Project 11: Target Code Generation

Due Date: Tuesday, May 14, 2006, Noon
Duration: Two weeks

Overview

The goal of this project is to finish the PCAT compiler by producing SPARC assembly code instructions from the intermediate representation (IR). This target code can then be assembled and executed.

File you are to create:

Emit.java

Files

The following files can be found via the class web page or FTPed from:

```
~harry/public_html/compilers/p11
```

Main.java
Altered to call **Emit.java**.

Peephole.java
This file relates to an optional extension: a "peephole optimizer" for the IR instructions. If you are not doing the extension, just copy this file into your directory and compile it, along with the other classes.

Lexer.java
Parser.java
Checker.java
Generator.java
Files you created.

Lexer.class
Parser.class
Checker.class
Generator.class
Compiled versions.

Token.java
StringTable.java
SymbolTable.java
Ast.java
PrintAst.java
PrettyPrintAst.java
IR.java
run
go
Unchanged.

tst
runAll
Same as before, but altered for this project.

go2
New file. This shell script will compile, assemble, and execute the program. It will print out the source and the **.s** file before beginning execution.

pc
New file. This shell script will compile and execute a PCAT program. "pc" stands for "PCAT Compile".

run2
runAll2
These shell scripts are for testing other code generation algorithms. Like **run** and **runAll**, they will compile a test PCAT program from the **tst** directory. However, these scripts will ignore the **.bak** files. Instead, these scripts proceed to execute the program (using **pc**) and make sure the output from the execution is correct.

makefile
Modified to deal with **Emit.java**.

Main.jar
The new "black box" code.

Organization

The goal is to run through the Intermediate Representation (IR) statements and produce target code, i.e., SPARC assembly code. This target code will be suitable for assembling and executing.

This document describes how to do this using "Code Generation Algorithm #1." You are free to try a more sophisticated algorithm... after you get "Code Generation Algorithm #1" working!

You will create a file called **Emit.java** containing class **Emit** with a method called **emitAll()**. **Main.java** has been modified to create an **Emit** object and to invoke **emitAll()** on it. The **emitAll()** method takes no arguments and returns no result.

Your **emitAll()** method (and any methods it invokes) should just print the assembly code to **stdout**. Later, this will allow us to pipe the output of your compiler straight into the assembler.

In **Main.java** I have commented out the calls to **printOffsets()** and to **printIR()**. For debugging, you may wish to un-comment these calls. I've also included the call to the **emitAll()** method, which you are going to write.

The "pc" Shell Script File

After a PCAT program has been compiled, it should be assembled and executed. I have provided a shell script file called **pc** which will do all of this. It will:

- run **Main** to produce the **.s** file
- abort if errors
- invoke **gcc** to assemble the **.s** file and produce an executable
- abort if errors
- execute the program

There is a test in **tst** called **simple**, so once you have finished **Emit.java** and compiled your compiler, you can type:

```
pc tst/simple
```

to compile, assemble, and run a PCAT program named **tst/simple.pcat**. (Of course, the assembly and execution steps will only work on a SPARC computer.) The **pc** command will also leave the files **tst/simple.s** and **tst/simple** behind, so the executable can be used again later. To re-execute the program, type

```
tst/simple
```

When the **pc** script invokes your compiler, it sends the **stdout** to a file with a name such as **simple.s**. This scheme depends on the compiler sending only the generated SPARC code to **stdout** and sending all error messages to **stderr**. The files I am providing print error messages on **stderr** instead of **stdout**, although they print other stuff (**printAst()**, **prettyPrint()**) on **stdout**. Whenever an error occurs during a compile, **Main.java** calls **System.exit(1)** to terminate with a non-zero error code. This will cause the **pc** script to terminate without proceeding to the assembly phase.

You can also use **pc** on PCAT programs of your own creation. If you create a file called **myprog.pcat** in the same directory as your code, just type:

```
pc myprog
```

The "run2" and "runAll2" Shell Script Files

The **run** and **runAll** scripts work the same way as in previous projects. They assume the existence of files like:

```
simple.pcat
simple.out.bak
simple.err.bak
```

In addition, I have provided a second set of scripts called **run2** and **runAll2**, which assume the existence of files with names like:

```
    simple.pcat
    simple.givenInput
    simple.givenOutput1
    simple.givenOutput2
```

which are included in the **tst** directory in this project.

The script called **run2** will first compile and assemble the PCAT program (using **pc**) and then run the executable. When running the executable, **run2** will pipe characters from the **givenInput** file to the running program. **Run2** will save the **stdout** and **stderr** output produced by the executable. After execution, **run2** will compare the output from the execution to the files containing the expected output (**givenOutput1** and **givenOutput2**) and will print any differences. The **run2** script will ignore the **.out.bak** and **.err.bak** files.

I have also provided a script called **runAll2**, which is the same as **runAll**, except that it applies **run2** to all of the test files, instead of **run**.

As long as you use Code Generation Algorithm #1 and follow the guidelines given in this document, your compiler should produce the same **.s** output file as expected. Therefore, you will not need to assemble or execute your output. If your output exactly matches the **.out.bak** and **.err.bak** files, it must necessarily assemble and execute in exactly the same way as expected. Therefore, you will not need to use **run2** or **runAll2**; they are only provided in case you attempt to use a different code generation algorithm.

