CS 4240 Project Phase 2

David Benas, Sean Collins, Casey Evanish

November 13, 2014

Run/Compile Instructions

Run code in the Antlr environment, should output IR to root directory and print symbol table to console.

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

We had a few ways of generating our intermediate code before settling on one that actually worked well. While originally, we had seperate output files that would handle the conversion into IR code, we determined that way was a bit too complicated and would involve too much for our tastes. So, we decided to integrate the code generation mostly in our tigerTreeWalker.g. As an example of how we did this, we can look at the following code:

```
while_stat returns [String breakLabel]
   @after {
      // Emit goto label to go back to top for checking
      irOutput.add(IRMap.gotoLabel(passThroughLabel));
      // Emit end label
      irOutput.add(IRGenerator.emitLabel(
                  ((BinaryExpression.EvalReturn) passThrough).condLabel));
      // Now check for break statements
      for (int line = 0; line < irOutput.size(); line++) {
         if (irOutput.get(line).contains("BREAK_LABEL_" + loopNestNum)) {
            irOutput.set(line, irOutput.get(line).replace("BREAK_LABEL_" + loopNestNum,
                  ((BinaryExpression.EvalReturn) passThrough).condLabel));
                  loopNestNum--;
                  break;
         }
      }
```

```
}
:   ^(WHILE_KEY expr {
      // Create an anchor to loop start
      passThroughLabel = IRGenerator.generateCondLabel($expr.binExpr);
      irOutput.add(IRGenerator.emitLabel(passThroughLabel));
      BinaryExpression.EvalReturn exprReturn = $expr.binExpr.eval(currentTemporary);
      currentTemporary = exprReturn.nextUnusedTemp;
      irOutput.add(exprReturn.irGen);
      passThrough = exprReturn;
      } stat_seq)
;
```

The way we do this is first, we assign values at the top, as such:
for $i := 1$ to 9000
$t(n) = 1$
$t(n+1) = 9000$

Then, we emite the loop label, followed by a conditional check that jumps to the end if false. After the body is interpreted, we emit an expression which increments the loop variable, followed by a goto top label. Finally, we have an end-label that we jump to when the label is finished.

In the end, this method is used for the other rules, which are used in conjunction in order to write to file a completed IR code output in .tigIR (haha) format.