

UNIT-II

SYNTAX ANALYSIS AND RUNTIME ENVIRONMENT

SYNTAX ANALYSIS

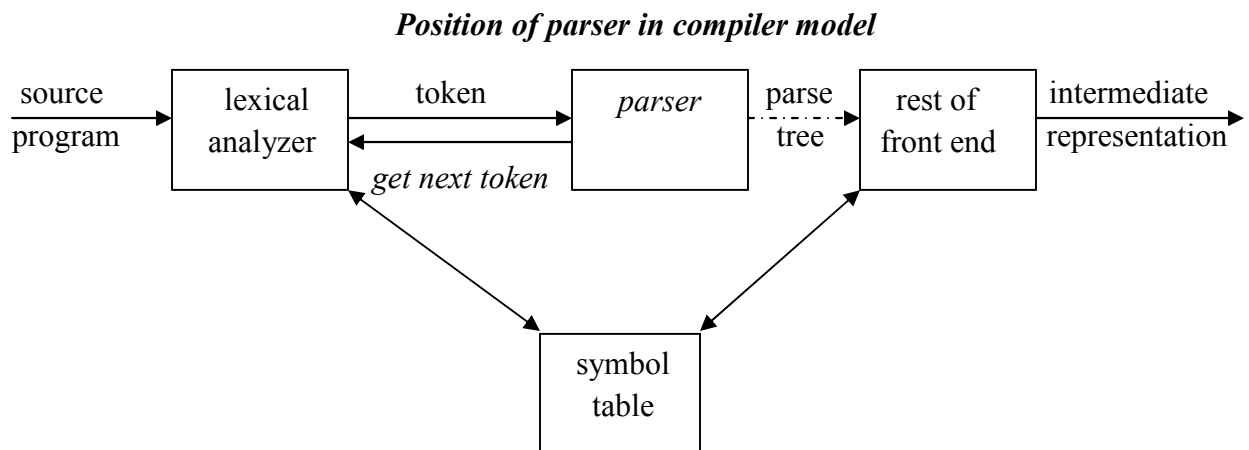
Syntax analysis is the second phase of the compiler. It gets the input from the tokens and generates a syntax tree or parse tree.

Advantages of grammar for syntactic specification :

1. A grammar gives a precise and easy-to-understand syntactic specification of a programming language.
2. An efficient parser can be constructed automatically from a properly designed grammar.
3. A grammar imparts a structure to a source program that is useful for its translation into object code and for the detection of errors.
4. New constructs can be added to a language more easily when there is a grammatical description of the language.

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.



Functions of the parser :

1. It verifies the structure generated by the tokens based on the grammar.
2. It constructs the parse tree.
3. It reports the errors.
4. It performs error recovery.

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.

Functions of error handler :

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 parser 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.

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:

$$expr \rightarrow expr\ op\ expr$$
$$expr \rightarrow (expr)$$
$$expr \rightarrow -\ expr$$
$$expr \rightarrow id$$
$$op \rightarrow +$$
$$op \rightarrow -$$
$$op \rightarrow *$$
$$op \rightarrow /$$
$$op \rightarrow \uparrow$$

In this grammar,

- **id** + - * / \uparrow () 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

1. $E \rightarrow - E$
2. $E \rightarrow - (E)$
3. $E \rightarrow - (E+E)$
4. $E \rightarrow - (id+E)$
5. $E \rightarrow - (id+id)$

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

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**.

Sentinels:

Given a grammar G with start symbol S, if $S \rightarrow \alpha$, where α may contain non-terminals or terminals, then α 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.

Ambiguity:

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

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

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

$E \rightarrow E + E$

$E \rightarrow id + E$

$E \rightarrow id + E * E$

$E \rightarrow id + id * E$

$E \rightarrow id + id * id$

$E \rightarrow E * E$

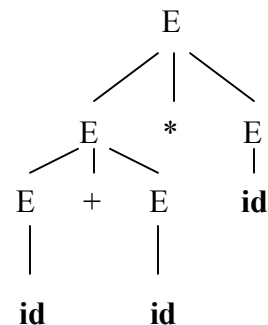
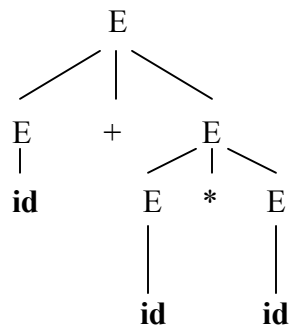
$E \rightarrow E + E * E$

$E \rightarrow id + E * E$

$E \rightarrow id + id * E$

$E \rightarrow id + id * id$

The two corresponding parse trees are :



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.

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 transition diagram .	It is used to check whether the given input is valid or not using derivation .
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.

- 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.

