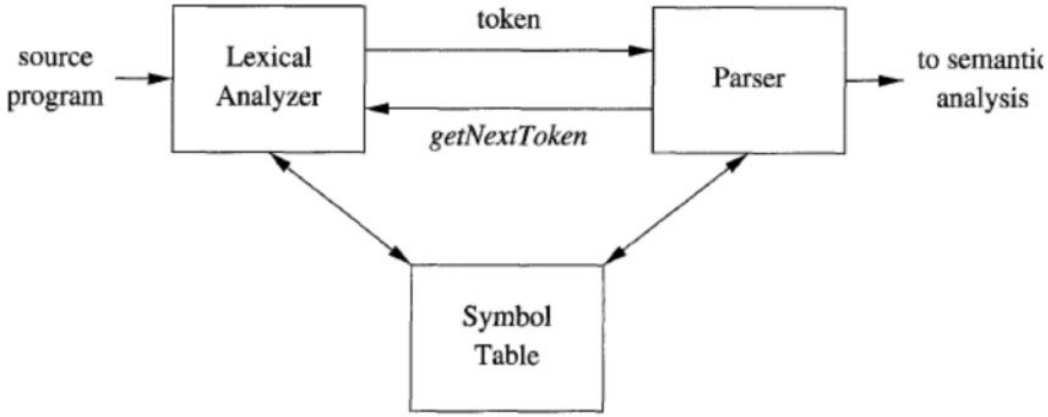
Assignment 2

Lexical Analysis

Lexical Analysis

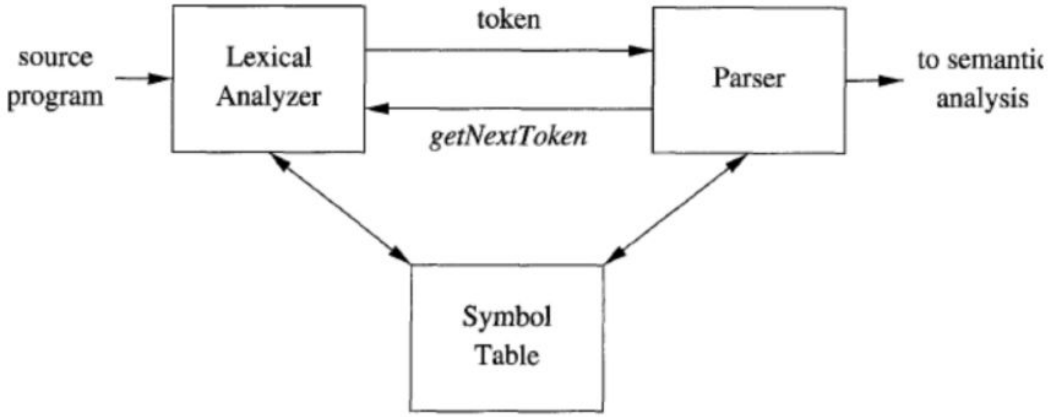
- Lexical Analysis is the first phase of compiler
- Converts sequence of characters (source program) into a sequence of tokens
- Used by the parser in the next stage to build an abstract syntax tree
  - Assignment 3 & 4



Interactions between the lexical analyzer and the parser

Lexical Analysis

- Also interacts with the symbol table (e.g. store information about identifiers etc.)



Interactions between the lexical analyzer and the parser

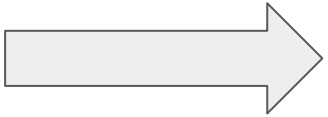
Lexical Analysis

- Identify lexemes and generate tokens
- Remove white spaces
- Detect some error and generate messages along with line no.

```
int f(int a){
    return 2*a;
    a=9;
}

int g(int a, int b){
    int x;
    x=f(a)+a+b;
    return x;
}

int main(){
    int a,b,i;
    a=1;
    b=2;
    a=g(a,b);
    for(i=0;i<4;i++){
        a=3;
        while(a--){
            b++;
        }
    }
    println(a);
    println(b);
    return 0;
}
```

