CS4031
Compiler Construction
Lecture 2

Mahzaib Younas

Lecturer, Department of Computer Science

FAST NUCES CFD

Outlines

- The role of lexical analyzer
- Input Buffering
- Specification of tokens
- Recognition of tokens
- Lexical Analyzer Generator Lex
- Finite Automata
- Design lexical Analyzer generator
- Optimization of DFA based pattern mactches

Lexical Analysis

- The main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program.

It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespaces or comments in the source code.

Lexeme

- A lexeme is a sequence of source code that matches one of the predefined patterns and thereby forms a valid token.
- Example:
- int c = 5;

Lexeme	Tokens
Int	Keyword
C	Identifier
=	Assignment operator
5	constant
;	symbol

Pattern

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

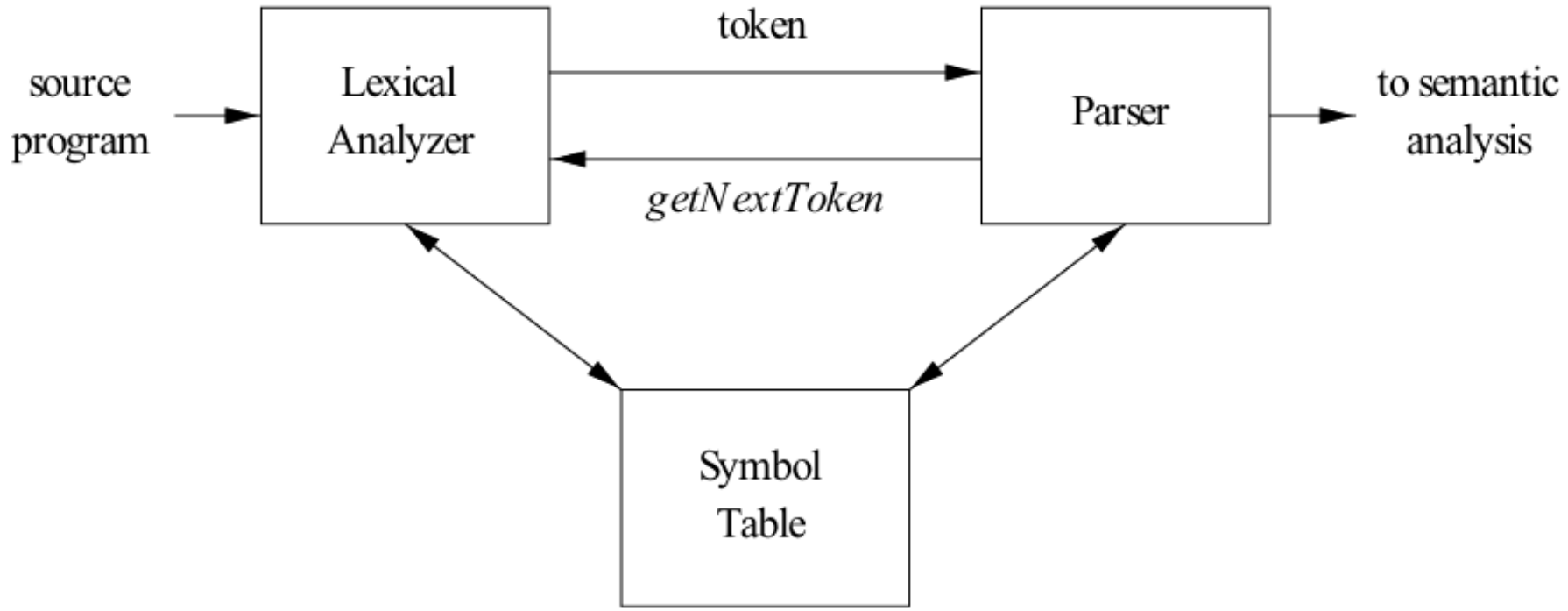
Tokens

A token is a pair consisting of a token name and an optional attribute value.

Partition input string into substring, and classify according to the rule

- **Identifier** x, y11, maxsize
- **Keywords** if else while for
- **Integers** 2 1000
- **Floats** 2.0 1000.0
- **Symbols** +) (> <
- **Strings** “enter x” “error”

Interaction between lexical Analyzer and parser



Symbol Table

- A symbol table is one of the most important data structures within a compiler, where all the identifiers used in a program are stored along with their type, scope, and memory locations.

