Project 8: Intermediate Code Generation (Part 1)

Due Date: Tuesday, February 7, 2006, Noon

Duration: One week

Overview

The goal of this project is to begin the translation of the PCAT source program into three-address instructions, which comprise the intermediate representation.

You will make use of an existing front-end which parses the source text and checks it for semantic errors. The input to the back-end is a well-formed Abstract Syntax Tree (AST). The back-end will work in two steps.

In the first step, the compiler will produce a list of instructions (called the "Intermediate Representation" or "IR instructions"). This intermediate code is halfway between the source-level code and the target machine (SPARC) code. In the second step, the back-end will translate the IR code into SPARC assembly code.

In this project, you will begin writing the code to produce the IR instructions from the abstract syntax tree. In later projects, you will complete the translation into intermediate code. After that, you will implement the second part of the back-end, which will translate the IR code to assembly code.

Each IR instruction is a "three-address instruction." Each IR instruction has an op-code (called **op**) and up to three operands (called **result**, **arg1**, **arg2**).

In this project, you'll generate IR instructions for:

- assignment statements
- procedure call statements
- arithmetic expressions
- variable initialization

In later projects, you'll generate code for expressions involving boolean values, flow of control statements, and other more complex constructs.

The File You Are To Create:

Generator.java

Other Files

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

```
~harry/public_html/compilers/p8
```

Main.java
The main routine which calls the parser, the type-checker, and the code generator. This file has been modified slightly from the previous version (i.e., in Porter's previous CS321) to call **generateIR()** and **printIR()**.

< Lexer.java >
< Parser.java >
< Checker.java >
The files you created last term. If you took CS-321 from me last term, ou may use your files, but these files are not available on the website; only compiled **.class** files are available.)

Lexer.class
Parser.class
Checker.class
You may use these compiled versions if there were any problems with yours or if you did not take CS321 from me last term. DO NOT DECOMPILE THESE FILES. If you have questions on their functionality, ask me!

Ast.java
The classes in this file describe the Abstract Syntax Tree.

PrettyPrint.java
Code to walk the AST and "pretty-print" it. (Modified slightly from previous versions.)

PrintAst.java
Code to walk the AST and print it in full, complete detail, which is useful in checking to make sure the AST is being constructed correctly. For all but the smallest ASTs, the output is difficult to read.

Token.java
FatalError.java
LogicError.java
StringTable.java
SymbolTable.java
Support files from last term, which you can pretty much ignore this term.

Generator0.java
A starter file to get you going on this project.

IR.java
A new file containing support code. The class in this file defines the intermediate representation (IR) instructions.

makefile
This file is used by the Unix **make** utility to compile everything.

tst
This directory contains the testing / debugging data which will be used to evaluate your code. It contains several PCAT programs, along with the output that your compiler should produce.

run
This is a shell script you can use to run a single test from the **tst** directory. It will print any differences between what your compiler produces and the "expected output."

runAll
This is a shell script you can use to run all the tests in the **tst** directory.

go
This is a shell script you can use to run a single test. It prints the program and the output it produced, but does not compare it to the "expected output."

Main.jar
This is the "black box" solution code, which was used to produce the "expected output" files in **tst**.

PCATDifferences.pdf
"A Review of the Abstract Syntax Tree: How a PCAT Program is Represented"
Please review this document, unless you took CS-321 from me last term.

ReviewOfAST.pdf
"PCAT Delta: Porter's Version vs. Tolmach's Version"
Discusses the differences between the PCAT languages used by me and Tolmach.

TolmachsAST.pdf
"The Abstract Syntax Tree: Differences Between Tolmach's and Porter's Representations"
This documents discusses the differences between the Absract Syntax Tree (AST) produced by my front-end and the AST produced by Andrew S. Tolmach (AST).

Getting Started

Create a new directory called "**p8**" for this project and copy in all the files I am providing.