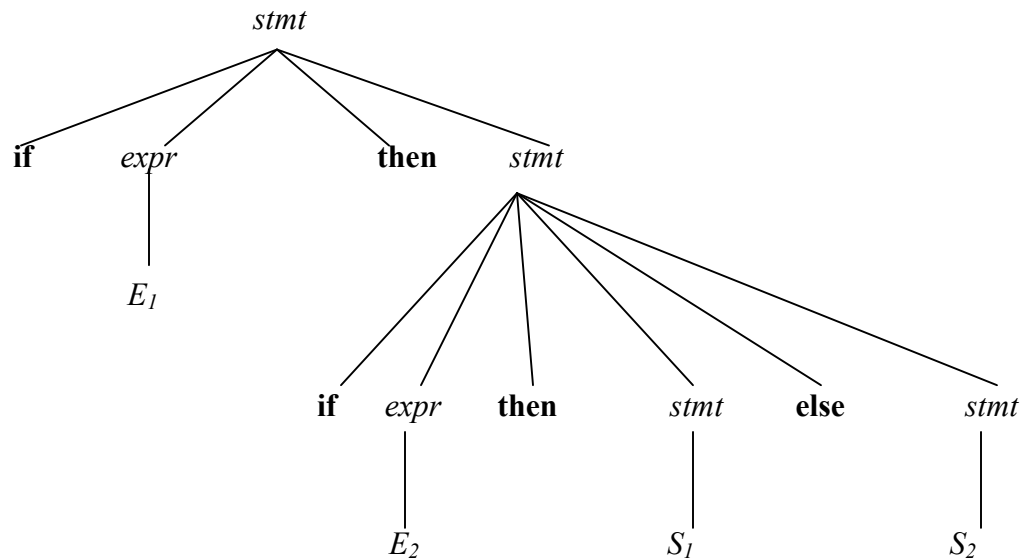
Eliminating ambiguity:

Ambiguity of the grammar that produces more than one parse tree for leftmost or rightmost derivation can be eliminated by re-writing the grammar.

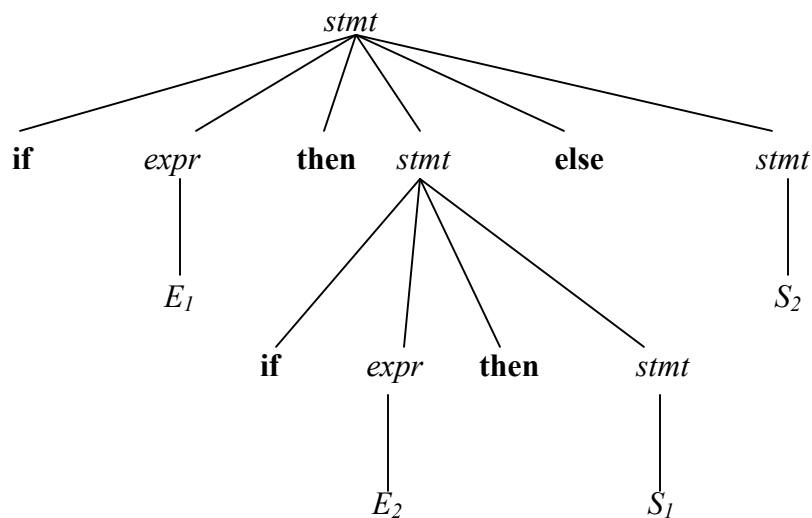
Consider this example, $G: stmt \rightarrow \text{if } expr \text{ then } stmt \mid \text{if } expr \text{ then } stmt \text{ else } stmt \mid \text{other}$

This grammar is ambiguous since the string **if E_1 then if E_2 then S_1 else S_2** has the following two parse trees for leftmost derivation :

1.



2.



To eliminate ambiguity, the following grammar may be used:

$$stmt \rightarrow matched_stmt \mid unmatched_stmt$$

$$matched_stmt \rightarrow \text{if } expr \text{ then } matched_stmt \text{ else } matched_stmt \mid \text{other}$$

$$unmatched_stmt \rightarrow \text{if } expr \text{ then } stmt \mid \text{if } expr \text{ then } matched_stmt \text{ else } unmatched_stmt$$

Eliminating Left Recursion:

A grammar is said to be *left recursive* if it has a non-terminal A such that there is a derivation $A \Rightarrow A\alpha$ for some string α . 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:

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

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

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

First eliminate the left recursion for E as

$$E \rightarrow TE'$$

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

Then eliminate for T as

$$T \rightarrow FT'$$

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

Thus the obtained grammar after eliminating left recursion is

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

Algorithm to eliminate left recursion:

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

Left factoring:

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. When it is not clear which of two alternative productions to use to expand a non-terminal A , we can rewrite the A -productions to defer the decision until we have seen enough of the input to make the right choice.

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

$$A \rightarrow \alpha A'$$

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

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

Left factored, this grammar becomes

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

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

$$E \rightarrow b$$

PARSING

It is the process of analyzing a continuous stream of input in order to determine its grammatical structure with respect to a given formal grammar.

Parse tree:

Graphical representation of a derivation or deduction is called a parse tree. Each interior node of the parse tree is a non-terminal; the children of the node can be terminals or non-terminals.

