Alexandru Bara 40132235 **Comp 442 Assignment 3**

Section 1: Attribute Grammar.

I took professor Paquets grammar in order to have a reference point I know works. Then I added SA# everywhere in the grammar in order to represent the semantic actions to do. One thing to note is that SA1 means make a node with the token that has previously been read which is why it is so common. SA3 means to make a node with null (this helps with read until epsilon when creating a leaf). For SA4 to SA40, I always use popuntilepsilon since I wanted to remove the hassle of guessing how many pops we need for each action. This is very easy to do simply by surrounding the statements with SA3 and SA# aftewrwards. I also have all my semantic actions to be SA, which represents a nonterminal in Ucalgary tool which makes the parsing table work (not the case if we use terminal).

ADDOP -> plus SA1 .
ADDOP -> minus SA1 .
ADDOP -> or SA1 .

APARAMS -> SA3 EXPR REPTAPARAMS1 SA33 .
APARAMS -> .

APARAMSTAIL -> comma EXPR .

ARITHEXPR -> SA3 TERM SA3 RIGHTRECARITHEXPR SA12 SA9 .

ARRAYOROBJECT -> SA3 REPTARRAYSIZE SA4 .
ARRAYOROBJECT -> lpar APARAMS rpar .

ARRAYSIZE -> lsqbr ARRAYSIZE2 .

ARRAYSIZE2 -> intlit SA1 rsqbr .
ARRAYSIZE2 -> rsqbr SA2 .

ASSIGNOP -> equal SA1 .

CLASSDECL -> class id SA1 SA3 OPTINHERITS SA25 lcurbr SA3 REPTMEMBERDECL SA26 rcurbr semi .

CLASSDECLORFUNCDEF -> SA3 CLASSDECL SA22 .
CLASSDECLORFUNCDEF -> SA3 FUNCDEF SA21 .

EXPR -> SA3 ARITHEXPR EXPR2 SA8 .

EXPR2 -> RELOP ARITHEXPR .
EXPR2 -> .

FACTOR -> id SA1 FACTOR2 REPTVARIABLEORFUNCTIONCALL .
FACTOR -> intlit SA1 .
FACTOR -> floatlit SA1 .
FACTOR -> lpar ARITHEXPR rpar .
FACTOR -> not SA1 SA3 FACTOR SA11 .
FACTOR -> SIGN SA3 FACTOR SA11 .

FACTOR2 -> lpar APARAMS rpar .
FACTOR2 -> REPTIDNEST1 .

FPARAMS -> SA3 id SA1 colon TYPE SA3 REPTFPARAMS3 SA4 SA3 REPTFPARAMS4 SA30 SA31 .
FPARAMS -> .

FPARAMSTAIL -> comma id SA1 colon TYPE SA3 REPTFPARAMSTAIL4 SA4 .

FUNCBODY -> lcurbr SA3 REPTLOCALVARORSTAT SA20 rcurbr .

FUNCDEF -> FUNCHEAD FUNCBODY .

FUNCHEAD -> SA3 function id SA1 SA3 FUNCHEADTAIL SA32 SA31 .

FUNCHEADMEMBERTAIL -> id SA1 lpar FPARAMS rpar arrow RETURNTYPE .
FUNCHEADMEMBERTAIL -> constructorkeyword lpar FPARAMS rpar .

FUNCHEADTAIL -> sr FUNCHEADMEMBERTAIL .
FUNCHEADTAIL -> lpar FPARAMS rpar arrow RETURNTYPE .

IDNEST -> SA3 dot id SA1 IDNEST2 SA35 .

IDNEST2 -> lpar APARAMS rpar .
IDNEST2 -> REPTIDNEST1 .

INDICE -> lsqbr SA3 ARITHEXPR SA34 rsqbr .

LOCALVARDECL -> SA3 localvar SA1 id SA1 colon TYPE ARRAYOROBJECT semi SA6 .

LOCALVARORSTAT -> LOCALVARDECL .
LOCALVARORSTAT -> SA3 STATEMENT SA7 .

MEMBERDECL -> SA3 MEMBERFUNCDECL SA28 .

MEMBERDECL -> SA3 MEMBERVARDECL SA29 .

MEMBERFUNCDECL -> MEMBERFUNCHEAD semi .

MEMBERFUNCHEAD -> function SA1 id SA1 colon lpar FPARAMS rpar arrow RETURNTYPE
.
MEMBERFUNCHEAD -> constructorkeyword colon lpar FPARAMS rpar .

MEMBERVARDECL -> attribute id SA1 colon TYPE SA3 REPTARRAYSIZE SA4 semi .

MULTOP -> mult SA1 .
MULTOP -> div SA1 .
MULTOP -> and SA1 .