In this project, you'll need to walk the AST so take a close look at the file called **PrettyPrint.java**, which does exactly that. I suggest that you use it as a starting framework when creating your **Generator.java** file. Study **PrettyPrint.java** carefully and consider using it as a guide when writing the code to walk the AST.

Also, during debugging, you may use **PrettyPrint.java** so it is helpful to understand what it does.

Testing

The **tst** subdirectory contains a bunch of files with names like:

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

The **.bak** files are the desired output files your compiler should produce. One is for the **stdout** and one is for the **stderr** produced by your compiler.

During testing you may use the executable shell script called **go**. The **go** script makes it easy to run your compiler on a particular test. Typing:

```
% go simple
```

or just:

```
% !g
```

is easier than typing:

```
% java Main < tst/simple.pcat
```

I am providing another shell script named **run**. You can use **run** to execute your compiler on a particular test and compare the results to the expected output (**.bak**) files. Any differences will be shown.

In the following example, you type **run simple** to see if your compiler produces the expected output. It does not. The < indicates what the expected output is and the > indicates what your compiler produced:

```
% run simple
simple0:
19c19
< ! ASSIGNMENT STMT...
---
> ! Assign STMT...
23,25c23,25
< ! ASSIGNMENT STMT...
<                 t4 := &j
<                 *t4 := k
---
> ! Assign STMT...
>                 t5 := &j
>                 *t5 := k
```

Another shell script I am providing is called **runAll**. It will run all the tests in the **tst** directory. If everything is okay, it will simply list all the tests that were run.

During your debugging, you are free to modify any of the files I distribute. For example, you might want to modify **Main.java** to call **printAST** or you might want to modify one of the files in **tst** to focus on testing a specific thing.

However, after you are finished and ready to submit, be sure to test your **Generator.java** with my files, exactly as they are distributed. When we test your code, we will use only your **Generator.java** file. It will be compiled with my files, *as they were distributed.*

I am also providing a working solution to this assignment in a file called **Main.jar**. This is the "black box" solution. You may use this to answer questions about exactly what the program should do in specific cases not covered by this document. (I used the black box code to produce the output **.bak** files in the **tst** directory.)

[My **Main.jar** "black box" program also has the code for Project 9 in it, as well as Project 8, so it will actually do more code generation than you are required to do for Project 8.]

The Intermediate Representation (IR) Instructions

A new file, called **IR.java**, is provided. It contains code to represent the IR instructions.

This file defines a class call **IR**; each instance of the **IR** class is a single intermediate representation (IR) instruction. Each instruction has an opcode and several operands/arguments. Here are the fields in each **IR** object:

```
int         op;
Ast.Node    result;
Ast.Node    arg1;
Ast.Node    arg2;
String      str;
int         iValue;
IR          next;
```

The opcode (**op**) is an integer telling which instruction it is. Here are the different opcodes. There are instructions for data movement, arithmetic, etc. [You will not use all of these in this project, but they are listed here for reference.]

```
Data movement:
   assign
   loadAddr
   store
   loadIndirect
Arithmetic computation:
   iadd
   isub
   imul
   idiv
   imod
   ineg
   fadd
   fsub
   fmul
   fdiv
   fneg
   itof
```

Procedures / Calling:

```
call
param
resultTo
mainEntry
mainExit
procEntry
formal
returnExpr
returnVoid
```

Testing and Branching:

```
label
goto
gotoiEQ
gotoiNE
gotoiLT
gotoiLE
gotoiGT
gotoiGE
gotofEQ
gotofNE
gotofLT
gotofLE
gotofGT
gotofGE
```

Misc:

```
comment
alloc
```

Input / Output:

```
readInt
readFloat
writeInt
writeFloat
writeString
writeBoolean
writeNewline
```

Each opcode is assigned an integer and there are a number of constants defined in the **IR.java** file:

```
static final int OPassign = 1;
static final int OPloadAddr = 2;
static final int OPstore = 3;
...
static final int OPwriteNewline = 48;
```

The **op** field will contain one of these integer values, indicating which instruction it is.