Types of parsing:

1. Top down parsing
 2. Bottom up parsing
-
- Top-down parsing : A parser can start with the start symbol and try to transform it to the input string.
Example : LL Parsers.
 - Bottom-up parsing : A parser can start with input and attempt to rewrite it into the start symbol.
Example : LR Parsers.

TOP-DOWN PARSING

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

Types of top-down parsing :

1. Recursive descent parsing
2. Predictive parsing

1. RECURSIVE DESCENT PARSING

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

Example for backtracking :

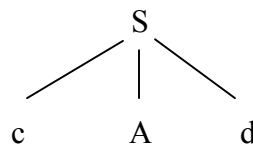
Consider the grammar $G : S \rightarrow cAd$
 $A \rightarrow ab \mid a$

and the input string $w=cad$.

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

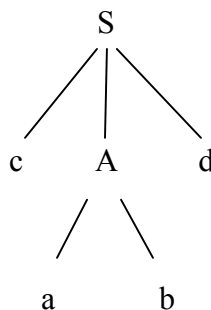
Step1:

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



Step2:

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



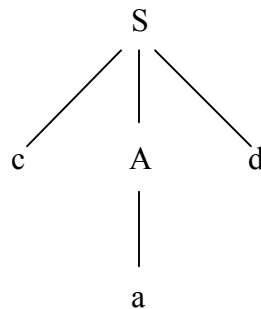
Step3:

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

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

Step4:

Now try the second alternative for A.



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

Example for recursive decent parsing:

A left-recursive grammar can cause a recursive-descent parser to go into an infinite loop. Hence, **elimination of left-recursion** must be done before parsing.

Consider the grammar for arithmetic expressions

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

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

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

After eliminating the left-recursion the grammar becomes,

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

Now we can write the procedure for grammar as follows:

Recursive procedure:

Procedure E()

begin

 T();

 EPRIME();

end

Procedure EPRIME()

begin

 If input_symbol='+' then

 ADVANCE();

 T();

 EPRIME();

end

Procedure T()

begin

 F();

 TPRIME();

end

Procedure TPRIME()

begin

 If input_symbol='*' then

 ADVANCE();

 F();

 TPRIME();

end

Procedure F()

begin

 If input-symbol='id' then

 ADVANCE();

 else if input-symbol='(' then

 ADVANCE();

 E();

 else if input-symbol=')' then

 ADVANCE();

end

else ERROR();

Stack implementation:

To recognize input **id+id*id** :

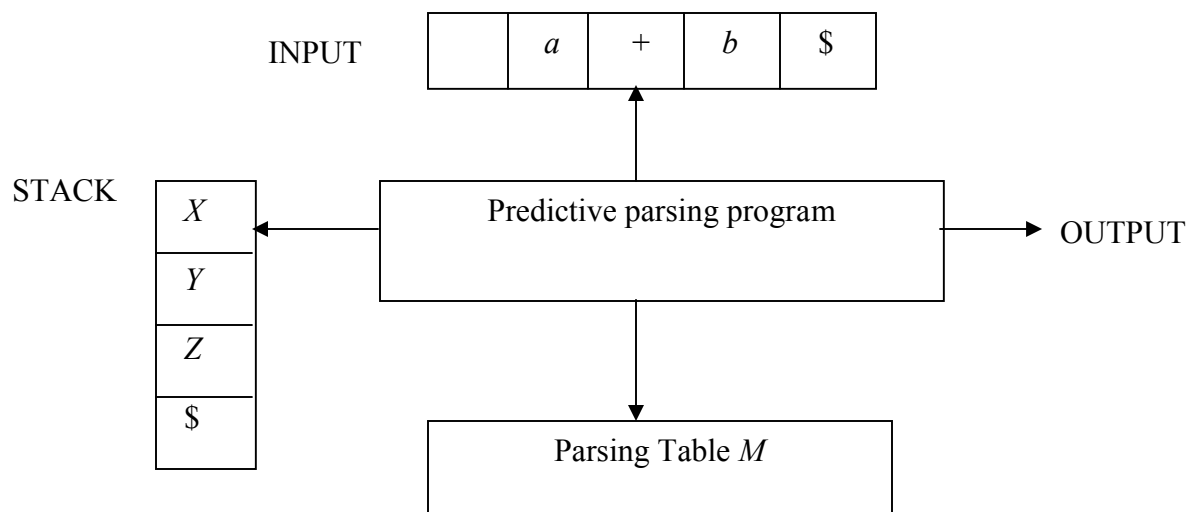
PROCEDURE	INPUT STRING
E()	<u>id</u> +id*id
T()	<u>id</u> +id*id
F()	<u>id</u> +id*id
ADVANCE()	id <u>+</u> id*id

TPRIME()	id+id*id
EPRIME()	id+id*id
ADVANCE()	id+ <u>id</u> *id
T()	id+ <u>id</u> *id
F()	id+ <u>id</u> *id
ADVANCE()	id+id* <u>id</u>
TPRIME()	id+id* <u>id</u>
ADVANCE()	id+id* <u>id</u>
F()	id+id* <u>id</u>
ADVANCE()	id+id* <u>id</u>
TPRIME()	id+id* <u>id</u>

2. PREDICTIVE PARSING

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

Non-recursive predictive parser



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

Input buffer:

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

Stack:

It contains a sequence of grammar symbols preceded by \$ to indicate the bottom of the stack. Initially, the stack contains the start symbol on top of \$.

