Unit-1

Introduction to Compiling:
1.1 INTRODUCTION OF LANGUAGE PROCESSING SYSTEM

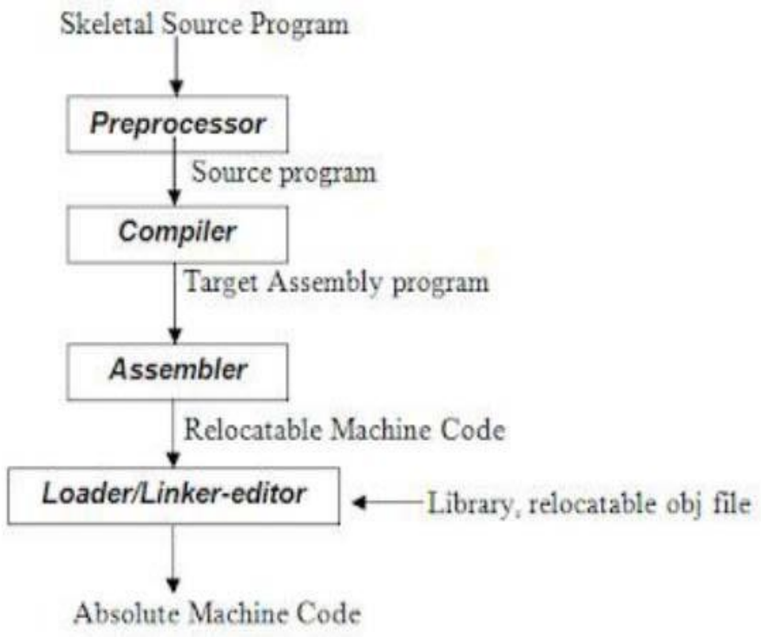


Fig 1.1: Language Processing System

Preprocessor

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

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

COMPILER

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

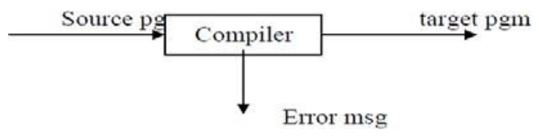


Fig 1.2: Structure of Compiler

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

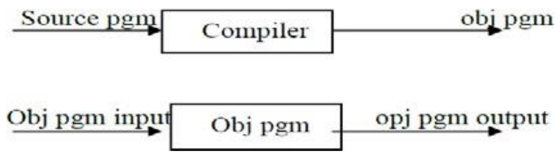


Fig 1.3: Execution process of source program in Compiler

ASSEMBLER

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

INTERPRETER

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

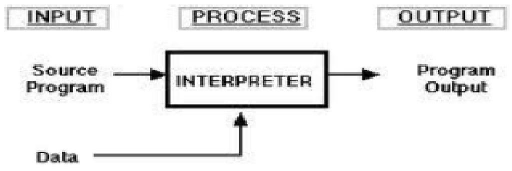


Fig1.4: Execution in Interpreter

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

Advantages:
Modification of user program can be easily made and implemented as execution proceeds.
Type of object that denotes a various may change dynamically.
Debugging a program and finding errors is simplified task for a program used for interpretation.
The interpreter for the language makes it machine independent.
Disadvantages:
The execution of the program is *slower*.
Memory consumption is more.

LOADER AND LINK-EDITOR:

Once the assembler 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 o they may be placed in arbitrary core locations is called relocation. Relocation loaders perform four functions.

1.2 TRANSLATOR

A translator is a program that takes as input a program written in one language and produces as output a program in another language. 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.

1.3 LIST OF COMPILERS

1. Ada compilers
2 .ALGOL compilers
3 .BASIC compilers
4 .C# compilers
5 .C compilers
6 .C++ compilers
7 .COBOL compilers
8 .Common Lisp compilers
9. ECMAScript interpreters
10. Fortran compilers
11 .Java compilers
12. Pascal compilers
13. PL/I compilers
14. Python compilers
15. Smalltalk compilers

1.4 STRUCTURE OF THE COMPILER DESIGN

Phases of a compiler: A compiler operates in phases. A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation. The phases of a compiler are shown in below
There are two phases of compilation.
a. Analysis (Machine Independent/Language Dependent)
b. Synthesis(Machine Dependent/Language independent)

Compilation process is partitioned into no-of-sub processes called **'phases'**.
Lexical Analysis:-
LA or Scanners reads the source program one character at a time, carving the source program into a sequence of automic units called **tokens.**

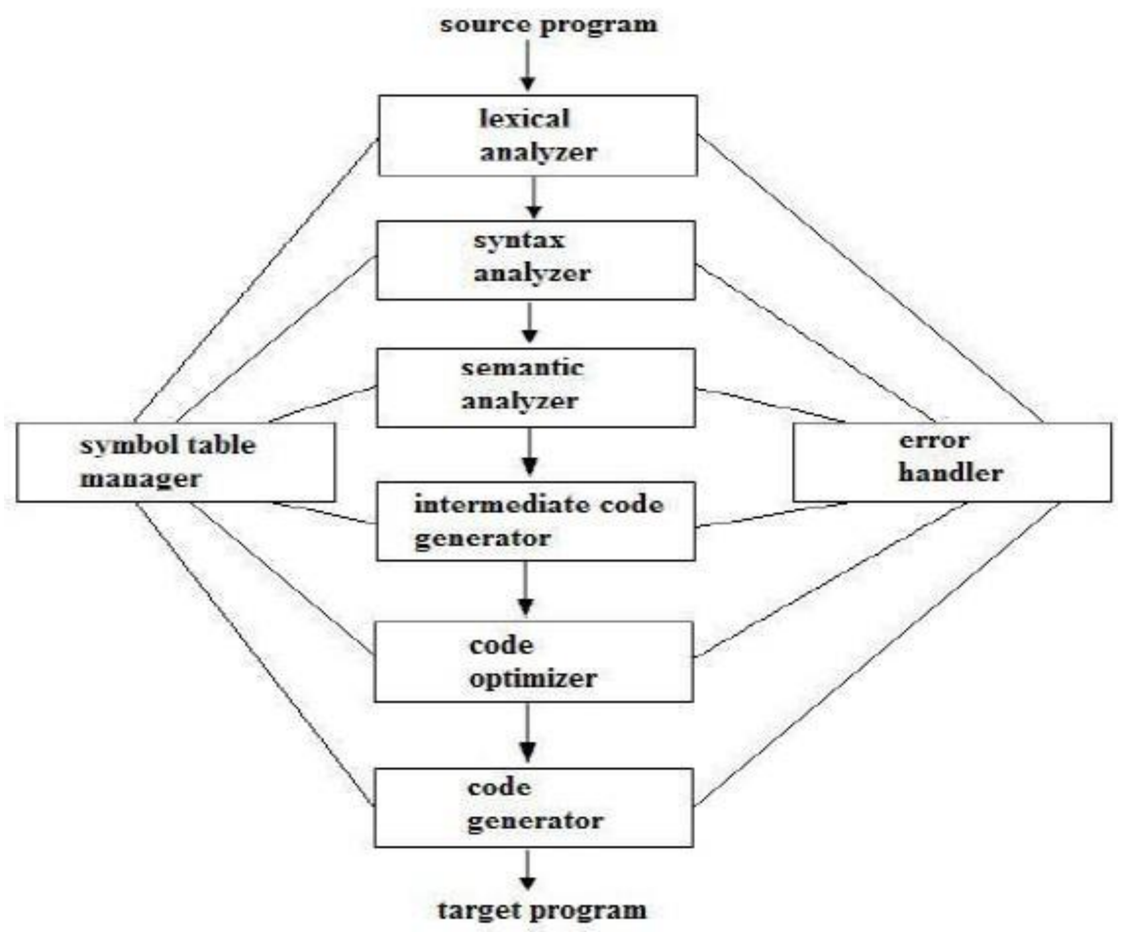


Fig 1.5: Phases of Compiler

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

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

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

Code Generation:-
The last phase of translation is code generation. A number of optimizations to **reduce the length of machine language program** are carried out during this phase. The output of the code generator is the machine language program of the specified computer.

