Project Phase 1: Front End

Design Internals:

tiger grammar: tiger.g4
main function code: tiger.cpp
symbol table generation code: SymbolTable.cpp
semantic checking code: SemanticCheck.cpp
IR generation code: IRGeneration.cpp
other helper function code and jar package: scope.cpp, antlr-4.7-complete.jar
Generated parser code is under **generated** directory

SymbolTable.cpp :

At first, it define a globalScope , and current scope is globalScope. When entering LET rule, it will create new scope , this scope will be the child of current scope and current scope is its parent. Then, it will take the place of original current scope and becomes the new current scope (curScope = newScope). After we leave the LET, current scope will become its parent (curScope = curScope->parent).

If we have declaration-segment in current scope, we insert the name and attributes of type, var and function into the map table of current scope, so that we can use them to do semantic check later.

In defining symbol table, we can do some simple semantic check, including duplicated definition, have return statement in no-return type function or no return statement in return type function, the type not defined.

SemanticCheck.cpp:

We do a lot of work in semantic check.

1, in symbol table, if we meet a function declaration, we record its parameter information. Here, we use this information to detect error related with parameter type and number.

2, in symbol table, we also record type and dimension for variable. Here, for operators in expression, we can detect problems like if their types are not identical, if aggregate operation happens, if assign float to int, e.t.

3, some others errors are also checked : indexer of array is not int; associated comparison(==) operation; second operator of EXP is not int; break not within loop; id not declared; id is not a function in rule <stat> -> <opt-prefix> id (<expr-list>), e.t.

IRGeneration.cpp:

We do not use listener in IR generation, instead we use our own mechanism, which is similar to visitor. For instructions that will generate IR code, we write their corresponding generation function, like genExpr(), genStatSeq(), genStat(), genFuncDec(), genVarDec(). In these functions, temporaries, jump_labels and assembly code will be generated and push into code vector. When needed, we call these functions and ensure the correctness of code sequence.

Moreover, to make sure that we execute only executing at levels that represent an actual instruction and avoid generation duplicated IR, we record the node we have entered. If we enter the same node later, we will find we have entered before, thus no IR will be generated.

scope.cpp:

It defines the data structure needed in a scope to record all kinds of information, including its parent scope, its children scope, the var defined in this scope.

How to Build:

By running **make**, the generated antlr code will be in ./generated and executable file **tiger** will be generated in current directory.

Run Examples:

If the test case **testcase1.tiger** is in current directory, you can run this command, the second parameter is test file name:

```
./tiger ./testcase1.tiger
```

If the test case **testcase2.tiger** is in other directory, you need to add its path:

```
./tiger ./test/case/path/testcase2.tiger
```

In current directory, we put many testcases under **project1_test_code_example** file, to use these test files, you can run command:

```
./tiger ./project1_test_code_example/testcase.tiger
```

All information (<tokentype, token>, the sequence of token type, symbol table, generated IR) will be printed to console. If we have lexical or parse error, the symbol table creation, semantic check and IR generation will not happen. If we have semantic error, IR generation will not happen and IR will not be printed.