Report
Compiler Design
comp-6421

Assignment 2
Syntax Analyser

Professor: Dr. Joey Paquet

Neona Sheetal Pinto
40172626

Rules Presented by the Professor:

```
<START> ::= <prog>
<prog> ::= {{<structOrImplOrfunc>}}
<structOrImplOrFunc> ::= <structDecl> | <implDef> | <funcDef>
<structDecl> ::= 'struct' 'id' [['inherits' 'id' {{',' 'id'}}]] '{' {{<visibility> <memberDecl>}} '}' ';'
<implDef> ::= 'impl' 'id' '{' {{<funcDef>}} '}'
<funcDef> ::= <funcHead> <funcBody>
<visibility> ::= 'public' | 'private'
<memberDecl> ::= <funcDecl> | <varDecl>
<funcDecl> ::= <funcHead> ';'
<funcHead> ::= 'func' 'id' '(' <fParams> ')' '->' <returnType>
<funcBody> ::= '{' {{<varDeclOrStat>}} '}'
<varDeclOrStat> ::= <varDecl> | <statement>
<varDecl> ::= 'let' 'id' ':' <type> {{<arraySize>}} ';'
<statement> ::= <assignStat> ';'
| 'if' '(' <relExpr> ')' 'then' <statBlock> 'else' <statBlock> ';'
| 'while' '(' <relExpr> ')' <statBlock> ';'
| 'read' '(' <variable> ')' ';'
| 'write' '(' <expr> ')' ';'
| 'return' '(' <expr> ')' ';'
| <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'
| '(' <arithExpr> ')'
| 'not' <factor>
| <sign> <factor>
<variable> ::= {{<idnest>}} 'id' {{<indice>}}
<functionCall> ::= {{<idnest>}} 'id' '(' <aParams> ')'
<idnest> ::= 'id' {{<indice>}} '.'
| 'id' '(' <aParams> ')' '.'
<indice> ::= '[' <arithExpr> ']'
<arraySize> ::= '[' 'intNum' ']' | '[' ']'
<type> ::= 'integer' | 'float' | 'id'
```

<returnType> ::= <type> | 'void'
<fParams> ::= 'id' ':' <type> {{<arraySize>}} {{<fParamsTail>}} | EPSILON
<aParams> ::= <expr> {{<aParamsTail>}} | EPSILON
<fParamsTail> ::= ',' 'id' ':' <type> {{<arraySize>}}
<aParamsTail> ::= ',' <expr>
<assignOp> ::= '='
<relOp> ::= 'eq' | 'neq' | 'lt' | 'gt' | 'leq' | 'geq'
<addOp> ::= '+' | '-' | 'or'
<multOp> ::= '*' | '/' | 'and'

Transformed grammar into LL(1) grammar

1. Removed all optionality EBNF constructs, all zero-or-more repetition EBNF constructs, all left-recursive constructs using the "*Grammar tool*".
2. 2 left-recursive constructs can also be found and replaced by the above tool
    a. Original left-recursive rule: <term>

```
<term> ::= <term> <multOp> <factor>  | <factor>
```

Replaced by rules: <term> and <rightrec-term>

```
<term> ::= <factor> <rightrec-term>
<rightrec-term> ::= EPSILON
<rightrec-term> ::= <multOp> <factor> <rightrec-term>
```

    b. Original left-recursive rule: < arithExpr >

```
<arithExpr> ::= <arithExpr> <addOp> <term> | <term>
```

Replaced by rules: < arithExpr > and < rightrec-arithExpr >

```
<arithExpr> ::= <term> <rightrec-arithExpr>
<rightrec-arithExpr> ::= EPSILON
<rightrec-arithExpr> ::= <addOp> <term> <rightrec-arithExpr>
```