The "go2" Shell Script File

The **go** script is unchanged and works the same way as in previous projects. You can use it to run your compiler on one of the test files. The **go** script will print the source. It will then run your compiler and print the **stdout**, and the **stderr**, It will ignore the **.out.bak** and **.err.bak** files.

I am also providing a script called **go2**, which will compile your program and then run it. For example, you can type:

```
    go2 simple
```

This script is provided for anyone using something besides Code Generation Algorithm #1.

The **go2** script will first print the PCAT source file and then invoke your compiler. The script then prints out the **.s** file before invoking the assembler, so you can look at what code your compiler generated. You will occasionally get assembler error messages if you don't generate legal SPARC code. The listing of the **.s** file is useful when such errors occur. Next, the **go2** script assembles the program into an executable file (with a name such as **simple**). Finally, if both the compiler and the assembler finish without exit errors, the script will run your program.

Files in /tst

Several of the files in **tst/** are PCAT programs requiring user input. They are: **error3**, **read1**, **read2**, **primes**, and **quicksort**. These files will prompt for input, after telling the user what to enter.

The files **fact**, **quicksort**, and **primes** are programs that actually compute something, rather than just testing a single aspect of the compiler. In the lectures, we used "quicksort" as an example; the file **quicksort.pcat** is a working version of this program. The program **fact.pcat** computes factorial numbers and **primes.pcat** computes prime numbers.

The program **speed.pcat** is a compute-bound benchmark designed to determine how fast the compiled PCAT code runs. It is not an "official" test and is not included in the **runAll** script. You may run it, but don't run it too often or you may irritate other users.

Strategy

You may adopt one of several code generation strategies. The black box compiler uses "code generation algorithm #1" and I recommend that approach as a good way to start. I expect that some students will try more complex code generation algorithms. Algorithm #1 is much simpler to implement but it generates many inefficient code sequences. The rest of this document discusses only code generation algorithm #1. The design of more complex code generation algorithms and optimizations is left entirely to you.

To do code generation algorithm #1, you need to walk the list of the IR instructions and, for each instruction, print out several SPARC instructions. The routine **printIR()** in **IR.java** happens to walk the list of instructions so I recommend using it as a starting skeleton for **emitAll()**.

Comments

You probably noticed in projects 8, 9 and 10 how nice it was to have the comments in the output. It makes it easier to locate the section of output you are interested in.

In this project two types of comments are now included in the output. First, the original comments produced by calls to **IR.comment()** are printed. Second, since **printIR()** contains code to print the IR instructions, it makes sense to just leave these print statements in **emitAll()**. The only modification is to prefix each line with a exclamation (!) and three spaces. An example is shown below.

The original comments were all capitals so they stand out visually. As you can see, one source code construct (like a PCAT WRITE statement), translates into several IR instructions, and each instruction is then translated into several SPARC instructions.

```
! MAIN...
!   mainEntry
        .global main
main:   save    %sp,-136,%sp
        set     display0,%o0
        st      %fp,[%o0]
! WRITE STMT...
!   writeString str1
        sethi   %hi(str1),%o0
        call    printf
        or      %o0,%lo(str1),%o0
!   writeNewline
        sethi   %hi(strNL),%o0
        call    printf
        or      %o0,%lo(strNL),%o0
```

Boilerplate

The first section of your target program will need to contain several fixed, predetermined lines of code. These lines will precede the SPARC instructions your compiler will generate for each IR instruction. You will need to start by printing out all these fixed lines, which I call the "boilerplate." In my own code, I created a single routine, called **emitBoilerplate()**, which just prints out all the stuff at the beginning.

Below is an example of how your output target file should begin, i.e., the output from **emitBoilerplate**(). (It follows this entire section; locate it now so you can follow along with this commentary.)

The first thing you see in the boilerplate is a comment. It is typical for compilers to add a comment to the output telling which version of the compiler was used, copyright notices, etc. Anyone looking at a strange **.s** file can tell immediately that it was produced by a compiler, which version of the compiler, etc.

Next, we need some **.global** directives since we'll be calling the routines **.div** and **.rem** to perform integer DIV and MOD.

Next, we'll need a temporary field and we can name it **temp**. This field will be used in the **writeFlt** routine (discussed below) and **temp** will also be needed during integer to float conversions since it is otherwise impossible to move a value from an integer register to a floating register.

Next, we'll need some format strings. To implement the **IR.writeInt**, **IR.writeFloat**, **IR.readInt**, and **IR.readFloat** instructions, we'll be calling **printf()** and **scanf()**, which are included in the C library. (The **gcc** command will automatically include these library routines as necessary during the linking phase.) They will need "format" strings, so we'll include three of them, called **strNL**, **strInt**, and **strFlt**. Also, when we print out booleans, we'll need the strings **strTrue** and **strFalse**. Finally, when runtime errors occur during execution, we'll be printing out a message before terminating, so we have the strings **message1**, **message2**, ... **message5**. (**Message5** should be all on one line, but is too long for this document.)

Notice how the compiler directives (**.data**, **.text**) put **temp** into the data segment, and the strings into the text segment, since they will be read-only. There are also **.align** directives to ensure that what follows will be aligned in memory.

