USN | | | | | | | | | | |

Solution to Internal Assessment Test I – May 2022

Sub:	System Software & Compiler Design					Sub Code:	18CS61	Branch:	CSE
Date:	06/05/2022	Duration:	90 min's	Max Marks:	50	Sem/Sec:	6/CSE(A,B,C)		OBE

1. **What is Compiler? Explain the various phases of a compiler with a neat diagram. Show the translations for an assignment statement A=B/C*D-6+E, clearly indicate the output of each phase.**

Solution:

Phases of Compiler

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler.

1.Lexical Analysis

The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes.

2.Syntax Analysis

The next phase is called the syntax analysis or **parsing**. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.

3.Semantic Analysis

Semantic analysis checks whether the parse tree constructed follows the rules of language.

Example:-Assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.

4.Intermediate Code Generation

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-

level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

5.Code Optimization

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

6.Code Generation

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

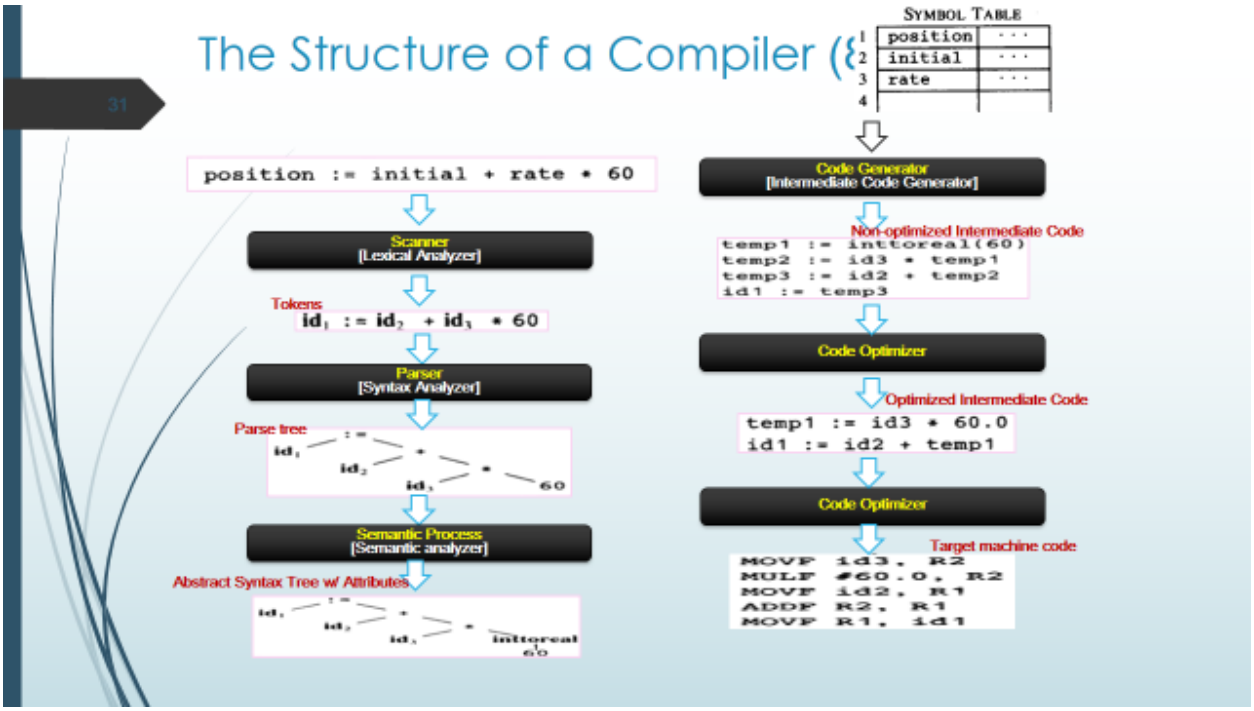
Symbol Table

It is a data-structure maintained throughout all the phases of a compiler.

