	
9	r3	r3		r3		r3	

Blank entries are error entries.

## **6. b. CONSTRUCTION OF LALR PARSING TABLE**

- ❖ Design an LALR Parser for the following Grammar and parse the input string w=\*id

$S \rightarrow L=R \quad S \rightarrow R \quad L \rightarrow *R \quad L \rightarrow id \quad R \rightarrow L$  (16) [Nov/Dec 2013]

- ❖ Find the LALR Parser for the following Grammar

$S \rightarrow L=R \quad S \rightarrow R \quad L \rightarrow *R \quad L \rightarrow id \quad R \rightarrow L$  (16) [Nov/Dec 2014]

**LALR PARSER:**

**Closure operation:**

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

\ Initially, every item in I is added to closure(I).

\ If  $A \rightarrow \alpha . B\beta$  is in closure(I) and  $B \rightarrow \gamma$  is a production, then add the item  $B \rightarrow . \gamma$  to I , if it is not already there. We apply this rule until no more new items can be added to closure(I).

**Goto operation:**

$Goto(I, X)$  is defined to be the closure of the set of all items  $[A \rightarrow \alpha X . \beta]$  such

that  $[A \rightarrow \alpha \cdot X\beta]$  is in  $I$ .

Steps to construct SLR parsing table for grammar  $G$  are:

4. Augment  $G$  and produce  $G'$
5. Construct the canonical collection of set of items  $C$  for  $G'$
6. Construct the parsing action function *action* and *goto* using the following algorithm that requires FOLLOW( $A$ ) for each non-terminal of grammar.

### The Sets of Items Construction:

```

procedure items( $G'$ );
begin
     $C = \{\text{closure}(\{[S' \rightarrow \cdot S, \$]\})\}$ ;
    repeat
        for each set of items  $I$  in  $C$  and each Grammar symbol  $X$ 
            Such that  $\text{goto}(I, X)$  is not empty and not in  $C$  do
                Add  $\text{goto}(I, X)$  to  $C$ 
        until no more sets of items can be added to  $C$ 
end

```

### Algorithm for construction of LALR parsing table:

**Input :** An augmented grammar  $G'$

**Output :** The LALR parsing table functions *action* and *goto* for  $G'$

#### Method :

1. Construct  $C = \{I_0, I_1, \dots, I_n\}$ , the collection of sets of LR(0) items for  $G'$ .
2. For each core present among the set of LR(1) items, find all sets having that core , and replace these sets by their union.
3. Let  $C'=\{J_0, J_1, \dots, J_m\}$  be the resulting sets of LR(1) items. State  $i$  is constructed from  $J_i$ . The parsing functions for state  $i$  are determined as follows:
  - (a) If  $[A \rightarrow \alpha \cdot a\beta]$  is in  $J_i$  and  $\text{goto}(J_i, a) = J_j$ , then set  $\text{action}[i, a]$  to "shift j". Here  $a$  must be terminal.
  - (b) If  $[A \rightarrow \alpha \cdot]$  is in  $J_i$  , then set  $\text{action}[i, a]$  to "reduce  $A \rightarrow \alpha$ " for all  $a$  in  $\text{FOLLOW}(A)$ .
  - (c) If  $[S' \rightarrow S \cdot]$  is in  $J_i$ , then set  $\text{action}[i, \$]$  to "accept".

If any conflicting actions are generated by the above rules, we say grammar is not LALR(1).

4. The goto table is constructed as follows. If  $J$  is the union of one or more sets of LR(1) items , that is , $J=I_1 \cup I_2 \cup \dots \cup I_k$ , then the cores of  $\text{goto}(I_1, X)$ ,  $\text{goto}(I_2, X)$  , .....  $\text{GOTO}(I_k, X)$  are the same, since  $I_1, I_2, \dots, I_k$  all have the same core. Let  $K$  be the union of all sets of items having the same core as  $\text{goto}(I_1, X)$ . Then  $\text{goto}(J, X)=K$ .

#### Example :

$S \rightarrow L=R \quad S \rightarrow R \quad L \rightarrow *R \quad L \rightarrow id \quad R \rightarrow L$

#### Augumented Grammar(I)

$S' \rightarrow S$

$S \rightarrow L=R$

$S \rightarrow R$

$L \rightarrow *R$

$L \rightarrow id$

$R \rightarrow L$

**Closure(I) :**

I<sub>0</sub>:  $S' \rightarrow . S, \$$

$S \rightarrow . L=R, \$$

$S \rightarrow . R, \$$

$L \rightarrow . *R, \$$

$L \rightarrow . id, \$$

$R \rightarrow . L, \$$

I<sub>1</sub>: goto(I<sub>0</sub>, S)

$S' \rightarrow S . , \$$

I<sub>2</sub>: goto(I<sub>0</sub>, L)

$S \rightarrow L . =R, \$$

$R \rightarrow L . , \$$

I<sub>3</sub>: goto(I<sub>0</sub>, R)

$S \rightarrow R . , \$$

I<sub>4</sub>: goto(I<sub>0</sub>, \*) , goto(I<sub>4</sub>, \*) , goto(I<sub>6</sub>, \*)

$L \rightarrow * . R, \$$

$R \rightarrow . L, \$$

$L \rightarrow . *R, \$$

$L \rightarrow . id, \$$

I<sub>5</sub>: goto(I<sub>0</sub>, id) , goto(I<sub>4</sub>, id) , goto(I<sub>6</sub>, id)

$L \rightarrow id . , \$$

I<sub>6</sub>: goto(I<sub>2</sub>, = )

$S \rightarrow L = . R, \$$

$R \rightarrow . L, \$$

$L \rightarrow . *R, \$$

$L \rightarrow . id, \$$

I<sub>7</sub>: goto(I<sub>4</sub>, R)

$L \rightarrow * R . , \$$

I<sub>8</sub>: goto(I<sub>4</sub>, L) , goto ( I<sub>6</sub>, L)

$R \rightarrow L . , \$$

I<sub>9</sub>: goto(I<sub>6</sub>, R)

$S \rightarrow L = R . , \$$

### Applying the Algorithm :

State	Action				Goto		
	=	*	id	\$	S	L	R
0		s4	s5		1	2	3
1				acc			
2	s6			r5			
3				r2			
4		s4	s5			8	7
5	r4			r4			
6		s4	s5			8	9
7	r3			r3			
8	r5			r5			
9				r1			

### Parsing the input String w= \*id :

Stack	Input	Action
0	*id\$	Shift 4
0 * 4	id\$	Shift 5
0 * 4 id 5	\$	Reduce by L→id
0 * 4 L	\$	Goto(4,L)=8
0 * 4 L 8	\$	Reduce by L→R
0 * 4 R	\$	Goto(4,R)=7
0 * 4 R 7	\$	Reduce by L→*R
0 L	\$	Goto(0,L)=2
0 L 2	\$	Reduce by R→L
0 R	\$	Goto(0,R)=3
0 R 3	\$	Reduce by S→R
0 S	\$	Goto(0,S)=1
0 S 1	\$	Accept

### 7.YACC-DESIGN OF A SYNTAX ANALYZER FOR A SAMPLE LANGUAGE

❖ Discuss about YACC for constructing a parser for a sample Language.

#### YACC:

- YACC Stands for Yet Another Compiler Compiler.
- YACC is a tool available for converting parse trees from source programs. The input to YACC is a CFG and is having a specific structure. For each production there is an action implemented.

- YACC parses a stream of tokens generated by LEX according to a user specified Grammar.

### **Structure of a YACC file :**

```
... definitions . . .
%%
. . . rules . . .
%%
. . . code . . .
```

#### **Definitions:**

There are three points to be considered.

##### **i. C Code:**

All code between %{ and % } is copied to the beginning of the resulting C File. This is used to define Variables Prototypes and routines etc..

##### **ii. Definition:**

The definition of a YACC file is concerned with tokens. These token definitions are written to a .h file when YACC compiles this file.

##### **iii. Associativity Rules:**

These handle associativity and priority of operators.

#### **Rules:**

A number of combinations of patterns and action is more than a single command then it needs to be in braces.

#### **Code:**

This can be very elaborate. It includes a call to yylex.

### **LEX – YACC Interaction :**

The LEX Code parses a file of characters and outputs a stream of tokens. YACC accepts a stream of tokens. YACC accepts a stream of tokens and parses it, performing appropriate actions.

If the LEX program is supplying tokenizer YACC program will repeatedly call the yylex routine. The rules will repeatedly call the yylex routine. The rules will probably function by calling return every time LEX have passed a token.

If LEX returns tokens then YACC will process them and they have to agree on what tokens there are. This is done as follows.

➤ The YACC file will have token definitions in the definition section.

% token NUMBER

➤ When the YACC file is translated with YACC -d , a header file y.tab.h is created.

```
#define NUMBER 258
```

This file can be included in both LEX and YACC Program.

- The Lex file can then call return NUMBER and the YACC program can match on this token.

### Return Values:

The LEX Parse can return a symbol that is put on top of the stack so that YACC can access it. This symbol is returned in the variable yylval.

By default this is defined as an int

```
Extern int lval;  
%%  
[0-9]+ {lval=atoi(yytext); return NUMBER ;}
```

### Rules Section:

The rules section contains the grammar of the language we want to parse. This looks like

```
name1 : THING something OTHERTHING {action}|othersomething THING {other  
action }  
name 2 : . . . . .
```

This is the general form of CFG with a set of actions associated with each matching right hand side. It is a good convention to keep non terminals in lower case and terminals in uppercase.

The terminal symbols get matched with return codes from the lex tokenizer. They are typically define coming from % token definitions in the YACC Program.

### User Code Section :

The minimal main program is

```
int main()  
{  
    yyparse();  
    return 0;  
}
```

In addition to the main program the code section will also contain subroutines to be used either in the YACC or the LEX Program.

## UNIT – IV

### **SYNTAX DIRECTED TRANSLATION & RUN TIME ENVIRONMENT**

Syntax Directed Definitions-Construction of syntax Tree-Bottom up Evaluation of S-Attribute Definitions-Design of predictive Translator-Type Systems-Specification of a simple type checker-Equivalence of Type Expressions-Type Conversions. RUN-TIME ENVIRONMENT : Source Language Issues—Storage organization-Storage Allocation-Parameter Passing-Symbol Tables. Dynamic Storage Allocation-Storage Allocation in FORTRAN.

#### PART-A

##### **1. Define syntax directed definition**

A Syntax Directed Definition(SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions. If X is a symbol and a is one of its attributes, then we write X.a to denote the value of a at a particular parse-tree node labeled X.

##### **2. List the various methods to evaluate semantic rules.**

- Parse Tree Methods
- Rule Based Methods
- Oblivious Method.

##### **3. What is Synthesized Attribute and Inherited Attribute??**

A **synthesized attribute** for a non terminal A at a parse-tree node N is defined by a semantic rule associated with the production at N. Note that the production must have A as its head.

- A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself.

An **inherited attribute** for a non terminal B at a parse-tree node N is defined by a semantic rule associated with the production at the parent of N. Note that the production must have B as a symbol in its body.

- An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself, and N's siblings.
- 

##### **4. What is an attribute Grammar?**

An SDD without side effects is sometimes called an attribute grammar. The rules in an attribute grammar define the value of an attribute purely in terms of the values of other attributes and constants.

##### **5. What is Dependency Graph?**

"**Dependency graphs**" are a useful tool for determining an evaluation order for the attribute instances in a given parse tree. While an annotated parse tree shows the values of attributes, a dependency graph helps us determine how those values can

be computed.

## 6. What is S – Attribute Definition and L-Attribute Definition?

### S – Attribute Definition

An SSD is *S-attributed* if every attribute is synthesized. Attributes of an S-attributed SSD can be evaluated in bottom-up order of the nodes of parse tree. Evaluation is simple using post-order traversal

### L-Attribute Definition

The idea behind this class is that, between the attributes associated with a production body, dependency-graph edges can go from left to right, but not from right to left(hence "L-attributed"). Each attribute must be either

- a. Synthesized, or
- b. Inherited, but with the rules limited as follows. Suppose that there is a production  $A \rightarrow X_1 X_2 \dots X_n$ , and that there is an inherited attribute  $X_i.a$  computed by a rule associated with this production. Then the rule may use only:
  - (a) Inherited attributes associated with the head  $A$ .
  - (b) Either inherited or synthesized attributes associated with the occurrences of symbols  $X_1 X_2 \dots X_{i-1}$  located to the left of  $X_i$
  - (c) Inherited or synthesized attributes associated with this occurrence of  $X_i$  itself, but only in such a way that there are no cycles in a dependency graph formed by the attributes of this  $X_i$ .

## 7. What are the two approaches used for de-allocation of blocks?

### a. Reference Count:

We keep track of the number of blocks that point directly to the present block. If this count drops to 0 then the block can be de-allocated.

### b. Marking Techniques:

This is used to suspend temporarily execution of the program and use the frozen pointers to determine which blocks are in use. We pour paint through these pointers. Any block that is reached by the paint is in use and the rest can be de-allocated.

## 8. What is 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.

## 9. What are the types of type checking? Write various static check methods.

**Static checking:** A compiler check the source program follows both the syntactic and semantic conventions of the source language at compile time

**Dynamic checking:** A compiler check the source program during execution of the target program

### Examples of static checks:

- Type checks
- Flow-of-control checks
- Uniqueness checks
- Name-related checks

**10. List the different storage allocation strategies and write down the limitations of static allocation.**

The strategies are:

- Static allocation
- Stack allocation
- Heap allocation

**Limitations of static allocation:**

- The size of a data object and constraints on its position in memory must be known at compile time
- Recursive procedure are restricted, because all activations of a procedure use the same binding for local name
- Data structures cannot be created dynamically since there is no mechanism for storage allocation at run time

**11. Define activation tree and what are the contents of activation record?**

**Activation tree**

- Control flows sequentially; the execution of a program consists of a sequence of steps, with control being at some specific point in the program at each step.
- Each execution of a procedure starts at the beginning of the procedure body and eventually returns control to the point immediately following the place where the procedure was called. This means the flow of control between procedures can be depicted using trees.

The **activation record** is a block of memory used for managing the information needed by a single execution of a procedure. Various fields of activation record are:

- Temporary variables.
- Local variables
- Saved machine registers
- Control link
- Access link
- Actual parameters
- Return values

**12. Write down syntax directed definition of a simple desk calculator. [Nov/Dec 2016]**

Production	Semantic Rules
1) $L \rightarrow E \ n$	$L.val = E.val$
2) $E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3) $E \rightarrow T$	$E.val = T.val$
4) $T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
5) $T \rightarrow F$	$T.val = F.val$
6) $F \rightarrow ( E )$	$F.val = E.val$
7) $F \rightarrow \text{digit}$	$F.val = \text{digit}.lexval$

Fig 5.1: Syntax Directed Definition of simple desk calculator

**13. List Dynamic Storage allocation techniques.** [Nov/Dec 2016]

- Explicit Allocation of fixed sized blocks.
- Explicit Allocation of variable sized blocks.
- Implicit Deallocation.

**14. What is coercion?**

Conversion from one type to another is said to be implicit, if it is done automatically by the

compiler. Implicit type conversions also called coercions are limited in many languages. Eg.

Integer can be converted to real but not vice-versa.

**15. What is dangling reference and when does it occurs? [May/June 2016]**

Whenever storage can be deallocated, the problem of dangling references arises.