Table Management (or) Book-keeping:- This is the portion to **keep the names** used by the 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 some job as the original, but in a way that saves time and / or spaces.

a. Local Optimization:-

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

If **A** > **B** goto **L2**

```
        Goto L3
        L2 :
```

This can be replaced by a single statement

```
    If A < B goto L3
```

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

```
                        A := B + C + D
                        E := B + C + F
```

Might be evaluated as

```
                        T1 := B + C
                        A := T1 + D
                        E := T1 + F
```

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

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

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

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

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

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

Example:

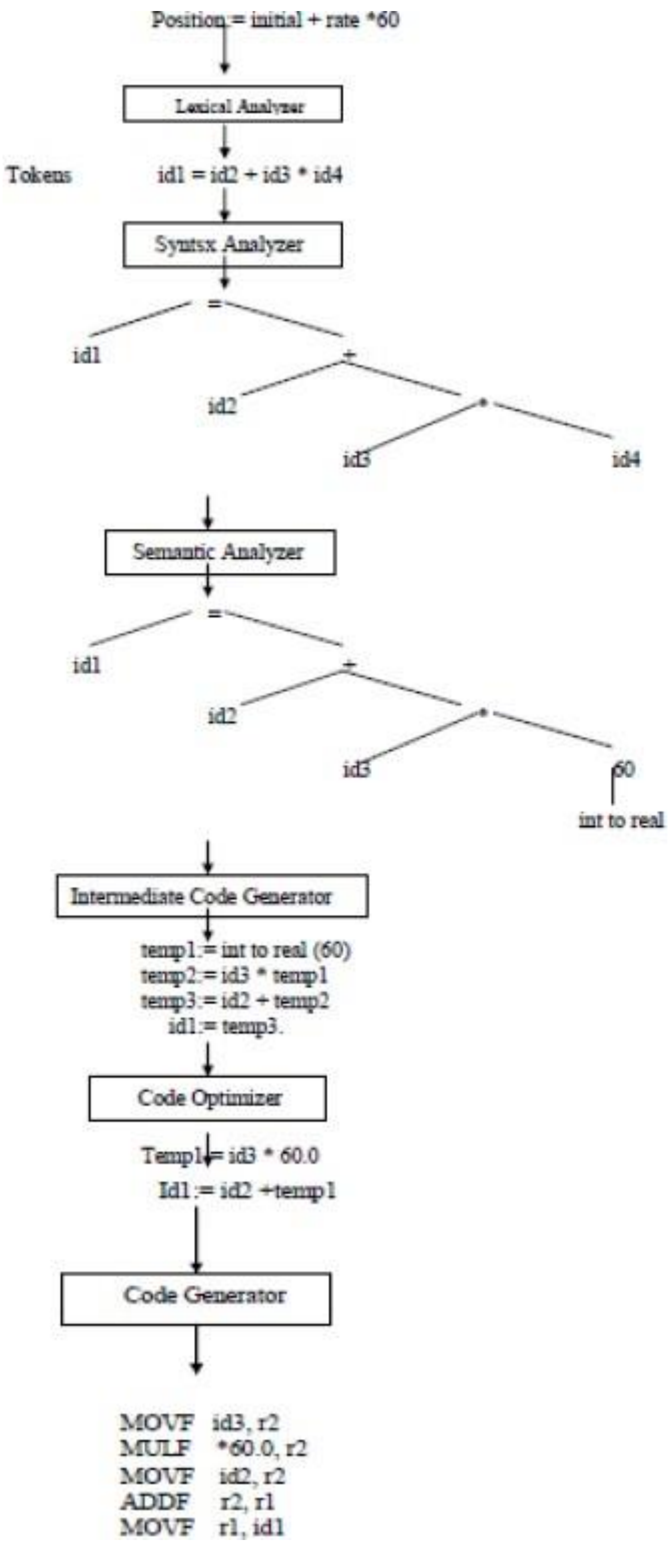


Fig 1.6: Compilation Process

2.0 LEXICAL ANALYSIS

- reads and converts the input into a stream of tokens to be analyzed by parser.
- lexeme : a sequence of characters which comprises a single token.
- Lexical Analyzer →Lexeme / Token → Parser

Removal of White Space and Comments

- Remove white space(blank, tab, new line etc.) and comments

Contsants

- Constants: For a while, consider only integers
- eg) for input 31 + 28, output(token representation)?
    input : 31 + 28
    output: <num, 31> <+, > <num, 28>
        num + :token
        31 28 : attribute, value(or lexeme) of integer token num

Recognizing

- Identifiers
    - Identifiers are names of variables, arrays, functions...
    - A grammar treats an identifier as a token.
    - eg) input : count = count + increment;
      output : <id,1> <=, > <id,1> <+, > <id, 2>;
      Symbol table

	tokens	attributes(lexeme)
0		
1	id	count
2	id	increment
3		

- Keywords are reserved, i.e., they cannot be used as identifiers

Then a character string forms an identifier only if it is no a keyword.

- punctuation symbols
    - operators : + - * / := < > …

Interface to lexical analyzer

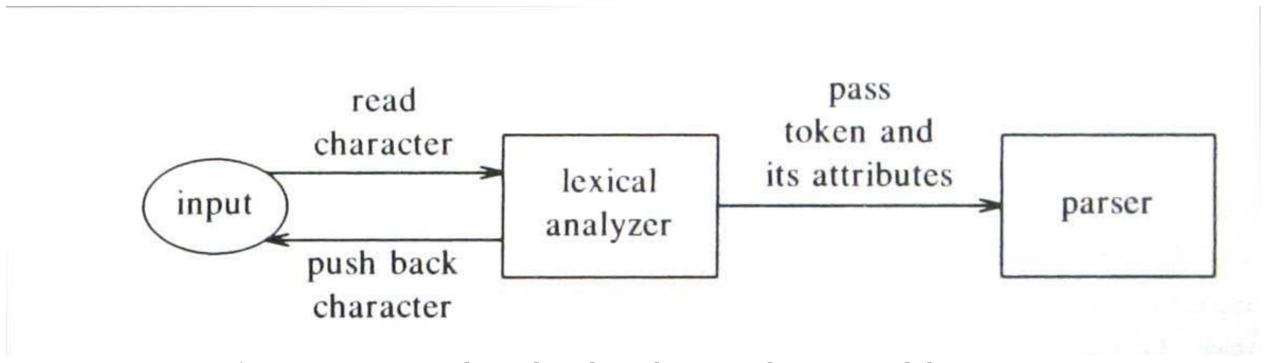
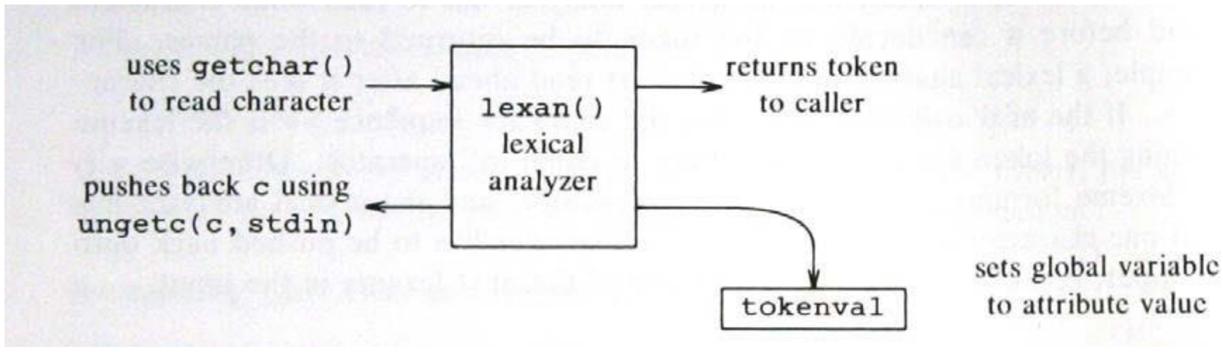


