

Compiler Project, Stage 4: Intermediate Code Generation

Computer Science 371
Amherst College
Fall 2015

This assignment, due on **Friday, November 20**, focuses on generating intermediate code.

1 Getting Ready

Go into your `cs371` directory and issue the following commands:

```
cp -r hw3 hw4
cp -r ~lamcgeoch/cs371/hw4/* hw4
cd hw4
rm Makefile
ln -s Makefile4 Makefile
```

In addition, issue the command `chmod -R a+w hw4` if you are working in a group directory. The effect of these commands is to give you a new directory, `hw4`. You have many new directories and files.

Try running `make`. Assuming that your `hw3` files compile correctly, it should run without errors.

2 A Tour of Some of the Directories and Files

minijava/Tree: This directory contains classes for all of the nodes that can appear in an intermediate tree. Class `Print` gives a useful way to print a human-readable version of an intermediate tree. If `root` is the root node of your tree, you can do something like:

```
Print print = new Print(System.out);
print.prStm(root);
```

You won't use class `StmList` at this point, but you'll probably use almost everything else.

minijava/Temp: This directory defines classes related to Temps (i.e. registers) and Labels. The "Maps" define how register names should be printed on the screen, in intermediate code, or in final code. Don't worry about them for now.

minijava/Machine: This directory defines the interface `Machine`. It defines operations that must be implemented elsewhere concerning the target architecture and assembly language.

minijava/Frame: This directory defines the interfaces `Frame` and `Access`.

minijava/Arch/Simple: This directory defines implementations of `Machine` and `Frame` for a simple target architecture.

minijava/Translate: This directory defines Expr and some subclasses. It also contains class Builtins, which contains Labels that can be used to access library functions for tasks such as printing and string manipulation. Class ICode can be used to produce human-readable intermediate code.

Makefile4: This is a new version of the Makefile.

minijava/Main4.java: This is revised from the previous release. You ought to be able to use it without change.

minijava/Interp.java: The main class for an interpreter, which will be described below.

tests4: There are several new test programs and scripts.

3 Random observations

A step-by-step approach to doing this assignment is described in a later section.

- There should be no null pointers within your intermediate tree! If you need a no-op statement, call

```
Stm noop() {  
    return new ESTM(new CONST(0));  
}
```

- You can SEQ together any number of Stms with

```
Stm seq(Stm... list) {  
    Stm result = null;  
    for (Stm s : list)  
        result = (result == null) ? s : new SEQ(result, s);  
    return result;  
}
```

This is an example of the rarely seen “varargs” feature in Java. You can call `seq` with any number of Stms as arguments. (Trivia question: what’s the one example of this that you saw in CS 11?)

- There are two constructors in BINOP. You could write either of the following:

```
new BINOP (BINOP.PLUS, e1, e2);  
new BINOP ("PLUS", e1, e2);
```

There are two similar constructors in CJUMP.

- You can call the ExpList constructor passing any number of Exps as parameters. (This is another example of the “varargs” feature.) If you have a linked list of Exps and you want to build an ExpList with those parameters, call the constructor with no arguments, and then one-by-one make calls to method `addLast()` for each parameter.

- Use `ExpList` containing zero elements in a `CALL` with no parameters.
- The `process()` methods for most of your parse tree statement nodes should have return type `Stm`.
- To print a string, for which the address is given by `Exp e`, use this `Stm`:

```
new ESTM(new CALL(new NAME(builtins.printString), new ExpList (e)));
```

- To convert a `int`, for which the value is given by `Exp e`, to a `String`, use this `Exp`:

```
new CALL(new NAME(builtins.intToString), new ExpList (e));
```

- To concatenate two strings, for which the addresses are given by `Exp left` and `Exp right`, use this `Exp`:

```
new CALL(new NAME(builtins.stringConcatenate), new ExpList (left, right));
```

- To get the length of a string, for which the address is given by `Exp s`, use this `Exp`:

```
new CALL(new NAME(builtins.stringLength), new ExpList (s));
```

- To index in a one-dimensional array, construct a tree fragment that takes a pointer to the first array element, as obtained with an `Access`, and then adds `machine.wordSize()` times the index to get a pointer to the actual element. That pointer then can be dereferenced with `MEM` to get the contained value. If there is another dimension to the array, the “contained value” is a pointer to another array. It can be adjusted by the second index, and so on.
- The length of an array is stored as an integer in position `-1` of the array.
- To create an array, for which the length is given by `Exp n`, use this `Exp`:

```
new CALL(new NAME(builtins.createArray), new ExpList (n));
```

Don't worry about accounting for the element in position `-1`. It is created automatically as an extra element of the array.

4 Getting started

You may find it helpful to build in small steps, recompiling frequently.