Parsing table:

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

Predictive parsing program:

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

1. If $X = a = \$$, the parser halts and announces successful completion of parsing.
2. If $X = a \neq \$$, the parser pops X off the stack and advances the input pointer to the next input symbol.
3. If X is a non-terminal, the program consults entry $M[X, a]$ of the parsing table M . This entry will either be an X -production of the grammar or an error entry.
If $M[X, a] = \{X \rightarrow UVW\}$, the parser replaces X on top of the stack by WVU .
If $M[X, a] = \text{error}$, the parser calls an error recovery routine.

Algorithm for nonrecursive predictive parsing:

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

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

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

```
set  $ip$  to point to the first symbol of  $w\$$ ;  
repeat  
    let  $X$  be the top stack symbol and  $a$  the symbol pointed to by  $ip$ ;  
    if  $X$  is a terminal or $ then  
        if  $X = a$  then  
            pop  $X$  from the stack and advance  $ip$   
        else  $error()$   
    else /*  $X$  is a non-terminal */  
        if  $M[X, a] = X \rightarrow Y_1 Y_2 \dots Y_k$  then begin
```

```

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

```

Predictive parsing table construction:

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

1. FIRST
2. FOLLOW

Rules for first():

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

Rules for follow():

1. If S is a start symbol, then $\text{FOLLOW}(S)$ contains $\$$.
2. If there is a production $A \rightarrow \alpha B \beta$, then everything in $\text{FIRST}(\beta)$ except ϵ 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 ϵ , 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 ϵ is in $\text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A, b]$ for each terminal b in $\text{FOLLOW}(A)$. If ϵ is in $\text{FIRST}(\alpha)$ and $\$$ is in $\text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A, \$]$.
4. Make each undefined entry of M be **error**.

Example:

Consider the following grammar :

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

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

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

After eliminating left-recursion the grammar is

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

First() :

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

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

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

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

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

Follow() :

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

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

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

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

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

Predictive parsing table :

NON- TERMINAL	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

Stack implementation:

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

LL(1) grammar:

The parsing table entries are single entries. So each location has not more than one entry. This type of grammar is called LL(1) grammar.

Consider this following grammar:

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

$$E \rightarrow b$$

After eliminating left factoring, we have

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

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

$$E \rightarrow b$$

To construct a parsing table, we need FIRST() and FOLLOW() for all the non-terminals.

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

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

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

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

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

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

Parsing table:

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

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

Actions performed in predictive parsing:

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

Implementation of predictive parser:

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

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 **abbcd e**.

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 $id_1 + id_2 * id_3$

The rightmost derivation is :

$E \rightarrow \underline{E+E}$
 $\rightarrow E + \underline{E * E}$
 $\rightarrow E + E * \underline{id_3}$
 $\rightarrow E + \underline{id_2} * id_3$
 $\rightarrow \underline{id_1} + id_2 * id_3$

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 γ_n is the n^{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:

- 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.

Conflicts in shift-reduce parsing:

There are two conflicts that occur in shift 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:

- α is a viable prefix of the grammar if there is w such that αw 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 ϵ or has two adjacent non-terminals.

Example:

Consider the grammar:

$$E \rightarrow EAE \mid (E) \mid -E \mid \text{id}$$

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

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

$$E \rightarrow E+E \mid E-E \mid E * E \mid E/E \mid E \uparrow E \mid -E \mid \text{id}$$

Operator precedence relations:

There are three disjoint precedence relations namely

$< \cdot$ - less than

$=$ - equal to

$\cdot >$ - greater than

The relations give the following meaning:

$a < \cdot b$ - a yields precedence to b

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

$a \cdot > b$ - a takes precedence over b

Rules for binary operations:

1. If operator θ_1 has higher precedence than operator θ_2 , then make

$$\theta_1 \cdot > \theta_2 \text{ and } \theta_2 < \cdot \theta_1$$

2. If operators θ_1 and θ_2 are of equal precedence, then make

$$\theta_1 \cdot > \theta_2 \text{ and } \theta_2 \cdot > \theta_1 \text{ if operators are left associative}$$

$$\theta_1 < \cdot \theta_2 \text{ and } \theta_2 < \cdot \theta_1 \text{ if right associative}$$