Fig 2.15. Inserting a lexical analyzer between the input and the parser

A Lexical Analyzer



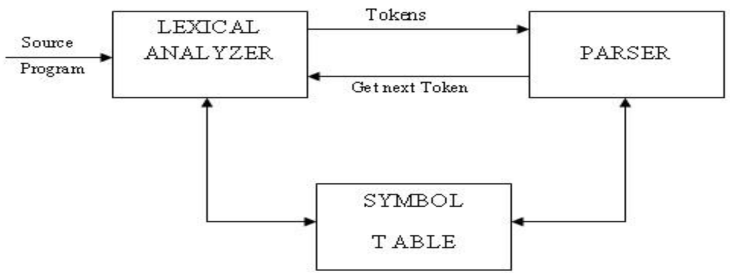
3. **Lexical Analysis:**

3.1 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.

3.2 ROLE OF LEXICAL ANALYZER

The LA is the first phase of a compiler. It main task is to read the input character and produce asoutput 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.

3.3 TOKEN, LEXEME, PATTERN:

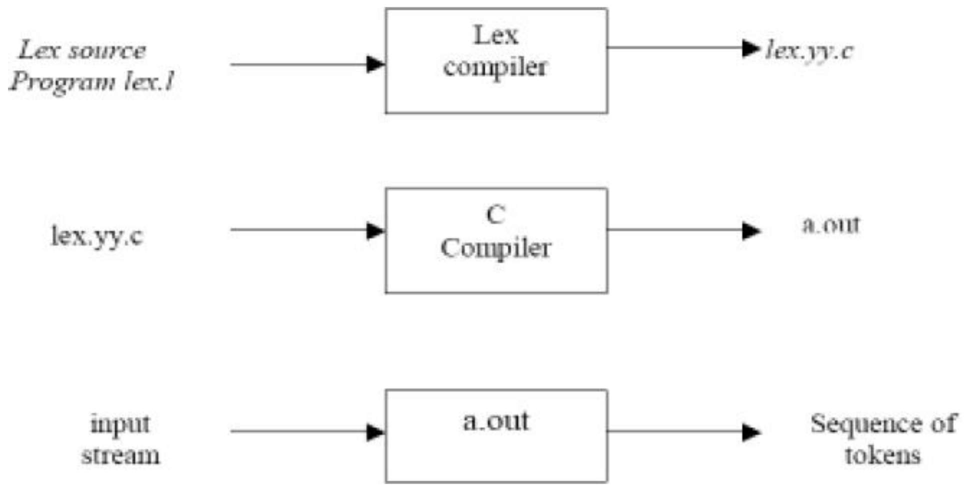
Token: Token is a sequence of characters that can be treated as a single logical entity.
Typical tokens are,
1) Identifiers 2) keywords 3) operators 4) special symbols 5)constants

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

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

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

3.4. Lexical Analyzer Generator



3.18. 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 {action 1}
   p2 {action 2}
   p3 {action 3}
   … …
   … …

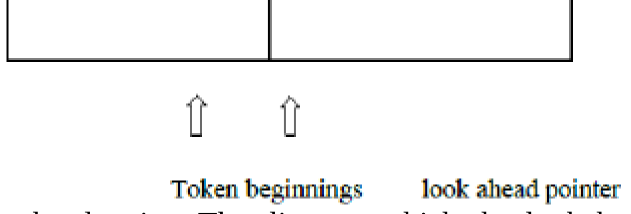
   Where, each *p* is a regular expression and each *action* is a program fragment describing what action the lexical analyzer should take when a pattern *p* matches a lexeme. In Lex the actions are written in C.
3. The third section holds whatever *auxiliary procedures* are needed by the *actions.*Alternatively 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.

3.19. 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 a t 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 thelexical analyzer to read its input from an input buffer. Figure shows a buffer divided into two haves 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 thecharacter next to be read. In practice each buffering scheme adopts one convention either apointer is at the symbol last read or the symbol it is ready to read.



Token beginnings look ahead pointerThe distance which the lookahead pointer may have to travel past the actual token may belarge. 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 look ahead 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 look ahead can be used before the next token is discovered. In the above example, ifthe look ahead 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 groupedinto 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-2

4.1 ROLE OF THE PARSER :

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

There are two ways of identifying an elementry sutree:

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

The two types of parsers employed are:

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

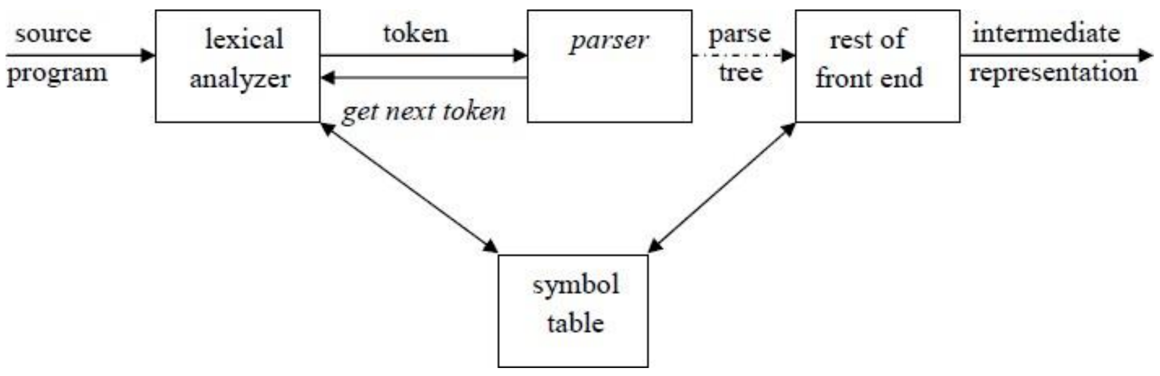


Fig . 4.1: position of parser in compiler model.

4.2 CONTEXT FREE GRAMMARS

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

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

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

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

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

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

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

A sentence of L(G) is a string of terminal symbols of G. If S is the start symbol of G then ω is a sentence of L(G) iff $S \Rightarrow \omega$ whereω is a string of terminals of G. If G is a context-free grammar, L(G) is a context-free language. Two grammar G_1 and G_2 are equivalent, if they produce same grammar.

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

4.2.1 Derivations

In general a derivation step is

$\alpha A\beta \Rightarrow \alpha\gamma\beta$ is sentential form and if there is a production rule $A\rightarrow\gamma$ in our grammar. where α and β are arbitrary strings of terminal and non-terminal symbols $\alpha 1 \Rightarrow \alpha 2 \Rightarrow ... \Rightarrow \alpha n$ (αn derives from $\alpha 1$ or $\alpha 1$ derives αn). There are two types of derivaion

1At each derivation step, we can choose any of the non-terminal in the sentential form of G for the replacement.