A dangling reference occurs when there is a reference to storage that has been deallocated

**PART-B**

**1.a.SOURCE LANGUAGE ISSUES**

- ❖ Describe the source language issues in detail.[Apr/May 2008]

**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.

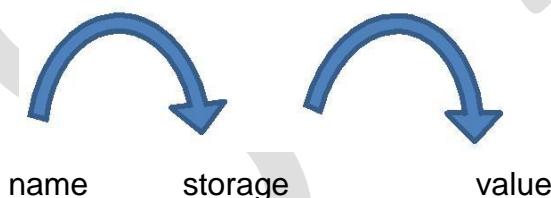
The portion of the program to which a declaration applies is called the **scope** of that declaration.

#### **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.

*environment state*



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*.

### **1.b. STORAGE ORGANIZATION**

- ❖ Mention in detail any 4 issues in storage organization.[Apr/May 2015]

### **STORAGE ORGANISATION**

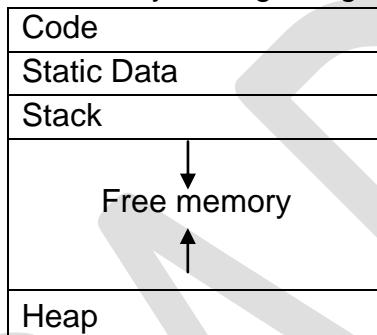
- The executing target program runs in its own logical address space in which each program value has a location.
- The management and organization of this logical address space is shared between the compiler, operating system and target machine. The operating

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.

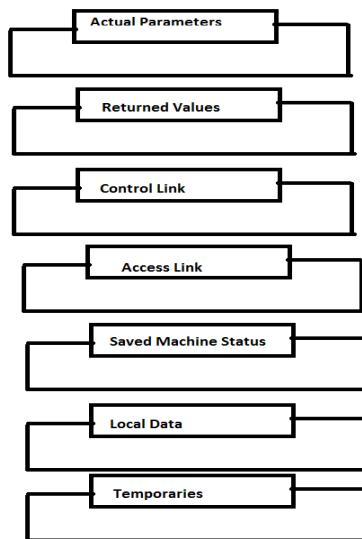
- The storage layout for data objects is strongly influenced by the addressing constraints of the target machine.
- A character array of length 10 needs only enough bytes to hold 10 characters, a compiler may allocate 12 bytes to get alignment, leaving 2 bytes unused.



- 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.

## 2.STORAGE ALLOCATION

- ❖ Discuss in detail about Storage Allocation Strategies.[Nov/Dec 2013] [May /June 2016]
- ❖ Discuss Run Time Storage Management in detail.[Nov/Dec 2013][Apr/May 2011].
- ❖ What are the different Storage Allocation Strategies?Explain.[Nov/Dec 2011]

### 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.

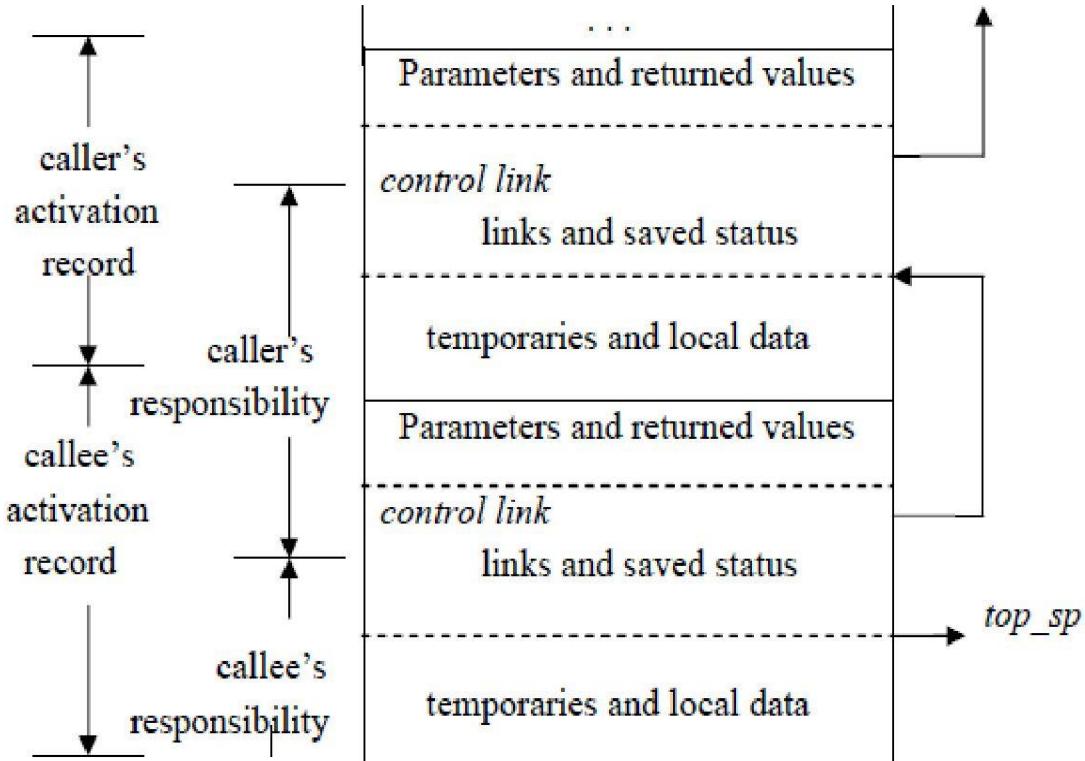
- Since the bindings do not change at run-time, everytime a procedure is activated, its names are bound to the same storage locations.
- Therefore values of local names are *retained* across activations of a procedure. That is, when control returns to a procedure the values of the locals are the same 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.

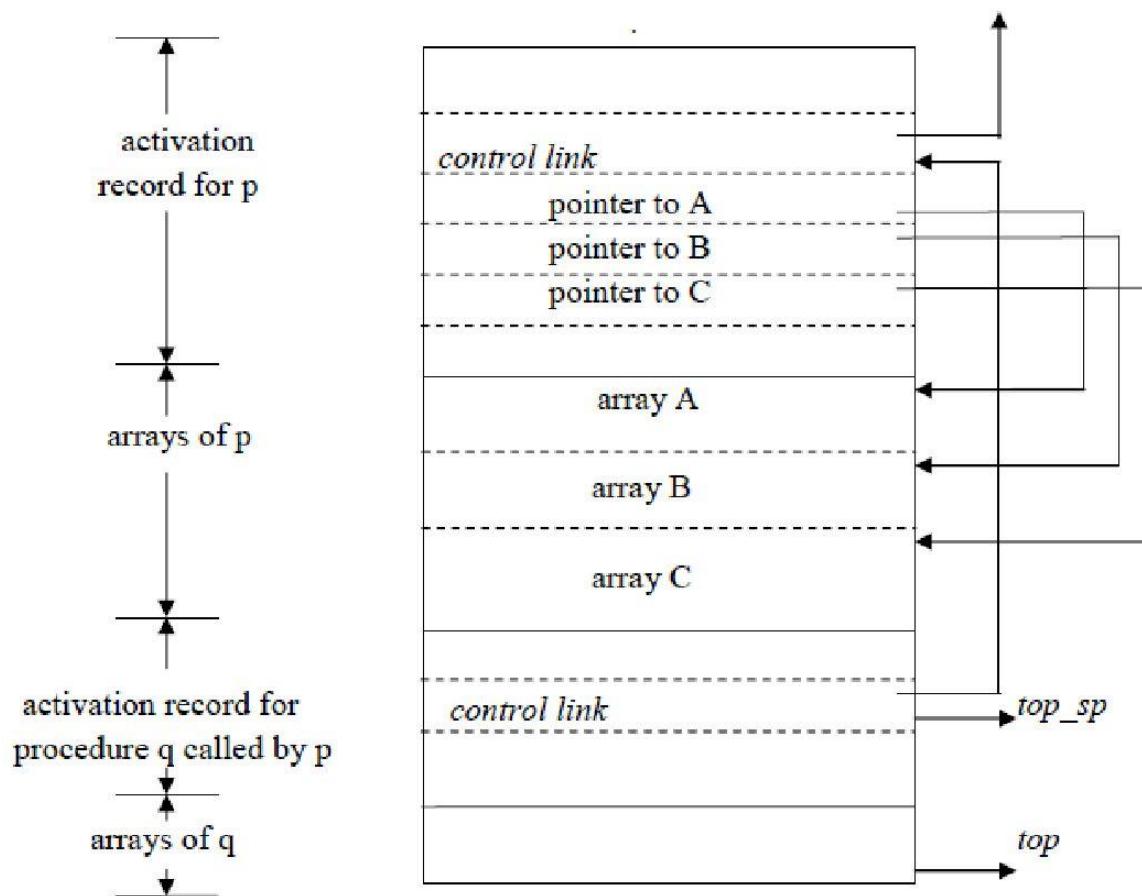
- The calling sequence and its division between caller and callee are as follows.  
The caller evaluates the actual parameters.
- The caller stores a return address and the old value of *top\_sp* into the callee's activation record. The caller then increments the *top\_sp* to the respective positions.
- 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.



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 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
		Retained activation record for r

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.

### 3.a.TYPES SYSTEMS

- ❖ Illustrate type checking with necessary diagram . [Nov/Dec 2016].

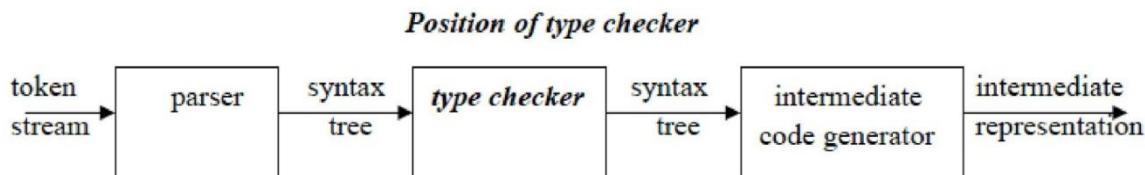
## TYPE CHECKING

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.



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.

## 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:

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

### Translation scheme:

$$\begin{aligned}
 P &\rightarrow D ; E \\
 D &\rightarrow D ; D \\
 D &\rightarrow id : T \{ \text{addtype}(id.entry, T.type) \} \\
 T &\rightarrow \text{char} \{ T.type := \text{char} \} \\
 T &\rightarrow \text{integer} \{ T.type := \text{integer} \} \\
 T &\rightarrow \uparrow T1 \{ T.type := \text{pointer}(T1.type) \} \\
 T &\rightarrow \text{array} [ \text{num} ] \text{ of } T1 \{ T.type := \text{array}(1... \text{num}.val, T1.type) \}
 \end{aligned}$$

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 ,  $\uparrow \text{integer}$  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.  $E \rightarrow \text{literal} \{ E.\text{type} : \text{char} \}$   $E \rightarrow \text{num} \{ E.\text{type} : = \text{integer} \}$

Here, constants represented by the tokens **literal** and **num** have type *char* and *integer*.

2.  $E \rightarrow \text{id} \{ E.\text{type} : = \text{lookup} ( \text{id}.entry ) \}$

*lookup* ( e ) is used to fetch type saved in symbol table entry pointed to by e.

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

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

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

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.  $E \rightarrow E_1 \uparrow \{ E.\text{type} : = \text{if } E_1.\text{type} = \text{pointer} (t) \text{ then } t \text{ else type_error} \}$

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:

$S \rightarrow \text{id} := E \{ S.\text{type} : = \text{if } \text{id}.\text{type} = E.\text{type} \text{ then void else type_error} \}$

#### 2. Conditional statement:

$S \rightarrow \text{if } E \text{ then } S_1 \{ S.\text{type} : = \text{if } E.\text{type} = \text{boolean} \text{ then } S_1.\text{type} \text{ else type_error} \}$

#### 3. While statement:

$S \rightarrow \text{while } E \text{ do } S_1 \{ S.\text{type} : = \text{if } E.\text{type} = \text{boolean} \text{ then } S_1.\text{type} \text{ else type_error} \}$

#### 4. Sequence of statements:

$S \rightarrow S_1 ; S_2 \{ S.\text{type} : = \text{if } S_1.\text{type} = \text{void} \text{ and } S_2.\text{type} = \text{void} \text{ then void else type_error} \}$

## Type checking of functions

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

```
E → E1 ( E2 ) { E.type := if E2.type = s and
E1.type = s → t then t
else type_error }
```

