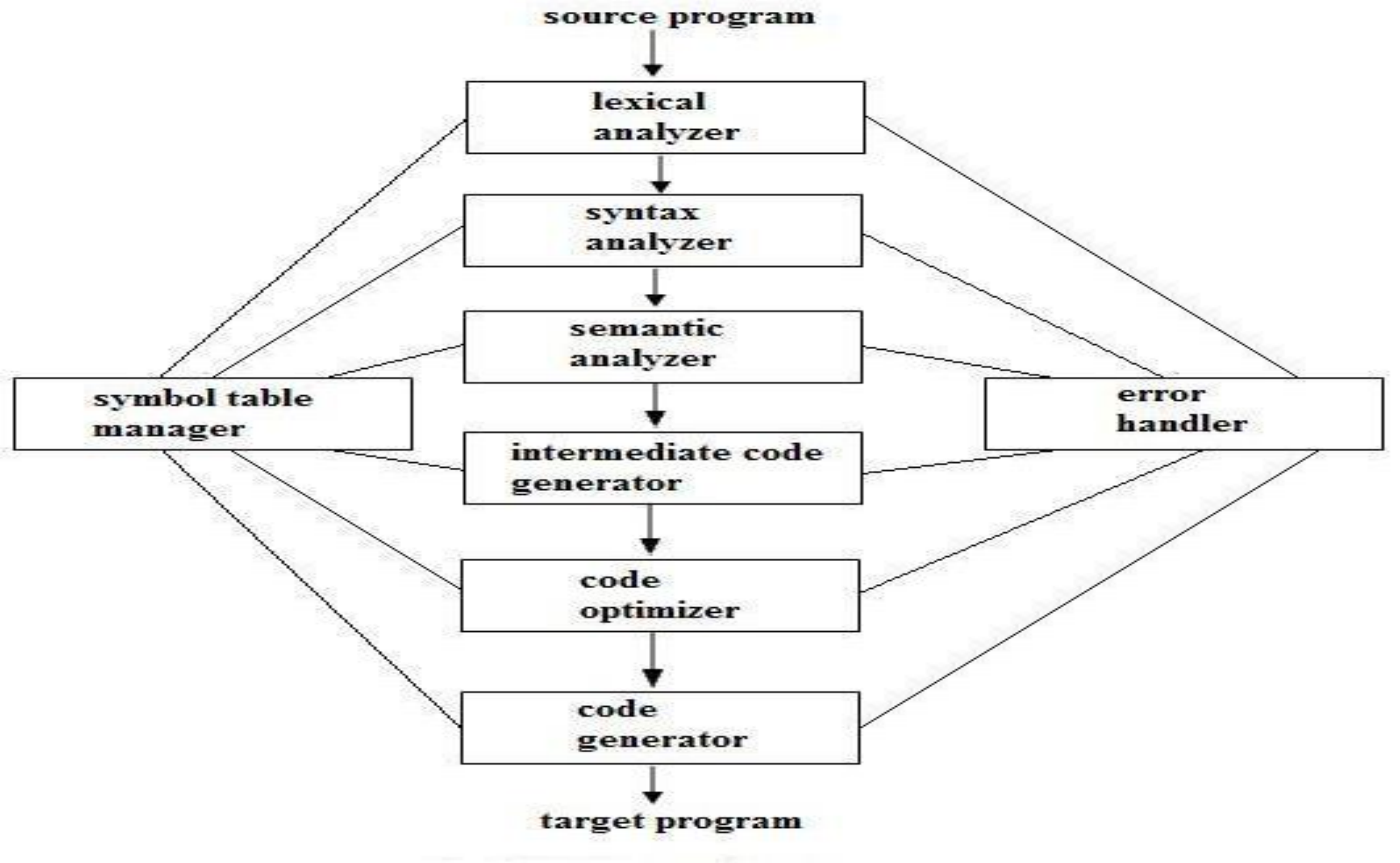
Compiler Tools

lex & yacc

Agenda

- Compiler Design Process
- Lex/Flex - Yacc/Bison Information
- lex
  - Definition Section
  - Rule - Expressions
  - User subroutines
  - Examples 1, 2, 3, 4
  - Exercises

Compiler Overview



Lexer/Scanner

- Lexical Analysis
  - process of converting a sequence of characters into a sequence of tokens.

