Assignment 2

Harley McPhee 27003226

Grammars Changes

Change Remove *

Reason: * is not supported, so we need to remove all * and make the non-terminals preceding the * recursive and give them an empty string to stop, or if this non-terminal is used elsewhere without a * we will need to introduce a new non-terminal to allow multiple elements preceding the *, and keep the original non-terminal preceding * as it is

OLD:

1. <prog> → <classDecl>*<progBody>
2. <classDecl> → class id {<varDecl>*<funcDef>*};
3. <varDecl> → <type>id<arraySize>*;
4. <arraySize> → [integer]
5. <funcDef> → <funcHead><funcBody>;
6. <fParams> → <type>id<arraySize>*<fParamsTail>* | ε
7. <fParamsTail> → ,<type>id<arraySize>*
8. <funcBody> → {<varDecl>*<statement>*}
9. <varDecl> → <type>id<arraySize>*;
10. Add statementLst because statement is used without <statement>*
11. <variable> → <idnest>*id<indice>*
12. <idnest> → id<indice>*.
13. <indice> → [<arithExpr>]
14. <factor> → <variable>
    | <idnest>*id(<aParams>)
    | num
    | (<arithExpr>)
    | not<factor>
    | <sign><factor>
15. <aParams> → <expr><aParamsTail>* | ε
16. <aParamsTail>→ ,<expr>
17. <statBlock> → {<statement>*} | <statement> | ε

CHG:

1. <prog> → <classDecl> <progBody>
2. <classDecl> → class id { <varDecl> <funcDef> } ; <classDecl> | ε
3. <varDecl> → <type> id <arraySize> ;
4. <arraySize> → [integer] <arraySize> | ε
5. <funcDef> → <funcHead> <funcBody> ; <funcDef> | ε
6. <fParams> → <type> id <arraySize> <fParamsTail> | ε

7. <fParamsTail> → , <type> id <arraySize> <fParamsTail> | ε
8. <funcBody> → { <varDecl> <statementLst> }
9. <varDecl> → <type> id <arraySize> ; <varDecl> | ε
10. <statementLst → <statement> <statementLst> | ε
11. <variable> → <idnest> id <indiceLst>
12. <idnest> → id <indiceLst> . <idnest> | ε
13. <indice> → [<arithExpr>]
14. <factor> → <variable>
    | <idnest> id (<aParams>)
    | num
    | (<arithExpr>)
    | not <factor>
    | <sign> <factor>
15. <aParams> → <expr> <aParamsTail> | ε
16. <aParamsTail>→ ,<expr> <aParamsTail> | ε
17. <statBlock> → { <statementLst> } | <statement> | ε
18. <indiceLst> → <indice> <indiceLst> | ε

Change Remove left-recursion

Reason: Left recursion causes non-determinism in a left to right parser. To remove the left recursion we add a new non-terminal for each RHS that has left recursion.

OLD:
1. <arithExpr> → <arithExpr> <addOp> <term> | <term>
2. Add <arithExprD> to remove left recursion from <arithExpr>
3. <term> → <term><multOp><factor> | <factor>
4. Add <termD> to remove left recursion from <term>

CHG:
1. <arithExpr> → <term><arithExprD>
2. <arithExprD> → <addOp><term><arithExprD> | ε
3. <term> → <factor><termD>
4. <termD> → <multOp><factor><termD> | ε

Change Allow <varDecl> or <funcDef> before or after each other in <classDecl>

Allow <classDecl> to have either <varDecl> and <funcDef> before or after each other and remove ambiguity that'll be created by it because either <varDecl> and <funcDef> both have id in their first sets

Reason: A variable can technically be declared after a function inside a class, the grammar as it is does not allow that, it requires that variables be declared before, also <classDecl> will then contain an ambiguity, the first set of <varDecl> and <funcDef> contains (int, float, id), which leads to non determinism in <classBody>

OLD:

1. <classDecl> → class id { <varDecl><funcDef>};<classDecl> | ε

CHG:

1. <classDecl> → class id {<classBody>};<classDecl> | ε
2. <classBody> → <classInDecl><classBody> | ε
3. <classInDecl> → <type>id<postTypeId>
4. <postTypeId> → <arraySize>; | (<fParams>)<funcBody>

Change Remove ambiguity caused by <idnest> and <variable> in <factor>

Reason:
These production cause non-determinism because <idnest> and <variable> first set will contain id and empty string, and both are the first non-terminals in one of factors RHS.