The IR instructions will be chained together in a linked list, linked together on their **next** field. The variables **firstInstruction** and **lastInstruction** point to the first and last instructions in the list of instructions constructed so far.

To create an instruction, you can invoke the constructor. For example:

```
IR inst;
...
inst = new IR (OPassign);
```

The constructor will add this instruction to the end of the growing list of instructions.

Some instructions will have no operands. For example, the **returnVoid** instruction has no operands. Some operands, such as **assign** and **iadd**, will have several operands. The **iadd** instruction has three operands and it looks like this:

```
x := y + z
```

One operand is called the **result** and the other two are called **arg1** and **arg2**.

The assign instruction has a **result** operand and another operand, which is stored in **arg1**:

```
x := y
```

Each IR instruction object has room for several operands. There is room for up to 5 operands (**result**, **arg1**, **arg2**, **iValue**, **str**), although no instruction uses more than three operands.

The **result**, **arg1**, and **arg2** operands will always point to nodes in the AST. In the above "assign" IR instruction, for example, the **result** operand would point to the **VarDecl** node representing the variable "x" and the **arg1** operand would point to the **VarDecl** representing the variable "y."

Some instructions have an operand that is an immediate integer; such an argument would be stored in the **iValue** field. Some instructions have an operand that is a Java "String"; such an argument would be stored in the **str** field.

To help you create new instructions, the **IR.java** file contains a number of static methods, such as:

```
static void assign (Ast.Node result, Ast.Node arg) {
    IR inst = new IR (OPassign);
    inst.result = result;
    inst.arg1 = arg;
}
...
static void iadd (Ast.Node result, Ast.Node arg1, Ast.Node arg2) {
    IR inst = new IR (OPiadd);
    inst.result = result;
    inst.arg1 = arg1;
    inst.arg2 = arg2;
}
```

Therefore, to generate an assign instruction, you can code something like this:

```
IR.assign (x, y);
```

The **IR.java** file also contains a static method called **printIR**, which will print out the entire list of IR instructions. The **main** method first calls **generateIR** to generate the intermediate code and then **main** calls **printIR** to print out the instruction list.

The Starter File: Generator0.java

I am providing a file called **Generator0.java** to help you get started and to provide the general framework. You will modify this file, calling it **Generator.java**. Be sure to replace
 <Your Name Here> -- <Date>
with your name and the date. You should always put your name and the date in your programs.

There will be a single instance of the class **Generator**, created by the **main** method, which will be called **generator**. The **Generator** class contains a method called **generateIR**, which the **main** method will call. **GenerateIR** will be passed a pointer to the Abstract Syntax Tree (AST); it will walk the AST, generating the IR code by calling methods like **IR.assign** and **IR.iadd**. You will need to write a number of additional methods to facilitate the code generation process.

Basically, you'll need something like a recursive walk of the AST and you'll need to create a method for each kind of AST node, much like we did for the **Checker** class in CS-321.

The starter file, **Generator0.java**, also contains several things that will be used in later projects:

```
//
// Constants
//
static final int INITIAL_VARIABLE_OFFSET      =  -4;
static final int VARIABLE_OFFSET_INCR         =  -4;
static final int INITIAL_FORMAL_OFFSET        = +68;
static final int FORMAL_OFFSET_INCR           =  +4;
static final int REGISTER_SAVE_AREA_SIZE      = +64;
static final int DISPLAY_REG_SAVE_AREA_OFFSET = +64;

static final int INTEGER_MODE = 1;
static final int REAL_MODE    = 2;
static final int STRING_MODE  = 3;
static final int BOOLEAN_MODE = 4;

//
// Fields
//
int lexicalLev = 0;
int maxLexicalLevel = 0;
Ast.Body currentBody;
Ast.StringConst stringList = null;
Ast.RealConst floatList = null;
int nextLabelNumber = 1;
int nextTempNumber = 1;
```

Ignore this material, but leave it in.

