TigerWorks

Part 1: Symbol Table Generation

The symbol table is divided into two HashMaps, one for each namespace. The first hashmap is for functions, and maps Strings to SymbolTableEntrys. Essentially, this is a map from function names to data structures representing Functions and their attributes.

The second map is for types and variables. Because types and variables can exist in many scopes, and there are typically many scopes in a tiger program, the second map maps Scope objects to maps that map Strings to SymbolTableEntrys. In essence, we first take the Scope of a variable or type and get the miniature Symbol table for that scope. Then we look at the String representing the variable or type's name, and get the data structure representing the variable or type and it's attributes.

If a variable or type is not found in the current scope, looking at the parent scope of your current scope is what would occur. This is because you can see things in more scopes with a more global presence. However, if you have definitions in several scopes, you always choose the most local scope.

When the symbol table is first created, we populate it with built in primitive types (int and fixedpt), as well as built-in standard library functions as described in the tiger specification document.

In our grammar, we had to modify several rules in order to properly insert functions, types, and variables into the symbol table. However, in the end, we were able to properly insert everything we needed to.

Upon a successful program parse, the compiler will print out the symbol table.

Part 2: Semantic Analysis

Semantic Analysis was primarily done through editing the grammar and checking key attributes of variables and functions. It required a big overhaul in specific parts of the grammar, especially the rewrite of the expression rules. Several new classes were created to aid in semantic checks.

The first class created was NameSpaceException. It was used to allow the Symbol Table to communicate a namespace collision to the parser. This is seen in SymbolTable.java, in the method put().

The second class created was arguably more important. We created a class in which the objects could represent the type and a few other attributes, called SemanticObject. This class radically simplified type checking when combined with the method evaluateType in TigerParser.java. It becomes easier to ensure types are correct

when you can pass around all the necessary attributes of variables in one object instead of passing around lots of individual attributes in a messy fashion.

Evaluating Type equivalence was simple: Name-Scope equivalence determined equality in the majority of cases. In addition, there was type promotion when an int and a fixedpt interacted.

The first thing done was evaluate the type of constants, values (as described in the specification), and expressions, as well as whether or not they were booleans. That information was passed to assignment expressions to confirm whether or not it was legal to assign the value of an expression to a variable, as well as conditionals to make sure that the expression being checked was a boolean.

The next thing we did was check to see if a function's return type agreed with the type of the value returned, as well as check to see if a function's arguments fit the definition of its parameters.

Upon a successful semantic analysis, the program will make no indication that anything is wrong. That is to say, it will continue to generating IR Code silently.

Part 3: IR Code Generation

In order to generate the intermediate code for a compiled Tiger program, we create an alternate grammar for traversing a generated AST and creating an intermediate code representation based on the contents of the AST. This secondary treewalking grammar, TigerTreeWalk.g, depends on a few external Java classes we wrote: Operator.java, an Enum listing the various binary operator; IRGenerator.java, a wrapper around a List object which stores lines of code in the intermediate language; and IRTranslator.java, a class which takes in inputs for the various intermediate instructions and generates the intermediate code for a certain instruction given its inputs.

In order to support code generation, the TigerTreeWalk grammar contains many rules which pass up return values relating to their data contents. At higher levels, these rules are used to create instructions. For example, expressions representing the usage of the various binary operators pass up information on which operators are inside of the expression, and the operator information is used to map a particular Tiger operation to a proper intermediate instruction. Throughout the process of code generation, TigerTreeWalk assigns temporary variables from the set {t0, t1, ... , ti} where i increases with each temporary variable assignment regardless of scope. Temporary variables are to be stored in the symbol table so that information on their types and other metadata related to them may be used in later stages of the code generation.

The output of the IR code is set to write to the file ir.tigir by default. The format of the final IR code is a 4-address code which generally has 4 parameter slots per an instruction except for in the case of function calls where it has a variable number of parameters per instruction.

Building and Running

Assuming you have antlr in your classpath,
To build:

```
java org.antlr.Tool Tiger.g
javac TigerCompiler.java
```

To compile:

```
java TigerCompiler <filename>
```

If your program has any issues, the compiler will complain and pinpoint exactly where it has an issue with your code. Otherwise, it will print out its symbol table and generate IR Code.