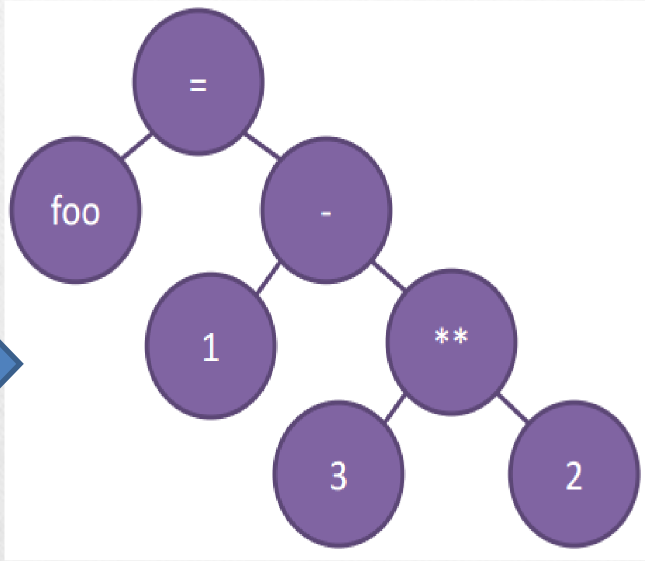
Example: foo = 1 - 3**2

Lexeme	**Token Type**
foo	Variable
=	Assignment Operator
1	Number
-	Subtraction Operator
**	Power Operator
2	Number

Parser

- Syntactic Analysis
  - The process of analyzing a sequence of tokens to determine its grammatical structure.
  - Syntax errors are identified during this stage.

Lexeme	**Token Type**
foo	Variable
=	Assignment Operator
1	Number
-	Subtraction Operator
**	Power Operator
2	Number



Semantic Analyzer

- Semantic Analysis
  - The process of performing semantic checks.
  - E.g. type checking, object binding, etc.

Code:

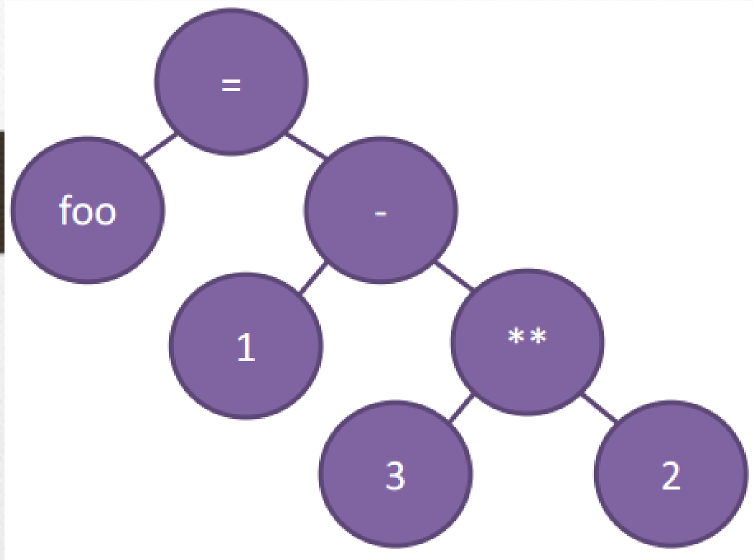
```
float a = "example";
```

Semantic Check Error:

```
error: incompatible types
in initialization
```

Intermediate Code Generator

- Generates intermediate code from annotated graph



```
t1 = 3**2
t2 = 1-t1
foo =t2
```

Optimizer(s)

- Compiler Optimizations
  - –– tune the output of a compiler to minimize or maximize some attributes of an executable computer program.
  - –Make programs faster, etc…