The starter file, **Generator0.java**, also contains a couple of methods that will come in handy during code generation. They are:

```
newTemp()
newLabel()
```

Methods in "IR.java"

In **IR.java** there are a number of static methods that can be used to generate IR instructions. Each method has the same name as one of the opcodes. (There is one exception. The method to create a **goto** instruction is called **go_to**, since "goto" is a Java keyword.)

You should call one of these methods whenever you wish to generate a new IR instruction. This method will create a new **IR** object and add it to the growing linked list of instructions. For example, the following method will create a new instruction to perform an "integer add":

```
static void iadd (Ast.Node result, Ast.Node arg1, Ast.Node arg2)
```

Notice that the types of the **result**, **arg1**, and **arg2** parameters are all pointers to **Ast.Node**s. For some of the instructions, such as **iadd**, the parameters will be pointers to nodes in the abstract syntax tree.

For other instructions, the parameters will be integers or Strings. For example, the **goto** instruction takes a single argument, which is a String.

```
String endLabel = ...;
...
IR.go_to (endLabel);
```

The **IR.java** file also contains the **printIR()** method, which is called by **main()** after your code has finished generating the list of IR instructions. It produces a printout looking something like this:

```
=====  Intermediate Code Follows  =====
! MAIN...
                mainEntry
! VAR INITIALIZATION...
                t1 := 9999
                i := t1
     .
     .
     .
! ASSIGNMENT STMT...
                t6 := &j
                *t6 := k
     .
     .
     .
! MAIN EXIT...
                mainExit
=======================================
```

Note that comments are included in the intermediate code sequence. (We'll use the SPARC convention that comments start with a "!" and go through end-of-line.) It turns out that the IR instructions get a little difficult to read since there are often so many of them and so many temporary variables. Adding comments to the output is relatively easy and it makes reviewing the output simpler.

There is a IR instruction opcode (called **OPcomment**) which is used to put comments within the linked list of IR instructions. You may generate comment "instructions" just like you generate other IR instructions. (Of course, when it comes time to translate IR instructions into SPARC code, these comments will end up producing zero SPARC instructions.)

Here is the method to generate a comment:

```
static void comment (String s) {
    IR inst = new IR (OPcomment);
    inst.str = s;
}
```

To generate part of the above IR code, your code might execute statements such as:

```
IR.comment ("VAR INITIALIZATION...");
temp = genExpr (p.expr);
IR.assign (p, temp);
```

The routine **genExpr()** is a method (which you will write) which is passed a pointer to an AST subtree representing a source code expression. It will generate whatever code is necessary to evaluate that expression and will return the identity of a variable containing the result. I'll discuss **genExpr()** more later.

The starter file **Generator0.java** contains the methods **newLabel()** and **newTemp()**.

Labels are simply Java Strings with the following format:

```
"Label_xxx"
```

where *xxx* is a number. The method **newLabel()** will return new String each time it is called. It increments the number so that the label it returns will always be unique.

```
String newLabel () {...}
```

In some textbooks, the instructions are numbered and "goto" instructions use numbers to specify the target instruction. Our instructions are not numbered; instead we will use symbolic labels. These symbolic names will ultimately be copied into the target instructions and the SPARC assembler will take care of associating them with specific memory addresses.

There is an IR opcode (called **OPlabel**) which is used to add a **label** to the list of IR instructions. The **label** instruction is not really an instruction, since it will never "do" anything. Instead, **label**s are more like placeholders, acting as targets for **goto** instructions. **Label**s are treated like other instructions since they are created and added to the list of IR instructions.

For example, you could use the following code:

```
String lab = newLabel ();
IR.label (lab);
...
IR.goto (lab);
```

to generate the following IR code:

```
Label_17:
        ...
        goto Label_17
```

Note that the **str** field of the IR object is being used to hold the label. All fields that are not used will have their default values of zero.

