CS3423: Assignment 1
Lex and Yacc
Due Wednesday, September 25th, 2024 at 21:00 hrs (9:00 PM)

Implement a lexer and parser for a language with the following specifications. It should output an equivalent C++ code and log files for tokens and statements encountered.

Note: corrections are highlighted in yellow

1. Lexical Analyzer

The lexical analyzer must recognize all the tokens categorised below :

- **Identifier**
  It can contain alphabets (both upper and lowercase), numbers and underscores. The first character cannot be a number. (eg: demo, Demo, d3m0, _demo_)
- **Constant**
  int constant (eg: 122, 0, 42)
  float constant (eg: 34.2, .3, 123.456, .5)
- **Punctuation**
  [ ] { } ( ) : ; < > ?
- **Reserved Word**
  set if else size loop finally return func print
- **Type**
  int float small big
- **Access Operator**
  [ ] -> <-
- **Arithmetic Operator**
  + - * / %
- **Bitwise Operator**
  | & ^ ~
- **Logical Operator**
  or and not
- **Comparison Operator**
  < > <= >= <> (here <> is 'not equal to'. There is not != operator)
- **Single Line Comment**
  # everything to the right of a hashtag

2. Syntax Analyzer

The parser should transpile the input program to C++. The overall structure of the program is as follows. There is a setup section followed by the main section. The setup section consists of Set Statement and Function Definitions. Function definitions and main section consist of a *body* made up of statements of the categories listed below. Statements are separated by semicolons. Spaces, tabs or newlines between tokens are ignored. Single line comments start with a hashtag (#).

- Set Statement (only in setup section)
- Variable Declaration
- Assignment Statement
- Push/Pop Statement
- Conditional Statement
- Loop Statement
- Function Declaration (only in setup section)
- Return Statement (only in function definitions)
- Print Statement

Set Statement

```
set int small          # int will be transpiled to int in c++
set int big            # int will be transpiled to long long in c++
set float small        # float will be transpiled to float in c++
set float big          # float will be transpiled to double in c++
```

Variable Declaration

```
float a                # declares a float a
int a=3, b, c          # declares integers a,b,c
[int] example          # declares vector of int
{int: float} example2  # declares a map from int to float types
{int: [float]} Example_3  # map of int to vector of float
```

Assignment Statement

```
a = 2*(a+b)            # RHS is an expression
```

Expression:

These contain variables, constants, arithmetic operators, bitwise operators, access operator [ ], size, grouping and function calls using ( ).

Element Access:

Accessing vector/map elements is done using [ ] operator with an index.

The index itself is an expression.
a[i] = b[i+4] % 7

Function Call:
Functions are called with any number of arguments including zero.
The arguments are also expressions.
a = foo() + bar(b, 5*c)

Push/Pop Statement

```
myvec<-[expr]          # pushes expr to the back end of the vector
[expr]->myvec          # pushes expr to the front end of the vector
myvec->[var]           # pops back end of the vector into a variable var
[var]<-myvec           # pops front end of the vector into a variable var
myvec->[]              # pops back end of the vector, discarding the value
[]<-myvec              # pops front end of the vector, discarding the value
```

Note: expr is any expression as defined previously
Note: here a variable can be a single element of a vector/map also.
Note: size[x] gives count of elements in x where x is a vector or map

Conditional Statement
We have proposed two grammars for conditional statements. Implement one and explain what the issue is with the other. A conditional will always contain an if-equivalent statement. But the statements equivalent to else if and else are optional in a conditional block. Predicate: consists of expressions separated by logical and comparison operators.

Alternate 1:

```
< predicate ? body          #equivalent to if
  predicate ? body          # equivalent to else if
      ….
  predicate ? body          # equivalent to else if
  else : body >             # equivalent to else
```

Nested example:

```
< absent_days == 0 ?
  < marks > 70?
    grade =10;                  # body can have any number of statements
    good_kids=good_kids+1;
    marks > 50?
```

```
   grade =8;
   else: grade =4;
 >

 absent_days  < 4 ?  <          # compact indentation
   marks > 70? grade =8;
   marks > 50? grade =4;
   else: grade =0;
 >

 else: <
   marks > 70? grade =4;        # only if portion is mandatory
   else: grade =0;
 >
>
```

Alternate 2:

INDENT if *predicate*:
body
INDENT else if *predicate*:
body
….
INDENT else:
body

#Here *INDENT* is a non-zero number of '>' representing the depth of the if/else if/else.
#It will be empty for depth 0, > for depth 1, >> for depth 2 and so on

Nested example:

```
if absent_days == 0:
> if marks > 70:
grade=10;                              # body can have any number of statements
good_kids=good_kids+1;
> else if marks > 50:
grade = 8;
> else:
grade = 4;
```

```
else if absent_days  < 4:              # compact indentation
> if marks > 70: grade=8;
> else if marks > 50: grade = 4;
> else: grade = 0;

else:
> if marks > 70: grade=4;              # only if portion is mandatory
> else: grade = 0;
```

Loop Statement

This is similar to a for-loop in C++.
init - declaration or empty.
predicate - same as a predicate of a conditional.
update - assignment statement or empty.