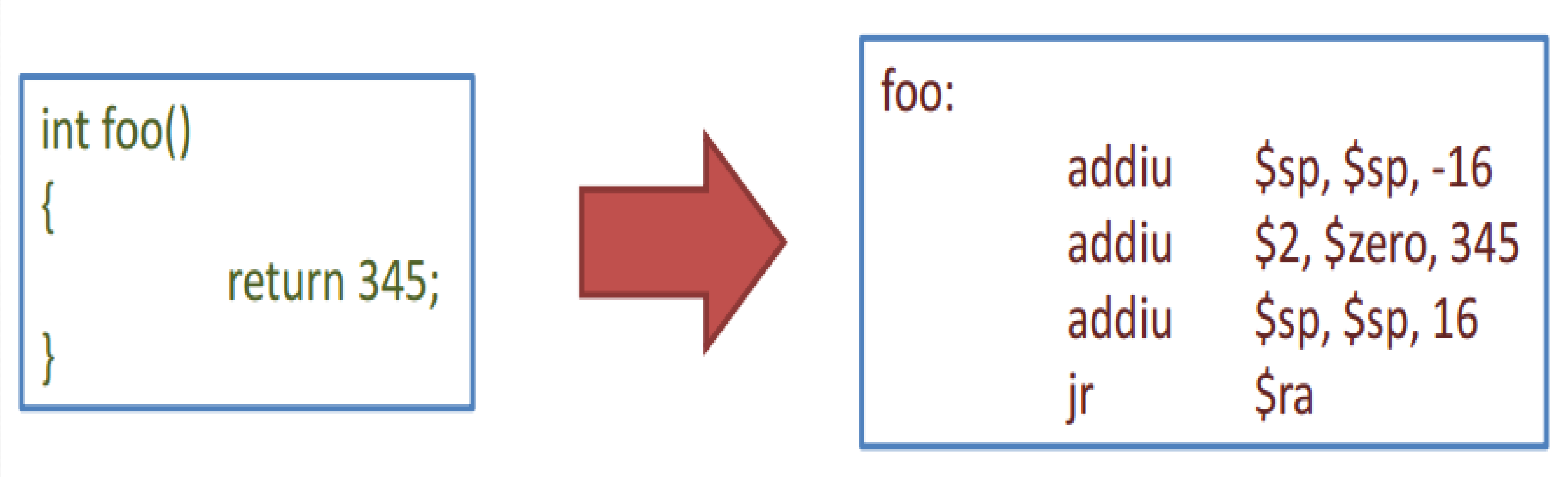
```
t1 = 3**2
t2 = 1-t1
foo =t2
```

→

```
t1 = 3**2
foo = 1-t1
```

Code Generator

- Code Generation
  - process by which a compiler's code generator converts some intermediate representation of source code into a form (e.g., machine code) that can be readily executed by a machine.

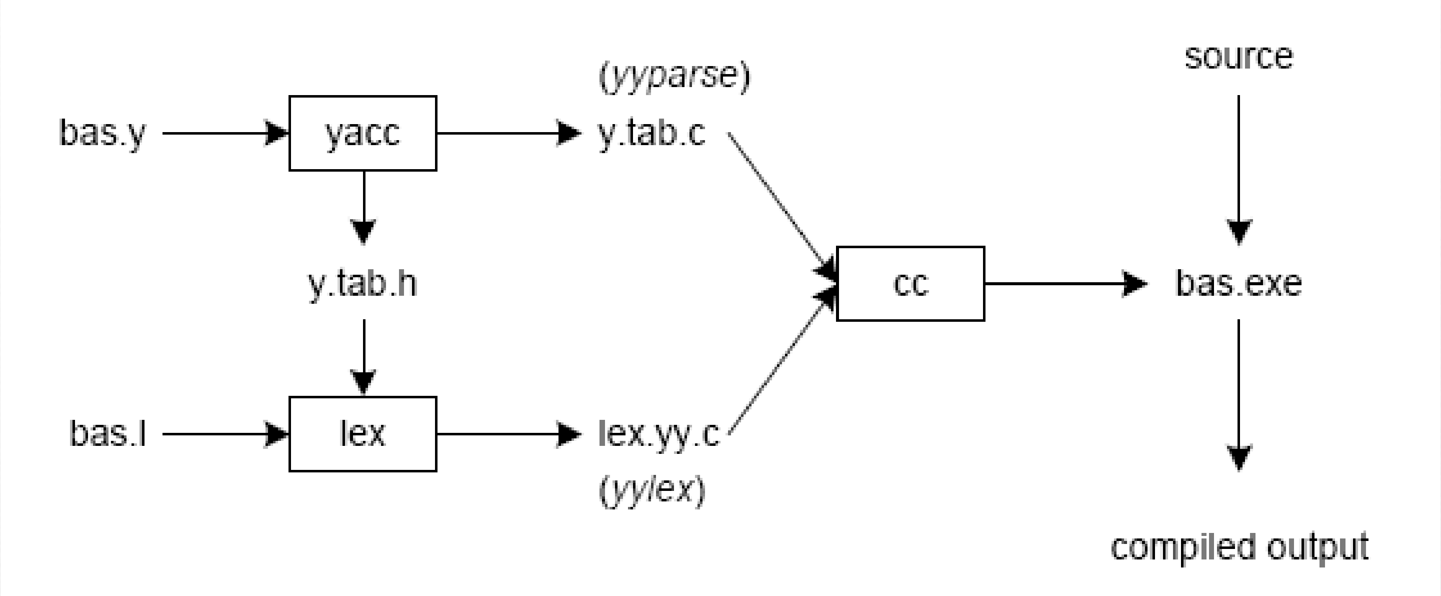


```
int foo()
{
        return 345;
}
```

```
foo:
        addiu   $sp, $sp, -16
        addiu   $2, $zero, 345
        addiu   $sp, $sp, 16
        jr      $ra
```