3. On using the output generated from the "*Grammar tool*" in the *ucalgary* format, the "*ucalgary* tool" helped in listing all the ambiguities. Using both the tools, to and forth, all the ambiguities were resolved manually.
4. The ambiguities and the solution are listed as follows:

```
a. <arraySize> ::= '[' 'intNum' ']'
   <arraySize> ::= '[' ']'


   Solution:
```

```
   <arraySize> ::= '[' <arraySize1>
   <arraySize1> ::=  'intNum' ']'
   <arraySize1> ::=  ']'
```

b.
```
   <expr> ::= <arithExpr>
   <expr> ::= <relExpr>
```

   Solution:

```
   <expr> ::= <arithExpr> <expr1>
   <expr1> ::= EPSILON
   <expr1> ::= <relOp> <arithExpr>
```

c.
```
   <factor> ::= <variable>
   <factor> ::= <functionCall>
```

   Solution:

```
   <factor> ::= 'id' <factor1>
   <factor1> ::= <variableTail> <factor2>
   <factor1> ::= <functionCallTail> <factor2>
   <factor2> ::= '.' 'id' <factor1>
   <factor2> ::= EPSILON
```

d.
```
   <idnest> ::= 'id' <rept-idnest1> '.'
   <idnest> ::= 'id' '(' <aParams> ')' '.'
```

   Solution

```
   <idnest> ::= 'id' <idnest1> '.'
   <idnest1> ::= <rept-idnest1>
   <idnest1> ::= '(' <aParams> ')'
```

e.
```
   <statement> ::= <assignStat> ';'
   <statement> ::= <functionCall> ';'
```

   Solution:

```
   <statement> ::= 'id' <statement1> ';'
   <statement1> ::= <variableTail> <statement2>
   <statement1> ::= <functionCallTail> <statement3>
   <statement2> ::= '.' 'id' <statement1>
   <statement2> ::= <assignOp> <expr>
   <statement3> ::= '.' 'id' <statement1>
```

```
   <statement3> ::= EPSILON
```

f.
```
   <variable> ::= <rept-variable0> 'id' <rept-variable2>
   <rept-variable0> ::= <idnest> <rept-variable0>
   <rept-variable0> ::= EPSILON
   <rept-variable2> ::= <indice> <rept-variable2>
   <rept-variable2> ::= EPSILON


   Solution:

   <variable> ::= 'id' <variable1>
   <variable1> ::= <variableTail> <variable2>
   <variable1> ::= <functionCallTail> <variable3>
   <variable2> ::= '.' 'id' <variable1>
   <variable2> ::= EPSILON
   <variable3> ::= '.' 'id' <variable1>
   <functionCallTail> ::= '(' <aParams> ')'
   <variableTail> ::= <indice> <variableTail>
   <variableTail> ::= EPSILON
```

g.
```
   <functionCall> ::= <rept-functionCall0> 'id' '(' <aParams> ')'

   Solution:

   Resolved in the statement and idnest1. Removed functionCall to keep
   it
```

5. The final transformed LL(1) Grammar

```
<START> ::= <prog>
<aParams> ::= <expr> <rept-aParams1>
<aParams> ::= EPSILON
<aParamsTail> ::= ',' <expr>
<addOp> ::= '+'
<addOp> ::= '-'
<addOp> ::= 'or'
<arithExpr> ::= <term> <rightrec-arithExpr>
<arraySize1> ::= 'intnum' ']'
<arraySize1> ::= ']'
<arraySize> ::= '[' <arraySize1>
<assignOp> ::= '='
<assignStat> ::= <variable> <assignOp> <expr>
<expr1> ::= EPSILON
<expr1> ::= <relOp> <arithExpr>
<expr> ::= <arithExpr> <expr1>
<fParams> ::= 'id' ':' <type> <rept-fParams3> <rept-fParams4>
<fParams> ::= EPSILON
<fParamsTail> ::= ',' 'id' ':' <type> <rept-fParamsTail4>
<factor1> ::= <variableTail> <factor2>
```