1. All the identifier's names along with their types are stored here.
2. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

1	Id1	-----
2	Id2	----
3	Id3	-----

1	Num1	----
2	Num2	-----



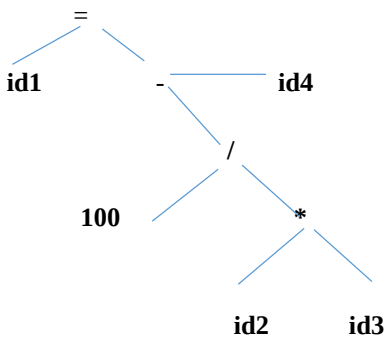
Translations for an assignment statement A=B/C*D-6+E, clearly indicate the output of each phase.

1.Lexical analysis:

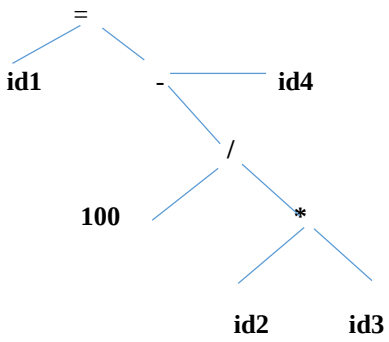
<id1>< = ><id2>< /><id3><*></><100>< - ><id4,4>

2.Syntax analysis:

Create the syntax tree:



3.**Semantic analysis:** remain same because no mismatch in data type



4.Intermediate Code Generation

T1=id2 * id3

T2=T1/100

T3=T2-id4

Id1=T3

5.Code Optimization

T1=id2 * id3

T2=T1/100

Id1=T2-id4

6. Code Generation

```
LDA r1,id2
LDA r2,id3
MUL r1,r1,r2
LDA r3, #100
DIV r1,r1,r3
LDA  r4,id4
SUB r1,r1,r4
STR id1,r1
```

7.Symbol Table

It is a data-structure maintained throughout all the phases of a compiler.

3. All the identifier's names along with their types are stored here.
4. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

1	Id1	-----
2	Id2	----
3	Id3	-----
4	Id4	-----

1	100	----

2.

a)Write the algorithm used for eliminating the left recursion.

Solution:-

Left recursion elimination algorithm:

Arrange the nonterminals in some order A1,A2,…,An.
For (each i from 1 to n)
{
For (each j from 1 to i-1)
{
Replace each production of the form Ai->Ajγ by the production Ai ->δ1 γ | δ2 γ | … |δk γwhere Aj->δ1 |δ2 | … |δkare all current Aj productions
}
Eliminate left recursion among the Ai-productions
}

b) Eliminate left recursion from the given grammar. A → Ba / Aa/ c B → Bb / Ab / d

Solution:

Step-01:

First let us eliminate the left recursion from A → Ba / Aa / c

Eliminating left recursion from here, we have-

A → BaA' / cA'

A' → aA' / ∈

Now, given grammar becomes-

A → BaA' / cA'

A' → aA' / ∈

B → Bb / Ab / d

Step-02:

Substituting the productions of A in B → Ab, we get the following grammar-

A → BaA' / cA'

A' → aA' / ∈

B → Bb / BaA'b / cA'b / d

Step-03:

Now, eliminating the left recursion from the productions of B, we get the following grammar-

A → BaA' / cA'

A' → aA' / ∈

B → cA'bB' / dB'

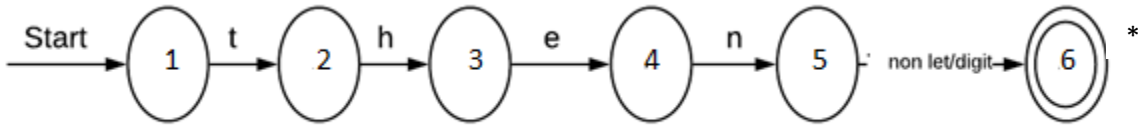
B' → bB' / aA'bB' / ∈

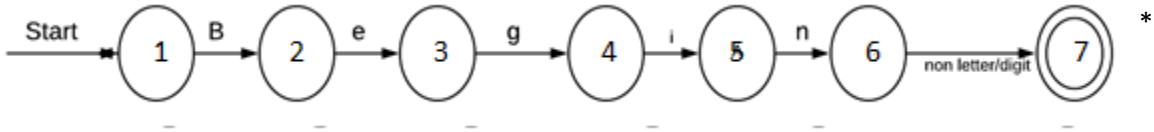
This is the final grammar after eliminating left recursion.