Lex & Yacc

- Lex
  - –generates C code for the lexical analyzer (scanner)
  - –Token patterns specified by regular expressions
- Yacc
  - –generates C code for a LR(1) syntax analyzer (parser)
  - –BNF rules for the grammar

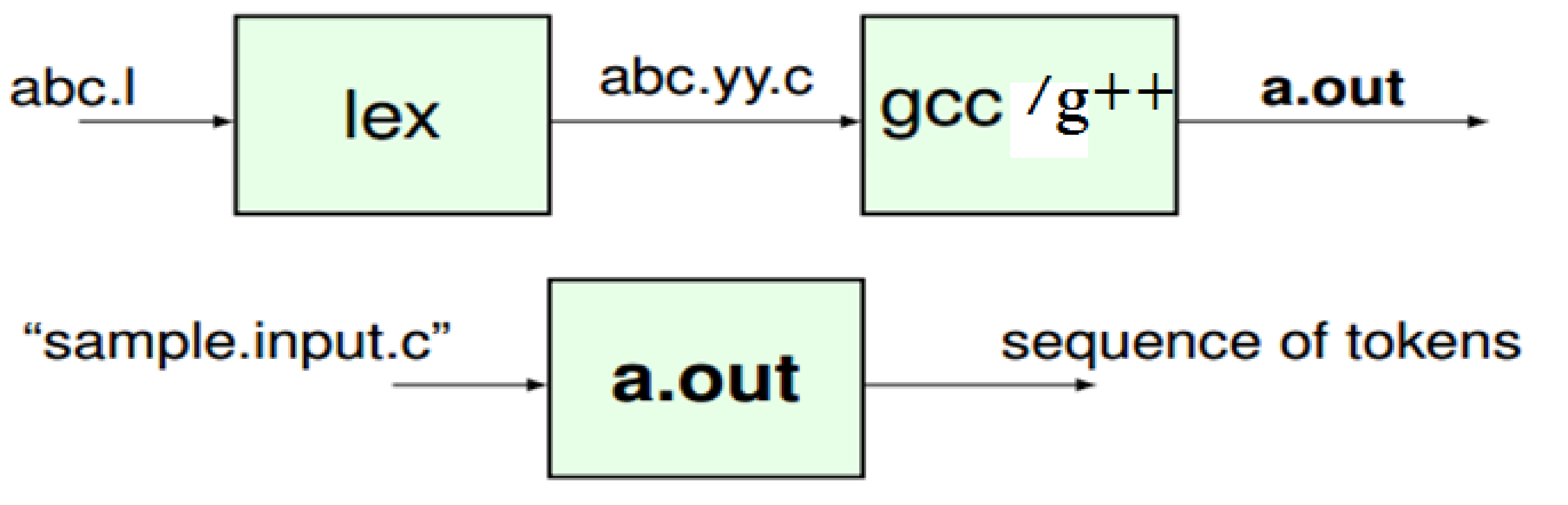


Lex

- Lex is a tool for generating scanners.
- Scanners are programs that recognize Lexical patterns in text.
- These lexical patterns (or regular expressions) are defined in a particular syntax.
- A matched regular expression may have an associated action.
  - This action may also include returning a token.
- If, no regular expression matches the pattern , further processing stops and Lex displays an error message.

Lex

- Lex and C are tightly coupled. A .lex file (files in Lex have the extension. Lex or .l) is passed through the lex utility, and produces output files in C.
- These file(s) are compiled to produce an executable version of the lexical analyzer.