<factor1> ::= <functionCallTail> <factor2>
<factor2> ::= '.' 'id' <factor1>
<factor2> ::= EPSILON
<factor> ::= 'id' <factor1>
<factor> ::= 'intnum'
<factor> ::= 'floatnum'
<factor> ::= '(' <arithExpr> ')'
<factor> ::= 'not' <factor>
<factor> ::= <sign> <factor>
<funcBody> ::= '{' <rept-funcBody1> '}'
<funcDecl> ::= <funcHead> ';'
<funcDef> ::= <funcHead> <funcBody>
<funcHead> ::= 'func' 'id' '(' <fParams> ')' 'arrow' <returnType>
<functionCallTail> ::= '(' <aParams> ')'
<idnest1> ::= <rept-idnest1>
<idnest1> ::= '(' <aParams> ')'
<idnest> ::= 'id' <idnest1> '.'
<implDef> ::= 'impl' 'id' '{' <rept-implDef3> '}'
<indice> ::= '[' <arithExpr> ']'
<memberDecl> ::= <funcDecl>
<memberDecl> ::= <varDecl>
<multOp> ::= '*'
<multOp> ::= '/'
<multOp> ::= 'and'
<opt-structDecl2> ::= 'inherits' 'id' <rept-opt-structDecl22>
<opt-structDecl2> ::= EPSILON
<prog> ::= <rept-prog0>
<relExpr> ::= <arithExpr> <relOp> <arithExpr>
<relOp> ::= 'eq'
<relOp> ::= 'neq'
<relOp> ::= 'lt'
<relOp> ::= 'gt'
<relOp> ::= 'leq'
<relOp> ::= 'geq'
<rept-aParams1> ::= <aParamsTail> <rept-aParams1>
<rept-aParams1> ::= EPSILON
<rept-fParams3> ::= <arraySize> <rept-fParams3>
<rept-fParams3> ::= EPSILON
<rept-fParams4> ::= <fParamsTail> <rept-fParams4>
<rept-fParams4> ::= EPSILON
<rept-fParamsTail4> ::= <arraySize> <rept-fParamsTail4>
<rept-fParamsTail4> ::= EPSILON
<rept-funcBody1> ::= <varDeclOrStat> <rept-funcBody1>
<rept-funcBody1> ::= EPSILON
<rept-idnest1> ::= <indice> <rept-idnest1>
<rept-idnest1> ::= EPSILON
<rept-implDef3> ::= <funcDef> <rept-implDef3>
<rept-implDef3> ::= EPSILON
<rept-opt-structDecl22> ::= ',' 'id' <rept-opt-structDecl22>
<rept-opt-structDecl22> ::= EPSILON
<rept-prog0> ::= <structOrImplOrfunc> <rept-prog0>
<rept-prog0> ::= EPSILON
<rept-statBlock1> ::= <statement> <rept-statBlock1>
<rept-statBlock1> ::= EPSILON
<rept-structDecl4> ::= <visibility> <memberDecl> <rept-structDecl4>

<rept-structDecl4> ::= EPSILON
<rept-varDecl4> ::= <arraySize> <rept-varDecl4>
<rept-varDecl4> ::= EPSILON
<returnType> ::= <type>
<returnType> ::= 'void'
<rightrec-arithExpr> ::= EPSILON
<rightrec-arithExpr> ::= <addOp> <term> <rightrec-arithExpr>
<rightrec-term> ::= EPSILON
<rightrec-term> ::= <multOp> <factor> <rightrec-term>
<sign> ::= '+'
<sign> ::= '-'
<statBlock> ::= '{' <rept-statBlock1> '}'
<statBlock> ::= <statement>
<statBlock> ::= EPSILON
<statement1> ::= <variableTail> <statement2>
<statement1> ::= <functionCallTail> <statement3>
<statement2> ::= '.' 'id' <statement1>
<statement2> ::= <assignOp> <expr>
<statement3> ::= '.' 'id' <statement1>
<statement3> ::= EPSILON
<statement> ::= 'id' <statement1> ';'
<statement> ::= 'if' '(' <relExpr> ')' 'then' <statBlock> 'else' <statBlock> ';'
<statement> ::= 'while' '(' <relExpr> ')' <statBlock> ';'
<statement> ::= 'read' '(' <variable> ')' ';'
<statement> ::= 'write' '(' <expr> ')' ';'
<statement> ::= 'return' '(' <expr> ')' ';'
<structDecl> ::= 'struct' 'id' <opt-structDecl2> '{' <rept-structDecl4> '}' ';'
<structOrImplOrFunc> ::= <structDecl>
<structOrImplOrFunc> ::= <implDef>
<structOrImplOrFunc> ::= <funcDef>
<term> ::= <factor> <rightrec-term>
<type> ::= 'integer'
<type> ::= 'float'
<type> ::= 'id'
<varDecl> ::= 'let' 'id' ':' <type> <rept-varDecl4> ';'
<varDeclOrStat> ::= <varDecl>
<varDeclOrStat> ::= <statement>
<variable1> ::= <variableTail> <variable2>
<variable1> ::= <functionCallTail> <variable3>
<variable2> ::= '.' 'id' <variable1>
<variable2> ::= EPSILON
<variable3> ::= '.' 'id' <variable1>
<variable> ::= 'id' <variable1>
<variableTail> ::= <indice> <variableTail>
<variableTail> ::= EPSILON
<visibility> ::= 'public'
<visibility> ::= 'private'