3. Make the following for all operators θ :

$$\theta < \cdot \text{id}, \text{id} \cdot > \theta$$

$$\theta < \cdot (, (< \cdot \theta$$

$$\cdot > \theta, \theta \cdot >)$$

$$\theta \cdot > \$, \$ < \cdot \theta$$

Also make

$(=), (< \cdot (,) \cdot >), (< \cdot id, id \cdot >), \$ < \cdot id, id \cdot > \$, \$ < \cdot (,) \cdot > \$$

Example:

Operator-precedence relations for the grammar

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

TABLE : Operator-precedence relations

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

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  $a \prec b$  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
\$	$<\cdot$ id+id*id \$	shift id
\$ id	$\cdot >$ +id*id \$	pop the top of the stack id
\$	$<\cdot$ +id*id \$	shift +
\$ +	$<\cdot$ id*id \$	shift id
\$ +id	$\cdot >$ *id \$	pop id
\$ +	$<\cdot$ *id \$	shift *
\$ + *	$<\cdot$ id \$	shift id
\$ + * id	$\cdot >$ \$	pop id
\$ + *	$\cdot >$ \$	pop *
\$ +	$\cdot >$ \$	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:

- ✓ 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 a syntactic error as soon as possible.

Drawbacks of LR method:

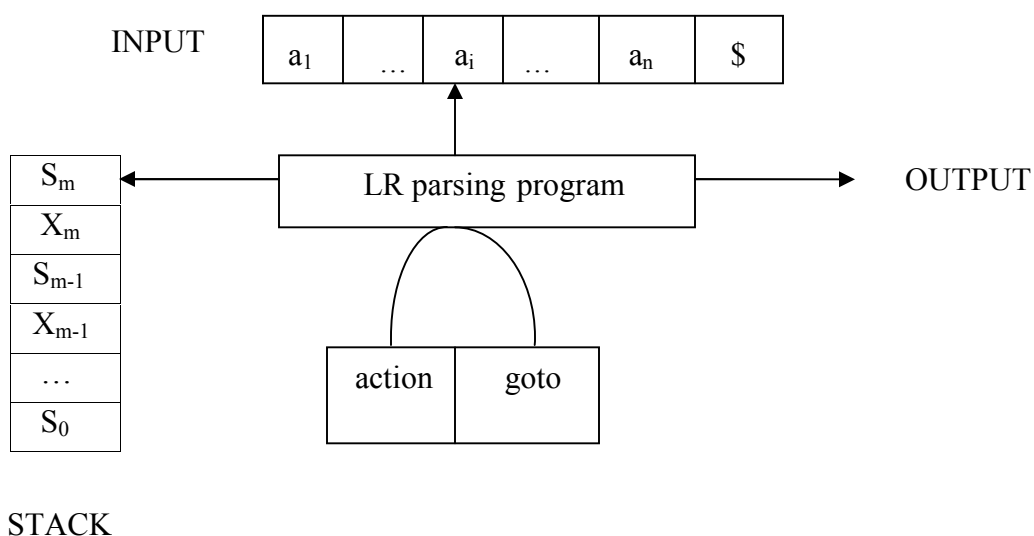
It is too much of work to construct a LR parser by hand for a programming language grammar. A specialized tool, called a LR parser generator, is needed. Example: YACC.

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 $action[s_m, a_i]$ 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  $A \rightarrow \beta$   
  end  
  else if  $action[s, a] = \text{accept}$  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 $A \rightarrow XYZ$ yields the four items :

$A \rightarrow \cdot XYZ$

$A \rightarrow X \cdot YZ$

$A \rightarrow XY \cdot Z$

$A \rightarrow XYZ \cdot$

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:

1. Initially, every item in I is added to $closure(I)$.
2. If $A \rightarrow \alpha \cdot B\beta$ is in $closure(I)$ and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow \cdot \gamma$ 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 $[A \rightarrow \alpha X \cdot \beta]$ such that $[A \rightarrow \alpha \cdot X\beta]$ 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.

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 $goto(I_i, a) = I_j$, then set $action[i, a]$ to “shift j”. Here a must be terminal.
 - (b) If $[A \rightarrow \alpha \cdot]$ is in I_i , then set $action[i, a]$ to “reduce $A \rightarrow \alpha$ ” for all a in FOLLOW(A).
 - (c) If $[S' \rightarrow S \cdot]$ is in I_i , then set $action[i, \$]$ 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 $goto(I_i, A) = I_j$, then $goto[i, A] = j$.
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 $[S' \rightarrow .S]$.

Example for SLR parsing:

Construct SLR parsing for the following grammar :

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

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

The given grammar is :

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

$E \rightarrow T$ ----- (2)

$T \rightarrow T * F$ ----- (3)

$T \rightarrow F$ ----- (4)

$F \rightarrow (E)$ ----- (5)

$F \rightarrow id$ ----- (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_4 , id)

$I_5 : F \rightarrow id .$

GOTO (I₀, T)
I₂ : E → T .
T → T . * F

GOTO (I₀, F)
I₃ : T → F .

GOTO (I₀, ()
I₄ : F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

GOTO (I₀, id)
I₅ : F → id .