## 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 T1 and T2 are type expressions, then their Cartesian product T1 X T2 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;
```

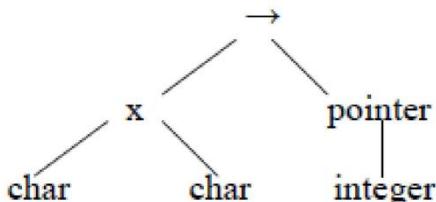
declares the type name *row* representing the type expression **record((address X integer) X (lexeme X array(1..15,char)))** and the variable *table* to be an array of records of this type.

**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.

### **3.b. SPECIFICATION OF A SIMPLE TYPE CHECKER**

- ❖ Explain the specification of a simple type checker.[Nov/Dec 2014] [May /June 2016]

#### **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:

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

#### **Translation**

#### **scheme:**

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

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 ,  $\uparrow \text{integer}$  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.  $E \rightarrow \text{literal} \quad \{ E.type := \text{char} \}$
- $E \rightarrow \text{num} \quad \{ E.type := \text{integer} \}$

Here, constants represented by the tokens **literal** and **num** have type *char* and *integer*.

2.  $E \rightarrow \mathbf{id} \quad \{ E.type := \text{lookup}(\mathbf{id}.entry) \}$

*lookup* (*e*) is used to fetch the type saved in the symbol table entry pointed to by *e*.

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

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

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

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  $\mathbf{array}(s,t)$  of  $E_1$ .

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

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:

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

#### 2. Conditional statement:

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

#### 3. While statement:

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

#### 4. Sequence of statements:

$S \rightarrow S_1 ; S_2 \quad \{ S.type := \mathbf{if } S_1.type = \mathbf{void} \mathbf{and } S_2.type = \mathbf{void} \mathbf{then void} \mathbf{else type\_error} \}$

### Type checking of functions

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

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

## 4. SYNTAX DIRECTED DEFINITION

- ❖ For the input expression  $3*5+4n$ , construct an annotated parse tree using syntax directed definition. [May /June 2016]

### Syntax-directed translation (SDT)

- SDT refers to a method of compiler implementation where the source language translation is completely driven by the parser, i.e., based on the syntax of the language. The parsing process and parse trees are used to direct semantic analysis and the translation of the source program. Almost all modern compilers are syntax-directed.
- SDT can be a separate phase of a compiler or we can augment our conventional grammar with information to control the semantic analysis and translation. Such grammars are called **attribute grammars**.
- We augment a grammar by associating **attributes** with each grammar symbol that describes its properties. With each production in a grammar, we give **semantic rules/ actions**, which describe how to compute the attribute values associated with each grammar symbol in a production.
- The general approach to Syntax-Directed Translation is to construct a parse tree or syntax tree and compute the values of attributes at the nodes of the tree by visiting them in some order. In many cases, translation can be done during parsing without building an explicit tree.
- A class of syntax-directed translations called "L-attributed translations" (L for left-to-right) includes almost all translations that can be performed during parsing. Similarly, "S-attributed translations" (S for synthesized) can be performed easily in connection with a bottom-up parse.

There are two ways to represent the semantic rules associated with grammar symbols.

- Syntax-Directed Definitions (SDD)
- Syntax-Directed Translation Schemes (SDT)

### 1. Syntax-Directed Definitions

- A syntax-directed definition (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions.
- An attribute has a name and an associated value: a string, a number, a type, a memory location, an assigned register, strings. The strings may even be long sequences of code, say code in the intermediate language used by a compiler. If  $X$  is a symbol and  $a$  is one of its attributes, then we write  $X.a$  to denote the value of  $a$  at a particular parse-tree node labeled  $X$ . If we implement the nodes of the parse tree by records or objects, then the attributes of  $X$  can be implemented by data fields in the records that represent the nodes for  $X$ . The attributes are evaluated by the semantic rules attached to the productions.

Example: PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + T$	$E.\text{code} = E_1.\text{code} \parallel T.\text{code} \parallel '+'$

- SDDs are highly readable and give high-level specifications for translations. But they hide many implementation details. For example, they do not specify order of evaluation of semantic actions.

## 2.Syntax-Directed Translation Schemes (SDT)

- SDT embeds program fragments called semantic actions within production bodies. The position of semantic action in a production body determines the order in which the action is executed.
- Example: In the rule  $E \rightarrow E_1 + T \{ \text{print } '+' \}$ , the action is positioned after the body of the production.

### Inherited and Synthesized Attributes

- Terminals can have synthesized attributes, which are given to it by the lexer (not the parser). There are no rules in an SDD giving values to attributes for terminals. Terminals do not have inherited attributes.
- A non terminal A can have both inherited and synthesized attributes. The difference is how they are computed by rules associated with a production at a node N of the parse tree.
- A **synthesized attribute** for a non terminal A at a parse-tree node N is defined by a semantic rule associated with the production at N. Note that the production must have A as its head.
  - A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself.
- An **inherited attribute** for a non terminal B at a parse-tree node N is defined by a semantic rule associated with the production at the parent of N. Note that the production must have B as a symbol in its body.
  - An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself, and N's siblings.

An inherited attribute at node N cannot be defined in terms of attribute values at the children of node N. However, a synthesized attribute at node N can be defined in terms of inherited attribute values at node N itself.

Production	Semantic Rules
1) $L \rightarrow E \ n$	$L.\text{val} = E.\text{val}$
2) $E \rightarrow E_1 + T$	$E.\text{val} = E_1.\text{val} + T.\text{val}$
3) $E \rightarrow T$	$E.\text{val} = T.\text{val}$
4) $T \rightarrow T_1 * F$	$T.\text{val} = T_1.\text{val} \times F.\text{val}$
5) $T \rightarrow F$	$T.\text{val} = F.\text{val}$
6) $F \rightarrow ( E )$	$F.\text{val} = E.\text{val}$
7) $F \rightarrow \text{digit}$	$F.\text{val} = \text{digit}.\text{lexval}$

Fig 5.1: Syntax Directed Definition of simple desk calculator

Example : The SDD in Fig. 5.1 is based on grammar for arithmetic expressions with operators + and \*. It evaluates expressions terminated by an endmarker n.

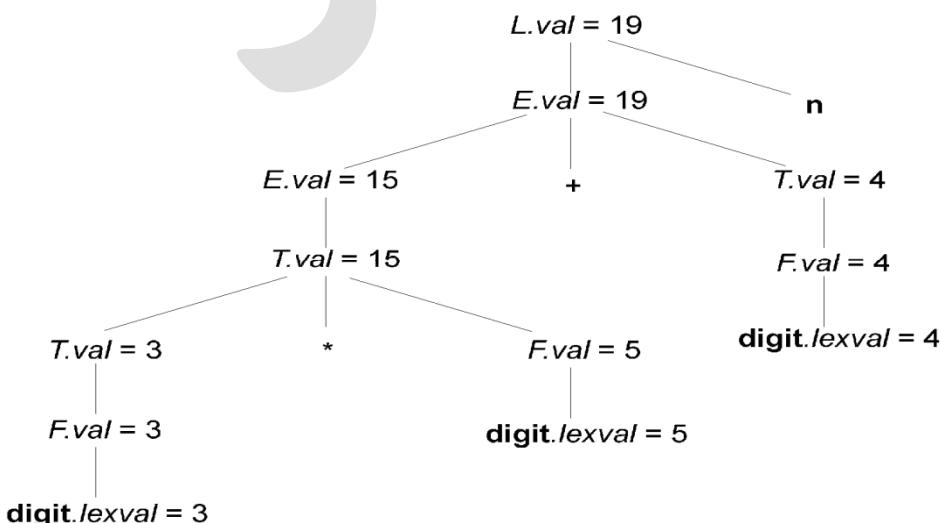
- In the SDD, each of the nonterminals has a single synthesized attribute, called val. We also suppose that the terminal digit has a synthesized attribute lexval, which is an integer value returned by the lexical analyzer.
- An SDD that involves only synthesized attributes is called **S-attributed**; the SDD in Fig. 5.1 has this property. In an S-attributed SDD, each rule computes an attribute for the nonterminal at the head of a production from attributes taken from the body of the production.

**Attribute Grammar:** An SDD without side effects is sometimes called an attribute grammar. The rules in an attribute grammar define the value of an attribute purely in terms of the values of other attributes and constants.

#### Evaluating an SDD at the Nodes of a Parse Tree

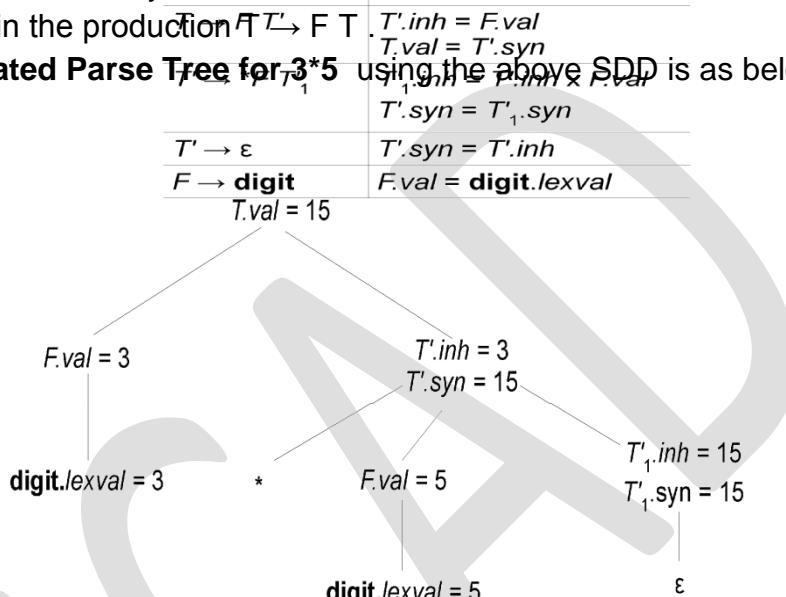
- Parse tree helps us to visualize the translation specified by SDD. The rules of an SDD are applied by first constructing a parse tree and then using the rules to evaluate all of the attributes at each of the nodes of the parse tree. A parse tree, showing the value(s) of its attribute(s) is called an **annotated parse tree**.
- With synthesized attributes, we can evaluate attributes in any bottom-up order, such as that of a postorder traversal of the parse tree.

Example: Annotated Parse Tree for  $3 * 5 + 4n$



- Inherited attributes are useful when the structure of a parse tree does not match the abstract syntax of the source code. They can be used to overcome the mismatch due to grammar designed for parsing rather than translation.
- In the SDD below, the nonterminal T has an inherited attribute *inh* as well as a synthesized attribute *val*. T inherits F.val from its left sibling F in the production  $T \rightarrow F T$ .

Annotated Parse Tree for  $3 * 5$  using the above SDD is as below.



An SDD with both inherited and synthesized attributes does not ensure any guaranteed order; even it may not have an order at all. For example, consider nonterminals A and B, with synthesized and inherited attributes A.s and B.i, respectively, along with the production and rules as in Fig.5.2. These rules are circular; it is impossible to evaluate either A.s at a node N or B.i at the child of N without first evaluating the other. The circular dependency of A.s and B.i at some pair of nodes in a parse tree is suggested by Fig.

e.g.

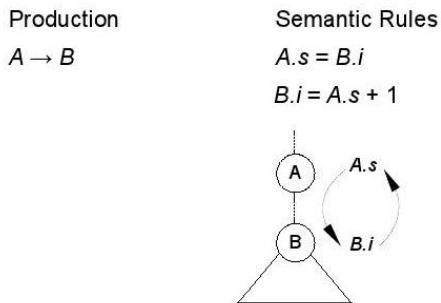


Fig . . . The circular dependency of  $A.s$  and  $B.i$  on one another .

### Evaluation Orders for SDD's

"Dependency graphs" are a useful tool for determining an evaluation order for the attribute instances in a given parse tree. While an annotated parse tree shows the values of attributes, a dependency graph helps us determine how those values can be computed.

A dependency graph shows the flow of information among the attribute instances in a particular parse tree; an edge from one attribute instance to another means that the value of the first is needed to compute the second. Edges express constraints implied by the semantic rules.

- Each attribute is associated to a node
- If a semantic rule associated with a production  $p$  defines the value of synthesized attribute  $A.b$  in terms of the value of  $X.c$ , then graph has an edge from  $X.c$  to  $A.b$
- If a semantic rule associated with a production  $p$  defines the value of inherited attribute  $B.c$  in terms of value of  $X.a$ , then graph has an edge from  $X.a$  to  $B.c$

**4 b) Construct a syntax directed definition scheme that takes strings of a's,b's and c's as input and produces as output the number of substrings in the input string that correspond to the pattern  $a(a/b^*c+(a/b)^*b)$ . For example the translation of the input string "abbcababcababc" is "3".**

1. Write a context free grammar that generate all strings of a's,b's and c's
2. Give the semantic attributes for the grammar symbols.
3. For each productions of the grammar a set of rules for evaluation of the semantic attributes.

Nov/Dec :2016

Solution:

1. The CFG  
 $G = \{\{a,b,c\}, \{S\}, S, P\}$  for all strings over the alphabet {a,b,c} with Pas the set of productions given below .  
 $S \rightarrow SAS \rightarrow SbS \rightarrow ScS \rightarrow aS \rightarrow bS \rightarrow c$ .
2. Given the grammar above

Define three synthesized attributes for the non terminals symbol S,namely nA1,nA2, and total.The idea of these attributes is that in the first attribute will capture the number of a's to left of a given c character, the second attribute, nA@, the number of a's to the right of that character.so that we can then add the value of a's to the right of c ,so that when find a new c,we copy the value of a's that were to the right of the first c and which are now to the left of the second c.

3. As suc ha set of rules is as follows, jere written as semantic actions given that their order of evaluation is done using a bottom-up depth-first search traversal.

```
S1-> S2 a {S1.nA1=S2.nA1 +1;S1.nA2=S2.total = S2.total;}
S1-> S2 b {S1.nA1=S2.nA1;S2.nA2=S2.nA2;S1.total=S2.total+S2.nA2;}
S1-> S2 c {S1.nA1=0;S1.nA2=S2.nA1;S1.total = S2.total;}
S1->a{S1.nA1=1;S1.nA2=0;S1.total=0;}
S1->b{S1.nA1=0;S1.nA2=0;S1.total=0;}
S1->c{S1.nA1=1;S1.nA2=0;S1.total=0;}
```

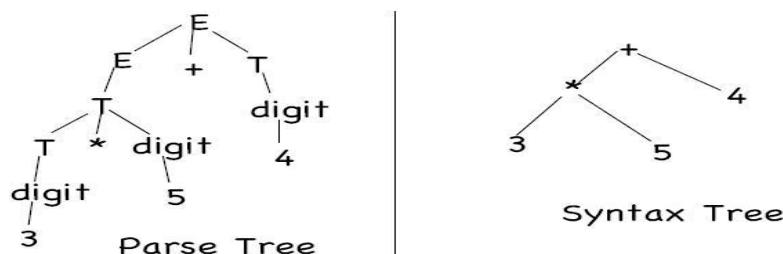
### 5.CONSTRUCTION OF SYNTAX TREE

- ❖ Describe the procedure to construct the syntax tree using syntax directed translation scheme.

## CONSTRUCTION OF SYNTAX TREE

### Construction of Syntax Trees

- SDDs are useful for is construction of syntax trees. A syntax tree is a condensed form of parse tree.
- Syntax trees are useful for representing programming language constructs like expressions and statements.



- They help compiler design by decoupling parsing from translation.
- Each node of a syntax tree represents a construct; the children of the node represent the meaningful components of the construct.
  - e.g. a syntax-tree node representing an expression  $E_1 + E_2$  has label + and two children representing the sub expressions  $E_1$  and  $E_2$
- Each node is implemented by objects with suitable number of fields; each object will have an *op field that is the label of the node* with additional fields as follows:

- If the node is a leaf, an additional field holds the lexical value for the leaf . This is created by function **Leaf(op, val)**
- If the node is an interior node, there are as many fields as the node has children in the syntax tree. This is created by function **Node(op, c1, c2,...,ck)** .

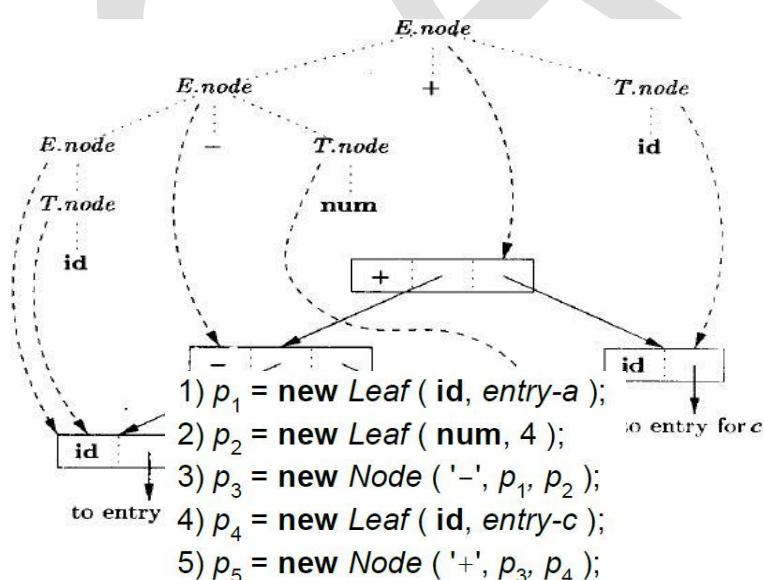
Example: The S-attributed definition in figure below constructs syntax trees for a simple expression grammar involving only the binary operators + and -. As usual, these operators are at the same precedence level and are jointly left associative. All nonterminals have one synthesized attribute *node*, which represents a node of the syntax tree.

Production	Semantic Rules
1) $E \rightarrow E_1 + T$	$E.\text{node} = \text{new Node} ('+', E_1.\text{node}, T.\text{node})$
2) $E \rightarrow E_1 - T$	$E.\text{node} = \text{new Node} ('-', E_1.\text{node}, T.\text{node})$
3) $E \rightarrow T$	$E.\text{node} = T.\text{node}$
4) $T \rightarrow (E)$	$E.\text{node} = T.\text{node}$

### Steps in the construction of the syntax tree for a-4+c

If the rules are evaluated during a post-order traversal of the parse tree, or with reductions during a bottom-up parse, then the sequence of steps shown below ends with p5 pointing to the root of the constructed syntax tree.

Syntax tree for a-4+c using the above SDD is shown below.



### Constructing Syntax Trees during Top-Down Parsing

With a grammar designed for top-down parsing, the same syntax trees are constructed, using the same sequence of steps, even though the structure of the parse trees differs significantly from that of syntax trees.

## 6.PARAMETER PASSING

❖ Explain the various parameter Passing Methods. [Regulation 13-New]

## PARAMETER PASSING

When one procedure calls another the usual method of communication between them is through non local names and through parameter of the called procedure.

### Various parameter passing methods:

- call by value: passing r-values.
- call by reference: passing l-values.
- call by copy-restore: hybrid between call by-value and call by-reference.
- call by name: passing via name substitution.

#### Call by Value

• Each actual argument is evaluated before call. On entry, the resulting value is copied and bound to the formal parameter; which behaves just like a local variable.

- Advantages: –

Simple; easy to understand

• Formal parameters can be used as local variables, Updating them doesn't affect actuals in calling procedure:

```
double hyp( double a, double b )  
{ a = a * a; b = b * b;  
return sqrt( a + b ); // use built-in sqrt() } //end
```

#### Call by Reference

Implemented by passing address of actual parameter

• On entry, the formal is bound to the address, providing a reference to actual parameter from within the subroutine

• If actual argument doesn't have an l-value (e.g., "2 + 3"), then either: – Forbid in language, i.e. treat as an error; compiler catches this – evaluate it into a temporary location and pass its address

- Advantages

No more large copying

Actual parameter can be updated

### Call by Copy-Restore

- Each actual argument is evaluated to a value before call
- On entry, value is bound to formal parameter just like a local
- Updating formal parameters doesn't affect actuals in calling procedure during execution
- Upon exit, the final contents of formals are copied into the actual
  - Thus, behaves like call by reference in most "normal" situations, but may give different results when concurrency or aliasing are involved:

```
type t is record a, b: integer; end record; r : t;  
procedure foo( s : in out t )  
begin r.a := 2; s.a := s.a + 3;  
end foo;  
r.a := 1; foo( r );  
print( r.a );
```

## 7. DESIGN OF A PREDICTIVE TRANSLATOR

- ❖ Explain Design of a Predictive Translator. (Regulation 2013)

### DESIGN OF A PREDICTIVE TRANSLATOR

The construction of predictive parsers to implement a translation scheme based on a grammar suitable for top-down parsing.

**Algorithm:** Construction of a predictive syntax-directed translator.

**Input.** A syntax-directed translation scheme with an underlying grammar suitable for predictive parsing.

**Output.** Code for a syntax-directed translator.

#### Method.

1. For each nonterminal A, construct a function that has a formal parameter for each inherited attribute of A and that returns the values of the synthesized attributes of A.
2. The code for nonterminal A decides what production to use based on the current input symbol.

3.The code associated with each production does the following:

We consider the tokens, nonterminals, and actions on the right side of the production from left to right.

- i) For token X with synthesized attribute x, save the value of x in the variable declared for X. x. then generate a call to match token X and advance the input.
- ii) For nonterminal B, generate an assignment c:=B(b<sub>1</sub>,b<sub>2</sub>,...,b<sub>k</sub>) with a function call on the right side, where b<sub>1</sub>,b<sub>2</sub>,...,b<sub>k</sub> are the variables for the inherited attributes of B and c is the variable for the synthesized attribute of B.
- iii) For an action , copy the code into the parser, replacing each reference to an attribute by the variable for that attribute.

### **Example: Translation Scheme for constructing Syntax Trees**

```

E → T      {R.i := T.nptr}
R      {E.nptr := R.s}

R → +
    T  {R1.i := mknnode ('+', R.i, T.nptr)}
    R1 {R.s := R1.s}

R → -
    T  {R1.i := mknnode ('-', R.i, T.nptr)}
    R1 {R.s := R1.s}

R → ε      {R.s := R.i}

T → (E)  {T.nptr := E.nptr}

T → id   {T.nptr := mkleaf (id, id.entry)}

T → num   {T.nptr := mkleaf (num, num.value)}

```

The above grammar is LL(1), and hence suitable for top-down parsing. From the attributes of the nonterminals in the grammar, we obtain the following types for the arguments and results of the functions for E, R, and T. since E and T do not have inherited attributes, they have no arguments.

```

function E :syntax_tree_node;
function R(i:syntax_tree_node):syntax_tree_node;
function T:syntax_tree_node;

```

we combine two of the r-productions in fig 4.1 to make the translator smaller.

The new productions use token **addop** to represent + and -:

**Parsing procedure for the productions R → addop T R | ε:**

$R \rightarrow addop$

T	{ $R_1.i := \text{mknnode}(addop.\text{lexeme}, R.i, T.\text{nptr})$ }
$R_1$	{ $R.s := R_1.s$ }
$R \rightarrow \epsilon$	{ $R.s := R.i$ }

The code for R is based on the parsing procedure is

**Procedure R;**

```

begin
if lookahead=addop then begin
  match(addop);T;R
end
else begin /*do nothing*/
end
end;

```

If the lookahead symbol is addop, then the production  $R \rightarrow addop T R$  is applied by using the procedure match to read the next input token after addop, and then calling the procedures for T and R. otherwise, the procedure does nothing, to mimic the procedure  $R \rightarrow \epsilon$ .

The procedure for R in

**Recursive-descent construction of syntax trees:**

```

function R(i: $\uparrow$ syntax_tree_node): $\uparrow$ syntax_tree_node;
  var nptr, i1, s1, s:  $\uparrow$ syntax_tree_node;
    addoplexeme : char;
begin
  if lookahead = addop then begin
    /* production  $R \rightarrow addop T R$  */
    match(addop);
    nptr := T;
    i1 := mknnode(addoplexeme, i, nptr);
    s1 := R(i1);
    s := s1;
  end

```

```

else s := i; /* production R → ε */

return s

end;

```

Contains code for evaluating attributes. The lexical value lexval of the token addop is saved in addoplexeme, addop is matched, T is called, and its result is saved using nptr. Variable i1 corresponds to the inherited attribute  $R_1.i$ , and s1 to the synthesized attribute  $R_1.s$ . the **return** statement returns the value of s just before control leaves the function. The functions for E and T are constructed similarly.

## UNIT – 5 CODE OPTIMIZATION AND CODE GENERATION

Principal Sources of Optimization-DAG-Optimization of Basic Blocks-Global Data Flow Analysis-Efficient Data Flow Algorithms-Issues in Design of a Code Generator-A simple Code Generator Algorithm.

### PART – A

#### **1. What are the properties of optimizing compilers? [May /June 2016]**

- The source code should be such that it should produce minimum amount of target code.
- There should not be any unreachable code.
- Dead code should be completely removed from source language.
- The optimizing compilers should apply following code improving transformations on source language.
  - Common subexpression elimination
  - Dead code elimination
  - Code movement
  - Strength reduction

#### **2. List the terminologies used in basic blocks.**

Define and use – the three address statement  $a:=b+c$  is said to define a and to use b and c.

Live and dead – the name in the basic block is said to be live at a given point if its value is used after that point in the program. And the name in the basic block is said to be dead at a given point if its value is never used after that point in the program.

#### **3. What is a DAG? Mention its applications. [May /June 2016]**

Directed acyclic graph(DAG) is a useful data structure for implementing transformations on basic blocks. DAG is used in

- Determining the common sub-expressions.

- Determining which names are used inside the block and computed outside the block.
- Determining which statements of the block could have their computed value outside the block.
- Simplifying the list of quadruples by eliminating the common sub-expressions and not performing the assignment of the form  $x := y$  unless and until it is a must.

#### 4. What is code motion and dead code elimination?

**Code motion** is an optimization technique in which amount of code in a loop is decreased. This transformation is applicable to the expression that yields the same result independent of the number of times the loop is executed. Such an expression is placed before the loop.

**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.

#### 5. What are basic blocks and flow graphs?

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

Eg.  $t1 := a * 5$

$t2 := t1 + 7$

$t3 := t2 - 5$

$t4 := t1 + t3$

$t5 := t2 + b$

**Flow graphs:** A flow graph is a directed graph in which the flow control information is added to the basic blocks.

- The nodes to the flow graph are represented by basic blocks.
- The block whose leader is the first statement is called initial block.
- There is a directed edge from block  $B_1$  to block  $B_2$  if  $B_2$  immediately follows  $B_1$  in the given sequence. We can say that  $B_1$  is a predecessor of  $B_2$ .

#### 6. What are machine idioms and constant folding?

##### Machine idioms:

Some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value. The use of these modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like  $i := i + 1$ .

##### Constant folding:

Deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.

Ex: int i=5; k=i+j; after, int i=5; k 5+j;

## 7. Define peephole optimization. List the characteristics of peephole optimization.

**Definition:** Peephole optimization is a simple and effective technique for locally improving target code. This technique is applied to improve the performance of the target program by examining the short sequence of target instructions and replacing these instructions by shorter or faster sequence.

**Characteristics of peephole optimization:** [Nov/Dec 2016]

- Redundant instruction elimination
- Flow of control optimization
- Algebraic simplification
- Use of machine idioms

## 8. How do you calculate the cost of an instruction?

The cost of an instruction can be computed as one plus cost associated with the source and destination addressing modes given by added cost.

MOV R0,R1	1
MOV R1,M	2
SUB 5(R0),*10(R1)	3

## 9. Identify the constructs for optimization in basic blocks. [Nov/Dec 2016]

Leaders : The first statement of basic blocks.

- The first statement is a leader
- Any statement that is the target of a conditional or unconditional goto is a leader.
- Any statement that immediately follows a goto or conditional goto statement is a leader.

## 9. Generate code for the following C Statement assuming three registers are available :

$x=a/(b+c)-d*(e+f)$ . [April/May 2015]

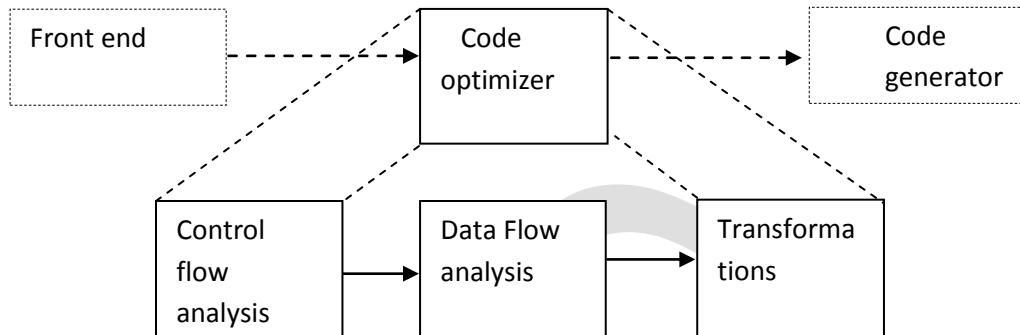
The three address code will be: Code will be :

t1 := b+c	MOV b, R0
t2 := a1/t1	ADD c, R0
t3 := t2-d	MOV a1, R1
t4 := e+f	DIV Ro , R1
t5 := t3 * t4	SUB d , R1
x := t5	MOv f , R2
	ADD e , R2
	MUL R1 , R2
	MOV R2 , x

## 11. What is register descriptor and address descriptor?

- For each available register, a register descriptor keeps track of the variable names whose current value is in that register.
- For each program variable, an address descriptor keeps track of the location or locations where the current value of that variable can be found.

**12. Give the block diagram of organization of code optimizer.**



**13. What do you mean by machine dependent and machine independent optimization?**

- The machine dependent optimization is based on the characteristics of the target machine for the instruction set used and addressing modes used for the instructions to produce the efficient target code.
- The machine independent optimization is based on the characteristics of the programming languages for appropriate programming structure and usage of efficient arithmetic properties in order to reduce the execution time.

**14. Mention the issues to be considered while applying the techniques for code optimization.**

- The semantic equivalence of the source program must not be changed.
- The improvement over the program efficiency must be achieved without changing the algorithm of the program.

**15. What are the different data flow properties?**

- Available expressions
- Reaching definitions
- Live variables
- Busy variables

**16. Write three address code sequence for the assignment statement. [May /June 2016]**

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

The three address code is,

$$\begin{aligned} t1 &:= a-b \\ t2 &:= a-c \\ t3 &:= t1 + t2 \\ t4 &:= t3 + t3 \end{aligned}$$

**d := t4**

## PART-B

### **1.PRINCIPAL SOURCES OF OPTIMIZATION**

- ❖ Write in detail about the Function Preserving Transformation.[Apr/May 2011]
- ❖ Explain loop optimization in detail and apply to an example. [Apr/May 2015]
- ❖ Explain Principal sources of optimization. [Nov/Dec 2014]
- ❖ Write in detail about loop optimization. [Nov/Dec 2013]
- ❖ Describe in detail the principal sources optimization. [Nov/Dec 2011]
- ❖ Explain the principal sources of optimization with examples. [May/Jun 2016]

### **PRINCIPAL SOURCES OF OPTIMIZATION**

- A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global.
- Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

### **Function-Preserving Transformations**

- There are a number of ways in which a compiler can improve a program without changing the function it computes.
- The transformations
  - Common sub expression elimination,
  - Copy propagation,
  - Dead-code elimination, and
  - Constant folding

are common examples of such function-preserving transformations. The other transformations come up primarily when global optimizations are performed.

- Frequently, a program will include several calculations of the same value, such as an offset in an array. Some of the duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language.

#### **➤ Common Sub expressions elimination:**

- An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation. We can avoid recomputing the expression if we can use the previously computed value.

- For example

t1:=4\*i t2:=a[t1]

t3:=4\*j t4:=4\*i

t5:=n

t6:=b [t 4] +t 5

The above code can be optimized using the common sub-expression elimination as

```
t1=4 * i
t2=a[t1]
t3=4*j
t5=n
t6=b[t1] + t5
```

The common sub expression  $t4 := 4*i$  is eliminated as its computation is already in  $t1$ . And value of  $i$  is not been changed from definition to use.

➤ **Copy Propagation:**

- Assignments of the form  $f := g$  called copy statements, or copies for short. The idea behind the copy-propagation transformation is to use  $g$  for  $f$ , whenever possible after the copy statement  $f := g$ . Copy propagation means use of one variable instead of another. This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate  $x$ .
- For example:  $x=Pi;$   
.....  
 $A=x*r*r;$

The optimization using copy propagation can be done as follows:

$A=Pi*r*r;$  Here the variable  $x$  is eliminated

➤ **Dead-Code Eliminations:**

- A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. A related idea is dead or useless code, statements that compute values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations. An optimization can be done by eliminating dead code.

Example:

```
i=0;
if(i=1)
{
    a=b+5;
}
```

Here, 'if' statement is dead code because this condition will never get satisfied.

➤ **Constant folding:**

- We can eliminate both the test and printing from the object code. More generally, deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.
- One advantage of copy propagation is that it often turns the copy statement into dead code.
- For example,

$a=3.14157/2$  can be replaced by

a=1.570 there by eliminating a division operation.

### **Loop Optimizations:**

- We now give a brief introduction to a very important place for optimizations, namely loops, especially the inner loops where programs tend to spend the bulk of their time. The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop.
- Three techniques are important for loop optimization:
  1. code motion, which moves code outside a loop;
  2. Induction -variable elimination, which we apply to replace variables from inner loop.
  3. Reduction in strength, which replaces an expensive operation by a cheaper one, such as a multiplication by an addition.

#### **1.Code Motion:**

- An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed ( a loop-invariant computation) and places the expression before the loop. Note that the notion “before the loop” assumes the existence of an entry for the loop. For example, evaluation of limit-2 is a loop-invariant computation in the following while-statement:

while (i <= limit-2) /\* statement does not change limit\*/

Code motion will result in the equivalent of

t= limit-2;

while (i<=t) /\* statement does not change limit or t \*/

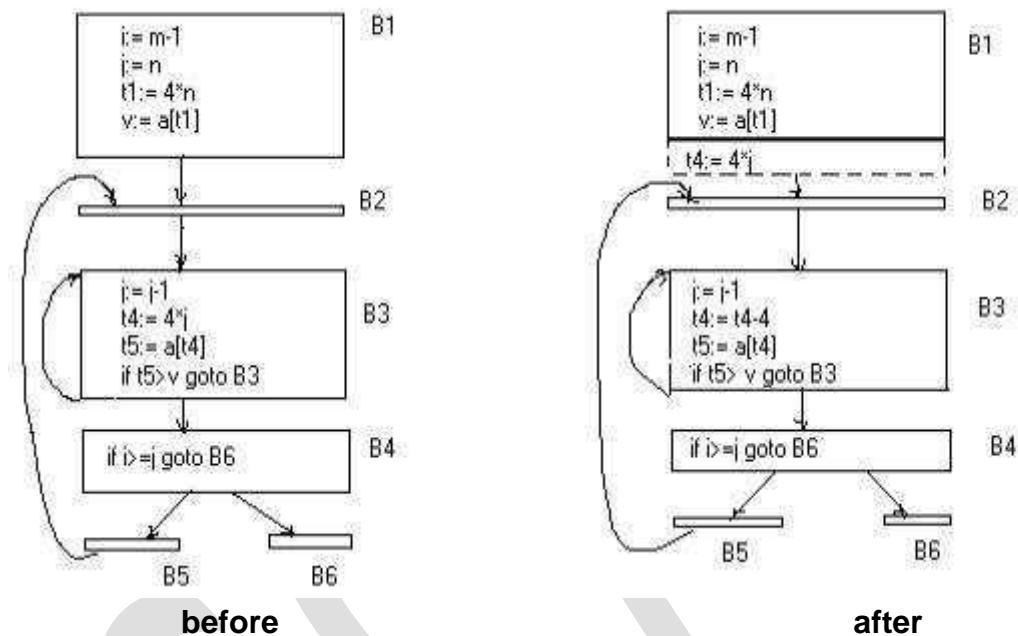
#### **2.Induction Variables :**

- Loops are usually processed inside out. For example consider the loop around B3.
- Note that the values of j and t4 remain in lock-step; every time the value of j decreases by 1, that of t4 decreases by 4 because 4\*j is assigned to t4. Such identifiers are called induction variables.
- When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination. For the inner loop around B3 in Fig. we cannot get rid of either j or t4 completely; t4 is used in B3 and j in B4.

However, we can illustrate reduction in strength and illustrate a part of the process of induction-variable elimination. Eventually j will be eliminated when the outer loop of B2 - B5 is considered.

Example:

As the relationship  $t4 := 4*j$  surely holds after such an assignment to  $t4$  in Fig. and  $t4$  is not changed elsewhere in the inner loop around B3, it follows that just after the statement  $j:=j-1$  the relationship  $t4 := 4*j-4$  must hold. We may therefore replace the assignment  $t4 := 4*j$  by  $t4 := t4-4$ . The only problem is that  $t4$  does not have a value when we enter block B3 for the first time. Since we must maintain the relationship  $t4=4*j$  on entry to the block B3, we place an initialization of  $t4$  at the end of the block where  $j$  itself is initialized, shown by the dashed addition to block B1 in second Fig.



- The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.

### 3.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.

For Example

In block B3

$j=j-1$   
 $t4=4*j$

Can be replaced by

j=j-1  
t4=t4-4

## 2.DAG

- ❖ What are the advantages of DAG Representation? Give example.  
[Apr/May 2015]
- ❖ Describe the algorithm for constructing DAG with an example.  
[Apr/May 2015]
- ❖ Generate DAG Representation with an example and list out the applications of DAG Representation.  
[Nov/Dec 2014]
- ❖ Construct DAG and three address code for the following C Code  
[Nov/Dec 2013]

```
prod=0
i=1
while(i<=20)
{ prod=prod+a[i]*b[i]
i=i+1
}
```

## THE DAG REPRESENTATION FOR BASIC BLOCKS

- A DAG for a basic block is a **directed acyclic graph** with the following labels on nodes:
  - Leaves are labeled by unique identifiers, either variable names or constants.
  - Interior nodes are labeled by an operator symbol.
  - 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

### 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.

### 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.

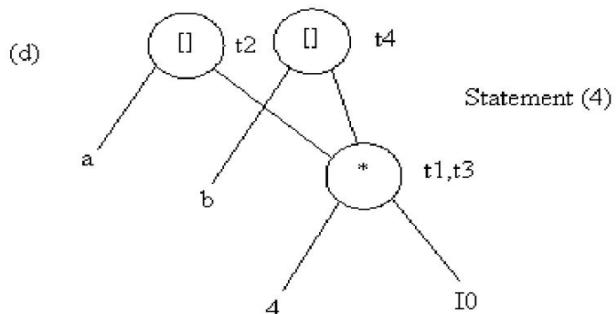
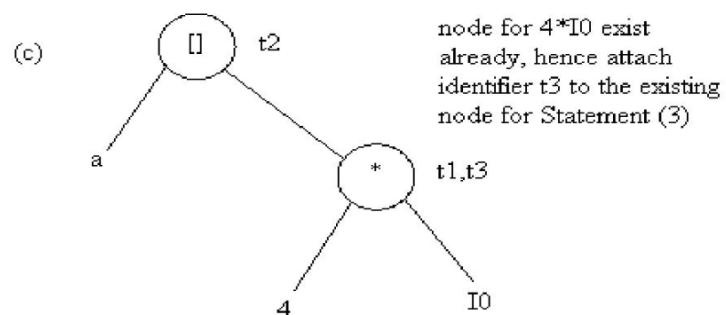
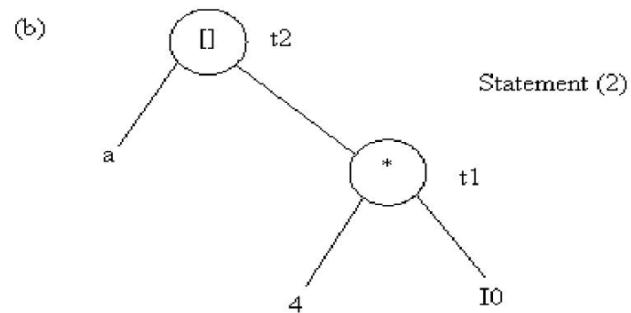
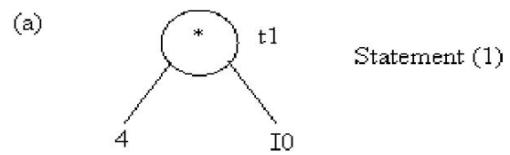
For case(iii), node  $n$  will be node( $y$ ).

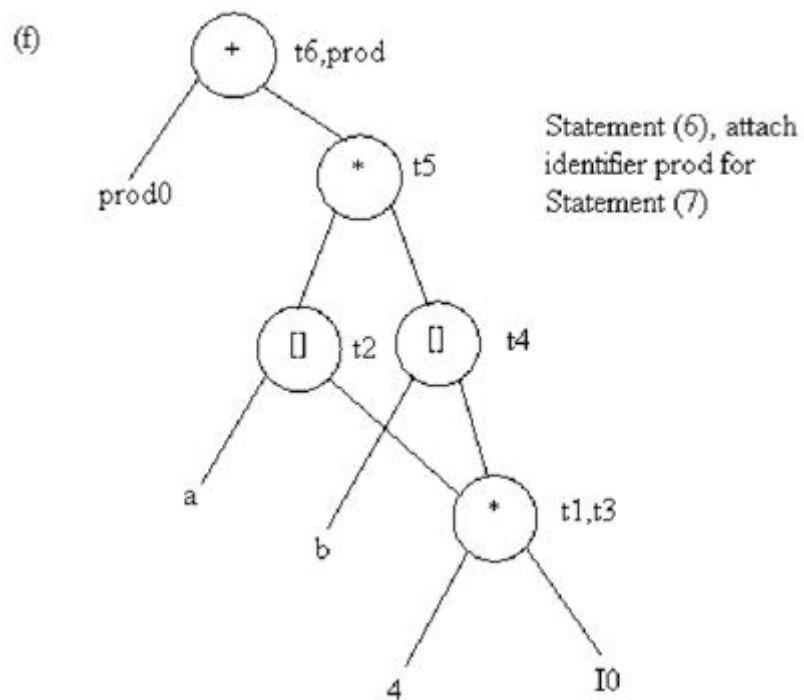
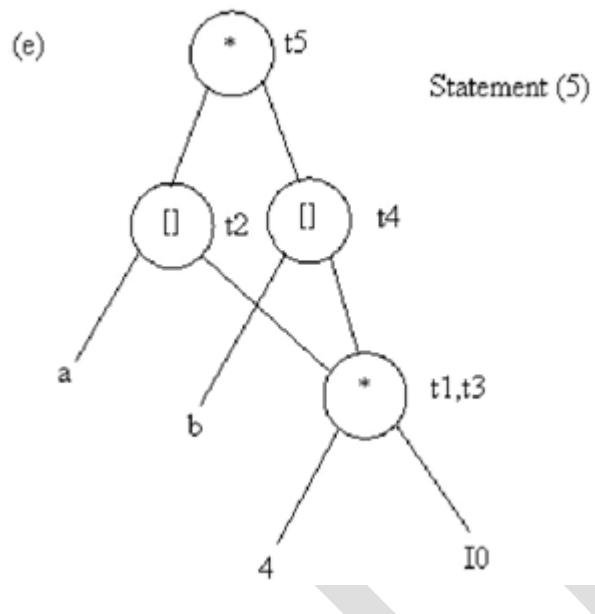
**Step 3:** Delete  $x$  from the list of identifiers for node( $x$ ). Append  $x$  to the list of attached identifiers for the node found in step 2 and set node( $x$ ) to  $n$ .

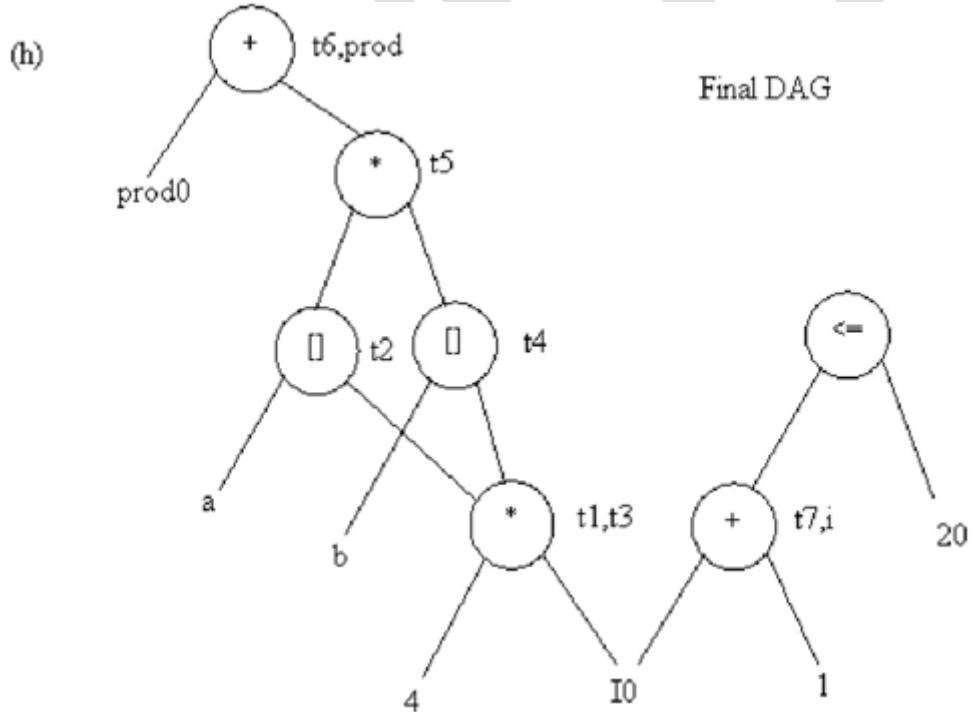
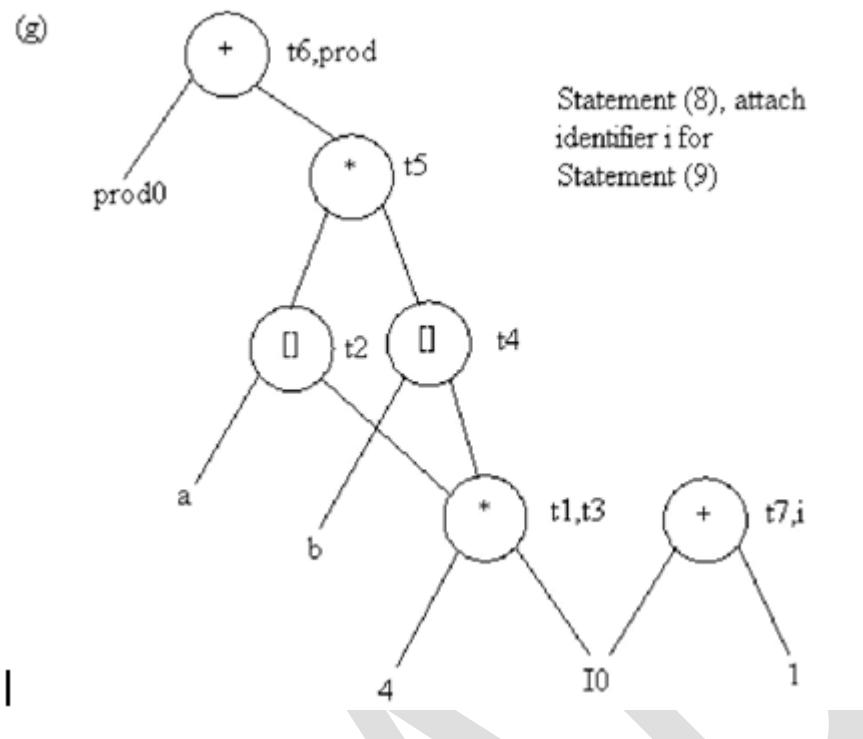
**Example:** Consider the block of three-address statements

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







### **3. OPTIMIZATION OF BASIC BLOCKS**

- ❖ Illustrate the optimization of basic blocks with an example. [Nov/Dec 2014]
- ❖ Discuss in detail the process of optimization of basic blocks. Give an example. [May/Jun 2014]
- ❖ Explain in detail the optimization of basic blocks. [Nov/Dec 2011]

#### **OPTIMIZATION OF BASIC BLOCKS**

There are two types of basic block optimizations. They are :

1. Structure -Preserving Transformations
2. Algebraic Transformations

#### **1. 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  
d=a-d  
c=d+c
```

#### **Dead code elimination:**

It's possible that a large amount of dead (useless) code may exist in the program. This might be especially caused when introducing variables and procedures as part of construction or error -correction of a program – once declared and defined, one forgets to remove them in case they serve no purpose. Eliminating these will definitely optimize the code.

#### **Renaming of temporary variables:**

A statement  $t := b + c$  where  $t$  is a temporary name can be changed to  $u := b + c$  where  $u$  is another temporary name, and change all uses of  $t$  to  $u$ .

In this we can transform a basic block to its equivalent block called normal-form block.

### **Interchange of two independent adjacent statements:**

Two statements

$t1 := b + c$

$t2 := x + y$

can be interchanged or reordered in its computation in the basic block when value of  $t1$  does not affect the value of  $t2$ .

