Concordia University
Department of Computer Science
and Software Engineering

Compiler Design (COMP 442/6421)
Winter 2021

Assignment 2, Syntactic Analyzer

Deadline:	Friday March 12th, 2021
Evaluation:	10% of final grade
Late submission:	penalty of 50% for each late working day

This assignment is about the design and implementation of a syntactic analyzer for the language specified by the grammar given below. The syntactic analyzer should use as input the token stream produced by the lexical analyzer that you have produced in assignment #1, and prove whether or not the token stream is a valid program according to the grammar given below. While doing so, it should locate, report, and recover from eventual syntax errors. The operation of the syntactic analyzer should produce an abstract syntax tree that is going to be used by the further processing steps to be implemented in assignment #3 (semantic analysis) and assignment #4 (code generation). It should also write to a file a trace of the derivation that is proving that the input token stream can be derived from the starting symbol of the grammar.

The assignment includes two grading source files used by the marker. These files should be used as-is and not be altered in any way. Completeness of testing is a major grading topic. You are responsible for providing appropriate test cases that test for a wide variety of valid and invalid cases in addition to what is in the grading source files provided.

Grammar

G = (N, T, S, R)

N – Nonterminal Symbols

START, aParams, aParamsTail, addOp, arithExpr, arraySize, assignOp, assignStat, classDecl, expr, fParams, fParamsTail, factor, funcBody, funcDecl, funcDef, funcHead, functionCall, idnest, indice, memberDecl, multOp, prog, relExpr, sign, statBlock, statement, term, type, varDecl, variable, visibility

T – Terminal Symbols

,, +, -, |, [, intLit,], =, class, id, {, }, ;, (,), floatLit, !, :, void, ., *, /, &, inherits, sr, main, eq, geq, gt, leq, lt, neq, if, then, else, read, return, while, write, float, integer, private, public, func, var, break, continue, string, qm, stringLit

S – Starting Symbol

START

R – Rules

```
<START>        ::= <prog>
<prog>         ::= {{<classDecl>}} {{<funcDef>}} 'main' <funcBody>
<classDecl>    ::= 'class' 'id' [['inherits' 'id' {{',' 'id'}}]] '{' {{<visibility> <memberDecl>}} '}' ';'
<visibility>   ::= 'public' | 'private' | EPSILON
<memberDecl>   ::= <funcDecl> | <varDecl>
<funcDecl>     ::= 'func' 'id' '(' <fParams> ')' ':' <type> ';'
                 | 'func' 'id' '(' <fParams> ')' ':' 'void' ';'
<funcHead>     ::= 'func' [['id' 'sr']] 'id' '(' <fParams> ')' ':' <type>
                 | 'func' [['id' 'sr']] 'id' '(' <fParams> ')' ':' 'void'
<funcDef>      ::= <funcHead> <funcBody>
<funcBody>     ::= '{' [[ 'var' '{' {{<varDecl>}} '}' ]] {{<statement>}} '}'
<varDecl>      ::= <type> 'id' {{<arraySize>}} ';'
<statement>    ::= <assignStat> ';'
                 | 'if'     '(' <relExpr> ')' 'then' <statBlock> 'else' <statBlock> ';'
                 | 'while'  '(' <relExpr> ')' <statBlock> ';'
                 | 'read'   '(' <variable> ')' ';'
                 | 'write'  '(' <expr> ')' ';'
                 | 'return' '(' <expr> ')' ';'
                 | 'break' ';'
                 | 'continue' '; '
                 | <functionCall> ';'
<assignStat>   ::= <variable> <assignOp> <expr>
<statBlock>    ::= '{' {{<statement>}} '}' | <statement> | EPSILON
<expr>         ::= <arithExpr> | <relExpr>
<relExpr>      ::= <arithExpr> <relOp> <arithExpr>
<arithExpr>    ::= <arithExpr> <addOp> <term> | <term>
<sign>         ::= '+' | '-'
<term>         ::= <term> <multOp> <factor> | <factor>
<factor>       ::= <variable>
                 | <functionCall>
                 | 'intLit' | 'floatLit' | 'stringLit'
                 | '(' <arithExpr> ')'
                 | 'not' <factor>
                 | <sign> <factor>
                 | 'qm' '[' <expr> ':' <expr> ':' <expr> ']'
<variable>     ::= {{<idnest>}} 'id' {{<indice>}}
<functionCall> ::= {{<idnest>}} 'id' '(' <aParams> ')'
<idnest>       ::= 'id' {{<indice>}} '.'
                 | 'id' '(' <aParams> ')' '.'
<indice>       ::= '[' <arithExpr> ']'
<arraySize>    ::= '[' 'intNum' ']' | '[' ']'
<type>         ::= 'integer' | 'float' | 'string' | 'id'
<fParams>      ::= <type> 'id' {{<arraySize>}} {{<fParamsTail>}} | EPSILON
<aParams>      ::= <expr> {{<aParamsTail>}} | EPSILON
<fParamsTail>  ::= ',' <type> 'id' {{<arraySize>}}
<aParamsTail>  ::= ',' <expr>
<assignOp>     ::= '='
<relOp>        ::= 'eq' | 'neq' | 'lt' | 'gt' | 'leq' | 'geq'
<addOp>        ::= '+' | '-' | 'or'
<multOp>       ::= '*' | '/' | 'and'
```