2If we always choose the left-most non-terminal in each derivation step, this derivation is called as left-most derivation.

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

Leftmost derivation :

$E \rightarrow E + E$

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

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

Rightmost derivation

$E \rightarrow E + E$

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

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

Sentence to be derived : – (id+id)

LEFTMOST DERIVATION	RIGHTMOST DERIVATION
E → - E	E → - E
E → - (E)	E → - (E)
E → - (E+E)	E → - (E+E)
E → - (id+E)	E → - (E+id)
E → - (id+id)	E → - (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:

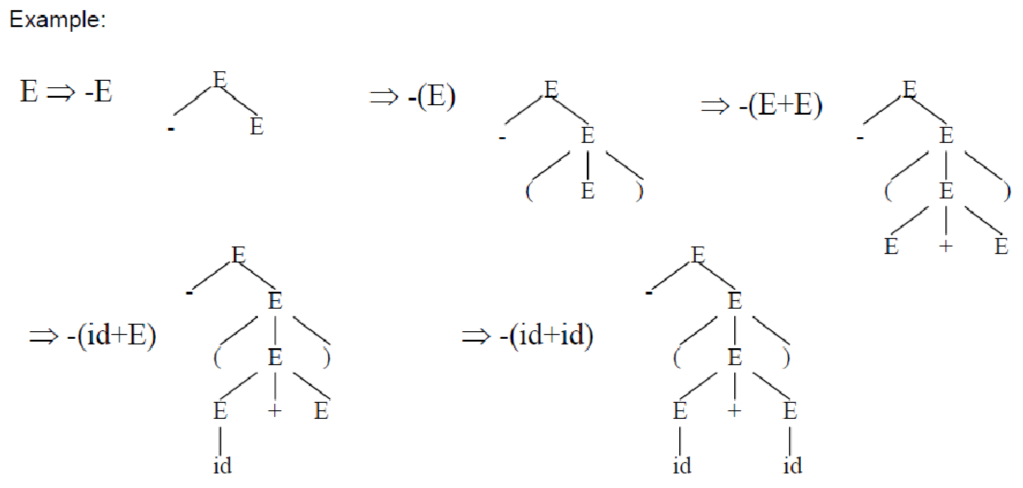
- Given a grammar G with start symbol S, if S → α , where α may contain non-terminals or terminals, then α is called the sentinel form of G.

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.

4.2.2 PARSE TREE

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



Ambiguity:

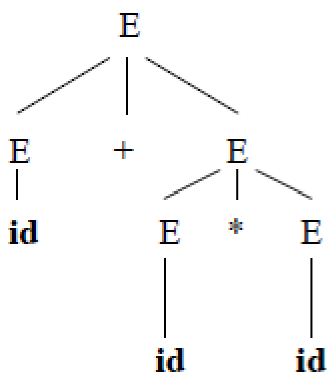
A grammar that produces more than one parse for some sentence is said to be **ambiguous grammar**.

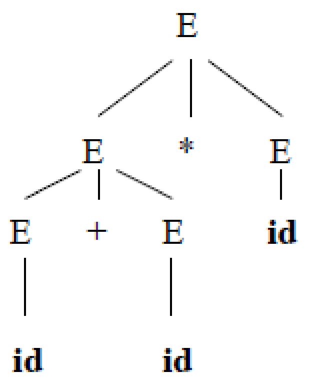
Example : Given grammar G : E → E+E | E*E | (E) | - E | id

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

E → E+ E	E → E* E
E → id + E	E → E + E * E
E → id + E * E	E → id + E * E
E → id + id * E	E → id + id * E
E → id + id * id	E → id + id * id

The two corresponding parse trees are :





Example:

To disambiguate the grammar E → E+E | E*E | E^E | id | (E), we can use precedence of operators as follows:

^ (right to left)

/,* (left to right)

-,+ (left to right)

We get the following unambiguous grammar:

E → E+T | T

T → T*F | F

F → G^F | G

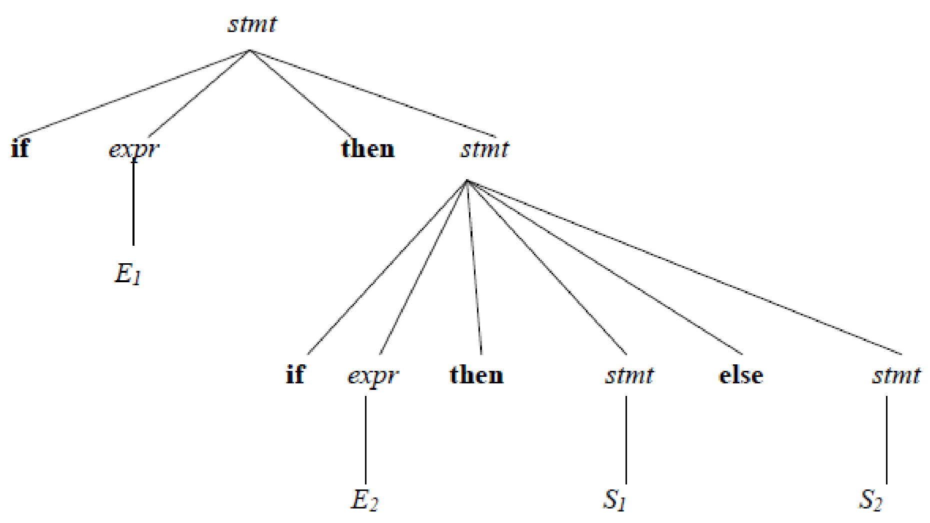
G → id | (E)

Consider this example, G: *stmt* → **if** *expr* **then** *stmt* |**if** *expr* **then** *stmt* **else***stmt* | **other**

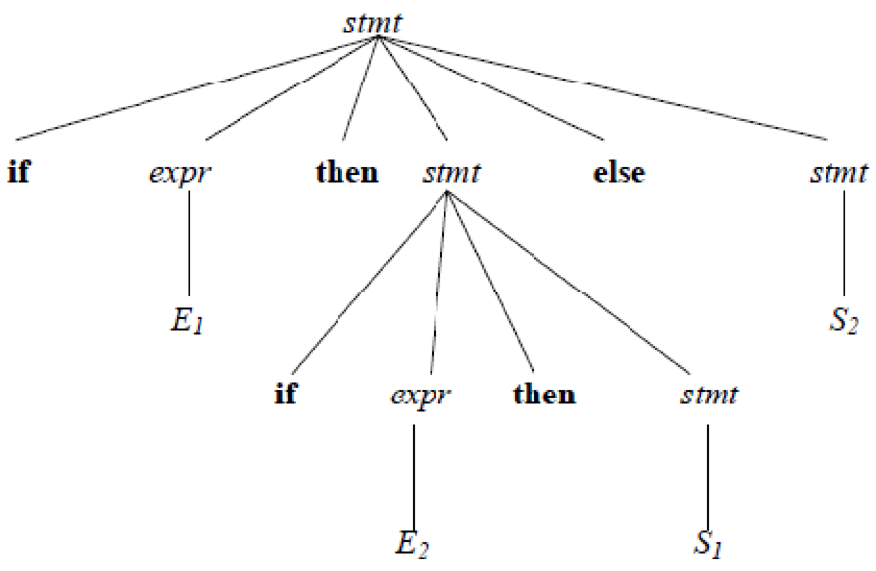
This grammar is ambiguous since the string **if E1 then if E2 then S1 else S2** has the following

Two parse trees for leftmost derivation :

1.



2.