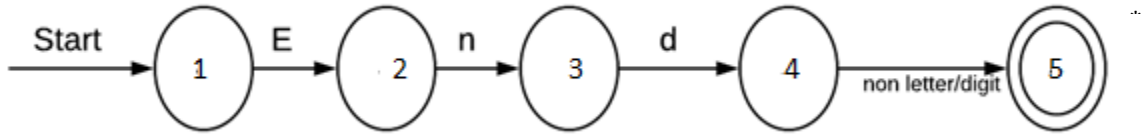
c)Construct transition diagram for keywords like Begin, end, if, then ,else and Identifiers and Relational operator

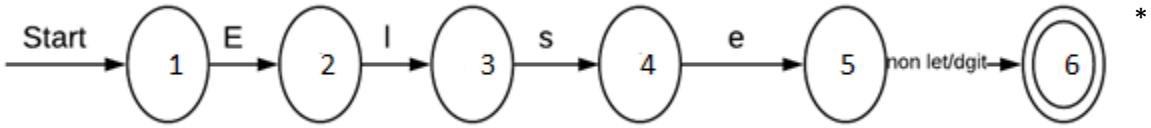
Solution:

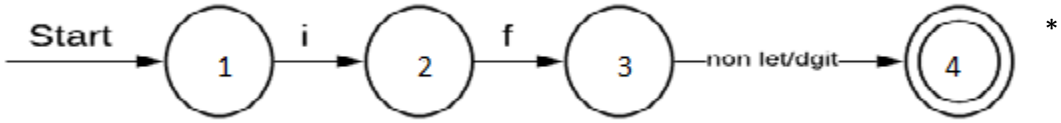
1.Keywords



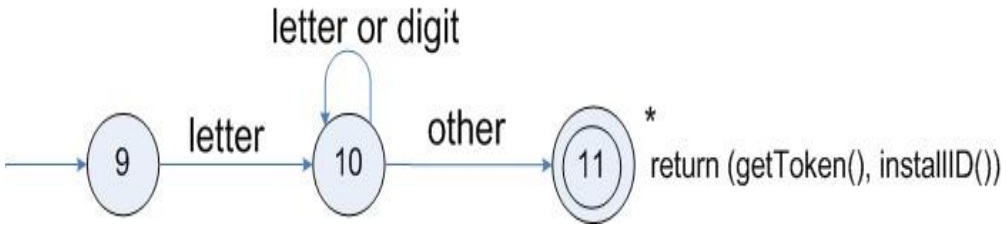








2.Identifiers:-

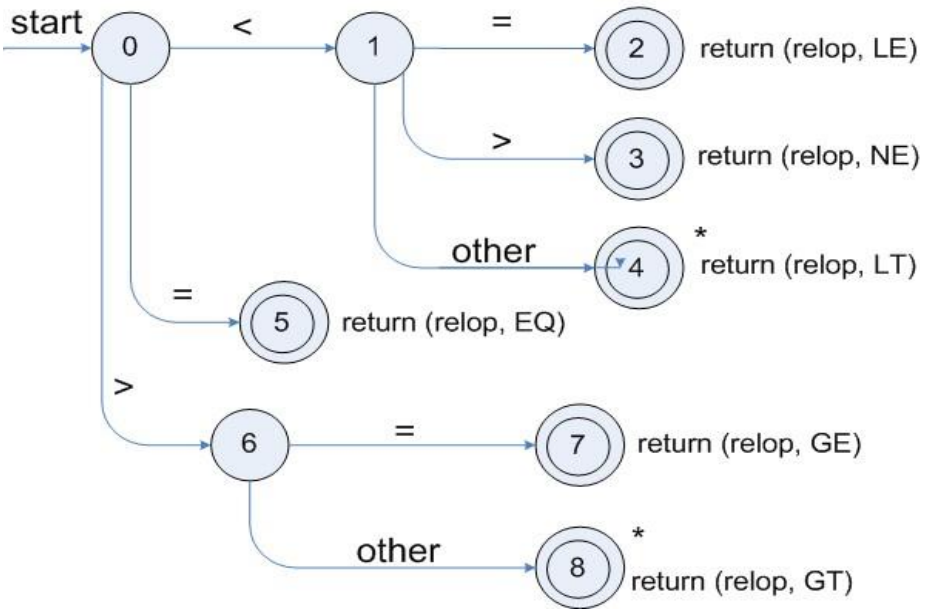


Recognition of Reserved Words and Identifiers

1. Install the reserved words in the symbol table initially
2. Create separate transition diagrams for each keyword

- **identifier-**a **call to installID**places it in the symbol table if it is not already there and returns a pointer to the symbol-table entry for the lexeme found.
- **identifier or reserve word -** Function ***getToken***examines the symbol table entry for the lexeme found, and returns - either id or one of the keyword tokens that was initially installed in the table.

3.Relation operator:



3 a)**What are the key problems with top down parsing?**

Solution:

Problem with top down parsing:

1. The key problem is that of determining the production to be applied for a nonterminal, say A. Once an A-production is chosen, the rest of the parsing process consists of "matching" the terminal symbols in the production body with the input string.
2. This may require backtracking to find the correct A-production tobe applied.
3. Elimination of left recursion of grammar is required
4. Left factoring is required

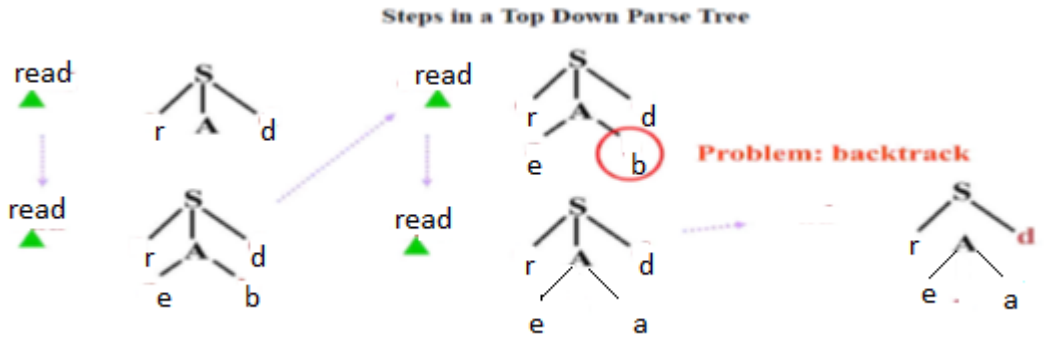
b)A recursive descent parser for the grammarS→ rXd X→ eb | ea for the input string **w=read**.

Solution:

1. General category of Parsing Top-Down