### **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  $\leq$ ,  $\geq$ ,  $<$ ,  $>$ ,  $+$  and  $=$  sometimes generate unexpected common sub expressions.
- Associative laws may also be applied to expose common sub expressions.

For example, if the source code has the assignments

$a := b + c$

$e := c + d + b$

the following intermediate code may be generated:

$a := b + c$

$t := c + d$

$e := t + b$

- Example:

$x := x + 0$  can be removed

$x := y^{**} 2$  can be replaced by a cheaper statement  $x := y * y$

- The compiler writer should examine the language carefully to determine 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  $(a + b) - c$ .

#### **4. GLOBAL DATA FLOW ANALYSIS**

- ❖ Explain Global Data Flow Analysis in detail. [Nov/Dec 2013,2016]
- ❖ What is Data Flow Analysis? Explain Data Flow Analysis with examples. [May/Jun 2014]
- ❖ Write about Data Flow Analysis of Structured Programs. [Nov/Dec 2011]
- ❖ How to trace Data Flow Analysis of Structured Programs. [May/Jun 2012]

#### **INTRODUCTION TO GLOBAL DATAFLOW ANALYSIS**

- In order to do code optimization and a good job of code generation , compiler needs to collect information about the program as a whole and to distribute this information to each block in the flow graph.
- A compiler could take advantage of “reaching definitions” , such as knowing where a variable like *debug* was last defined before reaching a given block, in order to perform transformations are just a few examples of data-flow information that an optimizing compiler collects by a process known as data-flow analysis.
- Data- flow information can be collected by setting up and solving systems of equations of the form :

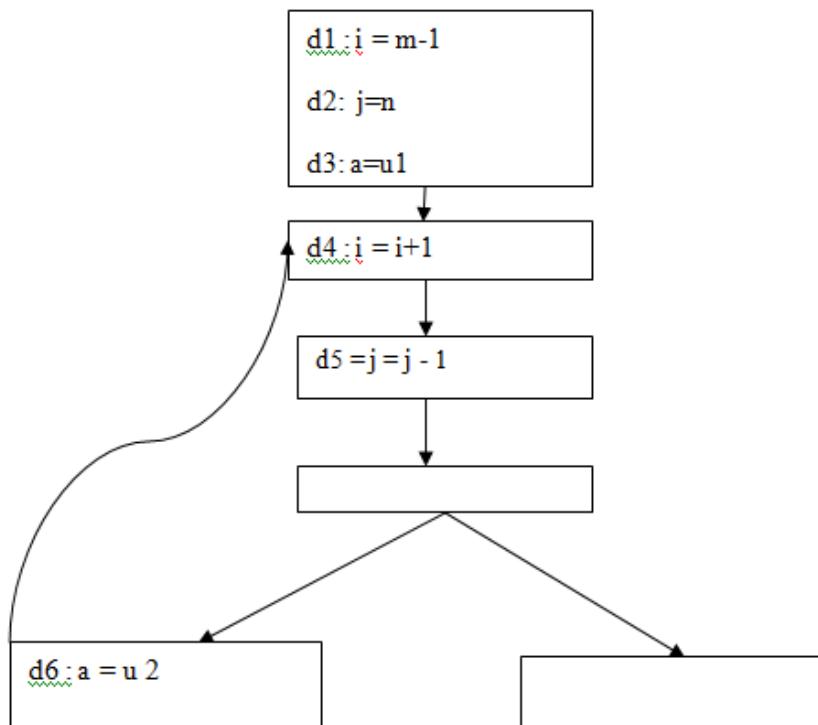
$$\text{out}[S] = \text{gen}[S] \cup (\text{in}[S] - \text{kill}[S])$$

This equation can be read as “ the information at the end of a statement is either generated within the statement , or enters at the beginning and is not killed as control flows through the statement.”

- The details of how data-flow equations are set and solved depend on three factors.
- The notions of generating and killing depend on the desired information, i.e., on the data flow analysis problem to be solved. Moreover, for some problems, instead of proceeding along with flow of control and defining  $\text{out}[s]$  in terms of  $\text{in}[s]$ , we need to proceed backwards and define  $\text{in}[s]$  in terms of  $\text{out}[s]$ .
- Since data flows along control paths, data-flow analysis is affected by the constructs in a program. In fact, when we write  $\text{out}[s]$  we implicitly assume that there is unique end point where control leaves the statement; in general, equations are set up at the level of basic blocks rather than statements, because blocks do have unique end points.
- There are subtleties that go along with such statements as procedure calls, assignments through pointer variables, and even assignments to array variables.

#### **Points and Paths:**

- Within a basic block, we talk of the point between two adjacent statements, as well as the point before the first statement and after the last. Thus, block B1 has four points: one before any of the assignments and one after each of the three assignments.



- Now let us take a global view and consider all the points in all the blocks. A path from  $p_1$  to  $p_n$  is a sequence of points  $p_1, p_2, \dots, p_n$  such that for each  $i$  between 1 and  $n-1$ , either
- $p_i$  is the point immediately preceding a statement and  $p_{i+1}$  is the point immediately following that statement in the same block, or
- $p_i$  is the end of some block and  $p_{i+1}$  is the beginning of a successor block.

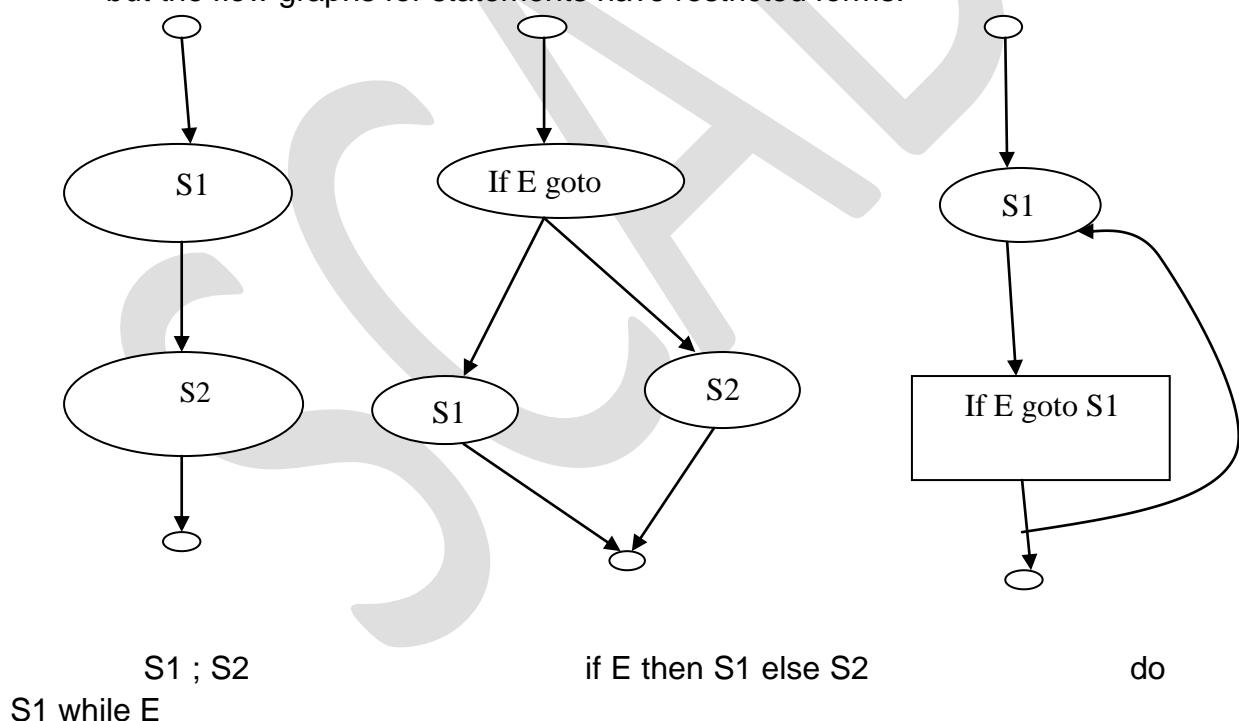
#### Reaching definitions:

- A definition of variable  $x$  is a statement that assigns, or may assign, a value to  $x$ . The most common forms of definition are assignments to  $x$  and statements that read a value from an i/o device and store it in  $x$ .
- These statements certainly define a value for  $x$ , and they are referred to as **unambiguous** definitions of  $x$ . There are certain kinds of statements that may define a value for  $x$ ; they are called **ambiguous** definitions. The most usual forms of **ambiguous** definitions of  $x$  are:
  - A call of a procedure with  $x$  as a parameter or a procedure that can access  $x$  because  $x$  is in the scope of the procedure.
  - An assignment through a pointer that could refer to  $x$ . For example, the assignment  $*q := y$  is a definition of  $x$  if it is possible that  $q$  points to  $x$ . we must assume that an assignment through a pointer is a definition of every variable.

- We say a definition d reaches a point p if there is a path from the point immediately following d to p, such that d is not “killed” along that path. Thus a point can be reached by an unambiguous definition and an ambiguous definition of the same variable appearing later along one path.

### Data-flow analysis of structured programs:

- Flow graphs for control flow constructs such as do-while statements have a useful property: there is a single beginning point at which control enters and a single end point that control leaves from when execution of the statement is over. We exploit this property when we talk of the definitions reaching the beginning and the end of statements with the following syntax.
- $S \rightarrow id := E | S; S | \text{if } E \text{ then } S \text{ else } S | \text{do } S$   
 $\text{while } E \rightarrow id + id | id$
- Expressions in this language are similar to those in the intermediate code, but the flow graphs for statements have restricted forms.



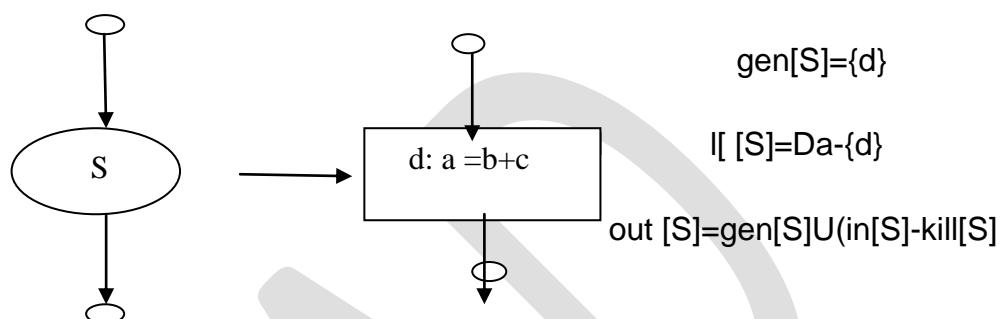
### Structured Control Constructs

- We define a portion of a flow graph called a *region* to be a set of nodes N that includes a header, which dominates all other nodes in the region. All edges between nodes in N are in the region, except for some that enter the header.
- The portion of flow graph corresponding to a statement S is a region that obeys the further restriction that control can flow to just one outside block

when it leaves the region.

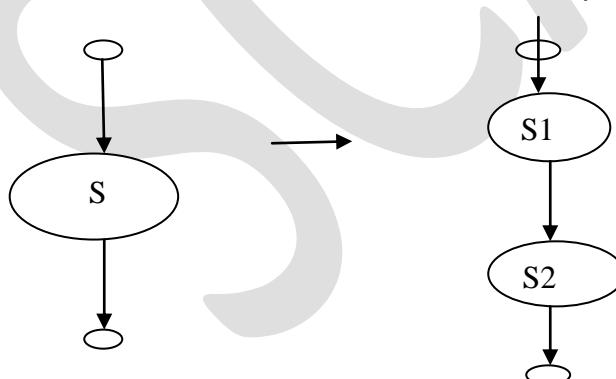
- We say that the beginning points of the dummy blocks at the entry and exit of a statement's region are the beginning and end points, respectively, of the statement. The equations are inductive, or syntax-directed, definition of the sets in[S], out[S], gen[S], and kill[S] for all statements S.
- **gen[S] is the set of definitions “generated” by S while kill[S] is the set of definitions that never reach the end of S.**
- Consider the following data-flow equations for reaching definitions :

i )



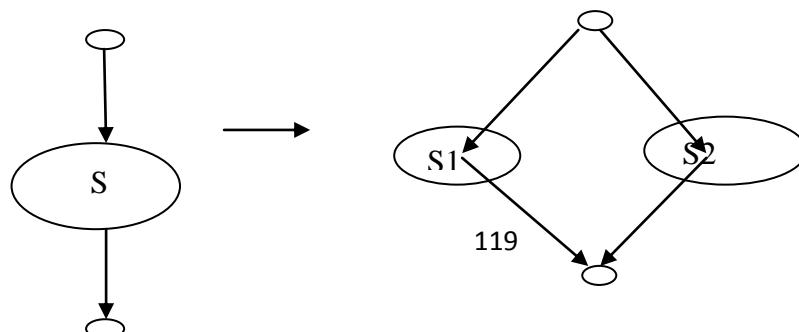
- Observe the rules for a single assignment of variable a. Surely that assignment is a definition of a, say d. Thus
- $\text{Gen}[S] = \{d\}$
- On the other hand, d “kills” all other definitions of a, so we write  $\text{Kill}[S] = \text{Da} - \{d\}$
- Where, Da is the set of all definitions in the program for variable a.

ii )



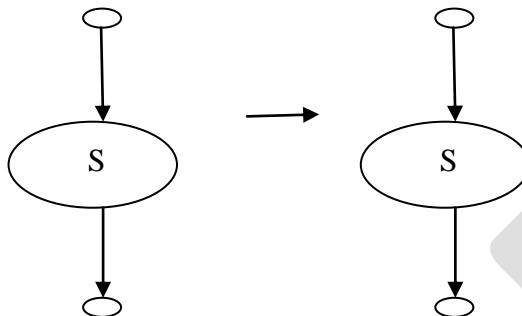
$$\begin{aligned} \text{gen}[S] &= \text{gen}[S_2] \cup (\text{gen}[S_1] - \text{kill}[S_2]) & \text{Kill}[S] &= \\ &= \text{kill}[S_2] \cup (\text{kill}[S_1] - \text{gen}[S_2]) & & \\ \text{in}[S_1] &= \text{in}[S] \text{ in } [S_2] = \text{out}[S_1] \text{ out } [S] = \text{out}[S_2] & & \end{aligned}$$

iii)



$\text{gen}[S] = \text{gen}[S_1] \cup \text{gen}[S_2]$   
 $\text{kill}[S] = \text{kill}[S_1] \cap \text{kill}[S_2]$   
 $\text{in}[S_1] = \text{in}[S]$   
 $\text{in}[S_2] = \text{in}[S]$   
 $\text{out}[S] = \text{out}[S_1]$

iv)



$\text{gen}[S] = \text{gen}[S_1]$   
 $\text{kill}[S] = \text{kill}[S_1]$   
 $\text{in}[S_1] = \text{in}[S] \cup \text{gen}[S_1]$   
 $\text{out}[S] = \text{out}[S_1]$

- Under what circumstances is definition d generated by  $S=S_1; S_2$ ? First of all, if it is generated by  $S_2$ , then it is surely generated by  $S$ . If d is generated by  $S_1$ , it will reach the end of  $S$  provided it is not killed by  $S_2$ . Thus, we write
- $\text{gen}[S] = \text{gen}[S_2] \cup (\text{gen}[S_1] - \text{kill}[S_2])$
- Similar reasoning applies to the killing of a definition, so we have  $\text{Kill}[S] = \text{kill}[S_2] \cup (\text{kill}[S_1] - \text{gen}[S_2])$

#### Conservative estimation of data-flow information:

- There is a subtle miscalculation in the rules for gen and kill. We have made the assumption that the conditional expression E in the if and do statements are “uninterpreted”; that is, there exists inputs to the program that make their branches go either way.
- We assume that any graph-theoretic path in the flow graph is also an execution path, i.e., a path that is executed when the program is run with at least one possible input.
- When we compare the computed gen with the “true” gen we discover that the true gen is always a subset of the computed gen. On the other hand, the true kill is always a superset of the computed kill.
- These containments hold even after we consider the other rules. It is natural to wonder whether these differences between the true and computed gen and kill sets present a serious obstacle to data-flow analysis. The answer

lies in the use intended for these data.

