CD: COMPILER DESIGN
CD: UNIT-1 20/09/2022

20 SEPTEMBER 2022 / IT-3rd year, Vth semester
FALL SEMESTER, YEAR (Vth, 3rd)
FALL SESSION (2022-23)
(CD)
MS. SHWETA TIWARI
Published: SEPTEMBER 20, 2022

PREPARED FOR

Engineering Students

All Engineering College

CD: COMPILER DESIGN

TOPIC On : UNIT-1

Introduction to Lexical Analyzer, Input Buffering, Specification of Tokens, Recognition of Tokens

By SHWETA TIWARI

Under On: INTRODUCTION TO COMPILER

TOPIC On : UNIT-1 Introduction to Lexical Analyzer, Input Buffering, Specification of Tokens, Recognition of Tokens

Role of lexical analysis
OR
How do the parser and scanner communicate

❑ **Role of lexical analysis OR How do the parser and scanner communicate.**

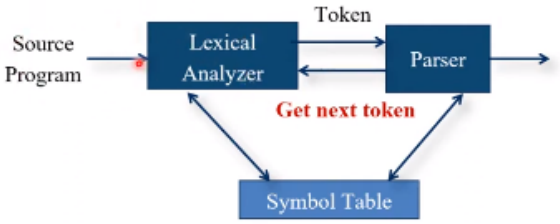


Fig. Communication between Scanner & Parser

- ✓ The lexical analyzer is the first phase of compiler. Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis.
- ✓ Upon receiving a **"Get next token"** command from parser, the lexical analyzer reads the input character until it can identify the next token.
- ✓ Lexical analyzer also stripping out comments and white space in the form of blanks, tabs, and newline characters from the source program.

❑ Why to separate lexical analysis & parsing?

- ✓ The Simplicity in design.
- ✓ Improves compiler efficiency.
- ✓ Enhance compiler portability.

❑ Token, Pattern & Lexemes

❖ **Token**

- ✓ Sequence of character having a collective meaning is known as token.
- ✓ 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**

- ✓ The sequence of character in a source program matched with a pattern for a token is called lexeme.

• Example: Rate, DIET, +, 100

❑ Token, Pattern & Lexemes

❖ Token

✓ Sequence of character having a collective meaning is known as token.

✓ 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

✓ The sequence of character in a source program matched with a pattern for a token is called lexeme.

- Example: Rate, DIET, +, 100

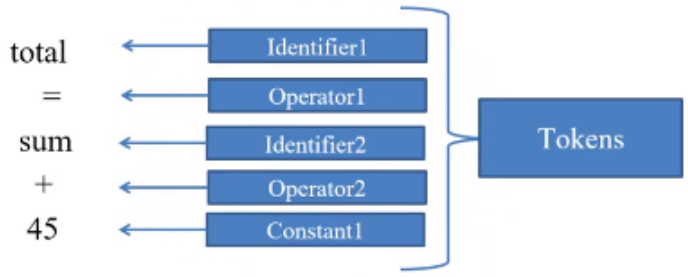
TOKEN, PATTERN & LEXEMES (EXAMPLE)

Example: total = sum + 45

❖ Lexemes

Lexemes for identifier: total, sum
Lexemes for operator: =, +
Lexemes for constant: 45

❖ Tokens



TOKEN, PATTERN & LEXEMES (EXAMPLE)

Consider the following code that is fed to Lexical Analyzer (separate lexeme and token)

```
int maximum(int x, int y)
{
// This will compare 2 numbers
if (x > y)
return x;
else
{
return y;
}
}
```

Lexeme	Token
Int	Keyword
maximum	Identifier
(	Operator
int	Keyword
x	Identifier
,	Operator
int	Keyword
Y	Identifier
)	Operator
{	Operator
If	Keyword
.	.
.	.
.	.
.	.

Input buffering

❑ Input Buffering in Compiler

- ✓ Lexical analyzer reads the source program character by character from the secondary storage but it is costly. Therefore, a block of data is first read into a buffer and then scanned by the lexical analyzer.
- ✓ It uses two pointer, begin ptr(bp) and forward ptr(fp) to keep track of the pointer of the input scanned.
- ✓ Initially both the pointers point to the first character of the input string as shown below:

Fig: initial configuration of buffer

❑ Input Buffering in Compiler

- ✓ Lexical analyzer reads the source program character by character from the secondary storage but it is costly. Therefore, a block of data is first read into a buffer and then scanned by the lexical analyzer.
- ✓ It uses two pointer, begin ptr(bp) and forward ptr(fp) to keep track of the pointer of the input scanned.
- ✓ Initially both the pointers point to the first character of the input string as shown below:

Fig: initial configuration of buffer

❑ Input Buffering in Compiler

- ✓ Lexical analyzer reads the source program character by character from the secondary storage but it is costly. Therefore, a block of data is first read into a buffer and then scanned by the lexical analyzer.
- ✓ It uses two pointer, begin ptr(bp) and forward ptr(fp) to keep track of the pointer of the input scanned.
- ✓ Initially both the pointers point to the first character of the input string as shown below:

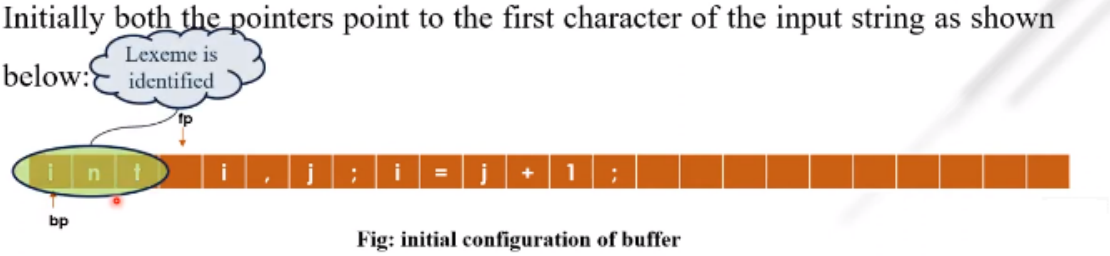


Fig: initial configuration of buffer

❑ There are mainly two techniques for input buffering:

1. ***Buffer pairs (one buffering scheme)***
2. ***Sentinels (two buffer scheme)***

1. Buffer pairs (one buffering scheme)

- ✓ Initially input buffer are divided into two halves of N characters. Where N is number of characters, N is usually the size of buffer(memory), for e.g: 1024 or 4096 bytes.

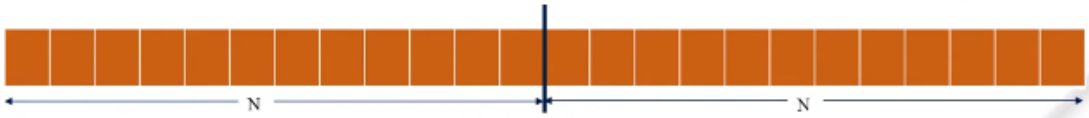


Fig: input buffer

1. Buffer pairs (one buffering scheme)

- ✓ Initially input buffer are divided into two halves of N characters. Where N is number of characters, N is usually the size of buffer(memory), for e.g: 1024 or 4096 bytes.

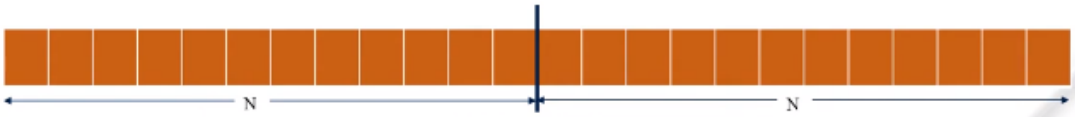


Fig: input buffer

- ✓ Consider the statement:

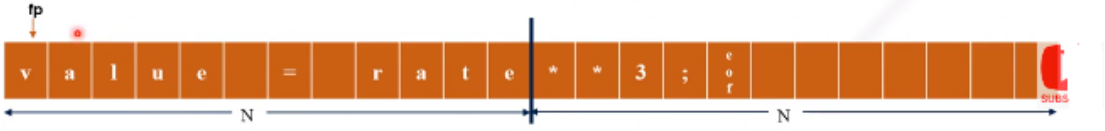
Fig: input buffer with buffer pair

1. Buffer pairs (Code to advance forward pointer)

```
if (forward pointer (fp) is at end of first half )
        {
                reload second half;
                fp = fp + 1;                    // now fp is at beginning of second half
        }
else if (forward pointer (fp) at end of second half)
        {
                reload first half;
                set fp to beginning of first half;
        }
else fp ++;                     //it takes two test for each advnace of forward pointer (fp)
```


1. Buffer pairs (Code to advance forward pointer)

```
if (forward pointer (fp) is at end of first half )
        {
                reload second half;
                fp = fp + 1;                    // now fp is at beginning of second half
        }
else if (forward pointer (fp) at end of second half)
        {
                reload first half;
                set fp to beginning of first half;
        }
else fp ++;                     //it takes two test for each advnace of forward pointer (fp)
```

2. Sentinels (two buffer scheme)

- ✓ *In buffer pairs we must check, each time we move the forward pointer that we have not moved off one of the buffers.*
- ✓ *Thus, for each character read, we make two tests.*
- ✓ To overcome this two tests, we add extra ***eof*** character at the end as sentinel, The sentinel is a special character that cannot be part of the source program.
- ✓ So, each buffer half will extended to hold a sentinel ***eof.***
- ✓ it optimize the code by reducing the number of tests to one per advance of forward pointer (fp)

---

2. Sentinels (two buffer scheme)

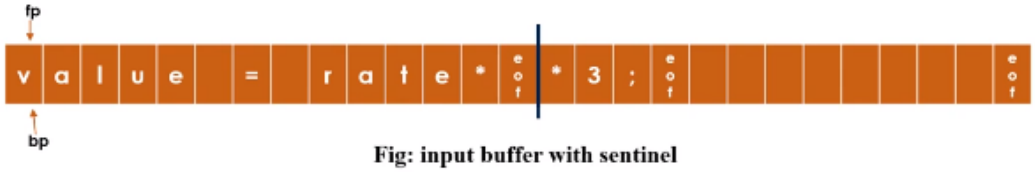


Fig: input buffer with sentinel

---

2. Sentinels (Code to advance forward pointer)

```
fp ++ ;

if (forward pointer (fp) = eof )
        {
            if (forward pointer (fp) is at end of first half )
                {
                    reload second half;
                    fp = fp + 1;
                }
            else if (forward pointer (fp) is at the second half)
                {
                    reload first half;
                    set fp to beginning of first half;
                }
            else
                {
                    terminate lexical analysis;
                }
        }
```

THANK YOU