OLD:

```
1. <variable>     →  <idnest> id <indiceLst>
2. <idnest>       → id <indiceLst> . <idnest> | ε
3. <factor>       → <variable>
                  | <idnest> id (<aParams>)
                  | num
                  | ( <arithExpr> )
                  | not <factor>
                  | <sign> <factor>
```

CHG:

```
1. <variable>     → id <variableIndice>
2. <idnest>       → . id <indice> <idnest> | ε
3. <factor>       → id <factorVarOrFunc> // Replace the two first rules with this
                  |  num
```

| (<arithExpr>)
| not <factor>
| <sign> <factor>

4. <factorVarOrFunc> → <factorVarArray> | <varOrFuncIdNest> | ε
5. <factorVarArray> → <indice> <indiceLst> <factorVarArrayNestId>
6. <factorVarArrayNestId> → . id <factorVarOrFunc> | ε
7. <varOrFuncIdNest> → (<aParams>) | . id <factorVarOrFunc>

More explanation:
<factorVarOrFunc> is used to determine if the statement is going to be a function call, or just a variable access. The reason we need to separate them is because a function call can not be accessed via an array access, ex: id.id[integer]() is not allowed.

Change terminal num in <factor> to non-terminal and add a new <num> non-terminal

Reason: My scanner does not have a num token, it has only integer or float so <num> will contain <float> or <integer>

OLD:

1. <factor> → id <factorVarOrFunc> // Replace the two first rules with this
   | num
   | (<arithExpr>)
   | not <factor>
   | <sign> <factor>

CHG:

1. <factor> → id <factorVarOrFunc> // Replace the two first rules with this
   | num
   | (<arithExpr>)
   | not <factor>
   | <sign> <factor>
2. <num> → <integer> | <float>

Change Allow <varDecl> or <statementLst> before or after each other in <funcBody>

Allow <funcBody> to have either <varDecl> and <statementLst> before or after each other and remove ambiguity that'll be created by it because both <varDecl> and <statementLst> have id in their first sets

Reason: A variable assignment, control flow statements, loops, and other types of statements can be used before a variable declaration, as long as the variable isn't used before its declaration but that's the purpose of the semantic analyzer to verify so the syntax analyzer should allow either to be before or after one another.

OLD:

1. <funcBody> → { <varDecl> <statementLst> }
2. <statement> → <assignStat> ;
   | if (<expr>) then <statBlock> else <statBlock> ;
   | for (<type> id <assignOp> <expr> ; <relExpr> ; <assignStat>) <statBlock> ;
   | get (<variable>) ;
   | put (<expr>) ;
   | return (<expr>) ;

New:

1. <funcBody> → { <funcInBody> }
2. <funcInBodyLst> → <funcInBody> <funcInBodyLst> | ε
3. <funcInBody> → id <varOrStat>
   | <statementRes>
   | <numType> id <arraySize> ; // Add for int or float variable declarations

4. <varOrStat> → id <arraySize> ; | <indiceLst> <idnest> <assignOp>
5. <statement> → <assignStat> ; | <statementRes>
6. <statementRes> → if (<expr>) then <statBlock> else <statBlock>;
   | for (<type> id <assignOp> <expr> ; <relExpr> ; <assignStat>) <statBlock> ;
   | get (<variable>) ;
   | put (<expr>) ;
   | return (<expr>) ;

More Explanation:

<varOrStat> is added because a variable declaration or assignment operation both can begin with an id, so we will need to look further ahead to see what it is.
<statement> was causing ambiguity, because it contains <assignStat> so it was removed from <funcInBody> and <funcInBody> used the new <statementRes> to use the reserved words statements, <statementRes> was also added to <statement> so whatever rules that were using <statement> would still have access to the reserved word statements.