Example of symbol table

- int semester;
- char x[ ] = “compiler construction”

Name	Type	Size	Dimension	Line Of Declaration	Line Of Usage	Address
semester	int	2	0	-	-	-
x	char	20	1	-	-	-

To see how these concepts are used in practice, in the C statemen

```
printf("Total = %d\n", score);
```

- both printf and score are lexemes matching the pattern for token id, and "Total = %d\n" is a lexeme matching literal.

Token	Informal Description	Sample Lexemes
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but ", surrounded by "'s	`"core dumped"`

Ad-hoc Lexer

- Ad-Hoc means using the concept of already known lanaguges.
- Hand-write code to generate tokens.
- Partition the input string by reading left-to-right.
- Recognize one token at a time.
- Ad-hoc Lexer required Look-a-head
- LOOK A HEAD

  It used to check where one token end and next token begins.

Example: simply we create a class which have ability to make the token of input stream.

```
class scanner
{
 Inputstream s;
 char next;                          //look ahead
 Lexer(Inputstream _s)
 {
        s = _s;
        next = s.read();
 }
```

Example: simply we create a class which have ability to make the token of input stream.

```
class scanner
{
 Inputstream s;
                                  //as we know inputstream is function in
     C++ which used for I/O.
 char next;                                  //look ahead
 Lexer(Inputstream _s)
 {
          s = _s;
          next = s.read();
 }
```

Example: simply we create a class which have ability to make the token of input stream.

```
class scanner
{
 Inputstream s;
 char next;                                  //look ahead
 Lexer(Inputstream _s)     //constructer of the class
 {
        s = _s;    //s is used to store the input stream
        next = s.read();
 }
```

How to perform the Tokenization via Program.

- Here we declare the method of class

```
Token nextToken() {

if( idChar(next) )
  return readId();

if( number(next) )
  return readNumber();

if( next == “” )
  return readString();
 ...
 ...
```

How to perform the Tokenization via Program.

- Here we declare the method of class

```
Token nextToken() {

if( idChar(next) )
  return readId();                  // if the letter is identifier mean follow the rule of
       identifier

if( number(next) )
  return readNumber();

if( next == “” )
  return readString();
 ...
 ...
```

How to Make the token of Identifier

```
Token readId() {
 string id = “”;
 while(true){
  char c = input.read();
  if(idChar(c) == false)
        return
          new Token(TID,id);
  id = id + string(c);
 }
}
```

Ad-Hoc Lexer using C++

- **Identifier** x, y11, maxsize
- **Keywords** if else while for
- **Integers** 2 1000
- **Floats** 2.0 1000.0
- **Symbols** +) (> <
- **Strings** “enter x” “error”

Firstly create the function for keywords

```
int isKeyword(char buffer[]) {
        char keywords[32][10] = {

"auto","break","case","char","const","continue","default","do","double","else","enum","extern","float

","for","goto", "if","int","long","register","return","short","signed",

"sizeof","static","struct","switch","typedef","union","unsigned","void","volatile","while" };
        int i, flag = 0;
        for (i = 0; i < 32; ++i) {
                if (strcmp(keywords[i], buffer) == 0) {
                        flag = 1;
                        break;
                }}
        return flag;
}
```

To check the operator

```
//decalration of operator
 operators[] = "+-*/%=";
for (i = 0; i < 6; ++i) {
if (ch == operators[i])
cout << ch << " is operator\n";
}
```

Check identifiers

```
if (isalnum(ch)) {
buffer[j++] = ch;
}
else if ((ch == ' ' || ch == '\n') && (j != 0)) {
buffer[j] = '\0';
j = 0;
if (isKeyword(buffer) == 1)
cout << buffer << " is keyword\n";
else
cout << buffer << " is indentifier\n";
}
}
```

How to describe the tokens?

Regular languages are the most popular for specifying the tokens.

1. Simple and useful theory
2. Easy to understand
3. Efficient implementations

Languages

- Let S be a set of characters. S is called the *alphabet*.
- A *language over* S is set of strings of characters drawn from S.