OPTINHERITS -> isa id SA1 REPTINHERITSLIST .
OPTINHERITS -> .

PROG -> SA3 REPTPROG0 SA24 .

RELEXPR -> SA3 ARITHEXPR RELOP ARITHEXPR SA14 .

RELOP -> eq SA1 .
RELOP -> neq SA1 .
RELOP -> lt SA1 .
RELOP -> gt SA1 .
RELOP -> leq SA1 .
RELOP -> geq SA1 .

REPTAPARAMS1 -> APARAMSTAIL REPTAPARAMS1 .
REPTAPARAMS1 -> .

REPTARRAYSIZE -> ARRAYSIZE REPTARRAYSIZE .
REPTARRAYSIZE -> .

REPTFPARAMS3 -> ARRAYSIZE REPTFPARAMS3 .
REPTFPARAMS3 -> .

REPTFPARAMS4 -> FPARAMSTAIL REPTFPARAMS4 .
REPTFPARAMS4 -> .

REPTFPARAMSTAIL4 -> ARRAYSIZE REPTFPARAMSTAIL4 .
REPTFPARAMSTAIL4 -> .

REPTIDNEST1 -> INDICE REPTIDNEST1 .

REPTIDNEST1 -> .

REPTINHERITSLIST -> comma id REPTINHERITSLIST .
REPTINHERITSLIST -> .

REPTLOCALVARORSTAT -> LOCALVARORSTAT REPTLOCALVARORSTAT .
REPTLOCALVARORSTAT -> .

REPTMEMBERDECL -> VISIBILITY SA3 MEMBERDECL SA27 REPTMEMBERDECL .
REPTMEMBERDECL -> .

REPTPROG0 -> CLASSDECLORFUNCDEF REPTPROG0 .
REPTPROG0 -> .

REPTSTATBLOCK1 -> SA3 STATEMENT SA7 REPTSTATBLOCK1 .
REPTSTATBLOCK1 -> .

REPTVARIABLE -> VARIDNEST REPTVARIABLE .
REPTVARIABLE -> .

REPTVARIABLEORFUNCTIONCALL -> IDNEST REPTVARIABLEORFUNCTIONCALL .
REPTVARIABLEORFUNCTIONCALL -> .

RETURNTYPE -> TYPE SA1 .
RETURNTYPE -> void SA1 .

RIGHTRECARITHEXPR -> ADDOP TERM RIGHTRECARITHEXPR .
RIGHTRECARITHEXPR -> .

RIGHTRECTERM -> MULTOP SA3 FACTOR SA11 RIGHTRECTERM .
RIGHTRECTERM -> .

SIGN -> plus SA1 .
SIGN -> minus SA1 .

START -> SA3 PROG SA23 eof .

STATBLOCK -> lcurbr SA3 REPTSTATBLOCK1 SA7 rcurbr .
STATBLOCK -> SA3 STATEMENT SA7 .
STATBLOCK -> .

STATEMENT -> id SA1 SA3 STATEMENTIDNEST SA36 semi .
STATEMENT -> SA3 if lpar RELEXPR rpar then STATBLOCK else STATBLOCK semi SA15 .
STATEMENT -> SA3 while lpar RELEXPR rpar STATBLOCK semi SA16 .

STATEMENT -> SA3 read lpar VARIABLE rpar semi SA17 .
STATEMENT -> SA3 write lpar EXPR rpar semi SA18 .
STATEMENT -> SA3 return lpar EXPR rpar semi SA19 .

STATEMENTIDNEST -> dot id SA1 SA3 STATEMENTIDNEST SA36 .
STATEMENTIDNEST -> lpar APARAMS rpar STATEMENTIDNEST2 .
STATEMENTIDNEST -> INDICE REPTIDNEST1 STATEMENTIDNEST3 .
STATEMENTIDNEST -> ASSIGNOP EXPR .

STATEMENTIDNEST2 -> .
STATEMENTIDNEST2 -> dot id SA1 SA3 STATEMENTIDNEST SA36 .

STATEMENTIDNEST3 -> ASSIGNOP EXPR .
STATEMENTIDNEST3 -> dot id SA1 SA3 STATEMENTIDNEST SA36 .

TERM -> SA3 SA3 FACTOR SA11 SA3 RIGHTRECTERM SA13 SA10 .

TYPE -> integer SA1.
TYPE -> float SA1.
TYPE -> id SA1.

VARIABLE -> SA3 id SA1 VARIABLE2 SA5 .

VARIABLE2 -> REPTIDNEST1 REPTVARIABLE .
VARIABLE2 -> lpar APARAMS rpar VARIDNEST .

VARIDNEST -> SA3 dot id SA1 VARIDNEST2 SA37 .

VARIDNEST2 -> lpar APARAMS rpar VARIDNEST .
VARIDNEST2 -> REPTIDNEST1 .