To eliminate ambiguity, the following grammar may be used:

stmt → *matched_stmt* | *unmatched_stmt*

matched_stmt → **if** *expr* **then** *matched_stmt* **else** *matched_stmt* | **other**

unmatched_stmt → **if** *expr* **then** *stmt*| **if** *expr* **then** *matched_stmt* **else** *unmatched_stmt*

Eliminating Left Recursion:

A grammar is said to be *left recursive* if it has a non-terminal *A* such that there is a derivation A=>Aα 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 → Aα | β it can be replaced with a sequence of two productions

A → βA'

A' → αA' | ε

Without changing the set of strings derivable from A.

Example : Consider the following grammar for arithmetic expressions:

E → E+T | T

T → T*F | F

F → (E) | id

First eliminate the left recursion for E as

E → TE'

E' → +TE' |ε

Then eliminate for T as

T → FT'

T'→ *FT' | ε

Thus the obtained grammar after eliminating left recursion is

E → TE'

E' → +TE' |ε

T → FT'

T' → *FT' | ε

F → (E) | id

Algorithm to eliminate left recursion:

```
1. Arrange the non-terminals in some order A1, A2 . . . An.
2.for i := 1 to n do begin
    for j := 1 to i-1 do begin
        replace each production of the form Ai → Aj γ
        by the productions Ai → δ1 γ | δ2γ | . . . | δk γ
        where Aj→ δ1 |δ2 | . . . |δk are all the current Aj-productions;
    end
        eliminate the immediate left recursion among the Ai-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 → αβ1 | αβ2 , it can be rewritten as

A → αA'

A' → β1 | β2

Consider the grammar , G : S→iEtS | iEtSeS | a

E → b

Left factored, this grammar becomes

S → iEtSS' | a

S' → eS | ε

E → b

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→cAd

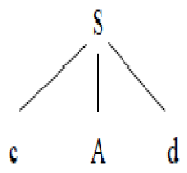
A → ab | a

and the input string w=cad.

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

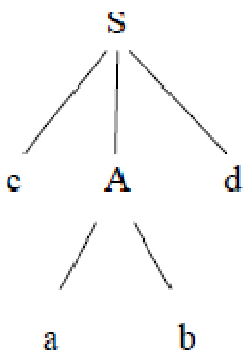
Step1:

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.



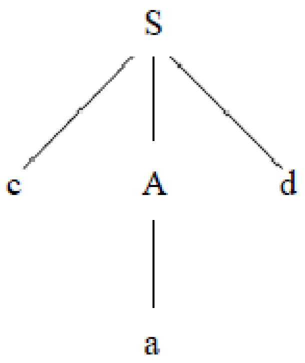
Step3:

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.

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.

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 FIRST(X) is {X}.
2. If $X \rightarrow \varepsilon$ is a production, then add ε 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 $X \rightarrow Y_1\ Y_2 \ldots Y_k$ is a production, then place a in FIRST(X) if for some i, a is in FIRST(Yi), and ε is in all of FIRST($Y1$),…,FIRST($Yi\text{-}1$); that is, $Y1,\ldots.Yi\text{-}1$ => ε. If ε is in FIRST(Y_j) for all j=1,2,..,k, then add ε to FIRST(X).

Rules for follow( ):

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

Example:

Consider the following grammar :

E → E+T | T

T→T*F | F

F → (E) | id

After eliminating left-recursion the grammar is

E → TE’

E’ → +TE’ |ε

T → FT’

T’ → *FT’ | ε

F → (E) | id

First( ) :

FIRST(E) = { (, id}

FIRST(E’) ={+ ,ε}

FIRST(T) = { (, id}

FIRST(T’) = {*, ε }

FIRST(F) = { (, id }

Follow( ):

FOLLOW(E) = { $,) }

FOLLOW(E’) = { $,) }

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

FOLLOW(T’) = { +, $,) }

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

Predictive parsing table :

NON-TERMINAL	id	+	*	(	)	$
E	E → TE’			E → TE’		
E’		E’ → +TE’			E’ → ε	E’→ ε
T	T → FT’			T → FT’		
T’		T’→ ε	T’→ *FT’		T’ → ε	T’ → ε
F	F → id			F → (E)		

Stack implementation:

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

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 → iEtS | iEtSeS | a
E → b
After eliminating left factoring, we have
S→iEtSS' | a
S'→eS |ε
E→b
To construct a parsing table, we need FIRST() and FOLLOW() for all the non-terminals.
FIRST(S) = { i, a }
FIRST(S') = {e,ε}
FIRST(E) = { b}
FOLLOW(S) = { $,e }
FOLLOW(S') = { $,e }
FOLLOW(E) = {t}

Parsing table:

NON-TERMINAL	a	b	e	i	t	$
S	S → a			S → iEtSS'		
S'			S' → eS S' → ε			S' → ε
E		E → 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 → aABe

A → Abc | b

B → d

The sentence to be recognized is **abbcde.**

REDUCTION (LEFTMOST)	**RIGHTMOST DERIVATION**
a**b**bcde (A → b)	**S**→ aA**B**e
a**Abc**de (A → Abc)	→ a**A**de
aA**d**e (B → d)	→ a**A**bcde
aABe(S → aABe)	→ abbcde
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 → E+E
E → E*E
E → (E)
E → id

And the input string $id_1+id_2*id_3$

The rightmost derivation is :

$E \rightarrow \underline{\mathbf{E}+\mathbf{E}}$
$\rightarrow E+\underline{\mathbf{E*E}}$
$\rightarrow E+E\mathbf{*}\underline{\mathbf{id_3}}$
$\rightarrow E+\underline{\mathbf{id_2}}\mathbf{*}id_3$
$\rightarrow \underline{\mathbf{id}_1}+id_2\mathbf{*}id_3$

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$w = y_n$, where y_n is then^{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→id