Temporary Variables

During IR code generation, it is frequently necessary to create a temporary. I am providing the method **newTemp()** for creating a new temporary variable.

```
Ast.VarDecl newTemp () {...}
```

Every time **newTemp()** is called, it will create a new temporary name, similarly to the way **newLabel()** creates a new label name each time it is called. The **newTemp()** method will then create a new variable with this name and return it.

Each temporary will be given a name (such as "`t15`", "`t16`", etc.). These names may conflict with source code variable names, but we don't care. After all, source code names may conflict with each other. (For example, the PCAT program may have two completely unrelated variables named "x." Thus, we will not be able to include source code variables names in the SPARC assembler file without first modifying them. The only place where confusion might arise is in the output from **printIR**(), but this will not affect the correctness of the compiler.

Each time the **newTemp()** routine is called, it will add a **VarDecl** node to the AST data structure for the procedure being compiled. Thus, it will look just the same as if the source code had included the declaration:

```
VAR t5: INTEGER := ...;
```

With temporaries, we will not bother with a "type" or initializing expression, so **newTemp()** will leave the **varDecl.typeName** and **varDecl.expr** fields set to NULL. It will also set the **varDecl.lexLevel** field to -1. Later, this will signal that we are dealing with a temporary variable, not a "real" variable.

As you know, **VarDecl** nodes are kept in a linked list in the **Body** node. In order for **newTemp()** to add a new variable to the end of this list, it needs to know which **Body** we are currently generating code for. Toward this end, there is a field in **Generator.java** that will point to the current **Body** node:

```
Ast.Body currentBody;
```

Your code must set and maintain this variable whenever it enters a new body. There is a **Body** node for the main program and a **Body** node for each procedure. Temporaries can be added to the main **Body** or to any of the other **Body**s.

The **newTemp()** routine returns a pointer to the **VarDecl** node that was just allocated.

Every time an IR instruction references (i.e., uses, accesses, or modifies) a variable (either a temporary variable or otherwise), that instruction must be set to point directly to the **VarDecl** describing the variable in question. In a subsequent project, we will compute offsets for the variables. Then, when we are ready to generate SPARC code, we will be able to look at the IR instruction, follow the pointer to the **VarDecl** node, and get the offset to use in the target instruction.

How to Get Started

I recommend you use the code from **PrettyPrint.java** as a starting skeleton. Make a copy of it, then go through and eliminate all "print" statements and anything relating to printing. Eliminate everything relating to indentation. Then rename each method from **ppXXX** to **genXXX**. For example, if you have a method named **ppExpr** rename it to **genExpr**. Now you should have a skeleton program that walks the entire abstract syntax tree in more-or-less logical, source-code order without doing anything.

Next, merge this code with the starter file, **Generator0.java** to give your initial **Generator.java** file. This should compile and should walk the AST, but will not generate any IR code yet.

Don't forget to modify the comments as well. It looks *really bad* to see comments that say "Print an expression" in a routine that generates IR instructions! If the comment in **PrettyPrint.java** said "Print an expression," change it to something like "Generate code for an expression." At the time you fill in the methods to perform the code generation, you can expand this comment, adding detail about parameters, results, etc.

Next, begin modifying these methods to generate the IR code, by adding calls to **newTemp()**, **newLabel()**, and the IR methods for generating instructions (e.g., **IR.assign**, **IR.go_to**, **IR.iadd**, etc).

If you start from **PrettyPrint.java**, you'll end up with several methods that will not be used in this project. For example, you might have a **genArrayType** method, that does nothing. Do not delete these methods; leave them in your code. In the later projects, we'll continue with this same file and add more code generation to it. Those methods will be needed then. In particular, we will need to walk the "type"-related nodes in the next project, so don't eliminate them.

Variable Initialization

The first thing to do in each procedure (or main) body is initialize the local variables. For each **VarDecl**, you'll need to generate IR code such as:

```
! VAR INITIALIZATION...
                t5 := ...
                i := t5
```

The **VarDecl** will contain an initializing expression. In this project, you'll be writing a routine called **genExpr()**, which will discussed in the next section. **GenExpr()** will generate the code for the expression and return the "place" where the result can be found. In general, **genExpr()** may generate several instructions; these are symbolized by "t5 := …" above. After calling **genExpr()**, you can generate the assignment statement, by calling **IR.assign()**.