Whenever a runtime error occurs, the code will take a branch to one of the labels **runtimeError1**, **runtimeError2**, ... **runtimeError5**. We need code to print out the right message and abort the program by calling the **exit()** routine. The code for each of these "error handlers" follows next.

Next come two routines **writeFlt** and **writeBool**. (Printing integers is so simple we'll just generate the necessary code inline.) Look over and understand these routines. As far as I can tell, **printf()** is incapable of printing single precision floating-point numbers. The code in **writeFlt** is passed a single-precision number, converts it to a double, then calls **printf** to print it. The code in **writeBool** tests its argument and prints either one string (**strTrue**) or another (**strFalse**).

Next, we need to emit a number of "display register" variables. ("Register" is somewhat of a misnomer.) We have previously calculated the maximum nesting level of the program; based on this number, **emitBoilerplate()** should print out as many of these **.word** directives as needed.

Originally, I thought we would be able to include floating-point constants as literal (i.e., immediate) values in the SPARC instructions, but there is apparently no way to do that, since there is apparently no way to move data from an integer register to a floating register without going through memory first. To deal with this problem, we must generate a single precision floating-point constant (using the **.single** directive) for each real constant (i.e., each **RealConst** node) that appears in the program. Recall that we have previously built a linked list of **RealConst** nodes (pointed to by **floatList** and linked on their **next** fields). We will walk the linked list of **RealConst**s and, for each, print out a **.single** directive, using its **nameOfConstant** and its **rValue**.

We also built a linked list of all the **StringConst**s in the program, using the **next** field in each **StringConst**. Next we run though this list and, for each string, print an **.asciz** directive for it. We have previously named each string and stored that name in the **nameOfConstant** field. Note that we end with an **.align** directive so that the SPARC instructions that follow will be aligned.

As you are coding this method, remember that you'll need to escape the double quote character (") and the backslash character (\) with backslashes.

```
!
! PCAT Compiler Version 1.0
!
             .global  .div
             .global  .rem
!
! Standard data fields
!
             .data
             .align   8
temp:        .double  0
             .text
strNL:       .asciz   "\n"
strInt:      .asciz   "%d"
strFlt:      .asciz   "%g"
strTrue:     .asciz   "TRUE"
strFalse:    .asciz   "FALSE"
message1:    .asciz   "Execution Error: Allocation failed!\n"
message2:    .asciz   "Execution Error: Pointer is NIL!\n"
message3:    .asciz   "Execution Error: Read statement failed!\n"
message4:    .asciz   "Execution Error: Array index is out of bounds!\n"
message5:    .asciz   "Execution Error: Count is not positive in array
                                                  constructor!\n"
             .align   8
```

```
!
! runtimeError1-5
!
! Branch to one of these labels to print an error message and abort.
!
runtimeError1:
             set      message1,%o0
             call     printf
             nop
             call     exit
             mov      1,%o0
runtimeError2:
             set      message2,%o0
             call     printf
             nop
             call     exit
             mov      1,%o0
runtimeError3:
             set      message3,%o0
             call     printf
             nop
             call     exit
             mov      1,%o0
runtimeError4:
             set      message4,%o0
             call     printf
             nop
             call     exit
             mov      1,%o0
runtimeError5:
             set      message5,%o0
             call     printf
             nop
             call     exit
             mov      1,%o0
! writeFlt
!
! This routine is passed a single precision floating number in %f0.
! It prints it by calling printf.  It uses registers %f0, %f1.
!
writeFlt:
        save    %sp,-128,%sp
        fstod   %f0,%f0
        set     temp,%l0
        std     %f0,[%l0]
        ldd     [%l0],%o0
        mov     %o1,%o2
        mov     %o0,%o1
        set     strFlt,%o0
        call    printf
        nop
        ret
        restore
```

```
! writeBool
!
! This routine is passed an integer in %i0/o0.  It prints "FALSE" if this
! integer is 0 and "TRUE" otherwise.
!
writeBool:
        save    %sp,-128,%sp
        cmp     %i0,%g0
        be      printFalse
        nop
        set     strTrue,%o0
        ba      printEnd
        nop
printFalse:
        set     strFalse,%o0
printEnd:
        call    printf
        nop
        ret
        restore
!
! Additional Fields
!
        .data
display0:       .word   0
display1:       .word   0
display2:       .word   0
display3:       .word   0
        .text
float3: .single 0r4.4
float2: .single 0r654.321
float1: .single 0r0
str3:   .asciz  "The computed answer is:"
str2:   .asciz  "Please enter a real value..."
str1:   .asciz  "Hello, world!"
        .align  8
```

When the **run** script compares the output from your compiler to the **.out.bak** files, it uses the **–w** option on the **diff** command, which causes it to ignore minor differences in whitespace. The relevant line in **run** is:

```
diff -w tst/$1.out.bak tst/$1.out
```

Theoretically, you could get by with using a single space wherever the boilerplate has any white space, but this would be difficult to read. For example, output like this would pass the tests:

```
!
! PCAT Compiler Version 1.0
!
 .global .div
 .global .rem
!
! Standard data fields
!
 .data
 .align   8
temp: .double 0
 .text
strNL: .asciz "\n"
strInt: .asciz "%d"
strFlt: .asciz "%g"
strTrue: .asciz "TRUE"
...
```

Please try to make your compiler format its output as close to the **.out.bak** files as possible.