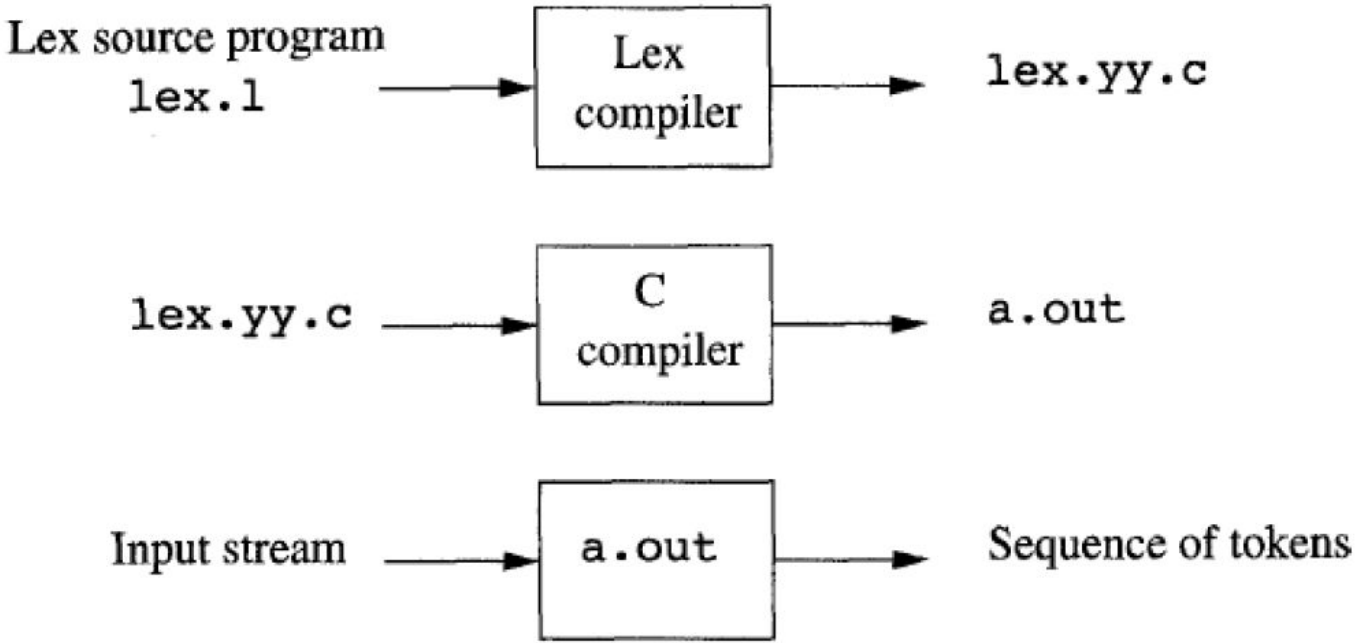


```
<INT> <ID, f> <LPAREN, (> <INT> <ID, a> <RPAREN,
)> <LCURL, {> <RETURN> <CONST_INT, 2> <MULOP, *>
<ID, a> <SEMICOLON, ;> <ID, a> <ASSIGNOP, =>
<CONST_INT, 9> <SEMICOLON, ;> <RCURL, }> <INT>
<ID, g> <LPAREN, (> <INT> <ID, a> <COMMA, ,>
<INT> <ID, b> <RPAREN, )> <LCURL, {> <INT> <ID,
x> <SEMICOLON, ;> <ID, x> <ASSIGNOP, => <ID, f>
<LPAREN, (> <ID, a> <RPAREN, )> <ADDOP, +> <ID,
a> <ADDOP, +> <ID, b> <SEMICOLON, ;> <RETURN>
<ID, x> <SEMICOLON, ;> <RCURL, }> <INT> <ID,
main> <LPAREN, (> <RPAREN, )> <LCURL, {> <INT>
<ID, a> <COMMA, ,> <ID, b> <COMMA, ,> <ID, i>
<SEMICOLON, ;> <ID, a> <ASSIGNOP, => <CONST_INT,
1> <SEMICOLON, ;> <ID, b> <ASSIGNOP, =>
<CONST_INT, 2> <SEMICOLON, ;> <ID, a> <ASSIGNOP,
=> <ID, g> <LPAREN, (> <ID, a> <COMMA, ,> <ID,
b> <RPAREN, )> <SEMICOLON, ;> <FOR> <LPAREN, (>
<ID, i> <ASSIGNOP, => <CONST_INT, 0> <SEMICOLON,
;> <ID, i> <RELOP, <> <CONST_INT, 4> <SEMICOLON,
;> <ID, i> <INCOP, ++> <RPAREN, )> <LCURL, {>
<ID, a> <ASSIGNOP, => <CONST_INT, 3> <SEMICOLON,
;> <WHILE> <LPAREN, (> <ID, a> <INCOP, -->
<RPAREN, )> <LCURL, {> <ID, b> <INCOP, ++>
<SEMICOLON, ;> <RCURL, }> <RCURL, }> <ID,
println> <LPAREN, (> <ID, a> <RPAREN, )>
<SEMICOLON, ;> <ID, println> <LPAREN, (> <ID, b>
<RPAREN, )> <SEMICOLON, ;> <RETURN> <CONST_INT,
0> <SEMICOLON, ;> <RCURL, }>
```