2. A recursive-descent parsing program consists of a set of procedures, one for each nonterminal.
3. Execution begins with the procedure for the start symbol, which halts and announces success if
4. its procedure body scans the entire input string.
5. Choose production rule based on input symbol
6. May require backtracking to correct a wrong choice.

Example: S ->rA d
A ->eb | ea



input: read

(c) What is Token, Lexeme and pattern? Explain with an example.

Solution:

*A **token*** is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes. We will often refer to a token by its token name.

*A **pattern***is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is *matched* by many strings.

*A **lexeme*** is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

Token	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters i, f	if
else	characters e, 1,s, e	else

Comparison	comparison< or > or <= or >= or == or ! =	<=
Id	letter followed by letters and digits	Total
Numeral	any numeric constant	3.4
Literal	anything but ", surrounded by “	“hello world”

In many programming languages, the following classes cover most or all of the tokens:

1. One token for each keyword. The pattern for a keyword is the same as the keyword itself.
2. Tokens for the operators, either individually or in classes such as the token
3. One token representing all identifiers.
4. One or more tokens representing constants, such as numbers and literal strings.
5. Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon

Example: Consider the following C statement

printf ("Total = %d\n”, score) ;

bothprintf and score are lexemes matching the pattern for token **id,** and

"Total = %d\n” is a lexeme matching literal.