- Overestimating the set of definitions reaching a point does not seem serious; it merely stops us from doing an optimization that we could legitimately do. On the other hand, underestimating the set of definitions is a fatal error; it could lead us into making a change in the program that changes what the program computes. For the case of reaching definitions, then, we call a set of definitions safe or conservative if the estimate is a superset of the true set of reaching definitions. We call the estimate unsafe, if it is not necessarily a superset of the truth.
- Returning now to the implications of safety on the estimation of gen and kill for reaching definitions, note that our discrepancies, supersets for gen and subsets for kill are both in the safe direction. Intuitively, increasing gen adds to the set of definitions that can reach a point, and cannot prevent a definition from reaching a place that it truly reached.

Decreasing kill can only increase the set of definitions reaching any given point.

### **Computation of in and out:**

- Many data-flow problems can be solved by synthesized translations similar to those used to compute gen and kill. It can be used, for example, to determine loop-invariant computations.
- However, there are other kinds of data-flow information, such as the reaching-definitions problem. It turns out that in is an inherited attribute, and out is a synthesized attribute depending on in. we intend that  $\text{in}[S]$  be the set of definitions reaching the beginning of S, taking into account the flow of control throughout the entire program, including statements outside of S or within which S is nested.
- The set  $\text{out}[S]$  is defined similarly for the end of s. it is important to note the distinction between  $\text{out}[S]$  and  $\text{gen}[S]$ . The latter is the set of definitions that reach the end of S without following paths outside S.
- Assuming we know  $\text{in}[S]$  we compute out by equation, that is
- $\text{Out}[S] = \text{gen}[S] \cup (\text{in}[S] - \text{kill}[S])$
- Considering cascade of two statements  $S_1; S_2$ , as in the second case. We start by observing  $\text{in}[S_1] = \text{in}[S]$ . Then, we recursively compute  $\text{out}[S_1]$ , which gives us  $\text{in}[S_2]$ , since a definition reaches the beginning of  $S_2$  if and only if it reaches the end of  $S_1$ . Now we can compute  $\text{out}[S_2]$ , and this set is equal to  $\text{out}[S]$ .
- Considering if-statement we have conservatively assumed that control can follow either branch, a definition reaches the beginning of  $S_1$  or  $S_2$  exactly when it reaches the beginning of S.
- $\text{In}[S_1] = \text{in}[S_2] = \text{in}[S]$
- If a definition reaches the end of S if and only if it reaches the end of one or both sub statements; i.e,

- $\text{Out}[S] = \text{out}[S_1] \cup \text{out}[S_2]$

#### **Representation of sets:**

- Sets of definitions, such as  $\text{gen}[S]$  and  $\text{kill}[S]$ , can be represented compactly using bit vectors. We assign a number to each definition of interest in the flow graph. Then bit vector representing a set of definitions will have 1 in position  $I$  if and only if the definition numbered  $I$  is in the set.
- The number of definition statement can be taken as the index of statement in an array holding pointers to statements. However, not all definitions may be of interest during global data-flow analysis. Therefore the number of definitions of interest will typically be recorded in a separate table.
- A bit vector representation for sets also allows set operations to be implemented efficiently. The union and intersection of two sets can be implemented by logical or and logical and, respectively, basic operations in most systems-oriented programming
- languages. The difference  $A - B$  of sets  $A$  and  $B$  can be implemented by taking the complement of  $B$  and then using logical and to compute  $A \cap \neg B$ .

#### **Local reaching definitions:**

- Space for data-flow information can be traded for time, by saving information only at certain points and, as needed, recomputing information at intervening points. Basic blocks are usually treated as a unit during global flow analysis, with attention restricted to only those points that are the beginnings of blocks.
- Since there are usually many more points than blocks, restricting our effort to blocks is a significant savings. When needed, the reaching definitions for all points in a block can be calculated from the reaching definitions for the beginning of a block.

#### **Use-definition chains:**

- It is often convenient to store the reaching definition information as "use-definition chains" or "ud-chains", which are lists, for each use of a variable, of all the definitions that reaches that use. If a use of variable  $a$  in block  $B$  is preceded by no unambiguous definition of  $a$ , then ud-chain for that use of  $a$  is the set of definitions in  $\text{in}[B]$  that are definitions of  $a$ . In addition, if there are ambiguous definitions of  $a$ , then all of these for which no unambiguous definition of  $a$  lies between it and the use of  $a$  are on the ud-chain for this use of  $a$ .

#### **Evaluation order:**

- The techniques for conserving space during attribute evaluation, also apply to the computation of data-flow information using specifications. Specifically, the only constraint on the evaluation order for the gen, kill, in and out sets for statements is that imposed by dependencies between these sets. Having chosen an evaluation order, we are free to release the space for a set after

all uses of it have occurred.

- Earlier circular dependencies between attributes were not allowed, but we have seen that data-flow equations may have circular dependencies.

#### **General control flow:**

- Data-flow analysis must take all control paths into account. If the control paths are evident from the syntax, then data-flow equations can be set up and solved in a syntax-directed manner.
- When programs can contain goto statements or even the more disciplined break and continue statements, the approach we have taken must be modified to take the actual control paths into account.
- Several approaches may be taken. The iterative method works arbitrary flow graphs. Since the flow graphs obtained in the presence of break and continue statements are reducible, such constraints can be handled systematically using the interval-based methods

However, the syntax-directed approach need not be abandoned when break and continue statements are allowed.

### **5. ISSUES IN THE DESIGN OF A CODE GENERATOR**

- ❖ Write in detail about the issues in the design of a code Generator.(10) [Apr/May 2011 ,May/Jun 2012] [Nov/Dec 2016].
- ❖ Discuss the various issues in the code Generation.[May/Jun 2014,Nov/Dec 2011]
- ❖ Explain various issues in the design of code generator. [May 2016]

### **ISSUES IN THE DESIGN OF A CODE GENERATOR**

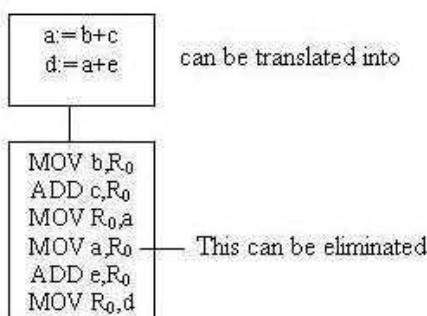
The following issues arise during the code generation phase :

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.
    - 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.

### 6.a. SIMPLE CODE GENERATOR ALGORITHM

- ❖ Explain code generation phase with simple code generation Algorithm.[Nov/Dec 2014]
- ❖ Explain in detail about the simple code generator. [Nov/Dec 2011]
- ❖ Explain code Generation Algorithm in detail.[May/Jun 2012]
- ❖ Write note on simple code generator. [May /June 2016]

## 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)

```

// move c from memory to Rj and
MOV c, Rj Cost = 3      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.

### **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

### 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(Ri), R	2
$a[i] := b$	MOV b, a(Ri)	3

### Generating Code for Pointer Assignments

The table shows the code sequences generated for the pointer assignments  $a := *p$  and  $*p := a$

### 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, R0 ADD z, R0 MOV R0,x CJ< z

Statements	Code Generated	Cost
a := *p	MOV *Rp, a	2
*p := a	MOV a, *Rp	2

## 7. PEEPHOLE OPTIMIZATION

- ❖ Explain peephole optimization in detail.
- ❖ Write an algorithm for constructing natural loop of a back edge. Nov : 16

### PEEPHOLE OPTIMIZATION

Many simple transformations can significantly improve the running time or space requirement of the target program.

A simple but effective technique for locally improving the target code is *peephole optimization*, which is done by examining a sliding window of target instructions (called the *peephole*) and replacing instruction sequences within the peephole by a shorter or faster sequence, whenever possible. Peephole optimization can also be applied directly after intermediate code generation to improve the intermediate representation. The peephole is a small, sliding window on a program.

The characteristics of Peephole Optimization:

- Redundant-instruction elimination
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms

### Eliminating Redundant Loads and Stores

If we see the instruction sequence

LD a, R<sub>0</sub>

ST R<sub>0</sub>, a

in a target program, we can delete the store instruction because whenever it is executed, the first instruction will ensure that the value of a has already been loaded into register R<sub>0</sub>. Note that if the store instruction had a label, we could not be sure that the first instruction is always executed before the second, so we could not remove the store instruction. Put another way, the two instructions have to be in the same basic block for this transformation to be safe.

### Eliminating Unreachable Code

Another opportunity for peephole optimization is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain code fragments that are executed only if a variable `debug` is equal to 1. In the intermediate representation, this code may look like

```
if debug == 1 goto L1  
goto L2  
L1: print debugging information  
L2:
```

One obvious peephole optimization is to eliminate jumps over jumps. Thus, no matter what the value of `debug`, the code sequence above can be replaced by

```
if debug != 1 goto L2  
print debugging information  
L2:
```

If `debug` is set to 0 at the beginning of the program, constant propagation would transform this sequence into

```
if 0 != 1 goto L2  
print debugging information  
L2:
```

Now the argument of the first statement always evaluates to **true**, so the statement can be replaced by **goto L2**. Then all statements that print debugging information are unreachable and can be eliminated one at a time.

### Flow-of-Control Optimizations

Simple intermediate code-generation algorithms frequently produce jumps to jumps, jumps to conditional jumps, or conditional jumps to jumps. These unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the sequence

```
goto L1  
L1: goto L2  
by the sequence  
goto L2  
L1: goto L2
```

If there are now no jumps to **L1**, then it may be possible to eliminate the statement **L1: goto L2** provided it is preceded by an unconditional jump. Similarly, the sequence