Useful Methods

I found that two methods made things much easier for me during code generation. They are: **getIntoAnyReg()** and **storeFromAnyReg()**. You may wish to create similar methods.

The idea is that you can call **getIntoAnyReg()**, passing it (1) a pointer to either a **VarDecl**, **Formal**, **IntegerConst**, or **RealConst**, and (2) the name of a register (such as "%o3"). It will generate whatever code is necessary to load the register.

Remember that a **VarDecl** may describe a local variable or a non-local variable. If you want to get the value of a local variable, you should generate code like this:

```
ld       [%fp+-148],%reg
```

where "-148" is the offset of the variable in the activation record.

If the variable access is non-local, you should generate code that uses the appropriate display register. Assume the variable was declared at lexical level 7 and is used non-locally (say at lexical level 9). The code might look like this:

```
set      display7,%reg2
ld       [%reg2],%reg2
ld       [%reg2+-148],%reg
```

In this case, a second register was necessary to hold addresses. The target register "reg" may be either an integer or float. Since addresses must always go into integer registers, the routine takes a third argument, "reg2," which must name an integer register.

Note that the caller of **getIntoAnyReg()** may supply the same register for both "reg" and "reg2." If, for example, it was called with reg=reg2="%o0", it would generate code this code:

```
set      display7,%o0
ld       [%o0],%o0
ld       [%o0+-148],%o0
```

To distinguish between locals and non-locals, you'll have to know what the lexical level of the current routine is. Then you can compare it to the lexical level at which the variable was defined. If they are equal, this variable must be a local. (Also remember that in a **VarDecl**, lexLevel == -1 indicates the variable is a temporary. If the variable is a temporary, then it must be a local reference.)

These methods should generate the same code for **Formal**s (local or non-local) as they generate for **VarDecl**s (local or non-local).

The code generated by **storeFromAnyReg()** is the same as the code generated by **getIntoAnyReg()**, except that the final instruction is a **st** instead of a **ld**. For **storeFromAnyReg()**, "reg2" must be different from "reg."

Here are the specifications for these methods:

```
// getIntoAnyReg (p, reg, reg2)
//
// This routine is passed p (a pointer to a VarDecl, Formal,
// IntegerConst, or RealConst) and two registers.  Each register will
// be a String such as "%i3".  This method generates instructions to
// load the desired quantity into "reg."  The register "reg" may be
// either int or floating.  The register "reg2" is an integer register
// that will be used as necessary.  This routine assumes that
// IntegerConsts will only be loaded into integer regs, so it does not
// check any mode information.
//

void getIntoAnyReg (Ast.Node p, String reg, String reg2) ...



// storeFromAnyReg (p, reg, reg2)
//
// This method is passed p (a pointer to a VarDecl or Formal) and two
// registers.  Each register will be a String such as "%i3".  This method
// generates instructions to store the "reg" into the variable.
// The register "reg2" must be an integer register and must be
// different from "reg"; it will be used if necessary.
//

void storeFromAnyReg (Ast.Node p, String reg, String reg2) ...
```

Approach to Implementing "Emit.java"

Here is the order in which I implemented the IR instructions.

The first test file to get working is **simple.pcat**. This PCAT program simply prints a couple of messages and it requires only four different IR instructions to function: **IR.mainEntry**, **IR.mainExit**, **IR.writeString**, and **IR.writeNewline**.

Next, work on compiling the file **write.pcat**, which additionally uses the **IR.writeInt**, **IR.writeFloat**, **IR.writeBoolean**, and **IR.assign** instructions.

Then, get the test programs **goto1.pcat**, **goto2.pcat**, and **goto3.pcat** working. These programs will require the conditional and unconditional branch instructions. After this you will have enough to compile programs with basic IF and WRITE statements; many of the remaining test files use IF and WRITE statements, so life will be simpler if you get them working first.

Then, move on to the **binary1.pcat**, **binary2.pcat**, **div.pcat**, **neg.pcat**, and **itof.pcat** tests; this should allow you to compile programs containing arithmetic and Boolean expressions.

Then move on to **call1.pcat**, which uses the **IR.procEntry** and **IR.returnVoid** IR instructions. The program **call2.pcat** will make sure that access to non-local variables is working and **call3.pcat** will exercise the **IR.returnExpr** instruction.

Then move on to the tests **param1.pcat**, **param2.pcat**, and **param3.pcat**, which will exercise parameter passing and returning, as well as non-local access. They will exercise the **IR.param** and **IR.formal** instructions.

Then move on to the test programs, **read1pcat** and **read2pcat**, which will require **IR.readInt**, **IR.readFloat**, and **IR.loadAddress** to be working.

Then, implement the **IR.alloc**, **IR.loadIndirect**, and **IR.store** instructions. Start by trying the **alloc1.pcat** test file, which is a basic test of these instructions, and move on to **alloc2.pcat**.

The **local.pcat** test will exercise local and non-local variable accessing.