Don't forget that for some variables (namely temporary variables) there will be no initializing expression. For them, generate nothing.

The genExpr() Routine

A key routine you'll need to write will be called **genExpr()**. This routine will be passed a pointer to an AST subtree representing an expression. The **genExpr()** routine will generate the IR instructions to evaluate the expression. The IR code produced by **genExpr()** will, when executed, evaluate the expression and move it into a variable.

In the case that the expression is a simple variable, **genExpr()** will generate no code and will return that variable.

In the case that the expression is more complex, involving operators such as addition or multiplication, **genExpr()** will create a new temporary variable and generate code that, when executed, will evaluate the expression and move the result into that temporary variable. **genExpr()** will return the temporary variable it created.

In the case that the expression is a constant (either integer or real), **genExpr()** will create a temporary variable, generate code to move the value into the temporary, and return the temporary.

[In the lecture notes, this returned variable is the ".place" attribute.]

When we say that **genExpr()** will "return a variable," we mean that **genExpr()** will return a pointer to a **VarDecl** node representing the variable.

Here is the header (i.e., method prototype) for **genExpr()**:

```
Ast.Node genExpr (Ast.Node t)
        throws FatalError
```

You should generate IR code for the following binary and unary operators.

+ - * / div mod unary- unary+ int-to-real

The operators yielding boolean results (**and**, **or**, **<**, **<=**, **>**, **>=**, **=**, and **<>**) will be handled in a later project. None of the test files use these operators. [These operators must be evaluated using "short-circuit" evaluation, which will require a more complex approach.]

There is an opcode for each of the arithmetic operations. For example,

```
IR.iadd (x, y, z)
```

generates the following IR instruction:

```
x := y + z        (integer)
```

The following instructions are relevant here:

```
IR.ineg        x := - y          (integer)
IR.iadd        x := y + z        (integer)
IR.isub        x := y - z        (integer)
IR.imul        x := y * z        (integer)
IR.idiv        x := y DIV z      (integer)
IR.imod        x := y MOD z      (integer)

IR.fneg        x := - y          (float)
IR.fadd        x := y + z        (float)
IR.fsub        x := y - z        (float)
IR.fmul        x := y * z        (float)
IR.fdiv        x := y / z        (float)

IR.itof        x := intToReal(y)
```

The **genExpr()** method should generate code for the following kinds of node: **BinaryOp**, **UnaryOp**, **FunctionCall**, **IntegerConst**, **RealConst**, **ValueOf**, and **IntToReal**.

We will deal with the following nodes in a later project: **BooleanConst**, **NilConst**, **ArrayConstructor**, **RecordConstructor**, **ArrayDeref**, **RecordDeref**, and **StringConst**. Don't worry about these now.

Assignment Statements and LValues

There are several IR instructions of interest here. They are:

```
IR.assign          x := y
IR.loadAddr        x := &y
IR.store          *x := y
IR.loadIndirect    x := *y
```

First consider the source code:

```
(* PCAT source: *)
x := y;
```

The IR code you should generate is:

```
t5 := &x
*t5 := y
```

You might ask: Why not generate the following IR instruction sequence instead? (It would work and clearly it has fewer IR instructions.)

```
x := y
```

The primary reason is that we want to use a general approach that can handle all cases in the same way. The thing on the lefthand side of an assignment is an "L-Value," but it won't always be a simple variable name. When it is more complex, we'll need to generate a sequence of instructions.

From the point of view of the assignment statement, we want to generate the same code, regardless of whatever L-Value we might find on the lefthand side of the assignment.

As an example, consider the following PCAT assignment statement.

```
a [ 55*foo(1,2,3) ] := y;
```

Obviously, lots of instructions will be required to compute which address to store into. (In fact, these instructions will involve a subroutine call and their execution may involve an *arbitrary* or even non-terminating amount of computation!)