```
if a < b goto L1  
L1: goto L2
```

Can be replaced by the sequence

```
if a < b goto L2  
L1: goto L2
```

Finally, suppose there is only one jump to **L1** and **L1** is preceded by an unconditional goto. Then the sequence

**goto L1**

**L1: if a < b goto L2**

**L3:**

may be replaced by the sequence

**if a < b goto L2**

**goto L3**

**L3:**

While the number of instructions in the two sequences is the same, we sometimes skip the unconditional jump in the second sequence, but never in the first. Thus, the second sequence is superior to the first in execution time.

These algebraic identities can also be used by a peephole optimizer to eliminate three-address statements such as

$x = x + 0$

or

$x = x * 1$

in the peephole. Similarly, reduction-in-strength transformations can be applied in the peephole to replace 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 approximated as multiplication by a constant, which may be cheaper.

### **Use of Machine Idioms**

The target machine may have hardware instructions to implement certain specific operations efficiently. Detecting situations that permit the use of these instructions can reduce execution time significantly. For example, some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value. The use of the modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like  $x = x + 1$ .

## Industrial / Practical Connectivity of the subject

### Industry Connectivity and Latest Developments

#### Industry Connectivity

We have collaborated with Infosys Pvt Limited, called Infosys campus connect. Based on the Industry requirement Problem Solving Techniques (PST) such as NFA, DFA design for tokens are practiced among the students.

#### Latest Development Tools

1. Selenium IDE
2. JFLAP 7.0



B.E/B.Tech DEGREE EXAMINATION, MAY/JUNE 2016

Sixth semester

Computer science and Engineering

CS6660 - COMPILER DESIGN

(Regulation 2013)

Answer ALL questions

PART- A (10 x 2 =20)

- |   |              |
|---|--------------|
| 1. What are the two parts of compilation?                           | Pg.No : 5    |
| 2. Illustrate diagrammatically how a language is processed.         | Pg .No : 7   |
| 3. Write a grammar for branching statements.                        | Pg. NO : 23  |
| 4. List the operations on languages.                                | Pg.NO : 23   |
| 5. Write an algorithm for FIRST and FOLLOW in parser.               | Pg.Nn : 39   |
| 6. Define ambiguous grammar.  | Pg.No : 38   |
| 7. What is DAG ?  | Pg .N o :101 |
| 8. when does dangling reference occurs?                             | Pg. No : 76  |
| 9. What are the properties of optimizing compilers?                 | Pg. No : 101 |
| 10. Write three address code sequence for the assignment statement. | Pg. No : 104 |

PART – B (5x16=80 Marks )

- 11 . a) Describe the various phases of compiler and trace it with the program segment (position =initial+rate \* 60 ). (16) Pg.No : 11

OR

- b) i) Explain Language processing system wit hneat diagram. (8) Pg.No:7  
ii) Explain the need for grouping of phases. (4) Pg.No : 19  
iii) Explain Various Error Encountered in different phases of compiler. (4)  
Pg.No : 16

- 12.a) i) Differentiate between Token, Pattern and Lexeme.(6) Pg.NO : 21  
ii) What are the issues of lexical analysis? (4) Pg.NO : 23  
iii) Write notes on Regular Expressions. (6) Pg.NO : 34

OR

- b) i) Write notes on Regular Expressions to NFA. Construct Regular Expressions to NFA for the sentence  $(a/b)^*a$ . (10) Pg.No : 34  
 ii ) Construct DFA to recognize the language  $(a/b)^*ab$ . (6) Pg.No : 34

13. a) i) Consider Stack implementation of shift reduce parsing for the grammar:

$$\begin{aligned} E &\rightarrow E+E \\ E &\rightarrow E^*E \\ E &\rightarrow (E) \\ E &\rightarrow \text{id} \text{ and the input string } id_1+id_2^*id_3 \end{aligned} \quad (8) \quad \text{Pg.No :57}$$

ii ) Explain LL(1) grammar for the sentence  $S \rightarrow iEtS \mid iEtSeS \mid a$  E->b( 8)  
 Pg .No:57

OR

- b) i)Write an algorithm for Non recursive predictive parsing. (6)Pg.No : 54  
 ii ) Explain context free grammars with examples. (10) Pg.No :44

14.a) i) Construct a syntax directed definition for constructing a syntax tree for assignment statements. (8) Pg.No:89

$$\begin{aligned} S &\rightarrow \text{id}:=E \\ E &\rightarrow E_1 +E_2 \\ E &\rightarrow E_1^*E_2 \\ E &\rightarrow -E_1 \\ E &\rightarrow (E_1) \\ E &\rightarrow \text{id} \end{aligned}$$

ii) Discuss specification of a simple type checker. (8) Pg.No : 87  
 OR

b) Discuss different storage allocation strategies. (16) Pg.NO:79

15. a) Explain principal sources of optimization with examples. (16) Pg.No : 104

Or

- b) i) Explain various issues in the design of code generator.(8) Pg.No :123  
 ii ) Write note on simple code generator. (8) Pg.No :125

B.E/B.Tech DEGREE EXAMINATION, NOV / DEC 2016

Sixth semester

Computer science and Engineering

CS6660 - COMPILER DESIGN

(Regulation 2013)

Answer ALL questions

PART- A (10 x 2 =20)

1. What is symbol Table? [Pg.No :6]
2. List the various compiler construction tools. [Pg.No :7]
3. List the rules that form the BASIS. [Pg.No :22]
4. Differentiate Token, Pattern and Lexeme. [Pg.No :21]
5. Construct the parse tree for -(id+id) [Pg.No :38]
6. What is meant by handle pruning? [Pg.No :40]
7. Write down syntax directed definition of a simple desk calculator. [Pg.No :75]
8. List Dynamic Storage allocation techniques. [Pg.No :76]
9. Identify the constructs for optimization in basic blocks. [Pg.No :103]
10. What are the Characteristics of peephole optimization? [Pg.No :102]

PART – B (5x16=80)

- 11.(a) (i) Explain the phases of compiler with a neat diagram.(10) [Pg.No :11]

- (ii) Write notes on compiler construction tools. (6) [Pg.No :19]

OR

- (b) (i) Explain the need for grouping of phases. (8) [Pg.No :19]

- (ii) Explain the various errors encountered in different phases of compiler.(8)

[Pg.No :16]

- 12.(a) (i) Discuss the role of lexical analyzer in detail with necessary examples. (8)

[Pg.No :23]

- (ii) Discuss how FA is used to represent tokens and perform lexical analysis with examples. (8) [Pg.No :27]

OR

- (b) (i) Conversion of regular expression  $(a/b)^*$  abb to NFA. (8) [Pg.No :34]

- (ii) Write an algorithm for minimizing the number of states of a DFA. (8)  
[Pg.No :35]

- 13.(a) (i) Construct parse tree for the input w= cad using top down parser.(6)

S->cAd [Pg.No :50]  
 A->ab| a

(ii) Construct parsing table for the grammar and find moves made by predictive parser on input id+id\*id and find FIRST and FOLLOW. (10) [Pg.No :55]

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

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

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

OR

(b) (i) Explain ambiguous grammar G :  $E \rightarrow E + E | E^* E | (E) | -E | id$  for the sentence id+id\*id. (6) [Pg.No :44]

(ii) Construct SLR Parsing table for the following grammar:

$$G : E \rightarrow E + T \mid T \quad T \rightarrow T * F \mid F \quad F \rightarrow (E) \mid id. \quad (10) \quad [Pg.No :64]$$

14 . (a) (i) Construct a syntax directed definition scheme that takes strings of a's,b's and c's as input and produces as output the number of substrings in the input string that correspond to the pattern  $a(a/b)^*c + (a/b)^*b$ . For example the translation of the input string "abbcabcababc" is "3".

- 4. Write a context free grammar that generate all strings of a's,b's and c's
- 5. Give the semantic attributes for the grammar symbols.

For each productions of the grammar a set of rules for evaluation o the semantic attributes. (8) [Pg.No :94]

(ii) Illustrate type checking with necessary diagram . (8) [Pg.No :83]

OR

(b) Explain the following with repect to code generation phase. (16)

- I. Input to code generator. [Pg.No :123]
- II. Target program.
- III. Memory management.
- IV. Instruction Selection.
- V. Registr allocation.
- VI. Evaluation order.

15. (a) (i) Write an algorithm for constructing natural loop of a back edge. (8)

[Pg.No :128]

(ii) Explain any four issues that crop up when designing a code generator. (8)

[Pg.No :123]

OR

(b) Expalin global data flow analysis with necessary equations. (16)

[Pg.No :116]

Reg. No. : 

--	--	--	--	--	--	--	--	--	--	--	--

## Question Paper Code : 71391

B.E./B.Tech. DEGREE EXAMINATION, APRIL/MAY 2015.

Sixth Semester

Computer Science and Engineering

CS 2352/CS 62/10144 CS 602 — PRINCIPLES OF COMPILER DESIGN

(Regulation 2008/2010)

(Common to PTCS 2352 – Principles of Compiler Design for B.E. (Part-Time) Fifth Semester – Computer Science and Engineering – Regulation 2009)

Time : Three hours

Maximum : 100 marks

Answer ALL questions.

PART A — (10 × 2 = 20 marks)

1. Describe the error recovery schemes in the lexical phase of a compiler.
2. Write a regular Definition to represent date in the following format : JAN-5<sup>th</sup> 2014.
3. What is the role of a passer?
4. Construct a decorated parse tree according to the syntax directed definition, for the following input statement:  $(4 + 7.5 * 3) / 2$
5. Write the 3-address code for ;  $x = *y$  ;  $a = \&x$ .
6. Place the above generated 3-address code in Triplets and Indirect Triplets.
7. What role does the target machine play on the code generation phase of the compiler?
8. How is Liveness of a variable calculated
9. Generate code for the following C statement assuming three registers are available :  $x = a/(b+c) - d*(e+f)$ .
10. Write the algorithm that orders the DAG nodes for generating optimal target code.

**PART B — (5 × 16 = 80 marks)**

11. (a) Prove that the following two regular expressions are equivalent by showing that the minimum state DFA's are same.

(i)  $(a / b)^*$  (8)

(ii)  $(a^* / b^*)^*$  (8)

Or

- (b) (i) Describe the error recovery schemes in the lexical phase of a compiler (8)

- (ii) Mention any four compiler construction tools with their benefits and drawbacks. (8)

12. (a) (i) Generate SLR Parsing Table for the following grammar (12)

$$S \rightarrow Aa \mid bAc \mid Bc \mid bBa$$

$$A \rightarrow d$$

$$B \rightarrow d$$

And parse the sentence "bdc" and "dd".

- (ii) Mention in detail any 4 issues in storage organization. (4)

Or

- (b) (i) Write down the algorithm to eliminate left-recursion and left-factoring and apply both to the following grammar (8)

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

$$T \rightarrow a \mid b \mid (E)$$

- (ii) Give a syntax-directed definition to differentiate expressions formed by applying the arithmetic operators + and \* to the variable x and constants ; expression :  $x^*(3*x + x*x)$ . (8)

13. (a) For the given program fragment  $A[i, j] = B[i, k]$  do the following :

- (i) Draw the annotated parse tree with the translation scheme to convert to three address code (6)

- (ii) Write the 3-address code (6)

- (iii) Determine the address of  $A[3,5]$  where, all are integer arrays with size of A as  $10 \times 10$  and B as  $10 \times 10$  with  $k=2$  and the start index position of all arrays is at 1. (Assume the base addresses) (4)

Or

- (b) (i) Apply Back-patching to generate intermediate code for the following input.

```
x : 2 + y ;
if x < y then x := x + y ;
repeat
```

```
y := y * 2 ;
while x >= 10 do x := x/2
```

until x < y  
Write the semantic rule and derive the Parse tree for the given code. (12)

- (ii) What is an Activation Record? Explain how its relevant to the intermediate code generation phase with respect to procedure declarations. (4)

14. (a) (i) Write the Code Generation Algorithm using Dynamic Programming and generate code for the statement  $x = a / (b - c) - s^*(e + f)$ . [Assume all instructions to be unit cost] (12)

- (ii) What are the advantages of DAG representation? Give example. (4)

Or

- (b) (i) Write the procedure to perform Register Allocation and Assignment with Graph Coloring. (8)

- (ii) Construct DAG and optimal target code for the expression  $x = ((a + b) / (b - c)) - (a + b)^* (b - c) + f$ . (8)

15. (a) Perform analysis of available expressions on the following code by converting into basic blocks and compute global common sub expression elimination

- (i)  $i := 0$
- (ii)  $a := n - 3$
- (iii) IF  $i < a$  THEN loop ELSE end
- (iv) LABEL loop
- (v)  $b := i - 4$
- (vi)  $c := p + b$
- (vii)  $d := M[c]$
- (viii)  $e := d - 2$
- (ix)  $f := i - 4$
- (x)  $g := p + f$
- (xi)  $M[g] := e$
- (xii)  $i := i + 1$
- (xiii)  $a := n - 3$
- (xiv) IF  $i < a$  THEN loop ELSE end
- (xv) LABEL end.

(16)

Or

- (b) (i) Explain Loop optimization in detail and apply it to the code in 15 (a). (10)

- (ii) What are the optimization techniques applied on procedures calls? Explain with example (6)

Reg. No. : 

--	--	--	--	--	--	--	--	--	--	--	--

## Question Paper Code : 91355

B.E./B.Tech. DEGREE EXAMINATION, NOVEMBER/DECEMBER 2014.

Sixth Semester

Computer Science and Engineering

CS 2352/CS 62/10144 CS 602 — PRINCIPLES OF COMPILER DESIGN

(Regulation 2008/2010)

(Common to PTCS 2352—Principles of Compiler Design for B.E. (Part-Time) Fifth Semester—Computer Science and Engineering—Regulation 2009)

Time : Three hours

Maximum : 100 marks

Answer ALL questions.

PART A — (10 × 2 = 20 marks)

1. What is the role of lexical analyzer?
2. Write regular expression to describe a languages consist of strings made of even numbers a and b.
3. List out the various storage allocation strategies.
4. Write a CF grammar to represent palindrome.
5. What are the types of intermediate languages?
6. Give syntax directed translation for case statement.
7. Differentiate between basic block and flow graph.
8. Draw DAG to represent  $a[i] = b[i]; a[i] = \& t;$
9. Represent the following in flow graph  
 $i = 1; sum = 0; while(i <= 10)\{sum += i; i++;}$
10. What is global data flow analysis?

PART B — (5 × 16 = 80 marks)

11. (a) (i) Explain the need for grouping of phases of compiler. (8)  
(ii) Explain a language for specifying the lexical analyzer. (8)

Or

- (b) (i) Write short notes on compiler construction tools. (8)  
(ii) Explain — specification and recognition of tokens. (8)

12. (a) (i) Explain the specification of simple type checker. (8)  
 (ii) Explain — runtime environment with suitable example. (8)

Or

- (b) Find the LALR for the given grammar and parse the sentence  $(a+b)^*c$ .  
 $E \rightarrow E + T / T, T \rightarrow T * F / F, F \rightarrow (E) / id$ . (16)

13. (a) Generate intermediate code for the following code segment along with the required syntax directed translation scheme

While ( $i < 10$ )

If ( $i \% 2 == 0$ )

Evensum = evensum + i;

Else

Oddsum = oddsum + i;

Or

- (b) Generate intermediate code for the following code segment along with the required syntax directed translation scheme. (16)  
 $s=s+a[i][j];$

14. (a) (i) Explain register allocation and assignment with suitable example. (8)

- (ii) Explain — code generation phase with simple code generation algorithm. (8)

Or

- (b) (i) Generate DAG representation of the following code and list out the applications of DAG representation. (8)

$i = 1; \text{while } (i \leq 10) \text{ do}$

$\text{sum} + \frac{a}{i};$

- (ii) Explain — Generating code from DAG with suitable example. (8)

15. (a) (i) Explain — principle sources of optimization. (8)

- (ii) Illustrate optimization of basic blocks with an example. (8)

Or

- (b) Explain peephole optimization and various code improving Transformations. (16)

Reg. No. :

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

## Question Paper Code : 51353

B.E./B.Tech. DEGREE EXAMINATION, MAY/JUNE 2014.

Sixth Semester

Computer Science and Engineering

CS 2352/CS 62/10144 CS 602 — PRINCIPLES OF COMPILER DESIGN

(Regulation 2008/2010)

(Common to PTCS 2352 – Principles of Compiler Design for B.E. (Part-Time) Fifth Semester – Computer Science and Engineering – Regulation 2009)

Time : Three hours

Maximum : 100 marks

Answer ALL questions.

PART A — (10 × 2 = 20 marks)

1. State any two reasons as to why phases of compiler should be grouped.
2. Why is buffering used in lexical analysis? What are the commonly used buffering methods?
3. Define Lexeme.
4. Compare the features of DFA and NFA.
5. What is the significance of intermediate code?
6. Write the various three address code form of intermediate code.
7. Define symbol table.
8. Name the techniques in loop optimization.
9. What do you mean by Cross-Compiler?
10. How would you represent the dummy blocks with no statements indicated in global data flow analysis?  
141

**PART B — (5 × 16 = 80 marks)**

11. (a) (i) Define the following terms : Compiler, Interpreter, Translator and differentiate between them. (6)  
 (ii) Differentiate between lexeme, token and pattern. (6)  
 (iii) What are the issues in lexical analysis? (4)

Or

- (b) Explain in detail the process of compilation. Illustrate the output of each phase of compilation for the input “ $a = (b + c)^* (b + c)^* 2$ ”.

12. (a) Consider the following grammar

$$S \rightarrow AS \mid b$$

$$A \rightarrow SA \mid a.$$

Construct the SLR parse table for the grammar. Show the actions of the parser for the input string “abab”.

Or

- (b) (i) What is an ambiguous grammar? Is the following grammar ambiguous? Prove  $E \rightarrow E + E \mid E * E \mid (E) \mid id$ . The grammar should be moved to the next line, centered.  
 (ii) Draw NFA for the regular expression  $ab^*/ab$ .

13. (a) How would you convert the following into intermediate code? Give a suitable example.

- (i) Assignment statements. (8)  
 (ii) ‘Case’ statements. (8)

Or

- (b) (i) Write notes on backpatching.  
 (ii) Explain the sequence of stack allocation processes for a function call.

14. (a) Discuss the various issues in code generation with examples.

Or

- (b) Define a Directed Acyclic Graph. Construct a DAG and write the sequence of instructions for the expression  $a + a * (b - c) + (b - c) * d$ .

15. (a) Discuss in detail the process of optimization of basic blocks. Give an example.

Or

- (b) What is data flow analysis? Explain data flow abstraction with examples.

B.E/B.Tech DEGREE EXAMINATION, MAY/JUNE 2012  
Sixth semester  
Computer science and Engineering  
CS2352/Cs62/10144 Cs602-PRINCIPLES OF COMPILER DESIGN  
(Regulation 2008)  
Answer ALL questions  
**PART- A**

1. Mention few cousins of compiler.
2. What are the possible error recovery actions in lexical analyzer?
3. Define an ambiguous grammar.
4. What is dangling reference?
5. Why are quadruples preferred over triples in an optimizing compiler?
6. List out the motivations for back patching.
7. Define flow graph.
8. How to perform register assignment for outer loops?
9. What is the use of algebraic identities in optimization of basic blocks?
10. List out two properties of reducible flow graph?

**PART B**

11. (a) (i) What are the various phases of the compiler? Explain each phase in detail.

(ii) Briefly explain the compiler construction tools.

OR

(b) (i) What are the issues in lexical analysis?

(ii) Elaborate in detail the recognition of tokens.

12. (a) (i) Construct the predictive parser for the following grammar.

S->(L)/a

L->L,S/S

(ii) Describe the conflicts that may occur during shift reduce parsing.

OR

(b) (i) Explain the detail about the specification of a simple type checker.

(ii) How to subdivide a run-time memory into code and data areas.

Explain

13. (a) (i) Describe the various types of three address statements.

(ii) How names can be looked up in the symbol table? Discuss.

OR

(b) (i) Discuss the different methods for translating Boolean expressions in detail.

(ii) Explain the following grammar for a simple procedure call statement

S->call id(enlist).

14. (a) (i) Explain in detail about the various issues in design of code generator.

(ii) Write an algorithm to partition a sequence of three address statements into basic blocks.

OR

- (b) (i) Explain the code-generation algorithm in detail.  
(ii) Construct the dag for the following basic block.

$d := b * c$

$e := a + b$

$b := b * c$

$a := e - d$

15. (a) (i) Explain the principal sources of optimization in detail.  
(ii) Discuss the various peephole optimization techniques in detail.

OR

- (b) (i) How to trace data-flow analysis of structured program?

- (ii) Explain the common sub expression elimination, copy propagation, and transformation for moving loop invariant computations in detail.