Introduction to Flex

- Flex is a tool to generate the lexer so that we do not have to build one from scratch
- Free, open source
  - sudo apt-get update
  - sudo apt-get install flex

Introduction to Flex



Creating a lexical analyzer with Lex

Lex file (.l)

- Example code : wordcount.l
- 3 Parts : separated by %%
  - Definition section
  - Rules section
  - Subroutine section
- Definition section contains option settings, declaration of start states and variables (to be used in the rules section), and some C codes
- In the declaration section, code inside of %{ and %} is copied through verbatim near the beginning of the generated C source file.
  - Include libraries for C code
  - Declare C variables and define functions to be used in the action part of the rules section

Lex File Divisions

- Rules section contains regular expression/pattern to be matched and corresponding action (C++ code) to be taken if the expression is matched
  - One or more lines of code in braces
- Each pattern must start at the beginning of the line (no spaces)
  - Flex considers any line that starts with whitespaces as code to be copied into the C program
- Subroutine section contains C code to be copied to the lex.yy.c file
  - Usually contains the main function
  - Calls yylex() , that is the flex's scanner routine.
  - Point yyin to the input file you are trying to scan (by default yyin is set to stdin).

Definition Section

- %option Provides some options: noyywrap, yylineno, nodefault etc.
- %option noyywrap (TLDR: Just put it and don’t worry about anything else)

> When a lex scanner reached the end of `yyin`, it called `yywrap()`. The idea was that if there was another input file, `yywrap` could adjust `yyin` and return 0 to resume scanning. If that was really the end of the input, it returned 1 to the scanner to say that it was done. Although subsequent versions of lex and flex have faithfully preserved `yywrap`, in 30 years I have never seen a use of `yywrap` that wouldn’t be better handled by flex’s other I/O management features. In practice, everyone used the default `yywrap` from the flex library, which always returns 1, or put a one-line equivalent in their programs. Modern versions of flex let you say `%option noyywrap` at the top of your scanner to tell it not to call `yywrap`, and from here on, we’ll always do that.

- yylineno keeps track of the no. of lines (set yylineno=1 before yylex())
- nodefault tells it not to add a default rule in case input does not match any pattern (not needed).

Definition Section

- You can declare named subpatterns (names for corresponding subpatterns) in the declaration section.
- Then in that way you can use the name (in curly braces) instead of the subpattern while you are writing patterns in the rules section.
    - Example: Letter in wordcount.l

Rules Section

- In the rules section you can declare regular expressions to match complex patterns in the input file
  - Detect lexeme and generate token in the action section
- Each pattern must start at the beginning of the line (no spaces).
- The pattern is followed by action section (one or more lines of C++ code in curly braces) that tells it what to do when a pattern is matched
  - The matched expression from the input file is stored in the **yytext** variable.
  - You can also use the variables declared in the declaration section (named subpatterns in curly braces in the regex part, and declared C++ variables declared in the code part).
  - You can also declare C++ variables inside the action part too.
- Each call of yylex() continues to match inputs until it returns from one of the action codes corresponding to a pattern matched
  - You will not be returning anything from the action codes in this assignment, so it parses through the whole file.

Special Characters

The characters with special meaning in regular expressions are:

.

Matches any single character except the newline character (`\n`).