In our approach, all of the address computation will be done in one place and then we will use either of the following two instructions, depending on whether we have an L-Value or an R-Value:

```
IR.store           *x := y
IR.loadIndirect     x := *y
```

The key is to create a single method to do all the address calculations:

```
Ast.Node genLValue (Ast.LValue t)
```

The **genLValue()** method will generate code to compute an *address*, not a value. It will generate code to store that *address* into some variable (always a temporary) and will return that temporary. Thus, any method that calls **genLValue()** to process an L-value will get back the identity of a variable holding an address.

In the case of the PCAT assignment statement

```
x := y;
```

we will generate the following code:

```
t5 := &x     <-- produced by genLValue
*t5 := y     <-- produced by genAssignStmt
```

But what about an L-value used as an R-value? The righthand side of this source code assignment statement (**x := y;**) contains an L-value (**y**) used as an R-Value. Why didn't we generate the following code, which first computes the address of **y** and then gets the value from that address?

```
t5 := &x
t6 := &y     <-- produced by genLValue
t7 := *t6    <-- get the R-Value from memory
*t5 := t7
```

Of course this code would execute correctly and might even be easier to generate, but we will implement a small optimization in the case where we have a simple variable (like **y**) occurring as an R-Value. (When we have array deref's or record deref's, we'll generate the longer code sequences, involving the **loadIndirect**, but these will be handled in the next project, not now.)

The optimization we'll perform will recognize the case where the R-Value is a simple variable and generate the shorter (two instruction) version shown previously. Since this occurs so often, the optimization is worthwhile and will dramatically reduce our IR sequences.

Here is how to proceed. Recall that we have **ValueOf** nodes, which "shroud" or "wrap" around L-values to turn them into R-values. The method

```
Ast.Node genValueOf (Ast.ValueOf t)
```

will be responsible for generating code to get the final value into a variable. This may involve arbitrary address calculation (i.e., a call to **genLValue()**) followed by the creation of a temporary, followed a **loadIndirect** instruction. The method **genValueOf()** will then return the name of the variable containing the result.

However, when the L-value is a simple variable, we want to avoid generating a **loadIndirect** instruction in **genValueOf** (). You can do this in **genValueOf()** by first looking to see what type of L-value we have. If it is a simple **Variable**, we can just return it. We generate no instructions. On the other hand, if **genValueOf()** sees that it has an **ArrayRecord** or a **RecordDeref**, it will need to generate code to compute the address, create a new temporary, generate a **loadIndirect** into that temporary, and return that temporary.

In this project, we will not deal with **ArrayDeref**s or **RecordDeref**s, so source statements like the following do not need to be handled yet:

```
r.f.g := a[4];
```

Procedure Invocation

In this project, you should generate code for all **FunctionCall** and **CallStmt** nodes.

Consider the following source code for a "call" statement:

```
foo (expr1, expr2, expr3, expr4, expr5, ...);
```

You should generate the following IR code:

```
t20 := ... expr1 ...
t21 := ... expr2 ...
t22 := ... expr3 ...
t23 := ... expr4 ...
t24 := ... expr5 ...
...
param 1,t20
param 2,t21
param 3,t22
param 4,t23
param 5,t24
...
call foo
```

To invoke a procedure, you should generate a sequence (zero or more) of **param** instructions, followed by a **call** instruction, and followed by an optional **resultTo** instruction (if the procedure returns a value).

Note that the above code sequence evaluates all arguments and moves their values into variables before generating the first **param** instruction. In general, the argument expressions could involve additional subroutine calls. You want to make sure that all the argument evaluations are complete before you generate the sequence of **param** instructions.

Next consider a statement, such as

```
x := ... expr ...;
```

For an assignment statement, you'll generate the following IR code:

```
t38 := &x
t39 := ... expr ...
*t38 := t39
```

The IR code in the box was generated by **GenExpr**, which returns the name of a temporary holding the resulting value. (In this case, **GenExpr** returned **t39**.)