A flex Input File

- Flex input files are structured as follows:

Definitions

%%

Rules

%%

User subroutines

Definition Section

- Defs, Constants, Types, #includes, etc. that can Occur in a C Program

```
%{
/* This is a comment inside the definition
*/
#include <math.h> // may need headers
#include <stdio.h> // for printf in BB
#include <stdlib.h> // for exit(0) in BB
%}
```

- Regular Definitions (expressions)
  - –name definition
    - DIGIT [0-9], ID [a-z][a-z0-9]*
  - –A subsequent reference to {DIGIT}+"."{DIGIT}* is identical to:

([0-9])+"."([0-9])*

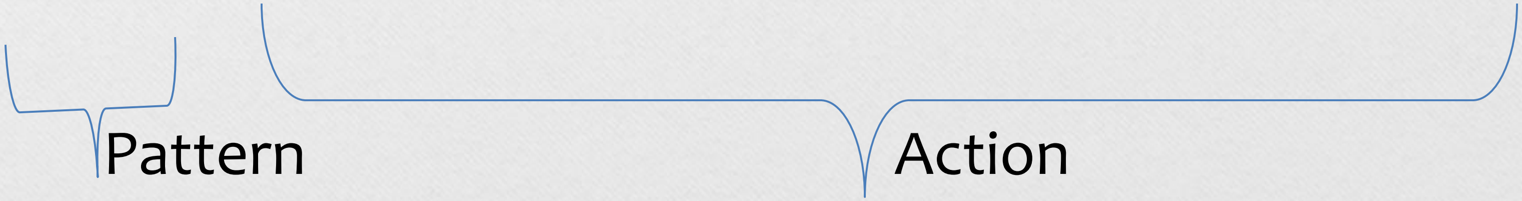
Rule

- The required **Rules** section is where you specify the patterns that identify your tokens and the action to perform upon recognizing each token.
- A rule has a regular expression (called the pattern) and an associated set of C statements in the form:

pattern action

  - The idea is that whenever the scanner reads an input sequence that matches a pattern, it executes the action to process it.

- Example
  - [0-9]+ { printf("Found an integer: %d“, yytext); }



Pattern

Action

Lex Regular Expressions

. matches a single character

[ ] matches any of the characters in the brackets. E.g, [abc] matches a, b or c

- represents intervening characters, so [0-9] matches any digit

+ means "one or more of the preceeding item", so [0-9]+ matches any integer

* means "zero or more of the preceeding item"

? means the preceeding items is optional

| means "or"

() group things

{} can be used to surround a name

Regular Expression	Meaning
[0-9]+("."[0-9]+)?	matches one or digits followed (optionally, hence the final ?) by a "." and one or digits
x	match the character ‘x’
rs	the regular expression r followed by the regular expression s;
r|s	either an r or an s
(r)	match an r; parentheses are used to provide grouping.
r*	zero or more r’s, where r is any regular expression r+ one or more r’s
[xyz]	a “character class”; in this case, the pattern matches either an ’x’, a ’y’, or a ‘z’.
[abj-oZ]	a “character class” with a range in it; matches an ‘a’, a ‘b’, any letter from ‘j’ through ‘o’, or a ‘Z’.
{name}	the expansion of the “name” definition.
"[+xyz]+\"+foo”	the literal string: [xyz]"foo

User subroutines

•Sections are used for ordinary designer defined C code that you want copied verbatim to the generated C file.

–subroutines are copied to the bottom of the file.

–E.g.

```
main() {
    yylex();
}
```

• If you do not need to include your own custom main() in your code, put the following line at the beginning of your lex specification file:

```
%option main
```

• This makes sure that a default main() and yywrap() are created for you.

