COMP 442: Compiler Design

Assignment 2: Syntactical Analyzer

Done by:

Abdulla Alhaj Zin, 40013496

Date:

19/02/2020

To:

Dr. Joey Paquet

Section 1: Transformed Grammar

The grammar was provided originally by the instructor. The java developed grammar tool was used to remove all EBNF notations and to remove the left recursions. After careful analyzation using the two tools Ucalgary and AtoCC, the grammar was transformed by hand to become LL1. About 10-15 productions were replaced/fixed for issues such as, first set conflicts, nullable productions, and first-follow set clashes.

The fully transformed grammar into LL1 can be found at the following path of the project: ~/TruCompiler/Grammar/[filenames_in_different_formats].grm

Please note that all files under that folder are the same, they're just in different formats to be used with the different tools.

Section 2: First Set/ Follow Set

The first and follow sets that were followed in designing the syntactical analyzer were generated by the UCalgary tool. The reason for using that one was because it was clearer than the AtoCC ones.

Here are the two sets:

nonterminal	first set	follow set	nullable	endable
ADDOP	plus minus or	intnum floatnum lpar not id plus minus	no	yes
ARRAYSIZEVALUE	intnum	rsqbr	yes	no
ASSIGNSTAT	equal	semi	no	no
ASSIGNOP	equal	intnum floatnum lpar not id plus minus	no	no
AFTEREXPR	eq neq lt gt leq geq	comma rpar semi	yes	no
FUNCDECL	lpar	public private rcurbr	no	no
AFTERFUNCDECL	void integer float id	public private rcurbr	no	no
FUNCHEAD	id	do local	no	no
FPARAMS	integer float id	rpar	yes	no
AFTERFUNCHEAD	void integer float id	do local	no	no
APARAMS	intnum floatnum lpar not id plus minus	rpar	yes	no
AFTERIDNEST	dot lsqbr lpar	equal mult div and rsqbr eq neq lt gt leq geq plus minus or comma rpar semi	yes	no
FUNCTIONCALL	lpar	equal mult div and rsqbr eq neq lt gt leq geq plus minus or comma rpar semi	no	no
AFTERMEMBERDECL	id lpar	public private rcurbr	no	no
OPTCLASSDECL2	inherits	lcurbr	yes	no
OPTFUNCBODY0	local	do	yes	no
OPTFUNCHEAD0	coloncolon	lpar	yes	no
FUNCBODY	do local	semi	no	no
RELOP	eq neq lt gt leq geq	intnum floatnum lpar not id plus minus	no	no
APARAMSTAIL	comma	comma rpar	no	no
REPTAPARAMS1	comma	rpar	yes	no
MEMBERDECL	id integer float	public private rcurbr	no	no
REPTCLASSDECL4	public private	rcurbr	yes	no
REPTFPARAMS2	lsqbr	rpar comma	yes	no
FPARAMSTAIL	comma	comma rpar	no	no
REPTFPARAMS3	comma	rpar	yes	no
REPTFPARAMSTAIL3	lsqbr	comma rpar	yes	no
REPTFUNCBODY2	if while read write return id	end	yes	no
REPTIDNEST0	dot	equal mult div and rsqbr eq neq lt gt leq geq plus minus or comma rpar semi	yes	no
IDNEST2	lpar lsqbr	equal mult div and rsqbr eq neq lt gt leq geq plus minus or comma rpar semi	yes	no
REPTOPTCLASSDECL22	comma	lcurbr	yes	no
BEFOREVARDECLNOTID	integer float	id	no	no
REPTOPTFUNCBODY01	id integer float	do	yes	no
CLASSDECL	class	class id main	no	no
REPTPROG0	class	id main	yes	no
FUNCDEF	id	id main	no	no
REPTPROG1	id	main	yes	no
ARRAYSIZE	lsqbr	semi lsqbr rpar comma	no	no
INDICE	lsqbr	equal mult div and rsqbr eq neq lt gt leq geq plus minus or comma rpar semi lsqbr	no	no
RIGHTRECARITHEXPR	plus minus or	rsqbr eq neq lt gt leq geq comma rpar semi	yes	no
MULTOP	mult div and	intnum floatnum lpar not id plus minus	no	no
SIGN	plus minus	intnum floatnum lpar not id plus minus	no	no
START	main class id	∅	no	no
PROG	main class id	∅	no	no
REPTSTATBLOCK1	if while read write return id	end	yes	no
STATEMENT	if while read write return id	else semi if while read write return id end	no	no
AFTERSTATMENT	semi equal	else semi if while read write return id end	no	no
ARITHEXPR	intnum floatnum lpar not id plus minus	rsqbr eq neq lt gt leq geq comma rpar semi	no	no
RELEXPR	eq neq lt gt leq geq	comma rpar semi	no	no
STATBLOCK	do if while read write return id	else semi	yes	no
IDNEST0	id	equal mult div and rsqbr eq neq lt gt leq geq plus minus or comma rpar semi	no	no
EXPR	intnum floatnum lpar not id plus minus	comma rpar semi	no	no
TERM	intnum floatnum lpar not id plus minus	rsqbr eq neq lt gt leq geq plus minus or comma rpar semi	no	no
FACTOR	intnum floatnum lpar not id plus minus	mult div and rsqbr eq neq lt gt leq geq plus minus or comma rpar semi	no	no
RIGHTRECTERM	mult div and	rsqbr eq neq lt gt leq geq plus minus or comma rpar semi	yes	no
TYPE	integer float id	do local semi id	no	no
TYPEFLOAT	float	do local semi id	no	no
TYPEID	id	do local semi id	no	no
TYPEINT	integer	do local semi id	no	no
VARDECL	id	id integer float public private do rcurbr	no	no
REPTVARDECL2	lsqbr	semi	yes	no
VARIABLE	lsqbr	equal mult div and rsqbr eq neq lt gt leq geq plus minus or comma rpar semi	yes	no
REPTVARIABLE2	lsqbr	equal mult div and rsqbr eq neq lt gt leq geq plus minus or comma rpar semi	yes	no
VISIBILITY	public private	id integer float	no	no