<varDecl> is no longer used anywhere so it is removed

Change <arithExpr> and <relExpr> in <expr> both derive <term>

Reason**:** These production cause non-determinism because <arithExpr> and <relExpr> both derive <term> so they will have the same first set

OLD:

1. <expr> → <arithExpr> | <relExpr>

CHG

1. <expr> → <term> <relOrAri>
2. <relOrAri> → <relOp> <arithExpr> | <arithExprD>

Change <statementRes> RHS if production to not use <statBlock>

Reason: An if statement doesn't always need an else and when it doesn't have an else the <statBlock> must have a semicolon after it.

OLD:

1. <statementRes> → if(<expr>)then<statBlock>else<statBlock>;

CHG

1. <statementRes> → if(<expr>)then<statBlock>else<statBlock>;
2. <statBlock> → { <statementLst> } ; | <statement> | EPSILON
3. <statThenBlock> → { <statementLst> } <statIfElseBlock> | <statement> <statElseBlock>
4. <statIfElseBlock> → else <statBlock> | ;
5. <statElseBlock> → else <statBlock> | EPSILON

Final Grammar

Please note that my tokens are a bit different from the terminals displayed in the grammar, I find that the terminals given are easier to process so I will leave them as is but a mapping of the tokens to terminals will be found at the end of this report. The syntax parser will use the tokens, but output the derivations in the terminals below.

```
<prog>                  → <classDeclLst> <progBody>
<classDeclLst>          → <classDecl> <classDeclLst> | ε
<classDecl>             → class id { <classBody> } ;
<classBody>             → <classInDecl> <classBody> | ε
<classInDecl>           → <type> id <postTypeId> ;
<postTypeId>            → <arraySize> | ( <fParams> ) <funcBody>
<progBody>              → program <funcBody> ; <funcDefLst>
<funcHead>              → <type> id ( <fParams> )
<funcDefLst>            → <funcDef> <funcDefLst> | ε
<funcDef>               → <funcHead> <funcBody> ;
<funcBody>              → { <funcInBodyLst> }
<funcInBodyLst>         → <funcInBody> <funcInBodyLst> | ε
<funcInBody>            → id <varOrStat> | <statementRes> | <numType> id <arraySize> ;
<varOrStat>             → id <arraySize> ; | <indiceLst> <idnest> <assignOp> <expr> ;
<statementLst>          → <statement> <statementLst> | ε
<statement>             → <assignStat> ; | <statementRes>
<statementRes>          → if ( <expr> ) then <statThenBlock>
                        | for ( <type> id <assignOp> <expr> ; <relExpr> ; <assignStat> )
                <statBlock>
                        | get ( <variable> ) ;
                        | put ( <expr> ) ;
                        | return ( <expr> ) ;
<statThenBlock>         →  { <statementLst> } <statIfElseBlock> | <statement> <statElseBlock>
<statIfElseBlock>       →  else <statBlock>  | ;
<statElseBlock>         →  else <statBlock> | ε
<assignStat>            → <variable> <assignOp> <expr>
<statBlock>             → { <statementLst> } | <statement> | ε
<expr>                  → <term> <relOrAri>
<relOrAri>              → <relOp> <arithExpr> | <arithExpr'>
<relExpr>               → <arithExpr> <relOp> <arithExpr>
<arithExpr>             → <term> <arithExpr'>
<arithExpr'>            → <addOp> <term> <arithExpr'> | ε
<sign>                  → + | -
<term>                  → <factor> <term'>
<term'>                 → <multOp> <factor> <term'> | ε
<factor>                → id <factorVarOrFunc> | <num> | ( <arithExpr> ) | not <factor> | <sign>
<factor>
<factorVarOrFunc>       → <factorVarArray> | <varOrFuncIdNest> | ε
<factorVarArray>        → <indice> <indiceLst> <factorVarArrayNestId>
<factorVarArrayNestId> → . id <factorVarOrFunc> | ε
<varOrFuncIdNest>       →  ( <aParams> ) | . id <factorVarOrFunc>
<variable>              → id <variableIndice>
```

