PROJECT REPORT

ON

COMPILER DESIGN

COURSE NAME: COMPILER DESIGN LABORATORY

COURSE NO: CSE 3212

DATE OF SUBMISSION: 15.06.2021

SUBMITTED TO

DOLA DAS

LECTURER

AHSAN HABIB NAYAN

LECTURER

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

KHULNA UNIVERSITY OF ENGINEERING & TECHNOLOGY

SUBMITTED BY

ASNUVA TANVIN

ROLL: 1707005

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

KHULNA UNIVERSITY OF ENGINEERING & TECHNOLOGY

FLEX

FLEX (Fast Lexical analyzer generator) is a tool for generating scanners. Lexical analysis is the first phase of a compiler. 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 whitespace or comments in the source code. Scanners perform lexical analysis by dividing the input into meaningful units. For a C program the units are *variables*, *constants, keywords, operators, punctuation* etc. These units also called as tokens.

BISON

Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Bison is used to perform semantic analysis in a compiler. Bison is a general-purpose parser generator that converts a grammar description for an LALR(1) context-free grammar into a C program to parse that grammar. Parsing involves finding the relationship between input tokens. Bison is upward compatible with Yacc: all properly-written Yacc grammars ought to work with Bison with no change. Interfaces with scanner generated by Flex. Scanner called as a subroutine when parser needs the next token.

Flex and Bison are aging Unix utilities that help to write very fast parsers for almost arbitrary file formats. Flex and Bison will generate a parser that is virtually guaranteed to be faster than anything that could be write manually in a reasonable amount of time. Second, updating and fixing Flex and Bison source files is a lot easier than updating and fixing custom parser code. Third, Flex and Bison have mechanisms for error handling and recovery, finally Flex and Bison have been around for a long time, so they far freer from bugs than newer code.

Compiler with Flex and Bison

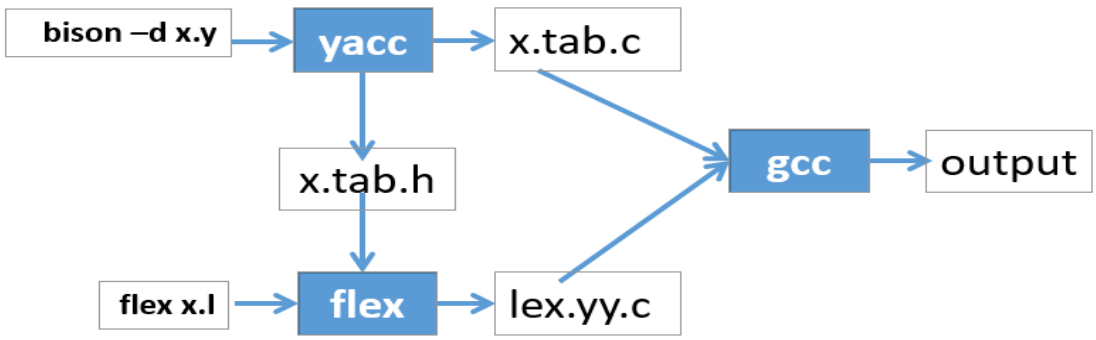

Fig: A diagram of how a compiler build with flex and bison works

Project Description:

DATA TYPES:

1. **Integer:**
   Range: -2,147,483,648 to 2,147,483,647.
   Tokens:
   - INT: Returned if regular expression for detecting integers match an expression
   - INTT: Returned if "int" found for declaring an integer

   Syntax: int a
   int a=10
2. **Float:**
   Range: 1.2E-38 to 3.4E+38
   Tokens:
   - FLOAT: Returned if regular expression for detecting floating point numbers match an expression
   - FL: Returned if "float" found for declaring a floating point number

   Syntax: float b

   float b=5.5
3. **String:**
   Range: a-z (small letters), A-Z (capital letters), 0-9 (digits) and symbols (: _ " ")
   Tokens:
   - STRING: Returned if regular expression for detecting strings match an expression
   - STT: Returned if "string" found for declaring a string

   Syntax: string str

   string st = "CSE"

VARIABLES:

Range: a-z (small letters), A-Z (capital letters), 0-9 (digits). A variable name has to start with a small or a capital letter and can only be 100 characters long.