4.a) Given the grammar: **D->T V_L T->int | float V_L ->id,V_L | id**
Construct the predictive parsing table and show the moves made by predictive parser on the input string **w=inta,b,c**
Solution:

1.D->T V_L 2.T->int | float 3.V_L ->id,V_L | id

<u>1.Apply left factor</u>
V_L ->id,V_L | id

V_L ->id V_L^1
V_L^1-> , V_L | ε

After left factor the grammar is
1.D->T V_L 2.T->int | float 3. V_L ->id V_L^1 4.V_L^1 -> , V_L | ε

2.First and Follow set

First(D) ={int ,float} Follow (D) ={$}
First(T) ={int ,float} follow (T) ={id}
First(V_L) ={id} follow (V_L) ={$}
First(V_L^1)= {, ε } follow (V_L^1) ={$}

Rule:
1.D->T V_L
2.T->int
3.T->float
4. V_L ->id V_L^1
5.V_L^1 -> , V_L
6.V_L^1 ->ε

3.Parsing table

NT	**ACTION**				
	int	**float**	**Id**	**,**	**$**
D	**D->T V_L**	**D->T V_L**			
T	**T->int**	**T-> float**			
V_L			**V_L ->id V_L^1**		
V_L^1				**V_L^1 -> , V_L**	**V_L^1 ->ε**

Tracing the string

Stack	**Input**	**action**
$D	**inta,b,c$**	**D->T V_L**
$V_L T	**inta,b,c $**	**T->int**
$V_L int	**inta,b,c$**	match
$ V_L^1	**a,b,c$**	**V_L ->id V_L^1**
$ V_L^1id	**a,b,c$**	match
$ V_L^1	**,b,c$**	**V_L^1 -> , V_L**
$ V_L ,	**,b,c $**	match
$ V_L	**b,c$**	**V_L ->id V_L^1**
$ V_L^1id	**b,c$**	match
$$V_L^1$	**,c $**	**V_L^1 -> , V_L**
$V_L ,	**,c $**	match
$ V_L	**c$**	**V_L ->id V_L^1**

$ V_L[1]id	**c$**	match
$ V_L[1]	**$**	**V_L[1] ->ε**
$	**$**	**accept**

5 (a) Write the algorithm to find FIRST and FOLLOW for the given grammar

Solution:

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

FIRST and FOLLOW sets

- If X is terminal, FIRST(X) = {X}.
- If X → ε is a production, then add ε to FIRST(X).
- If X is a non-terminal, and X → Y_1 Y_2 … Y_k is a production, and ε is in all of FIRST(Y_1), …, FIRST(Y_k), then add ε to FIRST(X).

FOLLOW(A) for a nonterminal A is defined as to be the set of terminals a that can appear immediately to the right of A in some sentential form

That is, the set of terminals a such that there exists a derivation of the form S => αAaβ , for some α and β

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

- If $ is the input end-marker, and S is the start symbol, $ ∈ FOLLOW(S).
- If there is a production, A → αBβ, then (FIRST(β) – ε) ⊆ FOLLOW(B).
- If there is a production, A → αB, or a production A → αBβ, where ε∈ FIRST(β), then FOLLOW(A) ⊆ FOLLOW(B).

