Pantho Haque
Roll: 1907075
Group : B1
Year : 3rd
Semester: 2nd

FINAL PROJECT ON COMPILER DESIGN USING FLEX AND BISON

SUBMISSION DATE: 21 NOV 2023

Course code: CSE 3212

Computer Science and Engineering

Khulna University of Engineering and Technology, Khulna - 9203

Objective:

1. Learn to perform lexical analysis on our code snippet and make a parser using bison.
2. Identify and categorize various language constructs.
3. Learn to design our own syntax of a programming language and justify the syntax with proper CFG and semantic rules.

Introduction:

Bison is a general-purpose parser generator that converts an annotated context-free grammar into a deterministic LR or generalized LR (GLR) parser employing LALR(1) parser tables. As an experimental feature, Bison can also generate IELR(1) or canonical LR(1) parser tables. Once you are proficient with Bison, you can use it to develop a wide range of language parsers, from those used in simple desk calculators to complex programming languages.

Bison is upward compatible with Yacc: all properly-written Yacc grammars ought to work with Bison with no change. Anyone familiar with Yacc should be able to use Bison with little trouble. You need to be fluent in C or C++ programming in order to use Bison. Java is also supported as an experimental feature.

Basic Structure:

== == indicating a single line comment

/=
... indicating a multiline comment
=/

beginning the start of the code
ending the end of the code

Display:

This function displays any data we give to it.

```
    | prnt expression eol
     {
          if($2 == 111){
               printf("Displaying a prime number\n");
          }else if($2 == 222){
               printf("Displaying a non-prime number\n");
          }
          else if($2 != "not"){
               printf("Displaying the value %d\n", $2);
          }
     }
      | prnt STRING eol
     {
          printf("Displaying the text %s\n\n", $2);
     }
```

ID and Type Design:

Variables have 2 data types .

in will store integer

1. **flt** will store floating.
   point number

Multiple variables can be declared in a single line separated by **comma**.

Array can be declared by adding **[]** after the variable name.

No duplicate variables are allowed here

```
"in"       {return int_type;}
"flt"       {return float_type;}
```

```
| int_type syn eol { if($2 == "not"){printf("of INTEGER type\n");}}
      | float_type synFlt eol
     {
          if ($2 == "not") {
               printf("of FLOAT type\n");
          }
     }
```

Operator Handling:

- Assignment operator (<-)
- Arithmetic Operators
  1.**add** for Addition
  2.**sub** for Subtraction
  3.**mul** for Multiplication
  4.**div** for Division
- Relational Operators
  1. **gt** for greater than
  2.**lt** for less than
  3. **ge** for greater equal
  4.**le** for less equal
  5.**ne** for not equal
  6.**eq** for equal to

```
"add"        {return plus;}
"sub"        {return minus;}
"**"         {return into;}
"div"        {return divi;}
"pow"        {return power;}
"mod"        {return mod;}
"root"       {return ssqrt;}

"ge"       {return ge;}
"le"       {return le;}
"eq"       {return eqeq;}
"gt"       {return gt;}
"ngt"      {return ngt;}
"lt"       {return lt;}
"nlt"      {return nlt;}
"ne"       {return neq;}
```

from-to Loop:

Loop is used to make an iteration through a data structure like array or string

from and **to** keywords are used to maintain the iteration sequence in loop

```
| from  fstatement to  INTEGER sbs statement sbf
     {
          if($2 != "not"){
               for(int a =$2;a<=$4; a= a+1 ){
                    printf("from to Looping %d\n",a);
               }
          }
          printf("From to loop finished\n\n");
     }
```

stopif Loop:

This loop is used to go through the statements again and again till a specific condition is satisfied. Once the condition returns false the loop will stop.

| | is holding the condition of this loop

```
| stopif ps VARIABLE ge INTEGER ps sbs VARIABLE dec INTEGER eol statement sbf
     {
          int a = VALUE[getValue($3)];
          for(int i=a;i>=$5;i=i-$12){
               printf("Stop-if loop current value %d\n",i);
          }
          printf("Stop-if loop finished\n\n");}
     | stopif ps VARIABLE le INTEGER ps sbs VARIABLE inc INTEGER eol statement
sbf
     {
          int a = VALUE[getValue($3)];
          for(int i=a;i<=$5;i=i+$12){
               printf("Stop-if loop current value %d\n",i);
          }
          printf("Stop-if loop finished\n\n");
     }
```

do-or-if conditional statement:

To do some action after taking a decision, we use conditional statement. The first condition will be initiated using **do-if** keyword. Then we use **or-if** keyword to check further conditions. Lastly the default action will be under **or** block.

```
| doif ps expression ps sbs statement sbf
    {
        if($3){
            printf("If condition true\n\n");

        }else{
            printf("If condition false\n\n");
        }
    }
    | doif ps expression ps sbs statement sbf elfb or sbs statement sbf
    {
        if($3){
            printf("If condition true\n\n");

        }else{
            printf("If condition false\n\n");
        }
    }
```

Function:

To make our code less redundance we use function which is identified by **func** keyword **| |** holding the parameters of our function

```
| ffunc VARIABLE ps param ps sbs statement sbf int_type
    {
        printf("%s Function Declared\n",$2);
    }
```

Directives:

##<> indicates importing a header file to our program.

End of a statement:

;; will indicate the end of each statement

Discussion:

Flex and Bison are essential tools for language processing. Flex (Fast Lexical Analyzer Generator) focuses on lexical analysis, breaking code into tokens. It uses rules based on regular expressions to identify patterns in code. Bison (GNU Parser Generator) deals with parsing code using formal grammar rules. It constructs a parser in C/C++ that understands code structure based on the defined grammar.

Conclusion:

Together, they streamline the process of language development by efficiently handling lexical analysis and parsing. Flex identifies tokens, making code manageable for Bison to understand syntax. This combination aids in creating compilers, interpreters, and language processors for various programming languages. Their integration allows for developing efficient language-processing tools.