[]

A *character class* that matches any character within the brackets. If the first character is a circumflex (^), it changes the meaning to match any character *except* the ones within the brackets. A dash inside the square brackets indicates a character range; for example, `[0-9]` means the same thing as `[0123456789]` and `[a-z]` means any lowercase letter. A - or] as the first character after the [is interpreted literally to let you include dashes and square brackets in character classes. POSIX introduced other special square bracket constructs that are useful when handling non-English alphabets, described later in this chapter. Other metacharacters do not have any special meaning within square brackets except that C escape sequences starting with \ are recognized. Character ranges are interpreted relative to the character coding in use, so the range `[A-z]` with ASCII character coding would match all uppercase and lowercase letters, as well as six punctuation characters

Special Characters

whose codes fall between the code for Z and the code for a. In practice, useful ranges are ranges of digits, of uppercase letters, or of lowercase letters.

[a-z]{-}[jv]

A differenced character class, with the characters in the first class omitting the characters in the second class (only in recent versions of flex).

- [a-zA-Z]
- [^\t\v]

Special Characters

\

Used to escape metacharacters and as part of the usual C escape sequences; for example, `\n` is a newline character, while `\*` is a literal asterisk.

*

Matches zero or more copies of the preceding expression. For example, `[ \t]*` is a common pattern to match optional spaces and tabs, that is, whitespace, which matches " ", " <tab><tab>", or an empty string.

+

Matches one or more occurrences of the preceding regular expression. For example, `[0-9]+` matches strings of digits such as `1`, `111`, or `123456` but not an empty string.

?

Matches zero or one occurrence of the preceding regular expression. For example, `-?[0-9]+` matches a signed number including an optional leading minus sign.

Special Characters

^

Matches the beginning of a line as the first character of a regular expression. Also used for negation within square brackets.

$

Matches the end of a line as the last character of a regular expression.

{}

If the braces contain one or two numbers, indicate the minimum and maximum number of times the previous pattern can match. For example, `A{1,3}` matches one to three occurrences of the letter *A*, and `0{5}` matches 00000. If the braces contain a name, they refer to a named pattern by that name.

- ^r : An r, but only at the beginning of a line (i.e., which just starting to scan, or right after a newline has been scanned).
- r$: an r, but only at the end of a line (i.e., just before a newline). Equivalent to "r/\n"
- {Letter}+
  - Letter [a-zA-Z] declared in declaration section

Special Characters

|

The *alternation* operator; matches either the preceding regular expression or the following regular expression. For example, `faith|hope|charity` matches any of the three virtues.

"..."

Anything within the quotation marks is treated literally. Metacharacters other than C escape sequences lose their meaning. As a matter of style, it's good practice to quote any punctuation characters intended to be matched literally.

()

Groups a series of regular expressions together into a new regular expression. For example, `(01)` matches the character sequence 01, and `a(bc|de)` matches *abc* or *ade*. Parentheses are useful when building up complex patterns with *, +, ?, and |.

- "[xyz]\"foo" : the literal string [xyz]“foo

Special Characters

/

Trailing context, which means to match the regular expression preceding the slash but only if followed by the regular expression after the slash. For example, `0/1` matches `0` in the string `01` but would not match anything in the string `0` or `02`. The material matched by the pattern following the slash is not "consumed" and remains to be turned into subsequent tokens. Only one slash is permitted per pattern.

The repetition operators affect the smallest preceding expression, so `abc+` matches *ab* followed by one or more *c*'s. Use parentheses freely to be sure your expressions match what you want, such as `(abc)+` to match one or more repetitions of *abc*.

<>

A name or list of names in angle brackets at the beginning of a pattern makes that pattern apply only in the given start states.

<<*EOF*>>

The special pattern `<<EOF>>` matches the end of file.

Ambiguous Patterns

How Flex Handles Ambiguous Patterns

Most flex programs are quite ambiguous, with multiple patterns that can match the same input. Flex resolves the ambiguity with two simple rules:

- Match the longest possible string every time the scanner matches input.
- In the case of a tie, use the pattern that appears first in the program.