Section 3: Design

The design of the syntactical analyzer was implemented in the predictive descent recursive method. All grammar productions were transformed into recursive method, following a one to one relation. All the programmatical grammar rules can be found under the syntactical analyzer folder in the class Rules.cs.
All productions are invoked from the single entry point method Start(), and then they recursively go through all the tokens that were passed from the Lexical Analyzer. A special tokens scanner was implemented to add special functionalities to facilitate enumerating over the tokens list throughout the recursive methods. Each method will create a Tree/Subtree/Node and add children to its parents using tail recursion. That is the main way used to build the Abstract Syntax Tree. The Abstract Syntax Tree is built on an underlying customized hand-written data structure. After the tree is built, the driver class takes care of generating the DOT file to have a nice visual graph of the parsed data. Both the derivations and the AST are written to the files with extensions .outderivations, and outast, respectively. Also, unit tests were implemented to test the critical parts of the analyzer. Moreover, the previous Lexical Analyzer had to be modified in its design to go recursively through blocks of code that do not have spaces in between them (That was a flaw in Assignment 1 that had to be fixed along with adding more test cases before implementing the Syntax Analyzer).

Section 4: Use of Tools

No external libraries were used outside of the built-in .Net core C# libraries. No special classes were used to handle any of the analysis; it was rather all done as a hand-written predictive descent recursive parser, and hand-written data structures.

Tools used to facilitate the implementation:

https://smlweb.cpsc.ucalgary.ca/ - A tool developed by UCalgary used to generate First and Follow Sets

http://www.atocc.de/cgi-bin/atocc/site.cgi?lang=de&site=main – A tool used to verify that the grammar is in LL1

Grammartool.jar - A tool developed by Dr. Joey Paquet used to remove left recursions, and generate .atocc and .ucalgary formatted files.

https://www.graphviz.org/ - A tool used to view DOT files

https://github.com/abdullahzen/TruCompiler - Github was used for source control