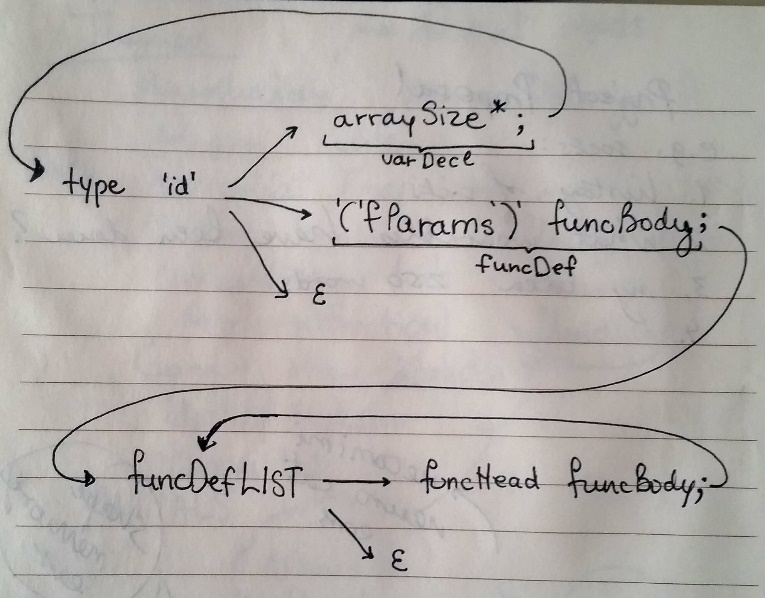
**Grammar**

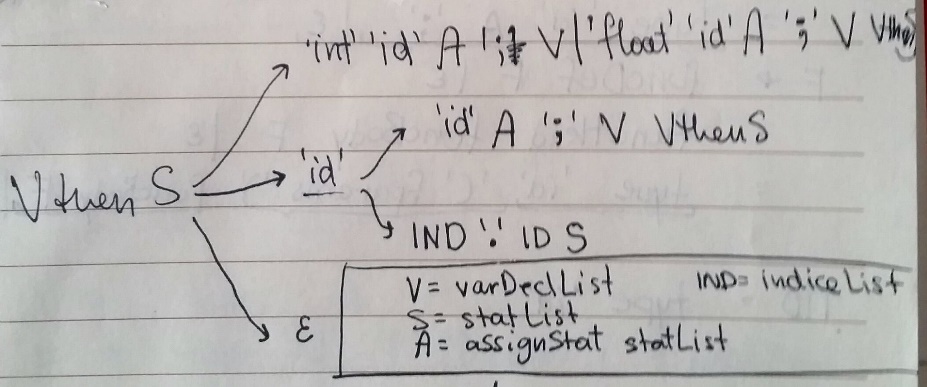
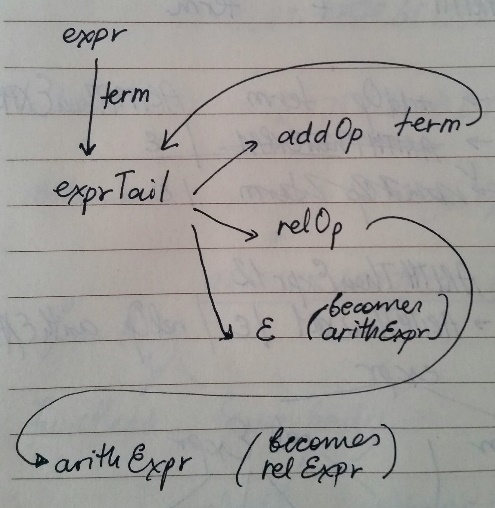
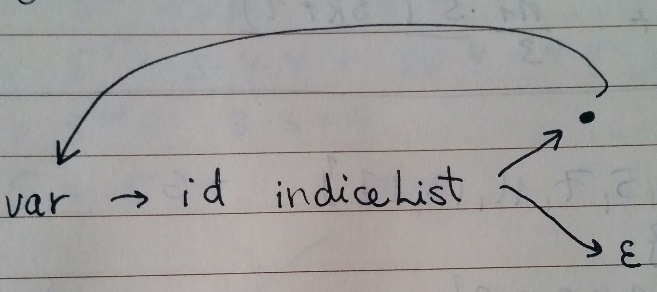
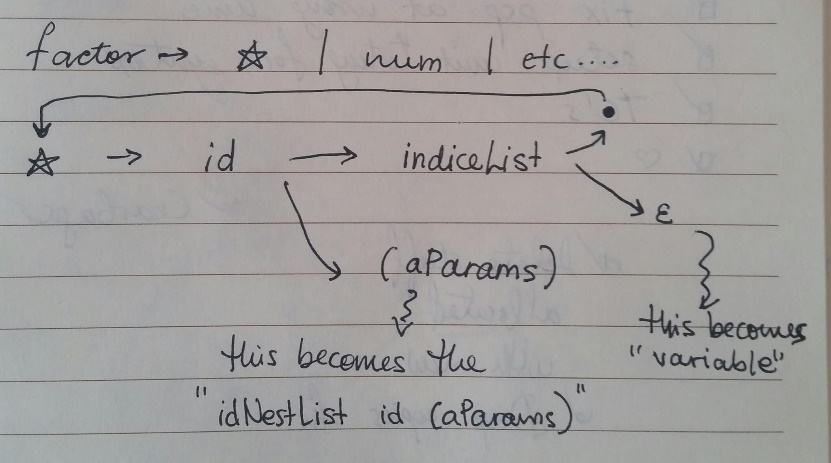
The meaning of the initial grammar was kept, I only transformed it to make it LL1.