6. The first and follow sets are then generated on the above LL(1) grammar using the "*ucalgary tool*" for all the non terminals.

nonterminal	first set	follow set	nullable	endable
ADDOP	plus minus or	id intnum floatnum lpar not plus minus	no	yes
ARRAYSIZE1	intnum rsqbr	semi lsqbr rpar comma	no	no
ASSIGNSTAT	id	∅	no	no
EXPR1	eq neq lt gt leq geq	semi comma rpar	yes	no
FACTOR2	dot	semi mult div and rsqbr eq neq lt gt leq geq plus minus or comma rpar	yes	no
FACTOR1	dot lpar lsqbr	semi mult div and rsqbr eq neq lt gt leq geq plus minus or comma rpar	yes	no
FUNCBODY	lcurbr	struct impl func rcurbr	no	no
FUNCHEAD	func	semi lcurbr	no	no
FPARAMS	id	rpar	yes	no
IDNEST	id	∅	no	no
IDNEST1	lpar lsqbr	dot	yes	no
APARAMS	id intnum floatnum lpar not plus minus	rpar	yes	no
FUNCDECL	func	rcurbr public private	no	no
ARITHEXPR	id intnum floatnum lpar not plus minus	semi rsqbr eq neq lt gt leq geq comma rpar	no	no
RELOP	eq neq lt gt leq geq	id intnum floatnum lpar not plus minus	no	no
APARAMSTAIL	comma	comma rpar	no	no
REPTAPARAMS	comma	rpar	yes	no

1				
REPTFPARAMS3	lsqbr	rpar comma	yes	no
FPARAMSTAIL	comma	comma rpar	no	no
REPTFPARAMS4	comma	rpar	yes	no
REPTFPARAMSTAIL4	lsqbr	comma rpar	yes	no
REPTFUNCBODY1	let id if while read write return	rcurbr	yes	no
REPTIDNEST1	lsqbr	dot	yes	no
REPTIMPLDEF3	func	rcurbr	yes	no
REPTOPTSTRUCTDECL22	comma	lcurbr	yes	no
REPTPROG0	struct impl func	∅	yes	no
MEMBERDECL	let func	rcurbr public private	no	no
ARRAYSIZE	lsqbr	semi lsqbr rpar comma	no	no
RETURNTYPE	void integer float id	semi lcurbr	no	no
RIGHTRECARITHEXPR	plus minus or	semi rsqbr eq neq lt gt leq geq comma rpar	yes	no
MULTOP	mult div and	id intnum floatnum lpar not plus minus	no	no
SIGN	plus minus	id intnum floatnum lpar not plus minus	no	no
START	struct impl func	∅	yes	no
PROG	struct impl func	∅	yes	no
REPTSTATBLOCK1	id if while read write return	rcurbr	yes	no
RELEXPR	id intnum floatnum lpar not	rpar	no	no

