T-PROGRAMMING

Rauful Islam Tamim

KUET

Project :A simple Compiler Project(T-Programming)

CSE 3212:Compiler Design Laboratory

Submited By:

MD.RAUFUL ISLAM TAMIM

ROLL:2007009

Table of Contents

Objectives:

- To acquire knowledge about Lexical Analysis and Syntax Analysis in the context of compiler construction.

- To gain hands-on experience with FLEX (Fast Lexical Analysis Generator) for performing lexical analysis.
- To identify, categorize, and analyze tokens, keywords, variables, and other language components using FLEX.
- To design a customized language syntax and implement a syntax analyzer using BISON for parsing.

Introduction:

Lexical analysis and syntax analysis are fundamental phases in the process of compiler Design. Lexical analysis generates tokens from source code, whereas syntax analysis focuses on the hierarchical structure of the code using bison for ensuring syntax and semantics validity.

1. Tokenization: Lexical analysis, while reading a sentence and splitting it into individual words, involves breaking down the source code into tokens. Tokens are the smallest meaningful units, encompassing keywords, identifiers, operators, and literals.

2. Whitespace and Comments: Lexical analysis, carried out using FLEX, disregards whitespace and comments, as they hold no significance in most programming languages. The focus is on extracting meaningful code elements using regular expressions defined in FLEX.

3. Error Detection: FLEX aids in identifying and reporting errors, such as syntax errors, when encountering invalid or unexpected tokens. This process assists programmers in locating and rectifying issues in their code.

4. Symbol Table: Lexical analysis, facilitated by FLEX, often maintains a symbol table or dictionary, tracking identifiers like variable and function names. This information is utilized in subsequent phases of the compilation process.

5. FLEX and BISON Integration: FLEX is employed for lexical analysis, employing regular expressions to recognize and categorize tokens. BISON, on the other hand, is utilized for parsing the syntax of the language, ensuring adherence to predefined grammar rules. Regular expressions in FLEX are complemented by Bison's parsing capabilities to create a comprehensive language processing system.

6. Output: The output of the combined FLEX and BISON approach is a structured representation of the program, comprising a token stream and a parsed syntax tree. This output serves as input for subsequent phases in the compilation or interpretation process.

In summary, the integration of FLEX for lexical analysis and BISON for syntax analysis allows for the creation of a comprehensive language processing system, encompassing

tokenization, error detection, and syntax parsing in the compilation or interpretation of a programming language.

Working of Flex and Bison:

Flex and Bison often work together in the process of creating a compiler or interpreter. Flex is used to generate a lexical analyzer that recognizes patterns in the input code, producing a stream of tokens.

Bison, on the other hand, generates a parser that analyzes the syntax of the code based on a context-free grammar. The output of the Flex lexer is fed into the Bison parser, creating a cohesive system that handles both lexical analysis and parsing, essential steps in the compilation process. This combination allows for the efficient development of language processors for custom programming languages.

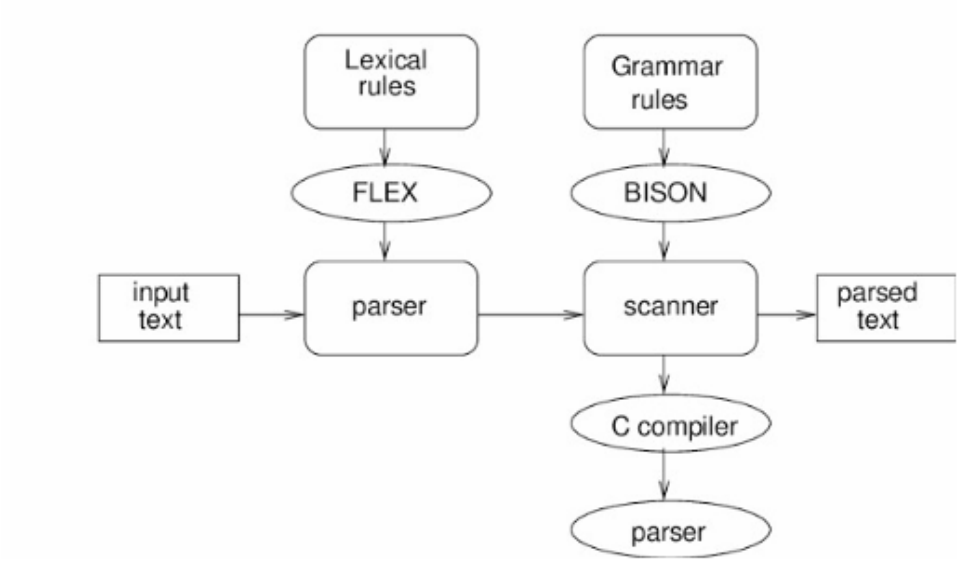

Figure-1: Flex and bison working together

-

Keywords

Keywords are reserved words in the language and cannot be used as identifiers. They have predefined purposes for language control, logic, and operations.