Examples

1.Write a lex program that identifies the following tokens CS, Compiler, , and .(dot)

2.Write a lex program which identifies whether the given number is even or add

3.Write a lex program that identifies floating-point numbers, integers and strings

4.Write a lex program that identifies whether a token is identifier or number (named Expressions)

5.Write lex program that counts the number of characters, words and lines

Example2

6. Write a lex program that identifies the consonants and vowels and count its number
7. Write a lex program which identifies whether the given string is upper case or lower case
8. Write a lex program that checks valid email
1.Write a lex program that identifies whether a token is identifier or number (named Expressions)
2.Write lex program that counts the number of characters, words and lines

Example1

```
%{
#include<stdio.h>
%}
%%
cs              printf("Computer science third Year
students\n");
compiler        printf("compiler For Cs third.\n");
DMU                  printf(“Debre Marko University");
\.              printf("Dote operator");
%%
main() {
     yylex();
}
int yywrap(){
     return 1;
     }
```

How to compile and run lex files

- Use any editor to write lex file
  - Save it with extension filename.l
- Process the flex file
  - lex filename.l
  - a c/c++ file named lex.yy.c is created
- Compile the newly created file
  - gcc lex.yy.c ….for C
  - g++ lex.yy.c … for C++
- Run
  - a.exe

Example 2

```
%{
#include <iostream>
using namespace std;
%}
%%
[ \n]      ;
[0-9]+\.[0-9]+      { cout << "Found a floating-point number:" << yytext
<< endl; }
[0-9]+          { cout << "Found an integer:" << yytext << endl; }
[a-zA-Z0-9]+        { cout << "Found a string: " << yytext << endl; }
[\t]            { return 0;}
%%
main() {
        cout<<"Press tab key to end running program";
        yylex();
}
int yywrap(){
        return 1;
        }
```

Example 3

```
%{
#include<stdio.h>
%}
NUM [0-9]+
ID [a-zA-Z][a-zA-Z0-9]*
ERROR [0-9][a-zA-Z0-9]*
%%
{NUM}       printf("Number"); //[0123456789]+
{ID}              printf("Identifier - %s", yytext);
{ERROR}       printf("Error\n");
[\t]       return 0;
%%
main() {
        yylex();
}
int yywrap(){
        return 1;
        }
```

Example 4

```
%{
#include<string.h>
int chars = 0,  words = 0, lines = 0;
%}
%%
\"              { return 0; }
[a-zA-Z]+   { words++; chars += strlen(yytext); }
\n              { chars++; lines++; }
.               { chars++; }
%%
main(int argc, char **argv)
{
yylex();
printf("Number of lines = %8d\nNumber of words =
%8d\nNumber of chars = %8d\n", lines, words, chars);
}
int yywrap(){  return 1; }
```

Exercises

1.Write a lexical analyzer that takes your name as input and display some message like "You write your Name"

1.Modify the first question to display "Your Name is Actual Name "

2.Scanner that replaces all numbers in a stream of text with a question mark.

3.Write the lexical analyzer for the following tokens

if ,for, else, int , float, string, while, do, break, switch, char, double … .

