Concordia University
Department of Computer Science and Software Engineering

Compiler Design (COMP 442/6421)
Winter 2014

Assignment 2, Syntactic Analyzer

Deadline:	Monday February 24th, 2014
Evaluation:	10% of final grade
Late submission:	penalty of 50% for each late working day

You have to design a syntactic analyzer for the language specified by the grammar below. We are using the following convention: Terminals (lexical elements) are represented with the bold courier font **`like this`**. Non-terminals are represented in angle brackets `<like this>`. The character ε (epsilon) represents an empty stream. The non-terminal `<prog>` is the starting symbol of the grammar.

Grammar

```
       <prog> ::= <classDecl>*<progBody>
  <classDecl> ::= class id {<varDecl>*<funcDef>*};
   <progBody> ::= program<funcBody>;<funcDef>*
   <funcHead> ::= <type>id(<fParams>)
    <funcDef> ::= <funcHead><funcBody>;
   <funcBody> ::= {<varDecl>*<statement>*}
    <varDecl> ::= <type>id<arraySize>*;
  <statement> ::= <assignStat>;
                |  if(<expr>)then<statBlock>else<statBlock>;
                |  for(<type>id<assignOp><expr>;<relExpr>;<assignStat>)<statBlock>;
                |  get(<variable>);
                |  put(<expr>);
                |  return(<expr>);
 <assignStat> ::= <variable><assignOp><expr>
  <statBlock> ::= {<statement>*} | <statement> | ε
       <expr> ::= <arithExpr> | <relExpr>
    <relExpr> ::= <arithExpr><relOp><arithExpr>
  <arithExpr> ::= <arithExpr><addOp><term> | <term>
       <sign> ::= + | -
       <term> ::= <term><multOp><factor> | <factor>
     <factor> ::= <variable>
                |  <idnest>*id(<aParams>)
                |  Num
                |  (<arithExpr>)
                |  not<factor>
                |  <sign><factor>
   <variable> ::= <idnest>*id<indice>*
     <idnest> ::= id<indice>*.
     <indice> ::= [<arithExpr>]
  <arraySize> ::= [ int ]
       <type> ::= int | float | id
    <fParams> ::= <type>id<arraySize>*<fParamsTail>* | ε
    <aParams> ::= <expr><aParamsTail>* | ε
<fParamsTail> ::= ,<type>id<arraySize>*
<aParamsTail> ::= ,<expr>
```

Operators and additional lexical conventions

```
<assignOp>  ::=  =
   <relOp>  ::=  == | <> | < | > | <= | >=
   <addOp>  ::=  + | - | or
  <multOp>  ::=  * | / | and

        id  ::=  follows specification for identifiers found in assignment#1
       num  ::=  follows specification for numbers found in assignment#1
       int  ::=  <nonZero><digit>*
 <nonZero>  ::=  1..9
   <digit>  ::=  <nonZero> | 0
```

For example, the non-terminal `<addOp>` is a generalization of the addition operators tokens **+**, **–** and **or**. The use of this notation here does not necessarily imply that you have to define a new type of token in your lexical analyzer. Also, **id** and **num** are tokens that refer to the lexical conventions given in the first assignment. Note that a new lexical convention for the token **int** has been added.

Work to be done

- Analyze the syntactical definition given on the first page (and the additional lexical definition for the token **int**). Remove all the * notations and replace them by list-generating productions. List in your documentation all the ambiguities and left recursions, or any error you may find in the grammar. Modify the productions so that the left recursions and ambiguities are removed without modifying the language. You should obtain a set of productions that can be parsed using the top-down predictive parsing method. Include the transformed grammar in your documentation.
- Derive the FIRST and FOLLOW sets for each non-terminal in your transformed grammar and list them in your documentation.
- Implement a predictive parser (recursive descent or table-driven) for your modified set of grammar rules.
- Your parser should optionally output to a file the derivation that derives the source program from the starting symbol.
- The parser should call your lexical analyzer as developed in assignment 1 when it needs a new token.
- The parser should properly identify the errors in the input program and print a meaningful message to the user for each error encountered. The parser should implement an error recover method that permits to report all errors. The error messages should be informative on the nature of the errors, as well as the location of the errors in the input file.
- In this assignment, you only check the syntactic correctness of the program, i.e., check whether the source program can be parsed according to the grammar. Do not check the semantic correctness of the program in this assignment.
- Write a set of source files that enable to test the parser for all syntactical structures involved in the language. Include cases testing for a variety of different errors to demonstrate the accuracy of your error reporting and recovery.

Example program

```
class Utility
{
	int var1[4][5][7][8][9][1][0];
	float var2;
	int findMax(int array[100])
	{
		int maxValue;
		int idx;
		maxValue = array[100];
		for( int idx = 99; idx > 0; idx = idx  - 1 )
		{
			if(array[idx] > maxValue) then {
				maxValue = array[idx];
			}else{};
		};
		return (maxValue);
	};
	int findMin(int array[100])
	{
		int minValue;
		int idx;
		minValue = array[100];
		for( int idx = 1; idx <= 99; idx = ( idx ) + 1)
		{
			if(array[idx] < maxValue) then {
				maxValue = array[idx];
			}else{};
		};
		return (minValue);
	};
};
program {
	int sample[100];
	int idx;
	int maxValue;
	int minValue;
	Utility utility;
	Utility arrayUtility[2][3][6][7];
	for(int t = 0; t<=100 ; t = t + 1)
	{
		get(sample[t]);
		sample[t] = (sample[t] * randomize());
	};
	maxValue = utility.findMax(sample);
	minValue = utility.findMin(sample);
	utility. var1[4][1][0][0][0][0][0] = 10;
	arrayUtility[1][1][1][1].var1[4][1][0][0][0][0][0] = 2;
	put(maxValue);
	put(minValue);
};
real randomize()
{
	real value;
	value = 100 * (2 + 3.0 / 7.0006);
	value = 1.05 + ((2.04 * 2.47)  - 3.0) + 7.0006 ;
	return (value);
};
```

Assignment submission requirements and procedure

You have to submit your assignment before midnight on the due date using the ENCS Electronic Assignment Submission system under the category "*programming assignment 2*". The file submitted must be a **.zip** file containing:

- all your code
- a set of input files to be used for testing purpose, as well as a printout of the resulting output of the program for each input file (derivation and error reporting, as described above)
- a simple document containing the information requested above

You are also responsible to give proper compilation and execution instructions to the marker in a README file. If the marker cannot compile and execute your programs, you might have to have a meeting for a demonstration.

Evaluation criteria and grading scheme

Documentation:		
	List of left recursions and ambiguities in the original grammar	3 pts
	Transformed grammar	2 pts
	FIRST and FOLLOW sets of non-terminals in the transformed grammar	3 pts
Program:		
	Correct implementation according to assignment statement	15 pts
	Accurate output of error messages	5 pts
	Output of derivation in a file	2 pt
	Error recovery	5 pts
	Completeness of test cases	15 pts
Total		50 pts