دانشگاه صنعتی اصفهان

دانشکده مهندسی برق و کامپیوتر

درس کامپایلر

تکلیف عملی سوم

تاریخ تحویل: ۱۸ خرداد

Question 1

Explanation:

In this question, we want to write a Flex-Bison parser that recognizes expression in Postfix or Prefix notations

This parser supports 5 operations $* \ - / + \hat{}$

It only supports positive integer or floating numbers, such as 56 or 1.23, you don't need to worry about unary subtract operator (such as - 3)

Important Points

- Your parser should output **error** for
    - Undefined characters
    - Syntax errors, for example missing operands

Input arguments and Output

Only argument is a **path** to a file containing the input for your parser.

The output of your parser should be an error in case of invalid input or the notation of expression (prefix or postfix)

Examples

Input: 5 4 + 1 + 9 + +

Output: syntax error

Input: 5 4 + 1 + 9 +

Output: Postfix

Input: * + 4.2 .5 9

Output: Prefix

Input: 5 4 %

Output: Undefined character error

Submission

Your submission **must** contain:

1. A lexer file for Flex

2. A parser file for Bison
3. A makefile for compiling your project

Question 2

We want to design and develop a Flex-Bison parser for a simple programming language called N++.

Statements

All statements in this language end with a semicolon ‘ ; ’

Data types

N++ supports integers, floats and Boolean which are indicated by keywords int, float and bool respectively.

Variables

N++ supports variable declaration, which is done with a data type and one or a series of variable names, for example:

```
int x;
float y1, x_123;
bool _isDone;
```

Note that multiple variables of the same data type can be declared using a comma-separated list. We'll discuss variable naming rules later in this document.

Assignment

Variables can be assigned a value **only after declaration.**
However, N++ supports assignment chaining, for example:

```
x = y1 = 2;
x_123 = y1 = 3.1;
_isDone = false;
```

- **Important point:** when assigning (specially in chained assignment) using literals as lvalue (left hand side of assignments) is not correct, for example:
  ```
  2 = x;
  X_123 = 3.1 = y1;
  ```

Condition

N++ supports control flow using if-else statements. The structure of these statements is as follow:

```
if (expression) {
        A block of code containing zero or more statements;
}
else {
        A block of code containing zero or more statements;
}
```

- **Important point:** an else block should always come after an if block, however, using an else block is not mandatory (“if” block can be alone), for example

  ```
  if (x > 2){
       y = 3;
  }
  ```

  is a correct statement, but

  ```
  x = 2;
  else {
     y = 5;
  }
  ```

  is not correct.
- **Important point:** the expression used in between parentheses of an if statement can contain assignments, for example

  ```
  if (x = z)
  if (y = x = (z > 5))
  ```

Loop

N++ supports loop using while statements, The structure of these statements is as follow:

```
while (expression) {
   A block of code containing zero or more statements;
}
```

- **Important point:** the expression used in between parentheses of an while statement can contain assignments, for example

  ```
  while (x = z)
  while (y = x = (z > 5))
  ```

Operators

N++ supports 5 groups of operators:

Arithmetic operators: / , - , * , + , %

Relational operators: < , > , <= , >= , == , !=

Logical operators: ! , && , ||

bitwise operators: ~ , & , | , ^

precedence overriding operators : ( )

- **Important point:** please note that some of these operators are binary and some are unary
- **Important point:** all expressions can contain a combination of all these operators, for example
  x = 2 / 3 * (y % z) ^ (3 & bitflag) | (~bitflag)
  if (2 || !(2 > (4 + ~y)))

Literals

Integer literals: contain only digits, for example: 123, 532, 415

Float literals: contain digits and a floating point, for example: 123**.**, **.**123, 123**.**456

Boolean literals: true, false

- **Important point:** all expressions can contain a combination of all these literals
- **Important point:** N++ does not support unary + or – operators, so you don't need to worry about them.

Variable naming

Variable identifiers can start with underscore and English alphabet and may contain underscore, alphabet and digits, **identifiers can't start with digits**, for example:

x_1, ___, _1_, _XXY123 are all valid identifiers.

Grammar

The following grammar is an approximation and might be ambiguous, you may need to change it.

Program ::= StatementList | ε

StatementList ::= StatementList Statement ; | ε

Statement ::= Declaration | Assignment | Conditional | Loop

Declaration ::= DataType VariableList

VariableList ::= VARIABLE , VariableList | VARIABLE

DataType ::= BOOLEAN | INT | FLOAT

Assignment ::= VARIABLE = RValue

RValue ::= VARIABLE = RValue | Expression

Conditional ::= IfBlock ElseBlock | IfBlock

IfBlock ::= IF (Condition) { StatementList }

ElseBlock ::= ELSE { StatementList }

Loop ::= WHILE (Condition) { StatementList }

Condition ::= Expression | Assignment

Expression ::= Expression BinaryOperator Expression | UnaryOperator Expression | (Expression) | VARIABLE | Literal

Literal ::= BooleanLiteral | INEGERLITERAL | FLOATLITERAL

BooleanLiteral ::= TRUE | FALSE

BinaryOperator ::= * | / | - | + | % | < | > | <= | >= | == | != | & | && | "|" | "||" | ^

UnaryOperator ::= ! | ~

Important Notes V1.1

- Your grammar **should not** have any conflicts

Input arguments and Output

Only argument is a **path** to a file containing the input for your parser.

The output of your parser should be the productions used

Example V1.1

Input: x = y = 2;

Output:

$$\begin{gathered} StatementList -> \ e \\ Literal -> \ INEGERLITERAL \\ Expression -> \ Literal \\ RValue -> \ Expression \\ RValue -> \ VARIABLE\ ASSIGN\_OP\ RValue \\ Assignment -> \ VARIABLE\ ASSIGN\_OP\ RValue \\ Statement -> \ Assignment \\ StatementList -> \ StatementList\ Statement\ SEMICOLON \\ Program \ \rightarrow \ StatementList \end{gathered}$$

Submission

Your submission **must** contain:

1. A lexer file for Flex
2. A parser file for Bison
3. A makefile for compiling your project

Additional Points (15%) V1.1

If your parser prints the reduced rules in correct order, you gain extra points!

For example, the output for the above example should look like this:

$$\begin{gathered} Program \ \rightarrow \ StatementList \\ StatementList -> \ StatementList\ Statement\ SEMICOLON \\ StatementList -> \ e \\ Statement -> \ Assignment \\ Assignment -> \ VARIABLE\ ASSIGN\_OP\ RValue \\ RValue -> \ VARIABLE\ ASSIGN\_OP\ RValue \\ RValue -> \ Expression \\ Expression -> \ Literal \\ Literal -> \ INEGERLITERAL \end{gathered}$$