Contents

Project Phase 2 Report

Abhijit Singh asingh811@gatech.edu

June 2021

1 About this project

This report describes the implementation of "phase 2" of the compiler. This includes generating symbol tables, semantic checking and IR code generation conforming to the project requirements for the tiger programming language

2 Preliminary notes

Tiger.g4 grammar file was cut down and changed to be able to define more visitors

3 Code hierarchy

The following classes define the main "actors" of this program 1) SymbolTableGenerator : generates symbols AND queues a bunch of semantic enforcement tasks as lambdas.

2) SemanticEnforcer : simply runs the list of semantic enforcement lambdas inherited from SymbolTableGenerator.

3) IRGenerator : mangles names on scope inherited from SymbolTableGenerator and generates IR code

The following data storage classes were used to help with code generation 1) Scope: defines a scope. Contains locally defined symbols (functions, vars, typedefs) , and statements in top-down order

2) Stat: a statement class (example ifstatement, return statement, break statement , function invocation etc)

Several classes inherit the "Stat" class such as StatAssignment, StatIfStmt etc

3) Symbol: defines symbols in the code. Other classes inherit the virtual class symbol and define more specific types (such as functions, variables, typedefs etc)

4) astNode : An abstract syntax tree node represents a mathematical equation. It contains left, right astnodes if it is not a leaf, and an "Operator" enum. which tells us that

```
currentAST node = left node OP right node
```

4 SymbolTableGenerator

Inherits the TigerBaseListener. Upon walking the input code file, it generates scopes, symbols and statements

It also queues semantic error checking lambdas under ErrorCheckingTask::tasks

Most importantly, it gathers any and all necessary data for IR generation (so the class name is a misnomer)

While walking through the code, it maintains a stack of active scopes,

4.1 Enter Listeners

The entry point listeners are used to create 1) variables, functions and typedefs (i.e. Symbols) in the current scope (i.e. scope on top of the static stack called scopeStack)

2) To create new subscopes when entering an eligible statement (such as an if statement or an else statement or a sub scope)

examples are:

```
enterTiger_program()
enterType_declaration()
enterVar_declaration()
enterFunct()
enterStat_seq_while()
enterStat_seq_for()
```

etc

4.2 Exit listeners

The vast majority of implementation is done on the "exit" portion of the listeners. The advantage of the exit point is that it gives us a nice post-order access to code syntax

The following are a few of several "exit listeners" virtual function overrides of the BaseListener:

```
exitPow_op_expr()
```

if there is a power operator in the given expression, it takes the last two ast nodes and connects them together into a third ast node and a power operator

Similar operator listeners exist for plus, minus, and, or etc and these consume as well as generate further AST nodes

```
exitFncall_stat()
```

Take the ast nodes from the ast stack matching the number of params of the function call and save them as params to be evaluated before execution of the function call Finally, add this statement to the scope (to be used later during IR generation)

Similar "statement exit" listeners exist for : loops (capture the condition AST, and scopes for inside the loops)

4.3 Error Handling queue

As the SymbolTableGenerator navigates the code in a pre-order post-order hybrid fashions (as mentioned above), it queues up tasks for semantic checking for later use

Take for example the following code snippet from SymbolTable.h

```
ErrorCheckingTask::tasks.push_back([aliasName,lineNum,charPos,back,sym](){
        std::string scopeName;
        auto res = back->getSymbol(aliasName, scopeName);
        if(res==nullptr){
                printErrorAndExit(lineNum,charPos, IRERROR_NO_SUCH_TYPE);
        }
        else if(res->getType()!=TYPE_TYPEDEF){
                printErrorAndExit(lineNum,charPos, IRERROR_NOT_ASSIGNABLE_TYPE);
        }
        else{
                sym->_returnSymbol=aliasName;//+scopeName;
        }
});
```

Here we queued an anonymous function that will check if a given ID corresponds to a Type definition. If not, it throws NO SUCH TYPE error. If the symbol exists but doesn't define a type, it throws NOT ASSIGNABLE TYPE error.

SymbolTable.cpp defines all the error messages corresponding to the different IRERRORs

4.4 Design of the symbol table

The symbol table enumerates

1) all type definitions
2) all vars and corresponding types (also whether it is static or not)
3) function definitions with (optional) return type

All of the above are indented by hierarchy of their respective scopes

The following is a sample symbol table

```
_Scope0:
    gen_random, func
        _Scope3:
            inc, var, int
            mod, var, int
            mult, var, int
            seed, var, int
            _Scope4:
                _t3, var, int, 1
                cur, var, int
                i, var, int
```

```
            prev, var, int
            _Scope5:
                _t4, var, int
                _t5, var, int
main, func, int
    _Scope6:
        _t10, var, int, 3
        _t11, var, int, 0
        _t6, var, int, 6
        _t7, var, int, 5
        _t8, var, int, 7
        _t9, var, int, 3
modulo, func, int
    _Scope1:
        a, var, int
        b, var, int
        _Scope2:
            _t0, var, int
            _t1, var, int
            _t2, var, int
            r, var, int
size, static, int, 0
```

5 SemanticEnforcer

The vast majority of work has already been done for the semantic enforcer by the SymbolTableGenerator. All it does is inherit the top level scope of the SymbolTableGenerator class. It does semantic Enforcement.

By simply executing the queued anonymous functions created by the SymbolTableGenerator, it checks for validity of various semantics (such as type equivalencies, duplicate vars, variable existence etc)

6 IRGenerator

Upon passing semantic enforcement, this inherits the top level scope from the other classes. It does the following

Name mangling: It calls on the top level scope to mangle names in a top down fashion. It maintains a dictionary of variable name -> stack of scopes and does pre-order reassignment of variable names. It pushes and pops scopes as we enter and exit scopes as part of various loop statements, functions etc

For example, if you are in a scope25 and want to access variable "a", and the most recent parent scope 16 defines it, the mangled name of the variable will be a-scope16 This way we can re use variable names in different scopes

Upon completing name mangling, it generates IR code. Each statement, symbol, scope and AST node handles its own logic to generate IR code. A scope will call its entire list of statements one at a time to generate their respective IR code.

The statement may call the IR code generation of the corresponding AST nodes (example assignments), as well as sub-scopes of the statements (example if-else conditions).

7 Limitations and room for improvement

1) Semantic checking beyond just the tests provided was done, however, there is still some scope for improvement. For example, I am not checking if the symbol of the function exists

2) all in-code constants are assigned to temporary variables. This is a waste of register space and if this wasn't time critical, I would correct this. For example, consider the following simple code snippet c = 1 +2 will be assigned as follows temp1 = 1 temp2 = 2

c= temp1 + temp2

3) When I started this assignment, I was determined to use clean design patterns. But due to time constraints I had to give up on them. You'll see that parts of the code use nice getters and setters. For the rest, I just made every member a public variable. Other terrible code problems include excessive use of static variables (Also I did not free any memory so expect memory leaks in this program).

4) Apologies for the loaded single header files and many commented out lines of code.

8 Acknowledgements

1) The professor's lectures- watching them carefully was key to doing anything for this assignment

2) Official class reference book- useful for more niche stuff

3) Piazza- quick TA responses were very much appreciated

4) The gallon of Redbull that got me through this and my full time job