Grammar transformations

* All \* notations: replaced by creating “List” productions instead:
  + Example: prog -> classDecl\* progBody
  + Correction:
    - prog -> classDeclList progBody
    - classDeclList -> classDecl classDeclList | EPSILON
* classDecl -> 'class' 'id' '{' varDeclList funcDefList '}' ';'
  + Problem: both varDecl and funcDef can start with **type id**
  + Correction: factored **type id** out:
    - classDecl -> 'class' 'id' '{' varThenFuncList '}' ';'
    - typeId -> type 'id'
    - varThenFuncList -> typeId varThenFuncList1 | EPSILON
    - varThenFuncList1 -> arraySizeList ';' varThenFuncList | '(' fParams ')' funcBody ';' funcDefList
    - DFA visualisation:
    - 
* funcBody -> '{' varDeclList statList '}'
  + both varDeclList and statList can start with **type id**
  + solved in a similar way to the one above:
  + separated assign statement from not assign statement:
    - statement -> assignStat ';' | statementNotAssign
    - statementNotAssign -> 'if' '(' expr ')' 'then' statBlock 'else' statBlock ';'

| 'for' '(' type 'id' assignOp expr ';' relExpr ';' assignStat ')' statBlock ';'

| 'get' '(' variable ')' ';' | 'put' '(' expr ')' ';'| 'return' '(' expr ')' ';'

* + factored out:
    - funcBody -> '{' varThenStatList '}'
    - varThenStatList -> 'int' 'id' arraySizeList ';' varThenStatList | 'float' 'id' arraySizeList ';' varThenStatList
    - | 'id' varThenStatList1 | statementNotAssign statementList | EPSILON
    - varThenStatList1 -> 'id' arraySizeList ';' varThenStatList | assignStatTail
    - assignStatTail -> indiceList assignStatTail1
    - assignStatTail1 -> assignOp expr ';' statementList | '.' assignStatTail
    - DFA visualisation:
    - 
* expr -> arithExpr | relExpr
  + relExpr starts with arithExpr! Ambiguity
  + built and implemented following DFA to distinguish between arithExpr and relExpr:
  + 
* Variable -> idNestList id indiceList
  + idNestList can start with id or can be epsilon. So when parser sees “id”, it doesn’t know if it’s from idNestList, or if idNestList is empty, and it’s the id that follows
  + DFA solution:
  + 
  + variable -> idNestListThenIdThenIndiceList
  + idNestListThenIdThenIndiceList -> 'id' indiceList idNestListThenIdThenIndiceList1
  + idNestListThenIdThenIndiceList1 -> '.' idNestListThenIdThenIndiceList | EPSILON
* factor -> variable | idnestList id (aParams) | …
  + Problem: both variable and idnestList start with the same thing
  + DFA solution:
  + 
  + Factor -> idNestListIdThenIndiceListOrAParams | num | etc..
  + idNestListIdThenIndiceListOrAParams -> 'id' idNestListIdThenIndiceListOrAParams1
  + idNestListIdThenIndiceListOrAParams1 -> indiceList idNestListIdThenIndiceListOrAParams2 | '(' aParams ')'
  + idNestListIdThenIndiceListOrAParams2 -> '.' idNestListIdThenIndiceListOrAParams | EPSILON
* Other grammar change
  + Changed num to numInt | numFloat, to match the corresponding tokens outputted by the scanner

**Grammar, FIRST and FOLLOW sets**

Please find them in the Documentation folder.