<variableIndice>	→ <indiceLst> <idnest>
<idnest>	→ . id <indiceLst> <idnest> | ε
<indice>	→ [<arithExpr>]
<indiceLst>	→ <indice> <indiceLst> | ε
<arraySize>	→ [integer] <arraySize> | ε
<type>	→ <numType> | id
<numType>	→ int | float
<fParams>	→ <type> id <arraySize> <fParamsTail> | ε
<aParams>	→ <expr> <aParamsTail> | ε
<fParamsTail>	→ , <type> id <arraySize> <fParamsTail> | ε
<aParamsTail>	→ , <expr> <aParamsTail> | ε
<assignOp>	→ =
<relOp>	→ == | <> | < | > | <= | >=
<addOp>	→ + | - | or
<multOp>	→ * | / | and
<num>	→ integer | float

Operators and additional lexical conventions

<assignOp>	→ =
<relOp>	→ == | <> | < | > | <= | >=
<addOp>	→ + | - | or
<multOp>	→ * | / | and

First and follow sets

Non-Terminals	First Set	Follow Set
<prog>	{class, program}	{ $ }
<classDeclLst>	{class, ε}	{ program }
<classDecl>	{class}	{class, program}
<classBody>	{int, id, float, ε}	{ } }
<classInDecl>	{int, id, float}	{ } , int, id, float}
<postTypeId>	{ [, (, ε}	{ ; }

<progBody>	{program}	{ $ }
<funcHead>	{int, id, float}	{ { }
<funcDefLst>	{int, id, float, ε}	{ $ }
<funcDef>	{int, id, float}	{ id, int, float, $ }
<funcBody>	{ { }	{ ; }
<funcInBodyLst>	{id, int, float, if, for, get, put, return, ε}	{ } }
<funcInBody>	{id, int, float, if, for, get, put, return}	{ id, int, float, if, for, get, put, return, } }
<varOrStat>	{ id, [, =, . }	{ id, if, for, get, put, return, int, float, } }
<statementLst>	{id, if, for, get, put, return, ε}	{ } }
<statement>	{id, if, for, get, put, return}	{ id, if, for, get, put, return, else, ;, } }
<statementRes>	{if, for, get, put, return}	{ id, if, for, get, put, return, else, ;, }, int, float }
<assignStat>	{id}	{), ;)
<statBlock>	{ { , id, if, for, get, put, return, ε }	{ ;, id, if, for, get, put, return, else, }}
<statThenBlock>	{ { , id, if, for, get, put, return, ε }	{ else }
<statThenBlock>	{ else, ; }	
<expr>	{id, integer, float, (, not, +, - }	{ , ,), ; }
<relOrAri>	{ ==, <>, <, >, <=, >=, +, -, or, ε}	{ , ,), ; }
<relExpr>	{(, id, not, float, integer, +, -}	{ ; }
<arithExpr>	{(, id, not, float, integer, +, -}	{],), ==, <>, <, >, <=, >=, ;, , }
<arithExpr'>	{+, -, or, ε}	{],), ==, <>, <, >, <=,

		>=, ;, , }
<sign>	{+ , -}	{ id, (, not, integer, float, +, - }
<term>	{id, integer, float, (, not, +, - }	{ +, -, or, ==, <>, <, >, <=, >=,],), ;, , }
<termD>	{*, /, and, ε}	{ +, -, or, ==, <>, <, >, <=, >=,],), ;, , }
<factor>	{id, integer, float, (, not, +, - }	{ *, /, and, +, -, or, ==, <>, <, >, <=, >=,],), ;, , }
<factorVarOrFunc>	{ [, ., (ε }	{ *, /, and, +, -, or, ==, <>, <, >, <=, >=,],), ;, , }
<factorVarArray>	{ [}	{ *, /, and, +, -, or, ==, <>, <, >, <=, >=,],), ;, , }
<factorVarArrayNestId>	{ . , ε }	{ *, /, and, +, -, or, ==, <>, <, >, <=, >=,],), ;, , }
<varOrFuncIdNest>	{ (, . , ε }	{ *, /, and, +, -, or, ==, <>, <, >, <=, >=,],), ;, , }
<variable>	{id}	{ =,) }
<variableIndice>	{ [, ., ε}	{ =,) }
<idnest>	{ ., ε}	{ =,) }
<indice>	{ [}	{ [, ., =, *, /, and, +, -, or, ==, <>, <, >, <=, >=,],), ;, , }
<indiceLst>	{ [, ε }	{., =,), *, /, and, +, -, or, ==, <>, <, >, <=, >=,], ;, ,}
<arraySize>	{ [, ε}	{ ,, ;,) }
<type>	{int, float, id}	{ id }