\$ E	+id_2*id_3 \$	shift
\$ E+	id_2*id_3 \$	shift
\$ E+id_2	*id_3 \$	reduce by E→id
\$ E+E	*id_3 \$	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

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→E+E | E*E | 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		

2. <u>Reduce-reduce conflict:</u>

Consider the grammar:

M → R+R | R+c | R
R → 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	\$				

Viable prefixes:

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

Example:

Consider the grammar:

E → EAE | (E) | -E | id
A → + | - | * | / | ↑

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

E → E+E | E-E | E*E | E/E | E↑E | -E | 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 θ_1 has higher precedence than operator $\theta_{2,}$ then make
$\theta_1 \cdot>\theta_2$ and $\theta_2 <\cdot \theta_1$

2. If operators θ_1 and $\theta_{2,}$ are of equal precedence, then make
$\theta_1 \cdot>\theta_2$ and $\theta_2 \cdot>\theta_1$ if operators are left associative
$\theta_1 <\cdot \theta_2$ and $\theta_2 <\cdot \theta_1$ if right associative

3. Make the following for all operators θ:
θ<· id , id·>θ
θ<·(, (<·θ
)·>θ , θ ·>)
θ·>$, $<· θ

Also make
(=) , (<· (,) ·>) , (<· id , id ·>) , $<· id , id ·>$, $<· (,) ·>$

Example:

Operator-precedence relations for the grammar

E → E+E | E-E | E*E | E/E | E↑E | (E) | -E | id is given in the following table assuming

1. ↑ 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

	+	-	*	/	↑	id	(	)	$
+	·>	·>	<·	<·	<·	<·	<·	·>	·>
-	·>	·>	<·	<·	<·	<·	<·	·>	·>
*	·>	·>	·>	·>	<·	<·	<·	·>	·>
/	·>	·>	·>	·>	<·	<·	<·	·>	·>
↑	·>	·>	·>	·>	<·	<·	<·	·>	·>
id	·>	·>	·>	·>	·>			·>	·>
(	<·	<·	<·	<·	<·	<·	<·	=	
)	·>	·>	·>	·>	·>			·>	·>
$	<·	<·	<·	<·	<·	<·	<·		

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 → E+E | E-E | E*E | E/E | E↑E | (E) | 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:

- ✓ 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:

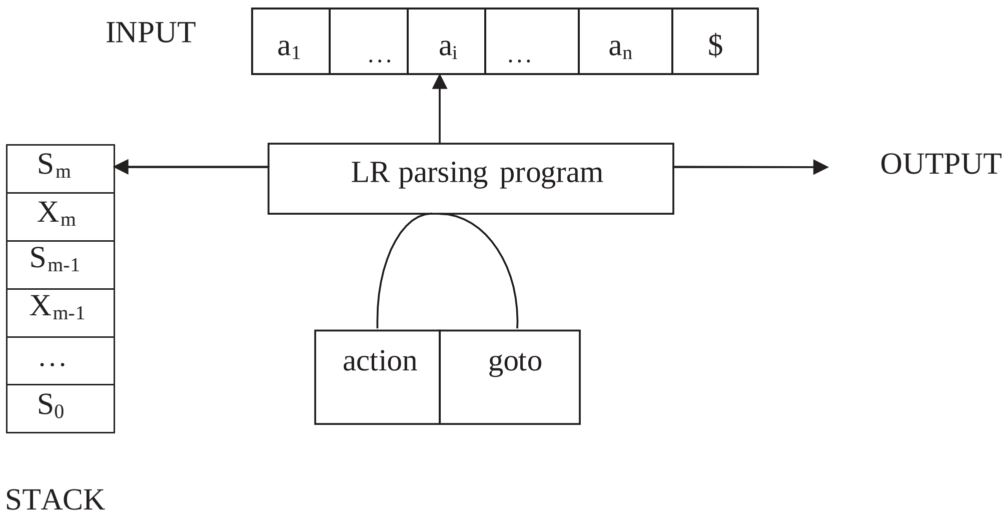
It is too much of work to construct a LR parser by hand for a programming language grammar. A specialized tool, called a LR parser generator, is needed. Example: YACC.

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 $s_0X_1s_1X_2s_2\ldots X_ms_m$, 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 → β,
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.

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(O) items:

An*LR(O) 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 :

A →**.**XYZ
A → X**.**YZ
A → XY**.**Z
A → 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:

1. Initially, every item in I is added to closure(I).
2. If A → a . Bβ is in closure(I) and B → y is a production, then add the item B → . y 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→ aX . β] such that [A→ a . Xβ] 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 = {$I_0, I_1, \ldots I_n$}, 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→a·*a*β] 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→a·] is in I_i , then set*action*[*i*,*a*] to “reduce A→a” for all*a*in FOLLOW(A).
   (c) If [S’→S.] 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. he*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’→.S].

Example for SLR parsing:

Construct SLR parsing for the following grammar :
G : E → E + T | T
T → T * F | F
F → (E) | id

The given grammar is :

G : E → E + T ------ (1)
E →T ------ (2)
T → T * F ------ (3)
T → F ------ (4)
F → (E) ------ (5)
F → id ------ (6)

Step 1 : Convert given grammar into augmented grammar.

Augmented grammar :

E’ → E

E → E + T

E → T

T → T * F

T → F

F → (E)

F → id

Step 2 : Find LR (0) items.

I_0 : E’ →**.**E

E →**.**E + T

E →**.**T

T →**.**T * F

T →**.**F

F →**.**(E)

F →**.**id

GOTO (I_0 , E)

I_1 : E’ → E**.**

E → E**.**+ T

GOTO (I_4 , id)

I_5 : F → id**.**

GOTO (I_0 , T)

$I_2 : E \rightarrow T.$

$\quad T \rightarrow T.* F$

GOTO (I_0 , F)

$I_3 : T \rightarrow F.$

GOTO (I_0 , ( )

$I_4 : F \rightarrow (.E)$

$\quad E \rightarrow .E + T$

$\quad E \rightarrow .T$

$\quad T \rightarrow .T * F$

$\quad T \rightarrow .F$

$\quad F \rightarrow .(E)$

$\quad F \rightarrow .id$

GOTO (I_0 , id)

$I_5 : F \rightarrow id.$

GOTO (I_1 , +)

$I_6 : E \rightarrow E +.T$

$\quad T \rightarrow .T * F$

$\quad T \rightarrow .F$

$\quad F \rightarrow .(E)$

$\quad F \rightarrow .id$

GOTO (I_2 , *)

$I_7 : T \rightarrow T *.F$

$\quad F \rightarrow .(E)$

$\quad F \rightarrow .id$

GOTO (I_4 , E)

$I_8 : F \rightarrow (E.)$

$\quad E \rightarrow E.+ T$

GOTO (I_4 , T)

$I_2 : E \rightarrow T.$

$\quad T \rightarrow T.* F$