Finally, make sure all the **error** test programs work correctly.

After this point, you should be done implementing all the IR instructions. The following programs should then run correctly with no further effort: **array1**, **array2**, **array3**, **for**, **fact**, **primes**, **sort**, and **speed**.

The last test file is called **yapp**. The file **yapp.pcat** is a 2100 line PCAT program, which yields 20,000 lines of SPARC assembly. If you pass the other tests, then you should passé this one, too. Since it takes quite a while to execute, you may wish to abort the **runAll** script rather than running the **yapp** test to completion. (By the way, **yapp.pcat** is an SLR parser generator. It implements the SLR parsing algorithms (include computation of FIRST and FOLLOW sets, sets of LR(0) items, closures, etc.) discussed last term.)

Next we discuss what code we want to generate for each instruction.

IR.mainEntry

The code to be generated (i.e., the output to be printed to **stdout**) for the **IR.mainEntry** instruction is shown by example below. The only variable is the **frameSize**, which is underlined in the material below.

As you can see, when executed this code will begin by allocating a frame of the correct size by subtracting from the **%sp** register. Then, we save a pointer to the base of this activation record (**%fp**) into the variable called **display0**.

```
!   mainEntry
         .global main
main:    save    %sp,-136,%sp
         set     display0,%o0
         st      %fp,[%o0]
```

The **frameSize** here is “136,” but this number will vary in other PCAT programs. Thus, you’ll just need print statements to print the above 5 lines. To find the **frameSize**, just follow the **arg1** field of the instruction to a **Body** node and use the **frameSize** that we saved in the **Body** in the previous project.

IR.mainExit

The code to be generated for the **IR.mainExit** instruction is this:

```
!   mainExit
       ret
       restore
```

IR.writeString

The code to be generated for the **IR.writeString** instruction is this:

```
!   writeString str123
         sethi   %hi(str123),%o0
         call    printf
         or      %o0,%lo(str123),%o0
```

The “str123” represents the name of the string being written and will vary among actual instructions. This code loads register **%o0** with the one, and only, argument to **printf()**, which it then calls.

IR.writeNewline

A WRITE statement may print several things; these should all be followed by a single new-line. For example:

```
WRITE ("a = ", a, "   b = ", b);
```

should produce a single line of output.

The code to be generated for the **IR.writeNewline** instruction is this:

```
!   writeNewline
         sethi   %hi(strNL),%o0
         call    printf
         or      %o0,%lo(strNL),%o0
```

IR.writeInt

The argument to the **IR.writeInt** instruction will be either an **IntegerConst**, **VarDecl**, or **Formal**. To generate code for this instruction, first call **getIntoAnyReg()** to load the value into **%o1**. In the example below, this produces the "`set`" instruction. Then produce the remaining three SPARC instructions, which call **printf**, passing **strInt** (the format string) as the first argument in **%o0** and the integer value as the second argument in **%o1**.

```
!   writeInt 12345
        set     12345,%o1
        sethi   %hi(strInt),%o0
        call    printf
        or      %o0,%lo(strInt),%o0
```

In this example, **getIntoAnyReg()** generated a single **set** instruction since the operand was an **IntegerConst**. If the operand had been a **VarDecl** or **Formal**, then **getIntoAnyReg()** would have generated several instructions to get it into **%o1**. (For **writeInt**, the operand will obviously never be a **RealConst**.) The routine **getIntoAnyReg()** requires two registers (called "reg" and "reg2"). We can just use reg = reg2 = **%o1**.

Regardless of what instructions **getIntoAnyReg()** generates, the final three instructions (which set **%o0** to point to the format string and call **printf**) will always be the same.

IR.writeFloat

The code to be generated for the **IR.writeFloat** instruction will be something like this. (Refer to the comments about the list of floats in the "Boilerplate" section of this document.) From this example, you can see that the code was generated by first calling **getIntoAnyReg()** with reg = **%f0** and reg2 = **%o1**. Then, the final two instruction (the "`call`" and the "`nop`"), which are always the same, were generated.

```
!   writeFloat 654.321
        set     float2,%o1
        ld      [%o1],%f0
        call    writeFlt
        nop
```

IR.writeBoolean

The code to be generated for the **IR.writeBoolean** instruction will be something like this:

```
!   writeBoolean 0
        set     0,%o0
        call    writeBool
        nop
```

IR.assign

The code to be generated for the **IR.assign** instruction will be something like this. First, we call **getIntoAnyReg()** to get the RHS operand into **%o0** (using reg2=**%o0** as a work register if necessary). Then, we call **storeFromAnyReg()** to generate code to move from **%o0** to wherever the result is (using **%o1** as a work register if necessary). If both the RHS operand and the result are local variables, we should get code like this:

```
!   x := t1
        ld      [%fp+-16],%o0
        st      %o0,[%fp+-4]
```

IR.procEntry

The code to be generated for the **IR.procEntry** instruction will be something like this. Note that "64" is the offset of the "display register" save area (DISPLAY_REG_SAVE_AREA_OFFSET) in the activation record. The only things that vary are the procedure name, the frameSize and the lexical level, which are underlined below.

```
!   procEntry p13_foo,lexLev=4,frameSize=136
p13_foo:   save    %sp,-136,%sp
           set     display4,%o0
           ld      [%o0],%o1
           st      %o1,[%fp+64]
           st      %fp,[%o0]
```