GOTO (I₁, +)
I₆ : E → E + . T
T → . T * F
T → . F
F → . (E)
F → . id

GOTO (I₂, *)
I₇ : T → T * . F
F → . (E)
F → . id

GOTO (I₄, E)
I₈ : F → (E .)
E → E . + T

GOTO (I₄, T)
I₂ : E → T .
T → T . * F

GOTO (I₄, F)
I₃ : T → F .

GOTO (I₆, T)
I₉ : E → E + T .
T → T . * F

GOTO (I₆, F)
I₃ : T → F .

GOTO (I₆, ()
I₄ : F → (. E)

GOTO (I₆, id)
I₅ : F → id .

GOTO (I₇, F)
I₁₀ : T → T * F .

GOTO (I₇, ()
I₄ : F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

GOTO (I₇, id)
I₅ : F → id .

GOTO (I₈,))
I₁₁ : F → (E) .

GOTO (I₈, +)
I₆ : E → E + . T
T → . T * F
T → . F
F → . (E)
F → . id

GOTO (I₉, *)
I₇ : T → T * . F
F → . (E)
F → . id

GOTO (I₄, ()

I₄ : F → (. E)

E → . E + T

E → . T

T → . T * F

T → . F

F → . (E)

F → id

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

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

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

SLR parsing table:

	ACTION						GOTO		
	id	+	*	()	\$	E	T	F
I ₀	s5			s4			1	2	3
I ₁		s6				ACC			
I ₂		r2	s7		r2	r2			
I ₃		r4	r4		r4	r4			
I ₄	s5			s4			8	2	3
I ₅		r6	r6		r6	r6			
I ₆	s5			s4				9	3
I ₇	s5			s4					10
I ₈		s6			s11				
I ₉		r1	s7		r1	r1			
I ₁₀		r3	r3		r3	r3			
I ₁₁		r5	r5		r5	r5			

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

TYPE CHECKING

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

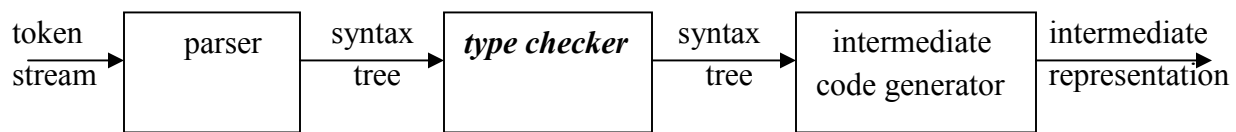
This checking, called *static checking*, detects and reports programming errors.

Some examples of static checks:

1. **Type checks** – A compiler should report an error if an operator is applied to an incompatible operand. Example: If an array variable and function variable are added together.

2. **Flow-of-control checks** – Statements that cause flow of control to leave a construct must have some place to which to transfer the flow of control. Example: An error occurs when an enclosing statement, such as `break`, does not exist in switch statement.

Position of type checker



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

TYPE SYSTEMS

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

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

Type Expressions

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

The following are the definitions of type expressions:

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

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

2. Since type expressions may be named, a type name is a type expression.
3. A type constructor applied to type expressions is a type expression.

Constructors include:

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

Products : If T_1 and T_2 are type expressions, then their Cartesian product $T_1 \times T_2$ is a type expression.

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

For example:

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

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

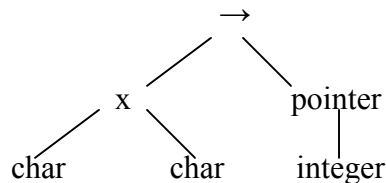
Pointers : If *T* is a type expression, then *pointer(T)* is a type expression denoting the type “pointer to an object of type *T*”.

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

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

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

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



Type systems

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

Static and Dynamic Checking of Types

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

Sound type system

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

Strongly typed language

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

Error Recovery

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

SPECIFICATION OF A SIMPLE TYPE CHECKER

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

A Simple Language

Consider the following grammar:

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

Translation scheme:

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

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.

- $$\begin{array}{ll} 1. E \rightarrow \mathbf{literal} & \{ E.type := char \} \\ E \rightarrow \mathbf{num} & \{ E.type := integer \} \end{array}$$

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

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

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

- $$3. E \rightarrow E_1 \text{ mod } E_2 \quad \{ E.type := \text{if } E_1.type = \text{integer and} \\ E_2.type = \text{integer then integer} \\ \text{else type error} \}$$

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

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

- $$5. E \rightarrow E_1 \uparrow \quad \{ E.type := \text{if } E_1.type = \text{pointer}(t) \text{ then } t \text{ else type error} \}$$

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:

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

2. Conditional statement:

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

3. While statement:

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

4. Sequence of statements:

$$S \rightarrow S_1 ; S_2 \quad \{ S.type := \text{if } S_1.type = \text{void} \text{ and } S_1.type = \text{void} \text{ then void else type_error } \}$$

Type checking of functions

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

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

SOURCE LANGUAGE ISSUES

Procedures:

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

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

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

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

Activation trees:

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

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

Control stack:

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

The Scope of a Declaration:

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

Declarations may be explicit, such as:

```
var i : integer ;
```

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

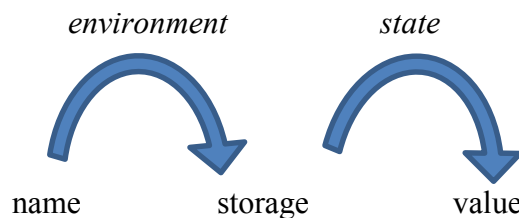
The portion of the program to which a declaration applies is called the *scope* of that declaration.