GOTO (I_4 , F)

$I_3 : T \rightarrow F.$

GOTO (I_6 , T)

$I_9 : E \rightarrow E + T.$

$\quad T \rightarrow T.* F$

GOTO (I_6 , F)

$I_3 : T \rightarrow F.$

GOTO (I_6 , ( )

$I_4 : F \rightarrow (.E)$

GOTO (I_6 , id)

$I_5 : F \rightarrow id.$

GOTO (I_7 , F)

$I_{10} : T \rightarrow T * F.$

GOTO (I_7 , ( )

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

$\quad E \rightarrow .E + T$

$\quad E \rightarrow .T$

$\quad T \rightarrow .T * F$

$\quad T \rightarrow .F$

$\quad F \rightarrow .(E)$

$\quad F \rightarrow .id$

GOTO (I_7 , id)

$I_5 : F \rightarrow id.$

GOTO (I_8 ,))

$I_{11} : F \rightarrow (E).$

GOTO (I_8 , +)

$I_6 : E \rightarrow E +.T$

$\quad T \rightarrow .T * F$

$\quad T \rightarrow .F$

$\quad F \rightarrow .(E)$

$\quad F \rightarrow .id$

GOTO (I_9 , *)

$I_7 : T \rightarrow T *.F$

$\quad F \rightarrow .(E)$

$\quad F \rightarrow .id$

GOTO (I_4 , ( )

I_4 : 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_O	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_{1O}		r3	r3		r3	r3			
I_{11}		r5	r5		r5	r5			

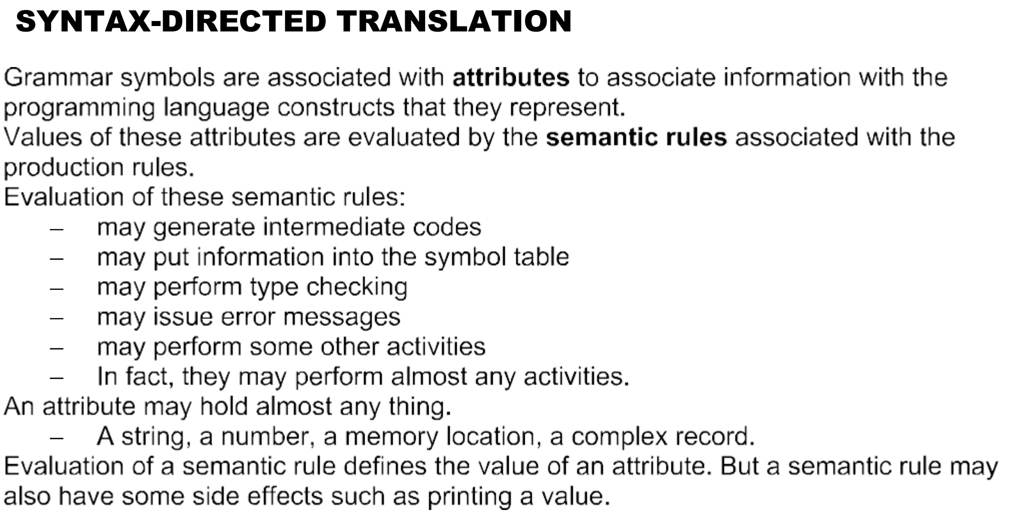
Blank entries are error entries.

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**

UNIT-3



SYNTAX-DIRECTED TRANSLATION

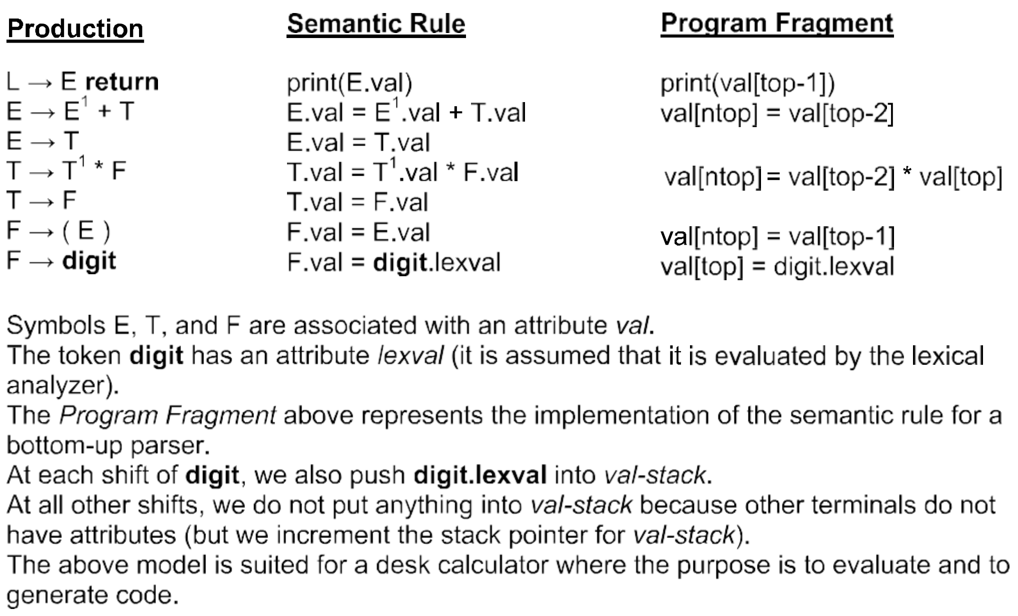
Grammar symbols are associated with **attributes** to associate information with the programming language constructs that they represent.
Values of these attributes are evaluated by the **semantic rules** associated with the production rules.
Evaluation of these semantic rules:

- may generate intermediate codes
- may put information into the symbol table
- may perform type checking
- may issue error messages
- may perform some other activities
- In fact, they may perform almost any activities.

An attribute may hold almost any thing.

- A string, a number, a memory location, a complex record.

Evaluation of a semantic rule defines the value of an attribute. But a semantic rule may also have some side effects such as printing a value.



Production	**Semantic Rule**	**Program Fragment**
L → E **return**	print(E.val)	print(val[top-1])
E → E^1 + T	E.val = E^1.val + T.val	val[ntop] = val[top-2]
E → T	E.val = T.val	
T → T^1 * F	T.val = T^1.val * F.val	val[ntop] = val[top-2] * val[top]
T → F	T.val = F.val	
F → (E)	F.val = E.val	val[ntop] = val[top-1]
F → **digit**	F.val = **digit**.lexval	val[top] = digit.lexval

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

2.1 SYNTAX-DIRECTED TRANSLATION(SDT)

A formalism for specifying translations for programming language constructs.(attributes of a construct: type, string, location, etc)

- Syntax directed definition(SDD) for the translation of constructs
- Syntax directed translation scheme(SDTS) for specifying translation

Postfix notation for an expression E

- If E is a variable or constant, then the postfix nation for E is E itself (E.t≡E).
- if E is an expression of the form E1 op E2 where op is a binary operator
    - E1' is the postfix of E1,
    - E2' is the postfix of E2
    - then E1' E2' op is the postfix for E1 op E2

- if E is (E1), and E1' is a postfix
  then E1' is the postfix for E

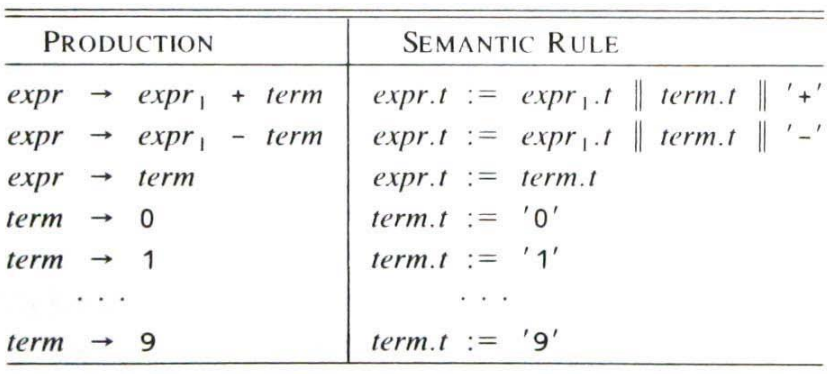
eg) 9 - 5 + 2 ⇒ 9 5 - 2 +

9 - (5 + 2) ⇒ 9 5 2 + -

Syntax-Directed Definition(SDD) for translation

- SDD is a set of semantic rules predefined for each productions respectively fortranslation.
- A translation is an input-output mapping procedure for translation of an input X,
  - construct a parse tree for X.
  - synthesize attributes over the parse tree.
    - Suppose a node n in parse tree is labeled by X and X.a denotes the valueof attribute a of X at that node.
    - compute X's attributes X.a using the semantic rules

associated with X.Example 2.6. SDD for infix to postfix translation



PRODUCTION	SEMANTIC RULE
$expr \rightarrow expr_1 + term$	$expr.t := expr_1.t \;\Vert\; term.t \;\Vert\; '+'$
$expr \rightarrow expr_1 - term$	$expr.t := expr_1.t \;\Vert\; term.t \;\Vert\; '-'$
$expr \rightarrow term$	$expr.t := term.t$
$term \rightarrow 0$	$term.t := '0'$
$term \rightarrow 1$	$term.t := '1'$
. . .	. . .
$term \rightarrow 9$	$term.t := '9'$

Fig 2.5. Syntax-directed definition for infix to postfix translation.

An example of synthesized attributes for input X=9-5+2

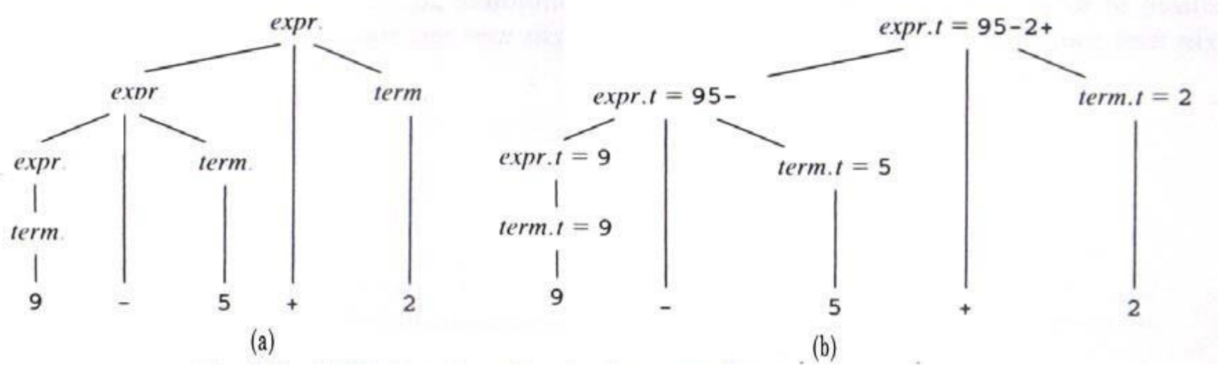


Fig 2.6. Attribute values at nodes in a parse tree.

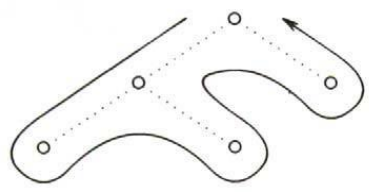
Syntax-directed Translation Schemes(SDTS)

- A translation scheme is a context-free grammar in which program

fragments calledtranslation actions are embedded within the right
sides of the production.