VISIBILITY -> public SA1 .
VISIBILITY -> private SA1 .

SA1 -> .
SA2 -> .
SA3 -> .
SA4 -> .
SA5 -> .
SA6 -> .
SA7 -> .
SA8 -> .
SA9 -> .
SA10 -> .

SA11 -> .
SA12 -> .
SA13 -> .
SA14 -> .
SA15 -> .
SA16 -> .
SA17 -> .
SA18 -> .
SA19 -> .
SA20 -> .
SA21 -> .
SA22 -> .
SA23 -> .
SA24 -> .
SA25 -> .
SA26 -> .
SA27 -> .
SA28 -> .
SA29 -> .
SA30 -> .
SA31 -> .
SA32 -> .
SA33 -> .
SA34 -> .
SA35 -> .
SA36 -> .
SA37 -> .
SA38 -> .
SA39 -> .
SA40 -> .

Here is a brief description of all of my semantic actions. This is taken from my code.
The base idea is that we can make a node, an empty token, null or a family. Whenever we make a family, it will pop until epsilon and it will make the node with all the children. I also give it a string for the make family which represents the semantical concept that is being built.

```
            case "SA1" -> AST.makeNode(previousToken);
            case "SA2" -> AST.makeNode(new Token(TokenType.EMPTY, "epsilon",
token.getLocation()));
            case "SA3" -> AST.makeNull();
            case "SA4" -> AST.makeFamily("array Size", -1);
            case "SA5" -> AST.makeFamily("variable", -1);
            case "SA6" -> AST.makeFamily("local variable", -1);
            case "SA7" -> AST.makeFamily("statement", -1);
```

```
case "SA8" -> AST.makeFamily("expression", -1);
case "SA9" -> AST.makeFamily("arithmetic expression", -1);
case "SA10" -> AST.makeFamily("term", -1);
case "SA11" -> AST.makeFamily("factor", -1);
case "SA12" -> AST.makeFamily("recursive arithmetic expression", -1);
case "SA13" -> AST.makeFamily("recursive term", -1);
case "SA14" -> AST.makeFamily("rel expression", -1);
case "SA15" -> AST.makeFamily("if statement", -1);
case "SA16" -> AST.makeFamily("while loop", -1);
case "SA17" -> AST.makeFamily("read", -1);
case "SA18" -> AST.makeFamily("write", -1);
case "SA19" -> AST.makeFamily("return", -1);
case "SA20" -> AST.makeFamily("function body", -1);
case "SA21" -> AST.makeFamily("function definition", -1);
case "SA22" -> AST.makeFamily("class definition", -1);
case "SA23" -> AST.makeFamily("start", -1);
case "SA24" -> AST.makeFamily("program", -1);
case "SA25" -> AST.makeFamily("inheritance", -1);
case "SA26" -> AST.makeFamily("rept member declaration", -1);
case "SA27" -> AST.makeFamily("member declaration", -1);
case "SA28" -> AST.makeFamily("member function declaration", -1);
case "SA29" -> AST.makeFamily("member variable declaration", -1);
case "SA30" -> AST.makeFamily("function params", -1);
case "SA31" -> AST.makeFamily("function head", -1);
case "SA32" -> AST.makeFamily("function tail", -1);
case "SA33" -> AST.makeFamily("argument params", -1);
case "SA34" -> AST.makeFamily("indice", -1);
case "SA35" -> AST.makeFamily("idnest", -1);
case "SA36" -> AST.makeFamily("statement idnest", -1);
case "SA37" -> AST.makeFamily("variable idnest", -1);
```

Section 2: Design

The first part was to enhance the parser. Now I have an if statement in my parser that checks if the top is a semantic action. If it is, then it goes in code that analyzes the semantic action and does the proper action. There is a big switch statement that will make the nodes, add nulls and make families based on the semantic action read.

There is also a class I added called AST. This is a class that can be seen as a node. It has the value of the node, it has the parent of the node and a list of all the childrens. It also has a static

stack that keeps all of the semantic nodes. I have all the proper functions to make nodes, update the death, make families and print the tree inside of this class. I then make static calls from my parser to the AST class and it takes care of all the semantic actions by pushing and popping from the static ast stack.

Section 3: Use of Tools

I used AtoCC in order to validate if my grammar is in LL1 format.
I used Ucalgary smlweb in order to create the parsing table and to get the first and follow sets. I tried to use the Ucalgary tool in order to fix my language, but I find that doing it by hand worked better.
I used google sheets to take the table from Ucalgary and make it in a csv file.
I used Joey Paquet's grammar tool to remove some ambiguities and left recursion.
Finally, I used the built in tools from Java like arraylist, hashmap and stacks in order to store my code.