IR.call

The code to be generated for the **call** instruction will be something like this:

```
!   call p13_foo
        call    p13_foo
        nop
```

IR.returnVoid

The code to be generated for the **IR.returnVoid** instruction will be something like this. Note that "64" is the offset of the "display register" save area (DISPLAY_REG_SAVE_AREA_OFFSET) in the activation record. Also note that we used display register "2" here, but this could be some other number. The underlined material will be whatever the lexical level of this routine is. Unfortunately, we don't have that information in this instruction. But we do have it when we process the **IR.procEntry** instruction. So you'll need to create a field (you might call it **currentLexLevel**) and set it when processing **IR.mainEntry** (to zero) and **IR.procEntry**.

```
!   return
        set     display2,%o0
        ld      [%fp+64],%o1
        st      %o1,[%o0]
        ret
        restore
```

IR.goto

The code to be generated for the **IR.goto** instruction will be something like this:

```
!   goto Label_1
        ba      Label_1
        nop
```

IR.label

The code to be generated for the **IR.label** instruction will be something like this:

```
!   Label_1:
Label_1:
```

IR.gotoiLE

The code to be generated for the **IR.gotoiLE** instruction will be something like this. Note that the first two loads were generated by two calls to **getIntoAnyReg()** and would have be somewhat different if the operands had been different.

```
!   if x <= t1 goto Label_2              (integer)
        ld      [%fp+-8],%o0
        ld      [%fp+-12],%o1
        cmp     %o0,%o1
        ble     Label_2
        nop
```

The code for the other integer conditional branches is similar.

IR.gotofLE

The code to be generated for the **IR.gotofLE** instruction will be something like this. The code for the other floating point branches is similar. The "s" at the end of **fcmps** indicates single-precision. One can also use **fcmpes** to do the comparison and cause an exception if there are problems (e.g., one operand is NaN). Note that there is a **nop** instruction inserted between the compare and branch instructions.

```
!   if x <= y goto Label_31             (float)
        ld      [%fp+-4],%f0
        ld      [%fp+-8],%f1
        fcmps   %f0,%f1
        nop
        fble    Label_31
        nop
```

IR.iadd

The code to be generated for the **IR.iadd** instruction will be something like this. First, we call **getIntoAnyReg()** to get the first operand into **%o0** (using reg2=**%o0** as a work register if necessary). Then, we call **getIntoAnyReg()** to get the second operand into **%o1** (using reg2=**%o1** as a work register if necessary). Then, we generate the "`add`" operation, taking its arguments from **%o0** and **%o1** and placing its result in **%o1**. Finally, we call **storeFromAnyReg()** to generate code to move whatever is in **%o1** to wherever the result should be (using **%o0** as a work register if necessary).

If the two operands are local variables, we get something like this:

```
!   t1 := y + z         (integer)
      ld      [%fp+-8],%o0
      ld      [%fp+-12],%o1
      add     %o0,%o1,%o1
      st      %o1,[%fp+-16]
```

If the two operands are in constants, we get something like this:

```
!   t2 := 123 + 456             (integer)
      set     123,%o0
      set     456,%o1
      add     %o0,%o1,%o1
      st      %o1,[%fp+-20]
```

If the two operands are non-locals, we get something like this:

```
!   t3 := y + z         (integer)
      set     display0,%o0
      ld      [%o0],%o0
      ld      [%o0+-8],%o0
      set     display0,%o1
      ld      [%o1],%o1
      ld      [%o1+-12],%o1
      add     %o0,%o1,%o1
      st      %o1,[%fp+-4]
```

Other binary operators are similar. In the case of floating operations, we need to get the first operand into **%f0** (using reg2=**%o0** as a work register) and the second operand into **%f1** (again using reg2=**%o0** as a work register). The "`fadds`" operation will place its result in **%f1** and then we need to call **storeFromAnyReg**() to move the result from **%f1** to wherever it is to be stored (using reg2=**%o0**).

IR.idiv, IR.imod

I implemented integer **div** and integer **mod** by calling the routines **.div** and **.rem** as shown below, since these routines have exactly the functionality we require for the **div** and **mod** operators in PCAT.

```
!   t17 := t15 DIV t16          (integer)
        ld      [%fp+-72],%o0
        ld      [%fp+-76],%o1
        call    .div
        nop
        st      %o0,[%fp+-80]

!   t25 := t23 MOD t24          (integer)
        ld      [%fp+-104],%o0
        ld      [%fp+-108],%o1
        call    .rem
        nop
        st      %o0,[%fp+-112]
```

Note that the loading of **%o1** (which was generated in **getIntoAnyReg()**) could certainly be moved into the delay slot. One approach is to design a peephole-style optimizer to run after the final code generation. It would work much like the peephole optimizer that ran over the IR instructions.

IR.itof

To implement the integer-to-float conversion, I proceeded as follows. First, generate code to get the integer value into **%o0**, by calling **getIntoAnyReg()**. Then, generate 2 instructions to store it in the variable called **temp**, allocated in the boilerplate code. Then, generate instructions to load the value into **%f0** and convert it from integer to single-precision floating using the **fitos** instruction, leaving it in **%f0**. Finally, I called **storeFromAnyReg()** to move the floating value to wherever it should be stored.