These turn out to do the right thing in the vast majority of cases. Consider this snippet from a scanner for C source code:

```
"+"                 { return ADD; }
"="                 { return ASSIGN; }
"+="                { return ASSIGNADD; }

"if"                { return KEYWORDIF; }
"else"              { return KEYWORDELSE; }
[a-zA-Z_][a-zA-Z0-9_]*  { return IDENTIFIER; }
```

For the first three patterns, the string += is matched as one token, since += is longer than +. For the last three patterns, so long as the patterns for keywords precede the pattern that matches an identifier, the scanner will match keywords correctly.

Start States

- Used when you are trying to match certain patterns only under certain contexts.
- Controls which patterns can be matched when.
- At any point, the scanner is in one start state and can match patterns active in that start state only.
- You can define as many start states as needed.
- Example: state.l
  - %x MYSTATE defines a start state
- %x means an ***exclusive*** start state, which means that when that state is active, only patterns specifically marked with the state can match.
- We can also define ***inclusive*** start states with %s, which means patterns not marked with any state can also match

Start States

- Flex itself defines the **INITIAL** state, the state in which you are initially.
- To switch to a different state in the action code, you must use the macro **BEGIN**
  - BEGIN MYSTATE
- Also make sure to return back to INITIAL state once your state related works are done (BEGIN INITIAL).
- Patterns are tagged with start-state names in angle brackets to indicate in which state(s) the pattern is active.
- The nameless patterns are active in the **INITIAL** state and any other inclusive state.

Assignment : Important Notes

- Integrating the symbol table
  - Use your symbol table program of the first assignment.
  - Include as a header file (.cpp or .h) in the definition section.
  - Remove all the console I/O and file I/O done in the first assignment, remove its main function too.
  - You can write code in it to output in the log file (discussed later) .
  - Declare a symbol-table object to use in the action codes
- Declare other C functions and variables you need to use in the %{ … %} part of the declaration section
- Declare start states and any named subpatterns you need in the declaration section.

Assignment : Important Notes

- Be careful about the order in which you write the rules
  - Remember how flex handles ambiguity
- While generating patterns , beware you regular expression
  - Should capture **all** input it is intended to catch
  - Should **not** capture any input it is not intended to catch.
- Character Literals:
  - Suppose you are reading the character literal ‘\n’
  - You will read ‘ , then \ and then n. (will not read *enter).* But as a token you have to say *enter* , again in the log file you have to print \ and n (not *enter*).
  - Also be careful about ‘\\’
  - While detecting multi-character literal error, make sure special characters like ‘\n’, ‘\\’ etc. don’t get interpreted as an error.

Assignment : Important Notes

- String literals

```
6   char str1[100] = "I own a dog.";
7   char str2[100] = "I own a dog.\
8   His name is Jack.";
9   char str3[100] = "He said, \" I own a dog\"\
10  so I asked, \"What is his name?\"";
```

Assignment : Important Notes

```
6     char str1[100] = "I own a dog.";
7     char str2[100] = "I own a dog.\
8     His name is Jack.";
9     char str3[100] = "He said, \" I own a dog\"\
10    so I asked, \"What is his name?\"";
```

- Special characters like ‘\a’ , ‘\b’ can also be inside the strings.
- Carefully distinguish between errors and what is correct, also properly extract and print the lexeme in the log file and token attribute in the token file (sample 1).
- Ensure the scanning continues to run properly even after detecting such complex strings.
- Depending on the C compiler in use and the OS, newlines could be ‘\n’ or ‘\r\n’. Handle those properly.

Assignment : Important Notes

- Single line comments: anything after \ prevents the next line from being commented.

```
12      //This is a single line comment.
13      //This is also a\
14      single line comment.
15      //This is also a /*Multi-line just for\
16      fun */ single line comment.
17      //This is another single line comment\
18      char str4[100] = "But this line is valid.";
19      //This is another /*Multi-line again\
20      char str5[100] = "This line is also valid.";
21      //Tony said, \"This is still easy.\"
22      char str6[100] = "Really Tony?";
```

Assignment : Important Notes

- Catch errors
  - 12.1ef
  - 12.efg
  - 12abcd
  - ‘\n
- Keep track of line count and error count.
- Your program should take a text file named ‘input.txt’ as input.

References

- Flex & Bison by John Levine.
  - Book provided in Moodle.
  - Chapter 1 & 2
- https://www.cs.princeton.edu/~appel/modern/c/software/flex/flex.html