5. (b) Compute FIRST and FOLLOW for the grammar
S ->ACB|Cbb|Ba A ->da|BC B->g|Є C-> h| Є
Solution:

FIRST Set:

FIRST(S) = FIRST(A) U FIRST(B) U FIRST(C) = { d, g, h, Є, b, a}
FIRST(A) = { d } U FIRST(B) = { d, g, h, Є }
FIRST(B) = { g, Є }
FIRST(C) = { h, Є }

FOLLOW Set

FOLLOW(S) = { $ }
FOLLOW(A) = { h, g, $ }
FOLLOW(B) = { a, $, h, g }
FOLLOW(C) = { b, g, $, h }

6 (a) Show that the following grammar is ambiguous. Write an equivalent unambiguous grammar for the same. **S→S + S | S * S | S ^ S |a**

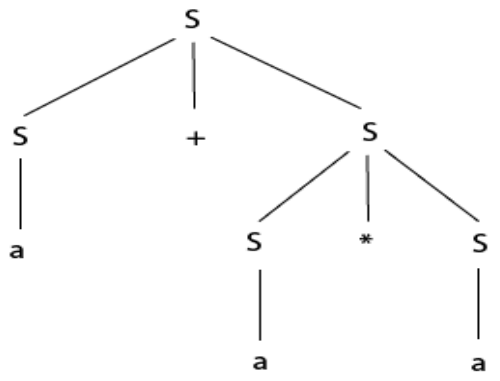
Solution:

Two LMD for same string so grammar is ambiguous

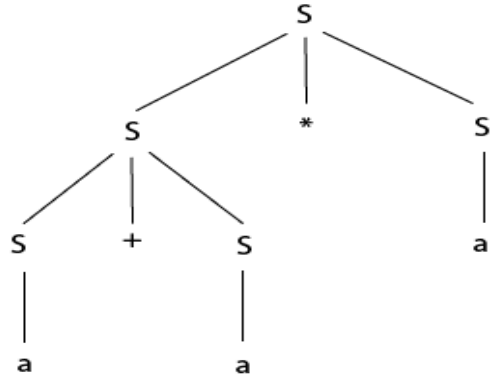
Example:

String a+ a*a can be derived in 2 ways-LMD

Solution: The given grammar is ambiguous because the derivation of string aab can be represented by the following string:



Parse tree 1



Parse tree 2

Unambiguous grammar will be:

1. S → S + A | A
2. A → A * B | B
3. B → C ^ B | C
4. C → a

6. **b)Eliminate left factoring from the following grammar:**
   S → bSSaaS / bSSaSb / bSb / a
   S → a / ab / abc/ abcd

1)S → bSSaaS / bSSaSb / bSb / a

Solution:
S → bSS' / a
S' →SaA |b
A→aS|Sb

2)S → a / ab / abc/ abcd

Solution:
S → aS'

S' → b / bc / bcd / ∈

S → aS'

S' → bA / ∈

A → c / cd / ∈

Final grammar is

S → aS'

S' → bA / ∈

A → cB / ∈

B → d / ∈

7.

(a) Explain the input buffering strategy used in lexical analysis phase.

Solution:

Input Buffering:

- Some efficiency issues concerned with the buffering of input.
- A two-buffer input scheme that is useful when lookahead on the input is necessary to identify tokens.
- Techniques for speeding up the lexical analyser, such as the use of sentinels to mark the buffer end.
- There are three general approaches to the implementation of a lexical analyser:
    1. Use a lexical-analyser generator, such as Lex compiler to produce the lexical analyser from a regular expression based specification. In this, the generator provides routines for reading and buffering the input.
    2. Write the lexical analyser in a conventional systems-programming language, using I/O facilities of that language to read the input.
    3. Write the lexical analyser in assembly language and explicitly manage the reading of input.

Buffer pairs:

- Because of a large amount of time can be consumed moving characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character.
- The scheme to be discussed:
- Consists a buffer divided into two N-character halves.

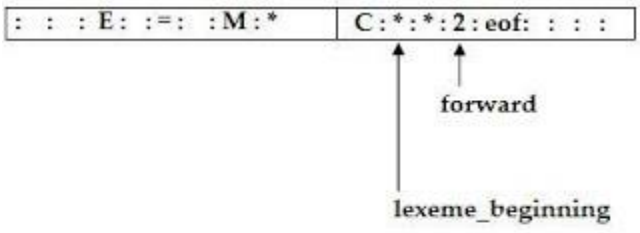


Fig 1.8 An input buffer in two halves

N – Number of characters on one disk block, e.g., 1024 or 4096.