1. In `Main4`, uncomment the two extra parameters in constructor call for `Typechecker`. Add matching code in the constructor to store the values.
2. Remove classes `Expr` and `ExprType` from directory `Typechecker`, if you have them. Add `import minijava.Translate.*;` as needed so that their new versions can be found. Be prepared to add numerous import lines as you work on this assignment.
3. For each global variable, create a label by calling `machine.makeLabel`, based on a string with prefix `v_`. Include the label in your `Var` object.

4. Do something similar for each method.
5. After the last parameter is found, call something like `currentMethod.makeFrame(machine)`. That method can call `machine.makeFrame(label)`, and the resulting `Frame` can be stored in the method. Call `frame.createParameterAccesses` to create `Access` objects for the parameters.
6. For each local variable, call `frame.allocLocal()` to create an `Access`.
7. Add a hidden local variable to each method that isn't `void`. Each return statement should yield code that assigns a value to that variable.
8. Create an extra "exit label" for each method. Whenever there is a return statement, the intermediate code should jump to that label. If the method is `void`, a `LABEL` for the exit label should be the last `Stm` represented in the intermediate tree. If the method isn't `void`, the last two `STMs` should be 1) the label, and 2) a `MOVE` that copies the return value from your return value `Temp` into the real return value register (obtained using method `RV()` in class `Frame`).
9. Add a `HashMap<String,Label>` to `Typechecker`. Use it to associate strings with labels. Whenever you encounter a string constant, check the map to find its label. It's not in the map, make a new `Temp.Label()` and add an entry to the map.
10. Add an instance variable `builtins` to `Typechecker`. Call `new Builtins(machine)` to create a value for it. (This will facilitate calls to built-in methods.)
11. Now plunge into `Phase2`, starting at the bottom. (If you're like me, your `Phase2` might have delegated some work to `Typechecker`, so be prepared to work there, too.) Create `Ex`, `CxRel`, or other objects (based on subclasses of `Expr`), as needed. One indication that a method needs work will be the presence of `null` as an `Expr`.
12. The `Stm` that represents the body of a method—don't forget to add the exit label—should be saved with the `Method` object.
13. Add code that's something like this to your `Method` class:

```

public String createICode() {
    Stm b = frame.procEntryExit1(body);

    return "method " + label + " "
        + b.idString() + " "
        + frame.getFrameInfo() + "\n"
        + new ICode(b, frame)
        + "endMethod\n\n";
}

```

14. Add code that's something like this to your `Typechecker` class:

```

Method findMainMethod() {
    for (Method m : methodList) {

```

```

        if (m.name.equals("main") &&
            m.paramTypes.size() == 1 &&
            m.paramTypes.get(0).equals(new ArrayType(Type.stringType)))
            return m;
    }
    throw new RuntimeException ("no main method!");
}

```

15. Add code that's something like this to your Typechecker class:

```

public String createICode() {
    StringBuffer sb = new StringBuffer();

    for (String s: stringMap.keySet())
        sb.append ("string " + stringMap.get(s) + " " + s + "\n");

    for (Var v : classVarMap.values())
        sb.append ("globalVar " + v.label + "\n");

    sb.append ("mainMethod " + findMainMethod().label + "\n\n");

    for (Method m : methodList)
        sb.append (m.createICode() + "\n");

    return sb.toString();
}

```

16. Uncomment the commented section at the end of the main method in Main4.java.

5 Compiling a Minijava Program and Running the Interpreter

To compile a test program and run the interpreter, you can type:

```

./compile -target simple Whatever.java
./interp Whatever.icode1

```

The `-target simple` option means that you want to use the simple architecture as your target machine.

Alternatively, you can type:

```

./doit Whatever.java

```

which will do the same thing.

6 Using the Interpreter, which is also a Debugger

When you start the interpreter, you'll get a prompt like this:

`interpreter>`

If you type `run`, your program will simply run. The various `debug` commands will print various things while you execute. (`debug 1` may be the most useful.)

Command `break` sets a breakpoint. For example you might type `break m_hanoi_0` to cause execution to stop when you reach a particular method. (`m_hanoi_0` is the label that was generated for method `hanoi` in my Towers of Hanoi program. Look at your intermediate code or run `methods` within the debugger to get a list of the methods in your program.) If you set breakpoints and then do `run`, the program will run until one of the breakpoints is reached.

Command `step`, an alternative to `run`, takes you through the program one step at a time. It's sometimes useful to run until a breakpoint and to take steps from there. Once you begin stepping, you can keep hitting the enter key to take more steps. If you type `b` or `?`, you'll be back at the `interpreter>` prompt.

Good luck! Questions? Let me know.

7 Submitting Your Work

Submit your work in the usual way.