productions(postfix)	SDD for postfix to infix notation	SDTS
list → list + term list.t = li	t.t || term.t || "+" list → lis	+ term {print("+")}

- {print("+");} : translation(semantic) action.
- SDTS generates an output for each sentence x generated by underlying grammar by executing actions in the order they appear during depth-first traversal of a parse tree for x.
  1. Design translation schemes(SDTS) for translation
  2. Translate :
     a) parse the input string x and
     b) emit the action result encountered during the depth-first traversal of parse tree.



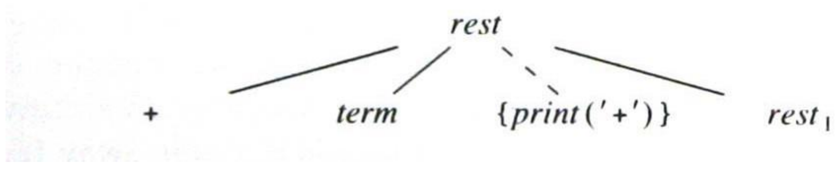


Fig 2.7. Example of a depth-first traversal of a tree. **Fig 2.8.** An extra leaf is constructed for a semantic action.

Example 2.8.
- SDD vs. SDTS for infix to postfix translation.

productions	SDD	SDTS

expr → list + term
expr → list + term
expr → term
term → 0
term → 1
…
term → 9
	expr.t = list.t || term.t || "+"
	expr.t = list.t || term.t || "-"
	expr.t = term.t
	term.t = "0"
	term.t = "1"
	…
	term.t = "9"
		expr → list + term printf{"+")}
		expr → list + term printf{"-")}
		expr → term
		term → 0 printf{"0")}
		term → 1 printf{"1")}
		…
		term → 9 printf{"0")}

- Action translating for input 9-5+2

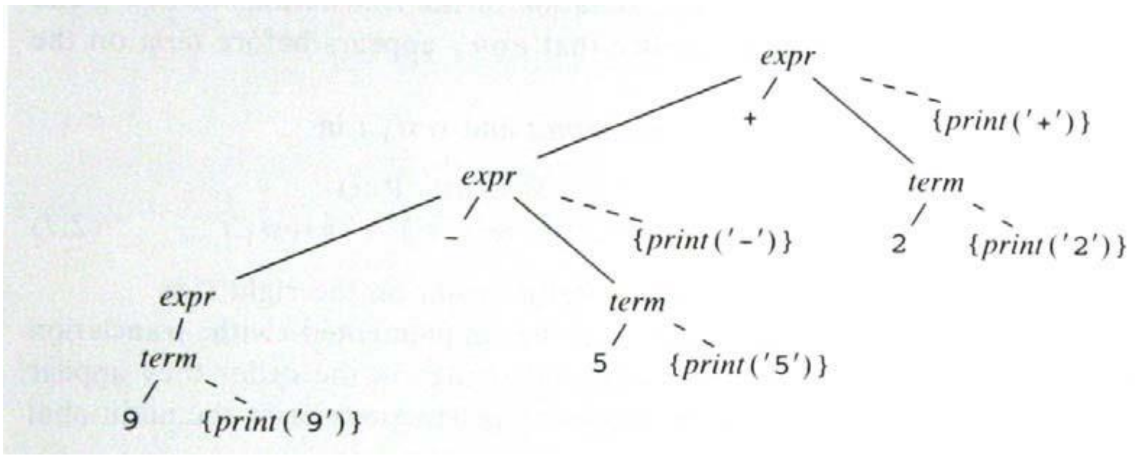


Fig 2.9. Actions translating 9-5+2 into 95-2+.

1) Parse.
2) Translate.

Do we have to maintain the whole parse tree ?

No, Semantic actions are performed during parsing, and we don't need the nodes (whosesemantic actions done).

Unit-4

Intermediate Code Generation

Intermediate codes are machine independent codes, but they are close to machine instructions.

The given program in a source language is converted to an equivalent program in an intermediate language by the intermediate code generator.

Intermediate language can be many different languages, and the designer of the compiler decides this intermediate language.

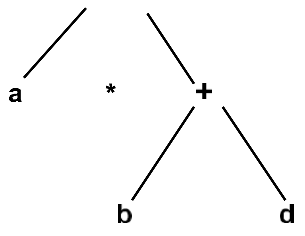
- syntax trees can be used as an intermediate language.
- postfix notation can be used as an intermediate language.
- three-address code (Quadraples) can be used as an intermediate language
    - we will use quadraples to discuss intermediate code generation
    - quadraples are close to machine instructions, but they are not actual machine instructions.

Syntax Tree

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

Production	**Semantic Rule**
op	E.val = NODE (op, E1.val, E2.val)
(E1)	E.val = E1.val
E1	E.val = UNARY (- , E1.val)
	E.val = LEAF (id)

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



Postfix Notation

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

Production	Semantic Rule	Program Fragment
E → **op**	E.code = E1.code ||	print op
E →(E1)	E.code = E1.code	print id
E → E1	E.code = id	

Three-Address Code We use the term "three-address code" because each statement usually contains three addresses (two for operands, one for the result).

The most general kind of three-address code is:

x **:=** y ***op*** z

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

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

op y,z,x

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

Representation of three-address codes

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

Example:

A = -B * (C + D)

Three-Address code is as follows:

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

Translation of Assignment Statements

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

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

Control-Flow Representation of Boolean Expressions

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

In A or B, if we determine A to be true, we need not evaluate B and can declare the entire expression to be true.

In A and B, if we determine A to be false, we need not evaluate B and can declare the entire expression to be false.

A better code can thus be generated using the above properties.

Example:

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

```
if A<B goto (4)
if C<D goto (4)
goto (6)
T = y + z
X = T
```

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

Generating 3-address code for Numerical Representation of Boolean expressions

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

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

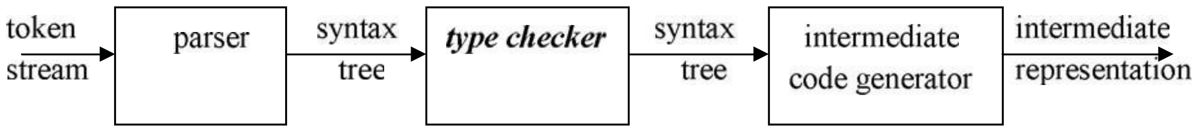
TYPE CHECKING

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.

The following are the definitions of type expressions:

Type Expressions

The type of a language construct will be denoted by a "type expression."

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.

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