Binding of names:

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

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

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

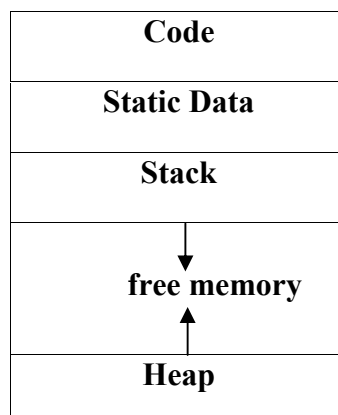


When an *environment* associates storage location s with a name x , we say that x is *bound* to s . This association is referred to as a *binding* of x .

STORAGE ORGANISATION

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

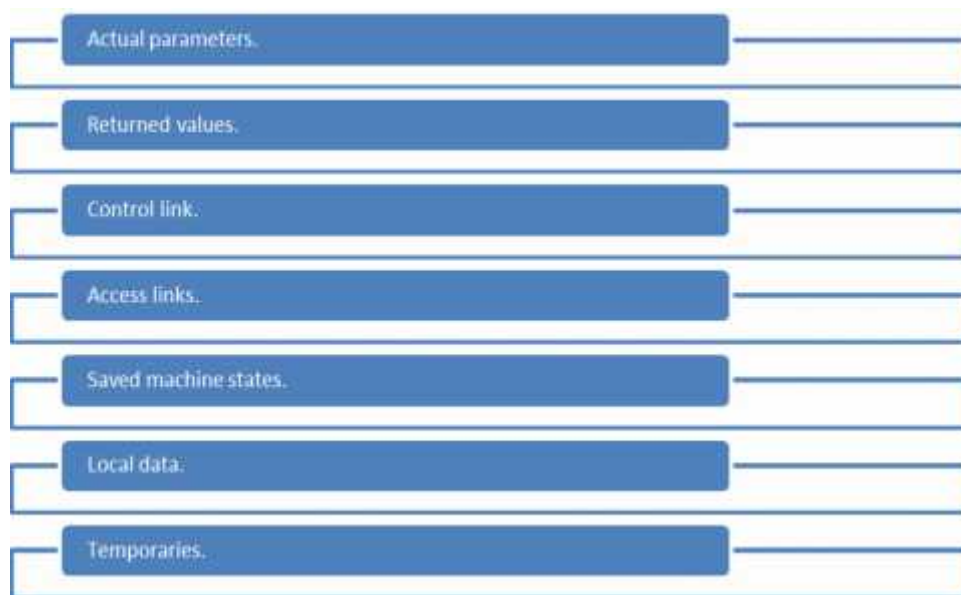
Typical subdivision of run-time memory:



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

Activation records:

- Procedure calls and returns are usually managed by a run time stack called the *control stack*.
- Each live activation has an activation record on the control stack, with the root of the activation tree at the bottom, the latter activation has its record at the top of the stack.
- The contents of the activation record vary with the language being implemented. The diagram below shows the contents of activation record.



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

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

STORAGE ALLOCATION STRATEGIES

The different storage allocation strategies are :

