CS 4240 Project Phase 2

David Benas, Sean Collins, Casey Evanish

November 12, 2014

Run/Compile Instructions

Instructions on how to run the code / compile and such will go here.

1 Symbol Table

As a group, we decided to have variables, types, and functions as symbols in our table. Variables would relate to an initialization value, types would have a backing type (obviously), and functions would have a return type. At first, we wanted functions to have their params in the symbol table as well, but we realized that since functions cannot share names, this was unecessary.

Ultimately, these three entries in the symbol table share the fact that they have a scope and an ID. We represented our Symbol Table as a Hashmap with a String key and a SymbolTableEntry(containing the parent scope and a String ID) value.

An example of how we add symbols into our table is as follows [from tiger.g]:

```
type[String id]
  : base_type {
    if ($base_type.text.equals("int")) {
      symbolTable.put(new TypeSymbolTableEntry(CURRENT_SCOPE,strip(id),
         TigerPrimitive.INT));
    } else if ($base_type.text.equals("fixedpt")) {
      symbolTable.put(new TypeSymbolTableEntry(CURRENT_SCOPE,strip(id),
         TigerPrimitive.FIXEDPT));
    }
  }
```

In order to deal with scoping, we first instantiate a scope object. Whenever we encounter a new block, we create a new scope object, while setting its parent to be the previous scope. The scope objects contain their parent scope, ID, and a counter for number of children. Essentially, we can think of our scopes as an upside-down tree, with each parent scope branching down to many children scopes. One of the ways we actually use scoping can be seen in output\SymbolTable.java 's put() method:

```
for (int index = 0; index < resultVarList.size(); index++) {
     while (curScope != null) {
        if (curScope.equals(resultVarList.get(index))) {
        // Found value in this or parent Scope! Reassign it.
        resultVarList.get(index).setValue(addVar.getValue());
        return;
        }
        curScope = curScope.getParent();
     }
}
```

This is important because we need to allow for variables with the same ID to exist in different scopes.

2 Semantic Checking

The semantic check and IR code generation section was a large portion of the project, taking us a long time and requiring extensive grammar revisions in order to make it work both correctly and effectively. The purpose of the semantic checking portion of the code is the verify that whatever was passed into the compiler is not only grammatically correct, but also syntactically correct. That is, we want to make sure that the words passed in make sense together.

A special modification we made to perform this task is the fact that we grouped every operation into an object to handle typing. This made the task of typing much easier than any other way that we discussed. While the majority of the code is fairly well documented or self-explanitory within the \tiger.g file, we'll examine the special case of handling when one type is int and the other type is fixedpt [a typing problem we encountered]:

```
(From tiger.g, public OperationObject getTyping()):
if (var1.isConstant() && var2.isConstant()) {
   newisConst = true;
   if (!(var1.getType().equals(var2.getType()))) {
      newType = symbolTable.getFixedPtType();
   } else {
      newType = var1.getType();
   }
   return new OperationObject(newisConst,newType,newId);
```

Code Break.

Continued On Next Page.

```
} else {
   newisConst = false;
   if ((var1.getType().getId().equals("int") && var2.getType().getId().equals("fixedpt"))
        ||(var1.getType().getId().equals("fixedpt")
        && var2.getType().getId().equals("int"))) {
      newType = symbolTable.getFixedPtType();
      return new OperationObject(newisConst,newType, newId);
   } else {
      if (!(var1.getType().equals(var2.getType()))) {
         System.out.println("Typing error between "+ var1.getId() +
                            " and " + var2.getId() + " on line "
                            + String.valueOf(lineNum));
         return null;
      } else {
         newType = var1.getType();
         return new OperationObject(newisConst,newType, newId);
      }
   }
}
```

The problem of casting the correct types is solved by this code block, wherein we can see that if the types of var1 and var2 match, then we perform a symbol table lookup in order to get the appropriate typing, in order to assign the appropriate type. The new type is essentially the typecast version of the original.

3 Intermediate/IR Code Generation

The bulk of our IR Code generation takes place inside output\IRGenerator.java and output\IRMap.java. Essentially, the IRMap provides an API in order to generate IR code based upon certain function calls. An addition statement, for example, would relate to the code:

```
public static String add(String a, String b, String target) {
   String base = "add, $a, $b, $target";

   base = base.replace("$a", a);
   base = base.replace("$b", b);
   base = base.replace("$target", target);

   return base;
}
```

This code takes any addition statement given params "a," "b," and "target" and translates it into a string of desired format. E.g., $z = x+y$, when converted, would become "add, x, y, z".

As for how we interpret other types of functions, control logic, assignments, etc., we must direct our attention to the IRGenerator code. While the specifics can be found in the file's comments, we can see what was done in general in control logic statements by looking at the following piece of code:

```
public static String while_stat(String expr1, String expr2,
                                Binop compare, List<String> statSeq) {
   String result = "";
   // Generate a unique label to loop to if statement is still true
   String startDo = String.valueOf(expr2.hashCode())
                   .substring((expr2.length() / 4), (expr2.length() / 4) + 5)
                   + "-while-do";

   // Generate a unique label to go to if statement is false
   String endDo = String.valueOf(expr2.hashCode())
                   .substring((expr2.length() / 2), (expr2.length() / 2) + 5)
                   + "-while-enddo";

   // Insert start label at beginning of loop.
   result += emitLabel(startDo);
   switch (compare) {
      case EQUAL:
         result += IRMap.breq(expr1, expr2, endDo);
         break;
       .
       .
      //CODE EMITTED FOR THE SAKE OF BREVITY
       .
       .

   }
   for (String stat : statSeq) {
      result += emit(stat);
   }
   // Insert end label at end of statSeq.
   result += emitLabel(endDo);
   return result;
}
```

Given the input string expressions and the binary operator comparing them, along with statSeq (the block of code to be executed if the statement is determined to be true), this code block takes and formats the code in the appropriate IR fashion, complete with unique label for the control block to goto based on the binary evaluation of true/false.

In the end, these methods are used in conjunction in order to write to file a completed IR code output in .tigIR (haha) format.