The body of the loop block runs for every iteration in which the predicate is true.
The body of the finally block runs exactly once after the loop ends. The variables in the finally block have the same scope as the loop block. (ie. Variables defined in the while loop are still accessible in the finally block). The finally block may be omitted.

```
loop (init; predicate; update):
<
  body
>
finally:
<
  body
>
```

Example:

```
loop (int i=0; i < size[myvec]-1 ; i=i+1) :
<
   myvec[i] = 2*myvec[i] + myvec[i+1] ;
   i++;
>
finally:
<
```

```
   myvec[i] = 2*myvec[i]  ;                # using value of i after last iteration
>
```

Function Declaration

- All functions are defined at the start of the program.
- A function is declared with the func keyword before its name.
- This is followed by a list of arguments and types, separated by commas.
- The return-type of the function is provided after a semicolon.
  It can be any valid type (primitives or complex) or void.
- The definition is provided in angled brackets. It has the same types of statements as the main code. Further, at least one return statement must exist anywhere in the body.

```
func identifier (type arg1, type arg2 … type arg3 ; return-type) <
    function-body
>
```

```
func total_area ( [float] h, [float] b ; float) <
  float sum=0;
  loop (int i=0 ; i < size[h] and i < size[b] 0; i=i+1)  <
    sum= sum+h[i]*b[i];
  >
  return sum;
>
```

Return Statement

It consists of a return statement followed by an expression that is returned.

```
return 2*3.14*r;                # returning value from an expression
return void;                    # for functions that do not return anything
```

Print Statement

```
print(a)                        # prints the value of an int or float
```

3. Logging

The input file will be a .txt file. The parser should take the input file name as the first command line argument.

During lexical analysis: every token should be printed along with its category as described in the specifications. Generate a file tokens.txt whose lines have the format:
LINE-NUM : CATEGORY : TOKEN

If a token has multiple uses, print both uses separated by " : " as shown below
4 : Punctuation : Access Operator : [
9 : Punctuation : Comparison Operator : <

During syntax analysis: every statement and function declaration should be categorised and printed in a file parsed.txt with each line having the format:
LINE-NUM : CATEGORY

Note: Nested Statements must be printed too.
LINE-NUM is the line number of the starting line of the statement in the input file. For example, in case of conditional statements or function declarations that span multiple lines, only use the line number from the first line.

TEST PROGRAM

```
set int small
set float small

func total_area ( [float] h, [int] b ; float)  # comment
<
  float sum=0;
  loop (int i=0 ; i < size[h] and i < size[b] 0; i=i+1)
  < sum = sum+h[i]*b[i]; >
  return sum;
>

[float] h, b;
a<-[0.5]; b<-[3];    # previously incorrect bracket a<-(0.5) was written
a<-[1.5]; b<-[2];
a<-[2.5]; b<-[1];
print( total_area(h,b) );
```

TOKEN LOG

1 : Reserved Word : set
1 : Type : int

1 : Type : small
2 : Reserved Word : set
2 : Type : int
2 : Type : small
4 : Reserved Word : func
4 : Identifier : total_area
4: Punctuation : (
4 : Punctuation : [
4 : Type : float
4 : Punctuation :]
4 : Identifier : h
…

STATEMENT LOG

1 : Set Statement
2 : Set Statement
4 : Function Declaration
6 : Variable Declaration
7 : Loop Statement
8 : Assignment Statement
9 : Return Statement
12 : Variable Declaration
13 : Push/Pop Statement
13 : Push/Pop Statement
14 : Push/Pop Statement
14 : Push/Pop Statement
15 : Push/Pop Statement
15 : Push/Pop Statement
16 : Print Statement

PRINTED RESULT

7

4. Test Cases

Submit 3 valid and 2 invalid test cases as input. They should be non-trivial.
Name them as t1.txt, t2.txt … t5.txt
Also submit:

- output .cpp file
- logs of tokens
- logs of parsed statements
- printed output from running the .cpp file
  (Note: all test programs must print a result)

5. Submission Details

Teams of 2 people are allowed. Your submission should be a gzip archive with the name: Asgn-1-ROLLNO1-ROLLNO2.tar.gz

It should have the following directory structure:

Asgn-1-ROLLNO1-ROLLNO2

```
├── report.pdf - provides justifications, assumptions and instructions
├── src - directory containing sources
├── build - directory containing makefile that should generate binaries here
└── test
    ├── input - directory for test input files like t1.txt
    ├── output - directory for transpiled c++ files like output1.cpp
    ├── printed - directory for result of c++ program like printed1.cpp
    ├── tokens - directory for token log files like tokens1.txt
    └── parsed - directory for statement log files like parsed1.txt
```

Mention the roll numbers of team members in the PDF.

Evaluation

- Lexer: 20 %
- Parsing:
  - Conditional Statement: 15%
  - Loop Statement: 15%
  - Function Declaration: 10%
  - Other Statements: 30%
- Test Cases: 10%