<numType>	{int, float}	{ id }
<fParams>	{ int, float, id, ε}	{) }
<aParams>	{id, integer, float, (, not, +, -, ε }	{) }
<fParamsTail>	{ , , ε}	{) }
<aParamsTail>	{ , , ε}	{) }
<assignOp>	{ = }	{ id, (, not, integer, float, +, -, if, for, get, put, return, int, } }
<relOp>	{ ==, <>, <, >, <=, >= }	{ id, (, not, integer, float, +, - }
<addOp>	{ +, -, or }	{ id, (, not, integer, float, +, - }
<multOp>	{ *, /, and}	{ id, (, not, integer, float, +, - }
<num>	{integer, float}	{ *, /, and, +, -, or, ==, <>, <, >, <=, >=,],), ;, , }

Token Mapping with terminals

Tokens	Lexemes
ID	letter alphanum*
INUM	nonzero digit* | 0
FNUM	integer fraction
SCMT	// alphanum* \n
CMT	/* (alphanum | \n)* */
EQUAL	=

EQUIV	==
NOTEQ	<>
GTEQ	>=
GT	>
LTEQ	<=
LT	<
ADD	+
SUB	-
MULTI	*
DASH	/
OPENPARA	(
CLOSEPARA	)
OPENBRA	[
CLOSEBRA	]
OPENCURL	{
CLOSECURL	}
INT	int
FLOAT	float
AND	and
NOT	not
OR	or
IF	if
THEN	then
ELSE	else

FOR	for
CLASS	class
GET	get
PUT	put
RETURN	return

Error Recovery

The syntax parser error recovery/reporter will indicate that an error has occurred when an unexpected token is received. At the beginning of a production procedure, skip_errors is called to check if the lookahead value matches a value in the first and follow set, but the follow set is used only if the first set contains and epsilon. The skip_errors will continue to skip over tokens until it finds a token that’s valid. Besides skip_errors, match will also report errors when an unexpected token is received. Match does not always recover from the error though and will stop the parsing from continuing, I added error handling for match in a few places though to check if there is a missing semicolon or a missing token, but this wasn’t done everywhere as there’s simply too many cases to cover.

Program Overview

The syntax parser was written in C++11, the main component is the SyntaxParser class, this class can be instantiated and is a recursive descent top down parser, this class contains all the procedures for the non terminals in the grammar.

Tools, Libraries and Techniques used

Tools

The following tools were used to help develop the lexical analyzer.

CLion is a cross platform IDE for c++, I used this instead of Visual Studio or Xcode as I work both on a Mac and Windows so I needed a cross platform solution.
https://www.jetbrains.com/clion/

Mingw allowed me to use gcc, a C++ compiler on Windows, this helped with my cross platform solution as gcc is a cross platform compiler for C++.
http://mingw.org/

CMake let me specify how to compile my program and the multiple files required to compile it.
https://cmake.org/

Atocc kFG program was used to help removed ambiguities in the language
http://www.atocc.de/cgi-bin/atocc/site.cgi?lang=de&site=main

Libraries

Google test was used to help test the lexical analyzer, I chose it because it’s one of the most popular c++ unit testing frameworks and was easy to setup and use.
https://github.com/google/googletest

Various C++ standard libraries were used to help develop the program, the following includes were used in my program

- #include <iostream>
- #include <vector>
- #include <map>
- #include <fstream>
- #include <algorithm>
- #include <string>

Techniques

The technique used was the Recursive descent predictive parser.