	plus minus			
STATBLOCK	lcurbr id if while read write return	else semi	yes	no
STATEMENT2	dot assign	semi	no	no
ASSIGNOP	assign	id intnum floatnum lpar not plus minus	no	no
EXPR	id intnum floatnum lpar not plus minus	semi comma rpar	no	no
STATEMENT3	dot	semi	yes	no
STATEMENT1	dot lpar lsqbr assign	semi	no	no
OPTSTRUCTDECL2	inherits	lcurbr	yes	no
REPTSTRUCTDECL4	public private	rcurbr	yes	no
STRUCTORIMPLORFUNC	struct impl func	struct impl func	no	no
STRUCTDECL	struct	struct impl func	no	no
IMPLDEF	impl	struct impl func	no	no
FUNCDEF	func	struct impl func rcurbr	no	no
TERM	id intnum floatnum lpar not plus minus	semi rsqbr eq neq lt gt leq geq plus minus or comma rpar	no	no
FACTOR	id intnum floatnum lpar not plus minus	semi mult div and rsqbr eq neq lt gt leq geq plus minus or comma rpar	no	no
RIGHTRECTERM	mult div and	semi rsqbr eq neq lt gt leq geq plus minus or comma rpar	yes	no
TYPE	integer float id	rpar lcurbr comma lsqbr semi	no	no

REPTVARDECL4	lsqbr	semi	yes	no
VARDECLORSTAT	let id if while read write return	let id if while read write return rcurbr	no	no
VARDECL	let	public private let id if while read write return rcurbr	no	no
STATEMENT	id if while read write return	else semi let id if while read write return rcurbr	no	no
VARIABLE	id	assign rpar	no	no
FUNCTIONCALLTAIL	lpar	semi mult div and dot rsqbr eq neq lt gt leq geq plus minus or comma rpar	no	no
VARIABLE2	dot	assign rpar	yes	no
VARIABLE3	dot	assign rpar	no	no
VARIABLE1	dot lpar lsqbr	assign rpar	yes	no
INDICE	lsqbr	semi mult div and lsqbr dot rsqbr eq neq lt gt leq geq assign plus minus or comma rpar	no	no
VARIABLETAIL	lsqbr	semi mult div and dot rsqbr eq neq lt gt leq geq assign plus minus or comma rpar	yes	no
VISIBILITY	public private	let func	no	no

Design

The design approach used for the syntax analyzer is the Table-driven predictive parsing as opposed to the recursive descent approach.
The parser table is generated using the online ”*ucalgary tool*” as per the parser algorithm. The parsing table which is created using the FIRST and FOLLOW sets tells the parser which right-hand-side of the rule to choose when the top of the stack is non-terminal.

The *Rule class* stores the rules in the form of the left-hand-side part and right-hand-side part.

The *Grammar class* separates information about the grammar with the parser. It utilizes the parsing table, FIRST, and FOLLOW sets created manually in the class that is derived from the use of tools. The files are placed in the resource folder. The “LL1.attrbute.grm” in the folder is the grammar rules.

The *SyntacticAnalyzer(parser) class* is responsible for parsing every test case, and writing to the output files (derivation output, error reporting). It contains a LexicalAnalyzer object which is implemented in Lexical Analyser from the assignment 1 for tokenizing the src file. A Grammar object is aggregated in it. Using the stack (parsing stack) is utilized to implement the parsing of the test case.

The *SyntacticAnalyzerDriver class* is a driver class to run the syntax analyzer which either opens every file(.src) in a folder or a single file(.src) and handles the task of parsing and generating output files.
Note: A more detailed explanation is given in the source code through comments.

Use of tools

1. *Grammartool.jar:* Tool provided by the professor to modify the grammar by removing the constraints and changing it to the “*ucalgary*” online tool form which will be used by the online tool.
2. *Ucalgary tool:* The ucalgary online tool is used in validating the LL(1) grammar and generating FIRST/FOLLOW sets and parsing tables. It helps to find all the ambiguities in the grammar.
3. *IntelliJ:* IDE platform for writing the source code. The IDE has smooth, efficient, and clever code suggestions which makes it easy to type. Debugging is much faster compared to other IDE’s.