Assignment 3

Harley McPhee 27003226

Grammar with Semantic actions

<prog> → "**createGlobalTable**" <classDeclLst> <progBody>
<classDeclLst> → <classDecl> <classDeclLst> | ε
<classDecl> → class id "**createClassEntryAndTable**" { <classBody> } ;
<classBody> → <classInDecl> <classBody> | ε
<classInDecl> → <type> id <postTypeId>
<postTypeId> → <arraySize> ; "**createVariableEntry**"
|"**createFunctionClassEntryAndFunctionTable**" (<fParams>)
<funcBody> ;
| ; "**createVariableEntry**"
<progBody> → program "**createProgramEntryAndTable**" <funcBody> ; <funcDefLst>
<funcHead> → <type> id "**createFunctionEntryAndTable**" (<fParams>)
<funcDefLst> → <funcDef> <funcDefLst> | ε
<funcDef> → <funcHead> <funcBody> ;
<funcBody> → { <funcInBodyLst> }
<funcInBodyLst> → <funcInBody> <funcInBodyLst> | ε
<funcInBody> → id <varOrStat>
| <statementRes>
| <numType> id <arraySize> ; "**createVariableEntry**"
<varOrStat> → id <arraySize> ; "**createVariableEntry**"
| <indiceLst> <idnest> <assignOp> <expr> ;
<statementLst> → <statement> <statementLst> | ε
<statement> → <assignStat> ; | <statementRes>
<statementRes> → if (<expr>) then <statThenBlock>
| for (<type> id "**createVariableEntry**" <assignOp> <expr> ;
<relExpr> ; <assignStat>) <statBlock> "createParameterEntry"
| 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> "**createParameterEntry**" <fParamsTail> | ε
<aParams> → <expr> <aParamsTail> | ε
<fParamsTail> → , <type> id <arraySize> "**createParameterEntry**" <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

Data Structures

SymbolTable

The symbol table is what is used to store all the function, variables and class declarations. It stores these in a list of SymbolRecords which is described below. The SymbolTable also contains all the functions to create new records of various type that are embedded in the grammars rules, I put them there because the parser already had all the procedures so it contained a lot of code already so moving them to SymbolTable would help prevent more bloating of the parser file, also the functions were mainly creation and manipulating records and then inserting them into its own list of SymbolRecords so it made sense to put it in SymbolTable.

The properties and function headers are found below:

```
bool second_pass_;
Token* current_token;
vector<string> errors_;
vector<SymbolRecord*> symbol_records_;
SymbolTable* parent_symbol_table_;
SymbolRecord* current_symbol_record_;

SymbolTable();
bool create_class_entry_and_table(string kind, string type, string name);
bool create_program_entry_and_table();
bool create_function_entry_and_table(SymbolRecord** record);
void print();
bool create_variable_entry(SymbolRecord* record);
bool create_function_class_entry_and_function_table(SymbolRecord **record);
bool create_parameter_entry(SymbolRecord* record);
bool insert(SymbolRecord *record);
void report_error_to_highest_symbol_table(string error_message);
SymbolRecord* search(string name);
void set_second_pass(bool second_pass);
void set_properly_declared(SymbolRecord *record);
```

SymbolRecord

The symbol record are what the SymbolTable data structure will have a list of, these are where all the functions, variables, and class information is stored. I decided against using inheritance and just having one type so I could avoid having to cast my data to various types to access the necessary properties, so some SymbolRecords will simply have properties that will

be empty because they have no use for it but are there for other types of SymbolRecords, it may not be the most elegant solution but it's a lot simpler to implement and saved me time as I used the other approach with the Scanners tokens and it was quite time consuming to get it right. The SymbolRecord also has a SymbolTable property for nested SymbolTables for storing classes and functions local variable/functions.

The properties are found below:

```
SymbolRecord(string kind, string type, string name);
SymbolRecord();
string kind_;
string type_;
vector<string> function_parameters;
string name_;
string structure_;
SymbolTable* symbol_table_;
bool properly_declared_;
vector<int> array_sizes;
string address;

bool set_kind(string kind);
bool set_type(string type);
bool set_name(string name);
bool set_structure(string structure);
bool append_to_type(string type);
bool add_function_parameter(string type);
bool generate_function_type();
bool add_array_size(IntegerToken integer_token);
```

Error Recovery

The parser error recovery/reporter will indicate that an error has occurred when a variable, class or function is being declared when the name used to declare it has already been declared and processed in the same scope, the parser will simply ignore this declaration and move on to the next one. Errors will also be reported when a variable is declared using an unknown type that has not been declared above in the class section, to recover from this error the program simply continues on.

Program Overview

The parser was written in C++11, the main component is the Parser 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, within the grammar rules and semantic actions that are used to create the symbol table. There is also the option to enable a two pass parser,

the first pass symbol generates the symbol table and will leave any variables declared flagged as not properly declared because the symbol table isn't complete at the time the variable itself was entered into the table so it's possible that its type hasn't been processed yet. The two pass parser will then verify that all flagged variables have their declared types in the symbol table and if not report an error. The two pass parser will also be used in the next assignment for type checking. The Parser has a global symbol table property that is of type SymbolTable, this is the main symbol table and contains all the records, and the records contain nested symbol tables too.

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 with semantic actions to create the symbol table, a second pass parser was also used to verify that various types were properly declared.