Ex

int

This is key word

Well come

This is string literal

3. Write the lexical analyzer that checks whether the input is operator or not.

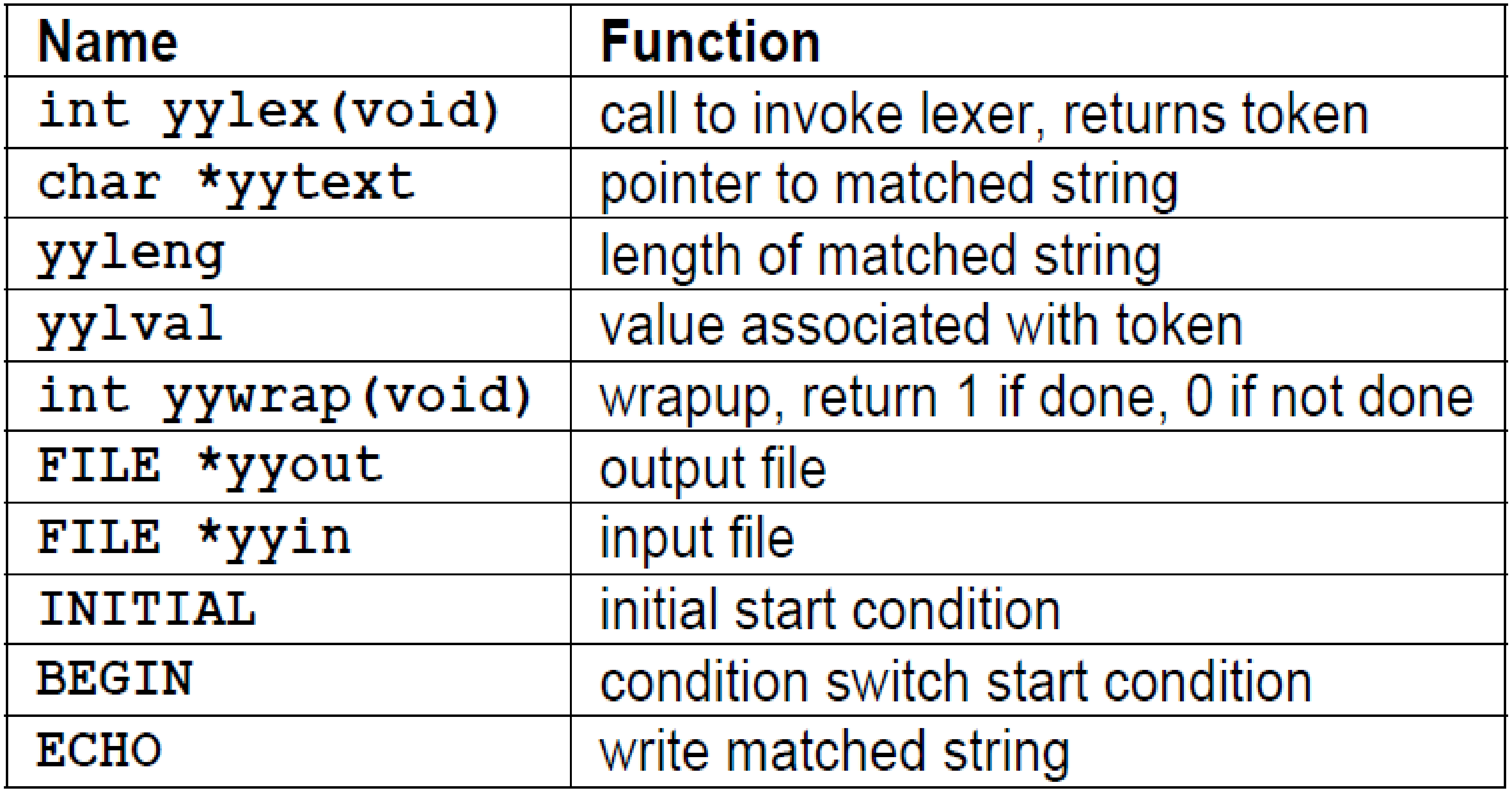
{ ----open Brace

= ----- equal to

etc.

flex Global Variables

- Holds more information needed about the token just read



Name	Function
`int yylex(void)`	call to invoke lexer, returns token
`char *yytext`	pointer to matched string
`yyleng`	length of matched string
`yylval`	value associated with token
`int yywrap(void)`	wrapup, return 1 if done, 0 if not done
`FILE *yyout`	output file
`FILE *yyin`	input file
`INITIAL`	initial start condition
`BEGIN`	condition switch start condition
`ECHO`	write matched string

Reading from a file

- To identify tokens and take some action most of the time you'd really like to pick a file to read from
- Flex reads its input from a global pointer to a C FILE variable called **yyin**, which is set to STDIN by default.
- All you have to do is set that pointer to your own file handle, and it'll read from it instead.
- Look at the example on next page

```
%{
#include <iostream>
using namespace std;
#define YY_DECL extern "C" int yylex()
%}
%%
[0-9]+\.[0-9]+   { cout << "FP number:" << yytext << endl; }
[0-9]+          { cout << “integer:" << yytext << en
[a-zA-Z0-9]+    { cout << " string: " << yytext <<
[\t]            { return 0; }
.          ;
%%
main() {
        // open a file handle to a particular file:
        FILE *myfile = fopen("example1.l", "r");
        if (!myfile) { // make sure it's valid:
         cout << "I can't open a.snazzle.file!" << endl;
        return -1;
        }
```

```
// set lex to read from it instead of
defaulting to STDIN:
        yyin = myfile;
        yylex();    // lex through the input:
}
int yywrap(){   return 1; }
```

Exercises…

1. Verify if a password is acceptable - A password is acceptable if it satisfies all of the following criteria.
    - –A password should contain at least one upper case and one lower case letter and one digit.
    - – A password should be at least 8 characters long.
    - –A password should contain no white space.
    - –Whitespace at the beginning and end of a password is ignored.
2. Write a flex code that takes c++ source code and calculates number of keywords, identifiers, and operators