I implemented **itof** by calling **getIntoAnyReg()** to start with. If it is passed an **IntegerConst**, it won't be able to generate code to move it into a floating reg. So instead I asked **getIntoAnyReg()** to put the integer into an integer register (**%o0**). Then, I generated code to move it from **%o0** to **temp** and then load it into a floating register in separate steps.

```
!   t1 := intToFloat (...)
        ...     ...,%o0
        set     temp,%o1
        st      %o0,[%o1]
        ld      [%o1],%f0
        fitos   %f0,%f0
        st      %f0,[%fp+-16]
```

Ideally, one would like to call **getIntoAnyReg()** to move the integer value straight into **%f0**, without messing with **temp**. There are better ways to generate a code sequence for this instruction, but I tried to keep the code as simple-minded as possible.

IR.formal

Originally, I had planned to pass parameters in registers **%o0**-**%o5**, following the SPARC conventions for the "C" language. This convention is very efficient if the subroutine can simply keep its parameters in the **%o0**-**%o5** registers and never move them into memory. Unfortunately, our simple-minded code generator would be significantly complicated by this approach. It is much easier to generate code if we can assume that all variables are stored in the activation record uniformly.

My original plan was to pass the first 6 arguments in the registers and then have the **IR.formal** instruction generate the code to move the argument values from the **%o0**-**%o5** registers into slots in the activation frame, where they would then be treated identically to the other local variables. Later, I realized that this plan would generate unnecessary loads and stores and that it would be simpler to have the caller just move all the arguments (into the caller's frame, with instructions in the caller's code). From there, they can be easily accessed by the callee's code. One benefit of this plan is that we don't have to distinguish between the first six parameters and the remaining parameters; they are all treated identically.

However, we still need to make sure that we allocate at least 6 words in each activation record for holding the first six parameters, regardless of whether we actually need the all six slots. (And we did this in the last project since it is the SPARC convention for calling "C" routines.) It might seem unnecessary if we only call routines with fewer than 6 arguments, but it is necessary since our PCAT routines may call "C" routines. (For example, we generate code that will call **printf**.) Our compiler makes sure there are always at least six slots present, just in case **printf** wants to make use of this space.

The bottom line is that all the arguments will be exactly where we want them–namely stored in the caller's frame–by the time we get into the called subroutine. Thus, the **IR.formal** instruction expands into zero SPARC instructions. You should still copy the comment to the output, however.

```
    !   formal 1,x
```

Long discussion, short code sequence. I always like that.

IR.param

I implemented **IR.param** as shown by example below. I called **getIntoAnyReg()** to move the value into **%o0**. Then I generated a "`st`" instruction to move it into the correct slot in the caller's activation record, depending on which parameter it is (e.g., parameter 1 goes into offset 68, parameter 2 goes into offset 72, etc.) Use the INITIAL_FORMAL_OFFSET and FORMAL_OFFSET_INCR constants.

```
!   param 1,123
        set     123,%o0
        st      %o0,[%sp+68]
!   param 2,456
        set     456,%o0
        st      %o0,[%sp+72]
```

IR.returnExpr and IR.resultTo

The test file **call3.pcat** tests the ability to return values from subroutines. To return values, we need the instructions **IR.returnExpr** and **IR.resultTo**.

The returned value can be returned in **%o0**, which is called **%i0** in the callee's register window. To generate code for **IR.returnExpr**, we first call **getIntoAnyReg()** to move the returned value into **%i0**. Then, we generate code similar to **IR.returnVoid** to restore the display register and return. For example:

```
!   return 123
        set     123,%i0
        set     display2,%o0
        ld      [%fp+64],%o1
        st      %o1,[%o0]
        ret
        restore
```

Directly after an **IR.call** instruction, we will have an **IR.resultTo** instruction, which will tell us where to put the returned value. We can generate the SPARC code by calling **storeFromAnyReg()** to move the value from **%o0**.

```
!   resultTo t2
        st      %o0,[%fp+-8]
```

IR.readInt

The code generated for **IR.readInt** is almost identical to the code for **IR.writeInt**. In the following example, note that **t1** will contain the address into which to store the result; we do not put the result in **t1** itself. Also, we need to test the value returned by **scanf()** in **%o0** to make sure it is non-zero. (The value is the number of characters scanned; if it is zero, there was an error in the input and no value was stored.)

```
!   readInt t1
        ld      [%fp+-12],%o1
        sethi   %hi(strInt),%o0
        call    scanf
        or      %o0,%lo(strInt),%o0
        cmp     %o0,0
        be      runtimeError3
        nop
```

IR.readFloat

The code generated for **IRreadFloat** is similiar:

```
!   readFloat t2
        ld       [%fp+-16],%o1
        sethi    %hi(strFlt),%o0
        call     scanf
        or       %o0,%lo(strFlt),%o0
        cmp      %o0,0
        be       runtimeError3
        nop
```