Keywords	Functionality
`inspect`	Equivalent to a print statement
base	Program start from here like Int main()
end	Program terminator

Control Structures:

Keyword	Functionality
if	Conditional statement
elif	Else-if condition
else	Else condition
repeat	Repetition structure (do)
while	Looping condition
from	Start value for iteration
to	End value for iteration
by	Step value for iteration

Logic & Math:

Keyword	Functionality
XOR	Exclusive OR logic operation
AND	Logical AND operation
OR	Logical OR operation
SIN	Sine function
COS	Cosine function
TAN	Tangent function
SQRT	Square root operation
log	Base-10 logarithm
log2	Base-2 logarithm
ln	Natural logarithm

Modules:

Keyword	Functionality
module	Define a reusable module

call	Call a defined module

Miscellaneous:

Keyword	Functionality
import<tamim.h>	Import a header file
choices for	Define conditional choices
none	Default condition
option	Individual choice condition
#asdf#	Indicating comment
:, ;	Deliminator

Data Types

Data Type	Keyword	Description
Integer	num	Represents whole numbers
Real Number	frac	Represents floating-point values
String	alpha	Represents text

Operators

Arithmetic Operators

Increment (++), Decrement (--), Addition (+), Subtraction (-), Multiplication (*), Division (/), Modulus (%), Power (^)

Comparison Operators

Equality (==), Inequality (!=), Less than or equal (<=), Greater than or equal (>=), Arrow operator (>>>)

Logical Operators

Exclusive OR (XOR), AND (AND), OR (OR), NOT (~)

Special Symbols

: (Colon) – Used to indicate the start of a block or statement.
; (Semicolon) – Marks the end of a statement.
{} (Curly Braces) – Used for grouping statements.
() (Parentheses) – Used for grouping expressions and function calls.
[] (Square Brackets) – Used for arrays or indexing.

Comments

Single-line Comments: Enclosed in # symbols (e.g., # This is a comment #).

Context-Free Grammars(CFG):

Context-free grammars (CFGs) are used to describecontext-free languages. A context-free grammar is a set of recursive rules used to generatepatterns of strings. A context-free grammar can describe all regular languages and more, but itcannot describe all possible languages.

Context-free grammars can generate context-free languages.They do this by taking a set of variables that are defined recursively, in terms ofone another, by a set of production rules. Context-free grammars are named as such because anyof the production rules in the grammar can be applied regardless of context—it does not dependon any other symbols that may or may not be around a given symbol that is having a ruleapplied to it. Context-Free Grammars used in Compiler:

Grammer Rules

%token BASE END START VARIABLE EOL ARROW

%token INTEGER REAL STRING INT_TYPE REAL_TYPE STRING_TYPE

%token SEE

%token AND OR NOT XOR LOG LOG2 LN SIN COS TAN FACTORIAL SQRT

%token IF ELIF ELSE CHOICE DEFAULT OPTION

%token FOREACH FROM TO DO WHILE BY AS

%token COMMENT MODULE CALL

%token IMPORT

%type <integer> INTEGER BASE END START program while_conditions

%type <string> VARIABLE INT_TYPE REAL_TYPE STRING_TYPE STRING COMMENT

%type <real> expr REAL statements statement

%nonassoc ELIF

%nonassoc ELSE

%left PPLUS MMINUS

%left AND OR XOR NOT

%left LOG LOG2 LN SQRT

%left '<' '>' EQL NEQL LEQL GEQL

%left '+' '-'

%left '*' '/' '%'

%right '^' FACTORIAL

%left SIN COS TAN

Rules:

%token BASE END START VARIABLE EOL ARROW

%token INTEGER REAL STRING INT_TYPE REAL_TYPE STRING_TYPE

```
%token SEE

%token AND OR NOT XOR LOG LOG2 LN SIN COS TAN
FACTORIAL SQRT

%token IF ELIF ELSE CHOICE DEFAULT OPTION

%token FOREACH FROM TO DO WHILE BY AS

%token COMMENT MODULE CALL

%token IMPORT


%type <integer> INTEGER BASE END START program
while_conditions

%type <string> VARIABLE INT_TYPE REAL_TYPE STRING_TYPE
STRING COMMENT

%type <real> expr REAL statements statement

%nonassoc ELIF

%nonassoc ELSE

%left PPLUS MMINUS

%left AND OR XOR NOT

%left LOG LOG2 LN SQRT

%left '<' '>' EQL NEQL LEQL GEQL

%left  '+' '-'

%left  '*' '/' '%'
```

```
%right  '^' FACTORIAL
%left SIN COS TAN


%%
program:     IMPORT BASE START statements END
statements:
          | statements statement
          ;
statement:
          EOL
          | COMMENT
          | declaration EOL
          | assigns EOL
          | show EOL
          | expr EOL
          | if_blocks
          | choice_block
          | loop_block
          | module_declare
          | module_call
    ;
```

```
declaration:
        INT_TYPE int_variables
        | REAL_TYPE real_variables
        | STRING_TYPE string_variables
   ;
int_variables:
        int_variables ',' int_var
        | int_var
   ;

int_var:
        VARIABLE '=' expr
        | VARIABLE
   ;
real_variables:
        real_variables ',' real_var
        | real_var
   ;
real_var:
        VARIABLE '=' expr
        | VARIABLE
```

```
    ;
string_variables:
        string_variables ',' string_var
        | string_var
    ;
string_var:
        VARIABLE '=' STRING
        | VARIABLE
    ;
show:
        SEE ARROW print_vars
    ;
print_vars:
        print_vars ',' VARIABLE
        | VARIABLE
    ;
assigns:
        assigns ',' assign
        | assign
    ;
assign:
        VARIABLE '=' expr
```

```
    ;
if_blocks:
        IF if_block else_statement
    ;

if_block:
        expr START statement END
    ;

else_statement:
        | elif_statement
        | elif_statement single_else
        | single_else
    ;
single_else:
        ELSE START statement END
    ;
elif_statement:
        elif_statement single_elif
        | single_elif
    ;
single_elif:
```

```
        ELIF expr START statement END
    ;
choice_block:
        CHOICE choice_variable START options END
    ;
choice_variable:
        VARIABLE
    ;
options:
        optionlist default
        | default
    ;

default:
        DEFAULT START statement END
    ;
optionlist:
        optionlist option
        | option
    ;
option:
        OPTION expr START statement END
```

```
    ;
loop_block:
        FROM expr TO expr BY expr START statement END
        | WHILE while_conditions START statement END
        | DO START expr END WHILE while_conditions EOL
    ;
while_conditions:
         VARIABLE PPLUS '<' expr
        | VARIABLE PPLUS LEQL expr
        | VARIABLE PPLUS NEQL expr
        | VARIABLE MMINUS '>' expr
        | VARIABLE MMINUS GEQL expr
        | VARIABLE MMINUS NEQL expr
    ;
module_declare:
         MODULE module_name '(' module_variable ')' START
statements END
    ;
module_name:
        VARIABLE
    ;
module_variable:
```

```
        module_variable ',' single_var
        | single_var
    ;
single_var:
        INT_TYPE VARIABLE
        | REAL_TYPE VARIABLE
        | STRING_TYPE VARIABLE
    ;
module_call:
        CALL user_module_name '(' user_parameters ')'
    ;

user_module_name:
        VARIABLE
    ;

user_parameters:
        user_parameters ',' single_param
        | single_param
    ;
single_param:
        VARIABLE
```

```
   ;
expr:
        INTEGER
        | REAL
        | VARIABLE
        | '+' expr
        | '-' expr
        | PPLUS expr
        | MMINUS expr
        | expr '+' expr
        | expr '-' expr
        | expr '*' expr
        | expr '/' expr
        | expr '^' expr
        | expr '%' expr
        | expr '<' expr
        | expr '>' expr
        | expr LEQL expr
        | expr GEQL expr
        | expr EQL expr
        | expr NEQL expr
        | expr AND expr
```

```
        | expr OR expr
        | expr XOR expr
        | NOT expr
        | '(' expr ')'
        | SIN '(' expr ')'
        | COS '(' expr ')'
        | TAN '(' expr ')'
        | LOG '(' expr ')'
        | LOG2 '(' expr ')'
        | LN '(' expr ')'
        | SQRT '(' expr ')'
        | VARIABLE PPLUS
        | VARIABLE MMINUS
        | expr FACTORIAL
    ;
%%
```

Makefile Commands:

```
final:
    bison -d tamim.y
    flex tamim.l
    gcc tamim.tab.c lex.yy.c
    ./a.exe
```

```
clean:
    del -f a.exe tamim.tab.c tamim.tab.h lex.yy.c output.txt
```

Discussion:

In this assignment, I learned flex and bison software. I created my own syntax for my language. I did Lexical Analysis with flex and generated token. I used regular expression to detect token. I passed the token to bison file for parsing.

All the syntax and rules for my language parsed using cfg defined with bison file. Although there are some shift/reduce and reduce/reduce conflict , my parser works for my language and give valid error when occurs with line number. I learned the idea how a compiler checks the given input is syntactically matched with the language's syntax or not. It's been a fascinating journey into the world of language design and compiler construction.

Conclusion: In conclusion, the exploration of flex and bison for the creation of my own language was found to be quite intriguing. Flex was employed for identifying words in the code, while bison was utilized for determining the structure.

Despite encountering some complexities with conflicts, mistakes are effectively captured by the code, and their

respective locations are communicated. Valuable insights were gained into the process of code sense-checking by compilers. This project contributed to a deeper understanding of language creation, and a continued enthusiasm for further learning in the domain of compiler development has been fostered. The journey proved to be an engaging experience. Reference:

References:

1. flex & bison - John R. Levin

2. LEX & YACC TUTORIAL - Tom Nieman

3. Compiler - Wikipedia 4. Flex (Fast Lexical Analyzer Generator) - GeeksforGeeks

5. GNU Bison – Wikipedia

6. Context Free Grammars | Brilliant Math & Science

Source Code:

https://github.com/tamim2007009/CompilerProject32-flex-bison-.git