

PROJECT REPORT

ON

**COMPILER DESIGN**



December 19, 2022

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING KHULNA UNIVERSITY OF ENGINEERING & TECHNOLOGY, KHULNA

Dipannita Biswas

Lecturer

Department of Computer Science and Engineering

Khulna University Engineering & Technology,Khulna

Dola Das

Assistant Professor

Department of Computer Science and Engineering

Khulna University Engineering & Technology,Khulna

## SUBMITTED BY

NAFIUL ALAM

ROLL: 1807005

COURSE NAME: COMPILER DESIGN LABORATORY

COURSE NO: CSE 3212

### 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

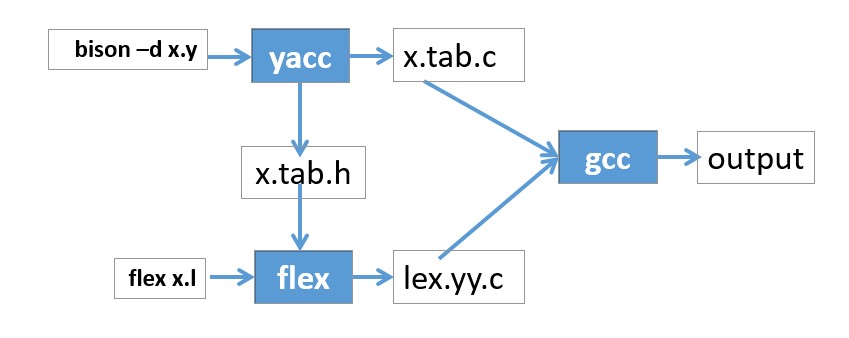


Fig: A diagram of how a compiler build with flex and bison works

**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 “inum” found for declaring an integer
  + Syntax: var a:inum

var a:inum << 10

1. **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 “fnum” found for declaring a floating point number
  + Syntax: var b:fnum

var b:fnum << 5.0

1. **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 “charry” found for declaring a string
  + Syntax: var str:charry

var str:charry << “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: var val:inum

var fval:fnum

var str:charry

**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 denumerator. | /(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. | -(a,-) |
| <= | 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 “fi” statement is found.
* ELIF: Returned when “fie” statement is found.
* ELSE: Returned when “fin” statement is found.

Syntax:

|  |  |
| --- | --- |
| ***IF***    fi[condition] {  any number of statements  }    ***IF-ELSE***    fi[condition] {  any number of statements  }  fin {  any number of statements  }    ***IF-ELIF-ELSE***    fi[condition]{  any number of statements  }  fie[condition] {  any number of statements  }  fie[condition] {  any number of statements  }    fin {  any number of statements  } | ***NESTED IF-ELIF-ELSE***    fi[condition] {  fi[condition] {  any number of statements  }  fin {  any number of statements  }  }  fie[condition] {  fi[condition] {  any number of statements  }  fie[condition] {  any number of statements  }  fin {  any number of statements  } }  fin {  fi[condition] {  any number of statements  }  } |

**2. SWITCH:**

Tokens:

* SW: Returned when “sinit” statement is found.  CA: Returned when “opt” statement is found.
* DF: Returned when “dd” statement is found.

Syntax:

sinit [expression]:

opt [expression]: statement

……………………….

……………………..

……………………..

opt [expression]: statement

dd: statement

**LOOPS:**

1. **FOR LOOP:**

Tokens:

* + FOR: Returned when “floop” statement is found.

Syntax:

floop [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))

1. **WHILE LOOP:**

Tokens:

* + WHILE: Returned if “wloop” statement is found.

Syntax:

wloop[condition] {

Any number of statements

}

1. **DO-WHILE LOOP:**

Tokens:

* + DO: Returned when “do” statement is found.
  + WHILE: Returned when “loop” statement is found.

Syntax: do {

Any number of statement

}loop [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:

var @array\_name:inum[array\_size]

1. **ASSIGNMENT:**

Array values have to be assigned manually.

Syntax:

{@array\_name, array\_index} << expression

**FUNCTION (USER DEFINED):**

Syntax:

return-type f->function-name (any number of parameters,):{

any number of statements

feedback expr

}

Return-type: inum, fnum, charry, !?. 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 (a:inum/b:fnum/c:charry). A function may not have any parameters.

Return: The value that the function returns.

Syntax: feedback a Token:

 RETURN: Returned if “feedback” statement is found.

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

**FUNCTION CALL:**

Syntax:

f->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 (a:inum/b:fnum/c:charry). A function may not have any parameters.

**BUILT IN FUNCTIONS:**

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Print string | outputs(expr1) | PFS | A string | Prints the input. |
| Integer Input | inputi() | INPI | Takes an integer as  input from user | No output |
| Float Input | inputf() | INPF | Takes a floating point number as  input from user | No output |
| Sine function | bf->SIN(expr) | SIN | The floating point value of the angle in degrees. | The sine value of the angle in degrees. |
| Cosine function | bf->COS(expr) | COS | The floating point value of the angle in degrees. | The cosine value of the angle in  degrees. |
| Tangent function | bf->TAN(expr) | TAN | The floating point value of the angle in degrees. | The tangent value of the angle in  degrees. |
| Logarithm | bf->LN(expr) | LN | A floating point value | The e based logarithm  value of the input. |
| Power function | bf->POW(expr1,expr2,) | POW | Two floating point values | Returns expr1 raised to the  power expr2. |
| Minimum | bf->MIN(expr1,expr2,) | MIN | Two integers | Returns the minimum  value of the two inputs. |
| Maximum | bf->MAX(expr1,expr2,) | MAX | Two integers | Returns the maximum  value of the two inputs. |
| Floor function | bf->FLOOR(expr1) | FLOOR | A floating point value | Returns the floor value of input in integer form |
| Ceil function | bf->CEIL(expr1) | CEIL | A floating point value | Returns the ceil value of input in integer form |
| GCD Function | bf->GCD(expr1,expr2,) | GCD | Two integers | Returns the gcd of the two Inputs. |
| Prime function | bf->PRIME(expr) | PRIME | An integer | Checks if input is prime or not. Prints answer. |
| String Length Function | bf->LEN(expr) | LEN | A string | Returns the length of the string |

**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:

Dep library.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 |