Type: A variable can contain either an integer value or a floating point number or a string.

Token:

- ID: Returned if regular expression for detecting a variable name matched an expression.

Syntax: int val

float num

string str

OPERATORS:

Token: *yytext is returned for all operators.

Suppose a and b are two integers:

Operators	Data Type	Type	Description	Syntax
+	Integer Float	Arithmetic	Adds two operands.	a+b
-	Integer Float	Arithmetic	Subtracts second operand from the first.	a-b
*	Integer Float	Arithmetic	Multiplies both operands.	a*b
/	Integer Float	Arithmetic	Divides numerator by de-numerator.	a/b
%	Integer	Arithmetic	Modulus Operator and remainder of after an integer division.	a%b
++	Integer	Arithmetic	Increment operator increases the integer value by one.	a++
--	Integer	Arithmetic	Decrement operator decreases the integer value by one.	b--
<=	Integer Float	Relational	True if the value of left operand is greater than or equal to the value of right operand.	(a<=b)
>=	Integer Float	Relational	True if the value of left operand is less than or equal to the value of right operand.	(a>=b)
>	Integer Float	Relational	True if the value of left operand is greater than the value of right operand.	(a>b)
<	Integer Float	Relational	True if the value of left operand is less than the value of right operand.	(a<b)

==	Integer Float	Relational	True if the values of two operands are equal.	(a==b)
!=	Integer Float	Relational	True if the values of two operands are not equal	(a!=b)

CONDITIONAL STATEMENTS;

1. IF-ELIF-ELSE

Tokens:

- IF: Returned when "if" statement is found.
- ELIF: Returned when "elif" statement is found.
- ELSE: Returned when "else" statement is found.

Syntax:

IF

```
if(condition) {
 any number of statements
}
```

IF-ELSE

```
if(condition) {
any number of statements
}
else {
any number of statements
}
```

IF-ELIF-ELSE

```
if(condition) {
any number of statements
}
elif(condition) {
any number of statements
}
elif(condition) {
any number of statements
}

else {
any number of statements
}
```

NESTED IF-ELIF-ELSE

```
if(condition) {
      if(condition) {
           any number of statements
      }
      else {
           any number of statements
      }
}
elif(condition) {
      if(condition) {
           any number of statements
      }
     elif(condition) {
           any number of statements
      }
     else {
           any number of statements
      }
 }
else {
       if(condition) {
            any number of statements
       }
}
```

2. **SWITCH:**

Tokens:

- SW: Returned when “switch” statement is found.
- CA: Returned when “case” statement is found.
- DF: Returned when “default” statement is found.

Syntax:

switch (expression):

case expression: statement

…….

……..

……..

case expression: statement

default: statement

LOOPS:

1. **FOR LOOP:**

Tokens:

- FOR: Returned when “for” statement is found.

Syntax:

for (expr1, expr2, expr3) {

Any number of statements

}

Suppose i is a loop control variable.

expr1: initial value of loop control variable (i = expr1)

expr2: upper bound of the loop control variable (i < expr2)

expr3: the value by which loop control variables will increment (i = i + expr3)

2. **WHILE LOOP:**

Tokens:

- WHILE: Returned if “while” statement is found.

Syntax:

while(condition) {

Any number of statements

}

3. **DO-WHILE LOOP:**
   Tokens:
   - DO: Returned when “do” statement is found.
   - WHILE: Returned when “while” statement is found.

   Syntax:

   do {

   Any number of statement

   }while (condition)

ARRAYS:

Arrays can only be of integer type.

1. **DECLARATION:**
   Tokens:
   - AN: Returned when a statement matches the regular expression to identify an array name. An array name starts with ‘@’ followed by any small/capital letters or digits.