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

STATIC ALLOCATION

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

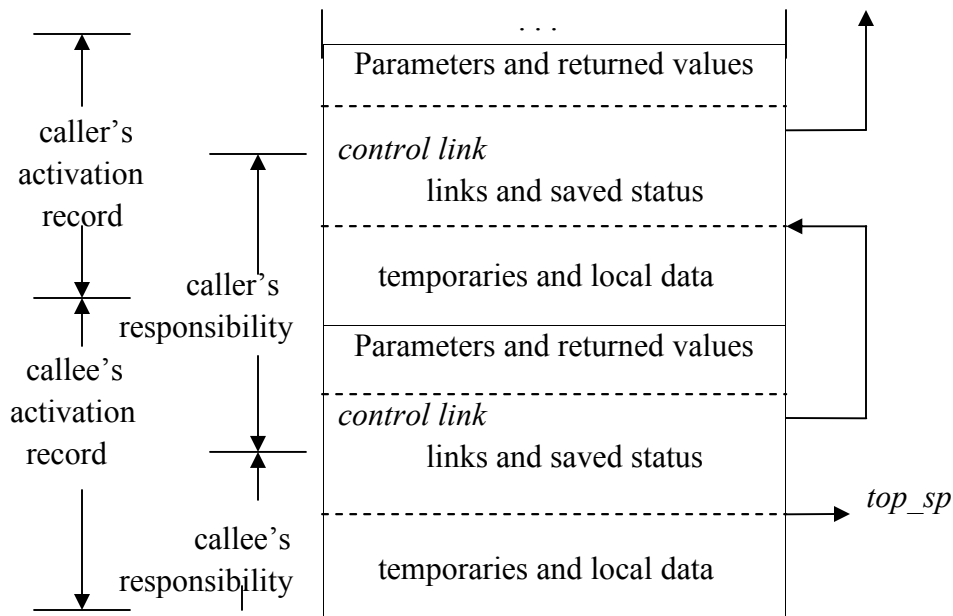
STACK ALLOCATION OF SPACE

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

Calling sequences:

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

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

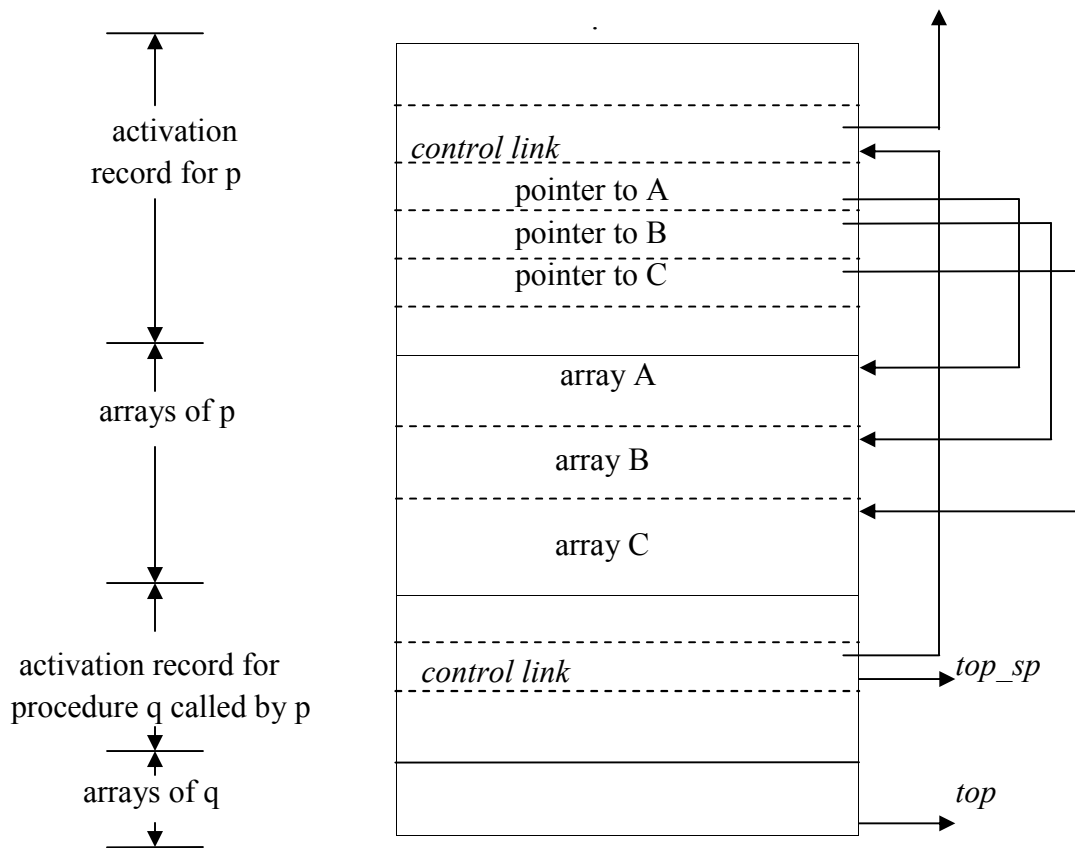


Division of tasks between caller and callee

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

Variable length data on stack:

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



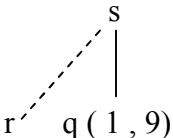
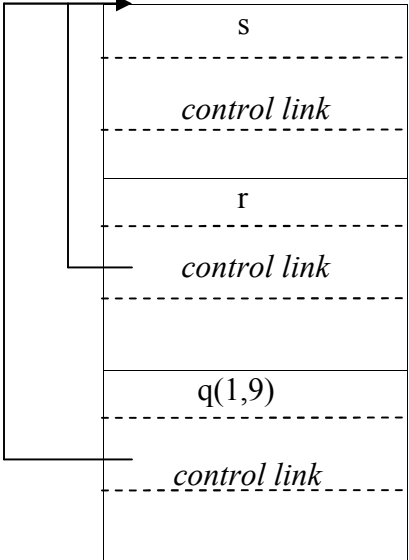
Access to dynamically allocated arrays

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

HEAP ALLOCATION

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

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

Position in the activation tree	Activation records in the heap	Remarks
		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.