IR.loadAddr

To implement the **IR.loadAddr** instruction, we must handle local and non-local variables. First, we generate code to get the desired address into **%o0**, then we can call **storeFromAnyReg()** to move it to wherever it should go. For example, asssuming **j3** is a local variable and **i3** is a non-local variable, the following code sequences would be generated:

```
!   t7 := &j3
        add      %fp,140,%o0
        st       %o0,[%fp+-12]

!   t10 := &i3
        set      display0,%o0
        ld       [%o0],%o0
        add      %o0,-12,%o0
        st       %o0,[%fp+-24]
```

IR.alloc

To implement the **IR.alloc** instruction, we must generate a call to **calloc()**, which takes two arguments. The first argument (in **%o0**) will be the count; the second argument (in **%o1**) will be the number of bytes. We can just use a count of one. The returned pointer will be in **%o0**. We do not need to test it for zero, since there will be separate IR instructions immediately following that will do the test.

```
!   t1 := allocate (...)
        set      1,%o0
        ...      ...,%o1
        call     calloc
        nop
        st       %o0,[%fp+-8]
```

IR.loadIndirect

To implement **IR.loadIndirect** instruction, you'll first need to call **getIntoAnyReg()** to move **y** (the address) into **%o0**. Then you can generate an additional **ld** instruction to perform the indirect fetch. Finally, you can call **storeFromAnyReg()** to move the fetched data to wherever it belongs.

```
!    x := *y
       ...      ...,%o0
       ld       [%o0],%o0
       ...      %o0,...
```

IR.store

To implement the **IR.store** instruction, you'll first need to load the value to be stored (**y**, in the following example) into a register (**%o0**). Then you'll need to load the address (**x**) into another register (**%o1**). Finally, you can generate the **st** instruction to store the data in the location indicated by the address.

```
!    *x := y
       ...      y,%o0
       ...      x,%o1
       st       %o0,[%o1]
```

Optional Extensions

I hope that some people will be able to complete this project well before the due date and will have time to do additional optimizations above and beyond the simple code generation done by the black box code. The exact nature and organization of your optimization code is up to you. Some ideas will be discussed in class and I will be happy to talk to you individually. I think the simplest form of optimization would be to implement the peephole optimizer if you did not get a chance to during project 10.

However, after getting the main program working, I would prefer that people spend any extra time going back and fixing other projects that they did not complete.

The output of your program will be compared to the output produced by the black box program. Please begin by getting the program working as described here so that its output exactly matches the **.bak** files. We could also assemble your target code and run it, comparing its output against the expected output (which is also provided in directory **tst**). But if your target code is exactly identical, there is no point; of course it will assemble and run the same way as the output from the black box compiler. Even if you do extension or add optimizations, you must still turn in this first version of **Emit.java** that exactly matches the **.bak** files.

You may also attempt doing some extensions or optimizations, but only after getting the base program working. If you are doing any extensions, the requirement for your compiler is that it should produce code that is functionally equivalent to the code produced by the black box compiler. To test this, the target code your compiler produces can be assembled and run, using **run2**. The output from the running program will be compared to the output of the executable produced by black box compiler and it must be identical.

Generator.class

If you were unable to complete the **Generator.java** file, you may use my **Generator.class** file instead. However, you'll need to modify the **makefile** to NOT compile **Generator.java**.

Grading

The primary consideration for grading will be correctness. The output of your program will be compared to the output produced by the "black box" program, **Main.jar**. Your output should match exactly.

Your code should also be well organized and clearly documented.

Be sure to follow my style guidelines for commenting and indenting your Java code. There is a link on the class web page called "Coding Style for Java Programs." Please read this document. Also look at the Java code I am distributing for examples of the style we are using in this class.

During testing, the grader will compile your **Emit.java** file and link it with my files, including my **Lexer.class**, **Parser.class**, **Checker.class**, and **Generator.class**.

[IF YOU DIDN'T TAKE CS-321 LAST TERM, IGNORE THE NEXT PARAGRAPH...]

I encourage you to use your own files during testing, but I also <u>strongly</u> encourage you to test your **Emit.java** with my **Lexer.class**, **Parser.class**, **Checker.class**, and **Generator.class**, just to make sure it works correctly with them. While there should be no difference, it still seems like a good idea.

Standard Boilerplate...

It is considered cheating to decompile or look inside any **.class** or **.jar** file I provide. If you have questions about what these files do, please ask me!

As before, email your completed program as a plain-text attachment to:

```
cs321-01@cs.pdx.edu
```

Don't forget to use a subject like:

```
Proj 11 - John Doe
```

<u>DO NOT EMAIL YOUR PROGRAM TO THE CLASS MAILING LIST!!!</u>

Your code should behave in exactly the same way as my code. If there is any question about the exact functionality required,

(1) Use my code (the “black box” **.jar** file) on test files of your own creation, to see how it performs.

(2) *Please ask* or talk to me!!! I will be happy to clarify any of the requirements.

Do not submit multiple times.

Please keep an unmodified copy of your file on the PSU Solaris system with the timestamp intact. This is required, in case there are any “issues” that arise after the due date.

In other words: **DO NOT MODIFY YOUR “Emit.java” FILE AFTER YOU SUBMIT IT**. You can create a **p10** directory, copy all files over and keeping working, if you need to.

Work independently: you must write this program by yourself.