1. Read N characters into each half of the buffer with one system read command.
2. If fewer than N characters remain in the input, then eof is read into the buffer after the input characters.
3. Two pointers to the input buffer are maintained.
4. The string of characters between two pointers is the current lexeme.
5. Initially both pointers point to the first character of the next lexeme to be found.
6. Forward pointer, scans ahead until a match for a pattern is found.
7. Once the next lexeme is determined, the forward pointer is set to the character at its right end.
8. If the forward pointer is about to move past the halfway mark, the right half is filled with N new input characters.

9. If the forward pointer is about to move past the right end of the buffer, the left half is filled with N new characters and the forward pointer wraps around to the beginning of the buffer.

Disadvantage of this scheme:

10. This scheme works well most of the time, but the amount of lookahead is limited.
11. This limited lookahead may make it impossible to recognize tokens in situations where the distance that the forward pointer must travel is more than the length of the buffer.
12. For example: DECLARE (ARG1, ARG2, … , ARGn) in PL/1 program;
13. Cannot determine whether the DECLARE is a keyword or an array name until the character that follows the right parenthesis.

Sentinels:

1. In the previous scheme, must check each time the move forward pointer that have not moved off one half of the buffer. If it is done, then must reload the other half.
2. Therefore the ends of the buffer halves require two tests for each advance of the forward pointer.
3. This can reduce the two tests to one if it is extend each buffer half to hold a sentinel character at the end.
4. The sentinel is a special character that cannot be part of the source program. (eof character is used as sentinel).

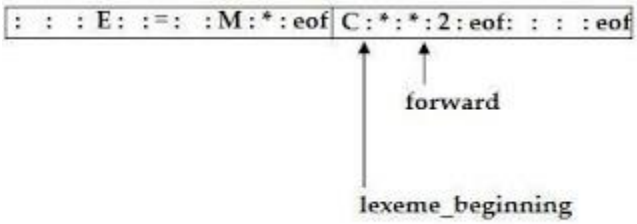


Fig 1.9 Sentinels at the end of each buffer half

1. In this, most of the time it performs only one test to see whether forward points to an eof.
2. Only when it reach the end of the buffer half or eof, it performs more tests.
3. Since N input characters are encountered between eof's, the average number of tests per input character is very close to 1.

(b) Explain the structure of LEX program.

Solution:

ALEXprogramconsistsofthreeparts:

declarations
%%
translation rules
%%
auxiliaryprocedures

Thedeclarationssectionincludesdeclarationsofvariables,constants,and regular definitions.

Thetranslation rulesofalexprogramarestatements oftheform

R1 {action1}
R2{action2}
...…….
Rn{actionn}whereeachRiisregularexpressionandeachactioni,isaprogramfragment describingwhatactionthelexicalanalyzershouldtakewhenpatternRimatcheslexeme.Typically,acti oniwillreturncontroltotheparser.InLexactionsarewritteninC;in general,however,theycanbeinanyimplementationlanguage.

Thethirdsectionholdswhatever auxiliaryproceduresareneededbytheactions.

c)Write a LEX program to recognize valid arithmetic expression. Identifiers in the expression could be only integers and operators could be + and *. Count the identifiers & operators present and print them separately.

```
%{
int count=0,ids=0,bracket=0;
%}
%%
[+] {printf("+");count++;} /*For recognizing the operators*/
[-] {printf("-");count++;}
[*] {printf("*");count++;}
[/] {printf("/");count++;}
[0-9]+ {ids++;} /*For recognizing the identifiers*/
[(] {bracket++;} /*For recognizing the brackets*/
[)] {bracket--;}
%%
int main()
{
printf("Enter the Arithmetic expression:\n");
yylex();
printf("Number of Operators=%d\n",count);
printf("Number of Identifiers=%d\n",ids);
if(count>=ids||bracket!=0||ids==1)
printf("Invalid expression\n");
else
printf("Valid expression\n");
}
```