Notations

- Languages are sets of strings (finite sequence of characters)
- Need some notation for specifying which sets we want
- For lexical analysis we care about *regular languages*.
- Regular languages can be described using *regular expressions*.

Regular languages

- Each regular expression is a notation for a regular language (a set of words).
- If **A** is a regular expression, we write **L(A)** to refer to language denoted by **A**.
- A *regular expression* (*RE*) is defined inductively

  a ordinary character from S

  ε the empty string

Basics of Regular expression

Functionalities	Purpose
R | S	Either R or S
RS	R followed by S
R*	Concatenation of R zero or more time
R?	E|R (Zero or one R)
R+	RR* (one or more R)
(R)	R (grouping)

Example

Integers

A non-empty string of digits

- Digits = 0|1|2|3|4|5|6|7|8|9|10
- Integer = digit digit*

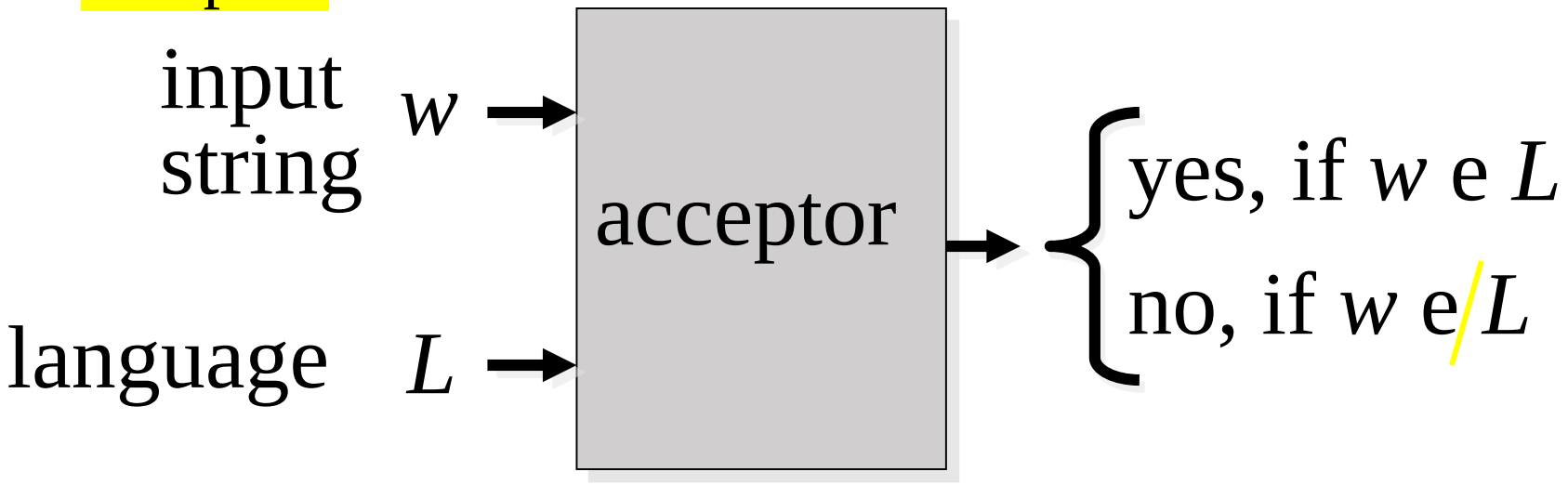
Identifiers

String or letter or digits starting with a letter

- C identifier
- a – z
- A – Z
- 0 - 9

How to use RE?

- We need mechanism to determine if an input string *w* belongs to *L(R)*, the language denoted by regular expression *R*. Such a mechanism is called acceptor.



Requirement

- **Specification**

  Regular Expression

- **Implementation**

  Finite Automata

Finite Automata

Finite Automaton consists of

- An input alphabet (**S)**
- A set of states
- A start (initial) state
- A set of transitions
- A set of accepting (final) states

Finite Automata

- A finite automaton ***accepts*** a string if we can follow transitions labelled with characters in the string from start state to some accepting state.
- **FA Example**: A FA that accepts any number of 1's followed by signle 0.