Notes

- Terminals (lexical elements, or tokens) are represented in single quotes *'likeThis'*.
- Non-terminals are represented between angle brackets *<likeThis>*.
- The empty phrase is represented by *EPSILON*.
- EBNF-style repetition notation is represented using double curly brackets *{{like this}}*. It represents zero or more occurrence of the sentential form enclosed in the brackets.
- EBNF-style optionality notation is represented using double square brackets *[[like this]]*. It represents zero or one occurrence of the sentential form enclosed in the brackets.
- *id* follows the specification for program identifiers found in assignment #1.
- *intLit, floatLit, stringLit* follow specification for integer, float, and string literals found in assignment #1
- *qm stands for ? and sr stands for ::*

Work to submit

Document

You must provide a short document that includes the following sections:

Section 1. **Transformed grammar into LL(1)** : Remove all the EBNF notations and replace them by right-recursive list-generating productions. Analyze the syntactical definition (using tools) and list in your documentation all the ambiguities and left recursions. Modify the grammar so that the left recursions and ambiguities are removed without modifying the language. Include in your documentation the set of productions that can be parsed using the top-down predictive parsing method, i.e. an LL(1) grammar.

Section 2. **FIRST and FOLLOW sets** : Derive the FIRST and FOLLOW sets for each non-terminal in your transformed grammar.

Section 3. **Design** : Give a brief overview of the overall structure of your solution, as well as a brief description of the role of each component of your implementation.

Section 4. **Use of tools** : Identify all the tools/libraries/techniques that you have used in your analysis or implementation and justify why you have used these particular ones as opposed to others.

Implementation

- **Parser** : Implement a predictive parser (recursive descent or table-driven) for your modified set of grammar rules.
- The result of the parser should be the creation of a tree data structure representing the parse tree as identified by the parsing process. This tree will become the intermediate representation used by the two following assignments. When parsing a file named, for example, **`originalfilename`**, the parser should write into a file named ***`originalfilename`*`.outderivation`** the derivation that corresponds to the original program, as well as write into a file named ***`originalfilename`*`.outast`** a text representation of the abstract syntax tree representing the original program. When syntax errors are found, error messages should be printed out in a file named ***`originalfilename`*`.outsyntaxerrors`.**
- **Derivation output** : Your parser should write to a file the derivation that proves that the source program can be derived from the starting symbol.
- **AST output**: The generated tree should be output to a file in any format that allows easy visualization of the structure of the tree. This can be a simple text representation of the tree structure, but it needs to allow the marker to quickly verify that the tree you generate is in fact correct with regards to the parsed program.
- **Error reporting** : The parser should properly identify all the errors in the input program and print a meaningful message to the user for each error encountered. The error messages should be informative on the nature of the errors, as well as the location of the errors in the input file.
- **Error recovery** : The parser should implement an error recovery method that permits to report all errors present in the source code.
- **Test cases** : 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.
- **Driver**: Include a driver that extracts the tokens from all your test files. For each test file, the corresponding **`outsyntaxerrors`**, **`outderivation`**, and **`outast`** files should be generated.

Assignment submission requirements and procedure

- Each submitted assignment should contain four components: (1) the source code, (2) a group of test files, (3) a brief report, and (4) an executable named **`parserdriver`**, that extracts the tokens from all your test files. For each test file, the corresponding **`outsyntaxerrors`**, **`outast`** and **`outderivation`** files should be generated.
- The assignment statement provides test files (**`.src`**) and their corresponding output files.
- The source code should be separated into modules using a comprehensible coding style.
- The assignment should be submitted through moodle in a file named: "**`A#_student-id`**" (e.g. **`A1_1234567`**, for Assignment #1, student ID 1234567) and the report must in a PDF format.
- You may use any language you want in the project and assignments but the only fully supported language during the lab is Java.
- You have to submit your assignment before midnight on the due date on moodle.
- The file submitted must be a **.zip** file

Evaluation criteria and grading scheme

Analysis:		
List of left recursions and ambiguities in the original grammar, and then resulting correctly transformed LL(1) grammar – document Section 1.	ind 2.1	3 pts
FIRST and FOLLOW sets of the transformed grammar – document Section 2.	ind 2.2	2 pts
Design/implementation:		
Description/rationale of the overall structure of the solution and the roles of the individual components used in the applied solution – document Section 3.	ind 4.3	2 pts
Correct implementation of a top-down predictive parser following the grammar given in this handout.	ind 4.4	10 pts
Output of clear error messages (error description and location) in an **`outsyntaxerrors`** file.	ind 4.4	3 pts
Output of a derivation in an **`outderivation`** file.	ind 4.4	5 pts
Implementation of an error recovery mechanism.	ind 4.4	2 pts
Creation of a tree data structure as intermediate representation of the program, which is then output as a text representation into an **`outast`** file.	ind 4.4	6 pts
Completeness of test cases.	ind 4.4	12 pts
Use of tools:		
Description of tools/libraries/techniques used in the analysis/implementation. Description of other tools that might have been used. Justification of why the chosen tools were selected – document Section 4.	ind 5.2	2 pts
Successful/correct use of tools/libraries/techniques used in the analysis/implementation.	ind 5.1	3 pts
Total		**50 pts**