**Solution structure**

GSymbol: abstract class for nonterminal and terminal grammar symbols. Allows us to put objects of GNonTerminal and GTerminal types into the same data structure and access their member functions easily through polymorphism.

GNonTerminal: extends GSymbol

* Has a NonTerminalType (e.g. classDeclList)

GTerminal: extends GSymbol

* Has a TerminalType (e.g. RETURN)
* ‘$’ is a terminal
* EPSILON is a terminal

Parser: a table-driven parser

* Has a Scanner, implemented in Assignment1
* Has a ParseTable
* Has a parsing stack, implemented and used according to algorithm shown in class
  + The stack contains GSymbols
* Has methods parse(), skipErrors() and inverseRHSMultiplePush(), implemented according to algorithm shown in class

ParseTable: a parsing table generated from my LL1 grammar

* Has a **std::list<GSymbol\*> \*rules**, an array of rules for parsing, generated from my LL1 rammar
  + Each rule is a list of GSymbols, where the leftmost one (first one) is the LHS of the rule, and the remaining ones are the RHS of the rule
  + To find out which rule to use, I check the table, which provides the index of that rule in the array
* Has a 2D array: **table**. This is the actual parsing table. One accesses data in the table by providing two parameters: the GNonTerminal that is currently on the top of the parsing stack, and the GTerminal that corresponds to the token that we just scanned from the input file. The corresponding row and column give the index of the rule that needs to be applied.
* Has a Map<string, int> that maps terminal types to indices in the **table**
* Has initializing methods to automatically initialize the **table,** the **rules**, and the mapping of terminals to indices from text files.
* Has getRule() and getRuleNo() public methods that the Parser will use when parsing

Driver: a small driver class that launches the parser with different test files.

**Tools used**

* <http://hackingoff.com/>: an online tool that takes a LL1 grammar as an input and outputs FIRST set, FOLLOW set, rules and parsing table.
* Git, GitHub for version control
* VisualStudio for creating the project solution and debugging

**Techniques used**

* Technique for transforming left recursion seen in class
* Factorization technique seen in class
* Table-driven parsing algorithm seen in class
  + Chose to do table-driven, because it is easier to change and to maintain, if I found an error or need to add another feature, I just have to modify the table

**Test cases**

All test cases can be found at **TestFiles/Syntax**

1. For some non-terminals, we will check what happens in three cases:
   * Next input token is from the FIRST set, this should give no errors
   * Next input token is from the FOLLOW set and FIRST set contains epsilon, this should give no errors
   * Next input token is from neither the FIRST nor the FOLLOW set, this should give an error
     + Will only test with one of all the possible invalid input tokens, for simplicity
   * Note: all **txt** files can be found at **TestFiles/Syntax**

|  |  |  |  |
| --- | --- | --- | --- |
| Current top of parsing stack | Input tokens from the FIRST set (correct) + names of files that tests them | If FIRST set contains epsilon: input tokens from the FOLLOW set + names of files that tests them | One of tokens from neither the FIRST nor the FOLLOW set (error) + names of file that tests it |
| prog | **‘class’** (full\_valid\_program.txt), **‘program’** (good1.txt) | **--** | **‘return’** (bad1.txt) |
| classDeclList | **‘class’** (full\_valid\_program.txt) | **‘program’** (good1.txt) | **‘]’** (bad2.txt) |
| classDecl | **‘class’** (full\_valid\_program.txt), | -- | Impossible, because bad input will get rejected at the classDeclList symbol |
| typeId | **'int'** (full\_valid\_program.txt), **'float'** (full\_valid\_program.txt), **'id'** (good3.txt) | -- | **‘-‘** (bad3.txt) |
| varThenFuncList | **'int'** (full\_valid\_program.txt), **'float'** (full\_valid\_program.txt), **'id'** (good3.txt) | ‘}’ (good4.txt) | ‘)’ (bad4.txt) |

1. Nested indiceLists and other messy things:
   * B = x[y][y+1][z[1+x]]; (good5.txt)
   * a.b.c.d.e.f[g.h.i.j[0][a]][1] = 5; (good5.txt)
   * different arithmetic and relational operators inside a for loop (good6.txt)
   * 1 < 2 > 1 (bad5.txt)