   Syntax:

   int array_name[array_size]

2. **ASSIGNMENT:**
   Array values have to be assigned manually.
   Syntax:
   <array_name, array_index>/*space*/ =/*space*/ expression
   <array_name, array_index>/*space*/ =/*space*/ Array Operation Result

3. **OPERATIONS:**
   Addition, subtraction, multiplication, division and remainder operation can be done on array values. The result can be saved into array variables or normal variables.
   Syntax:

   a =/*space*/<array_name, array_index1> + <array_name, array_index2>
   <array_name, array_index1>/*space*/=/*space*/<array_name, array_index2> + <array_name, array_index3>

FUNCTION (MAIN AND USER DEFINED):

Syntax:

return-type function-name (any number of parameters):

any number of statements

return expr

Return-type: Integer, float, string, void. Return if any expression matches tokens consecutively INTT, FL, STT, VOID.

Function-name: Same as variable name. Token is ID which is returned if any expression matches the regular expression to identify a variable name.

Parameters: Syntax is same as variable declaration (int a/float b/string c). A function may not have any parameters.

Return: The value that the function returns.

Syntax:

return a

Token:

- RETURN: Returned if "return" statement is found.

Return variable can be an integer, a floating point number, a string. Void type functions return nothing.

FUNCTION CALL:

Syntax:

function-name (any number of parameters)

Function-name: Same as variable name. Token is ID which is returned if any expression matches the regular expression to identify a variable name.

Parameters: Syntax is same as variable declaration (int a/float b/string c). A function may not have any parameters.

BUILT IN FUNCTIONS:

FUNCTION NAME	SYNTAX	TOKEN	INPUT	OUTPUT
Print Integer	pfi(expr1)	PFI	An integer	Prints the input and a new line.
Print Floating point number	pff(expr1)	PFF	A floating point number	Prints the input and a new line.

Print string	pfs(expr1)	PFS	A string	Prints the input.
Print Array Value	pfa(array_expression)	PFA	An array variable	Prints the value of the variable and a new line.
Integer Input	inpi()	INPI	Takes an integer as input from user	No output
Float Input	inpf()	INPF	Takes a floating point number as input from user	No output
Sine function	sin(expr)	SIN	The floating point value of the angle in degrees.	The sine value of the angle in degrees.
Cosine function	cos(expr)	COS	The floating point value of the angle in degrees.	The cosine value of the angle in degrees.
Tangent function	tan(expr)	TAN	The floating point value of the angle in degrees.	The tangent value of the angle in degrees.
Logarithm	ln(expr)	LN	A floating point value	The e based logarithm value of the input.
Power function	pow(expr1,expr2)	POW	Two floating point values	Returns expr1 raised to the power expr2.
Minimum	min(expr1,expr2)	MIN	Two integers	Returns the minimum value of the two inputs.
Maximum	max(expr1,expr2)	MAX	Two integers	Returns the maximum value of the two inputs.
Floor function	floor(expr1)	FLOOR	A floating point value	Returns the floor value of input in integer form

Ceil function	ceil(expr1)	CEIL	A floating point value	Returns the ceil value of input in integer form
Factorial Function	expr!	*yytext is returned for '!'	An integer(less than or equal to 20)	Returns the factorial of input.
GCD Function	gcd(expr1,expr2)	GCD	Two integers	Returns the gcd of the two Inputs.
Prime function	prime(expr)	PRIME	An integer	Checks if input is prime or not. Prints answer.
String Length Function	len(expr)	LEN	A string	Returns the length of the string
String Compare Function	cmp(expr1,expr2)	CMP	Two strings	Prints the result.
String Copy Function	cpy(expr1,expr2)	CPY	Two string variables	Copies second string to the first string variable.
String Concatenation Function	cat(expr1,expr2)	CAT	Two string variables	Combines two strings and stores them in the first string variable.

END LINE:

Token:

- END: Returned when "endl" statement is found.

Syntax:

endl

Usage: Prints a new line whenever called.

HEADER:

Any C header file is accepted.

Syntax:

#include<file_name.h>

Token:

HEAD: Returned when a statement matches the regular expression for a header statement.

COMMENT:

Syntax:

//This is a comment

Token:

CMT: Returns when a statement matches the regular expression for a comment.

SYMBOLS:

Symbol	TOKEN
(	*yytext returned
)	*yytext returned
{	*yytext returned
}	*yytext returned
,	*yytext returned
:	COL returned
Blank Space	No action taken
New Line(\n)	No action taken
Tab(\t)	No action taken