Any expression may be a function call, as in the following assignment statement:

```
x := bar (expr1, expr2, expr3, expr4, expr5, ...);
```

For this, you should generate the following IR code:

```
t38 := &x
t40 := ... expr1 ...
t41 := ... expr2 ...
t42 := ... expr3 ...
t43 := ... expr4 ...
t44 := ... expr5 ...
...
param 1,t40
param 2,t41
param 3,t42
param 4,t43
param 5,t44
...
call bar
resultTo t39
*t38 := t39
```

To generate the above IR instructions, you can use these methods from **IR.java**:

```
IR.param (param-number, temp);
IR.call (ptr-to-PROC_DECL);
IR.resultTo (temp);
```

This section discusses how to generate IR instructions for a **FunctionCall** node. The generation of instructions for a **CallStmt** node is quite similar.

The first and last instructions come from the fact that this is an assignment statement. Notice that the address of the L-value (**x**) is computed first, followed by the computation of each of the argument expressions. Each of these computations could involve lots of additional instructions. The order in which these computations is done is critical, since each could involve calls to other procedures with side-effects.

After computing the values of each of the argument expressions, we are ready to generate the code to invoke the procedure **foo**. First, we have a **param** instruction for each of the arguments. Later, we will translate the instruction

```
    param i,txxx
```

into SPARC code to move the value in the temporary into the appropriate %o register (or into the appropriate frame slot for parameters beyond six).

The **call** instruction contains a pointer to the **ProcDecl** node for the procedure being called.

(Later, we will need to rename each of the procedures, since the procedure names in the source code may not have unique names, while the procedures in the target file must have unique labels. We will rename "bar" to something like "proc23_bar." At that time, we will store the new name in the **ProcDecl** node and use it when doing the final SPARC code generation.)

Finally, in the case of a **FunctionCall** node, we need to generate a **resultTo** instruction. After the procedure returns, the result will be in %o0. The **resultTo** instruction tells us where to store it. If the procedure invocation is from a **CallStmt** node (i.e., we are calling a void procedure at the statement level), we will not generate the **resultTo** instruction.

Note that we need to make two passes over the list of **Argument**s associated with a **CallStmt** or **FunctionCall**. In the first pass, we will call **genExpr()** for each argument expression. As we look at each argument expression, we will not generate the **param** instruction. (Why? Consider the case where there are nested calls. The **param** instructions should all directly precede the **call** to which they apply.)

Then, we will make a second pass over the **Argument** list, and for each, generate the **param** instruction. One problem is that we need to keep a list of all the locations of the argument computations during the first pass. This is nothing more than a list of temporary variables, but it could be arbitrarily long. Do we need to create a new sort of linked list to deal with this?

Rather than messing with a new linked list, I have simply added a field (called **location**) to the **Argument** node. You may use this field to hold the temporary variable associated with the **Argument** as it is returned from **genExpr()**. In the second pass over the list of **Argument**s, the generation of the **param** instructions should be straightforward.

Grading

The primary consideration for any program is correctness: programs must not contain any bugs. Your program will be run on the test files in the **tst** directory. Your output must match exactly.

Your code should also be well organized and clearly documented. The indenting must follow my standards. Every method must have a comment describing what it does and the comments must match the code.

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 **Generator.java** file and link it with my files, including my **Lexer.class**, **Parser.class**, **Checker.class**, and **PrettyPrint.class**.

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

I encourage you to use your own Java code during testing, but I also strongly encourage you to test your **Generator.java** with my **Lexer.class**, **Parser.class**, **Checker.class**, and **PrettyPrint.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 8 - John Doe
```

DO NOT EMAIL YOUR PROGRAM TO THE CLASS MAILING LIST!!!

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 "Generator.java" FILE AFTER YOU SUBMIT IT**. You can create a **p9** directory, copy all files over and keeping working, if you need to.

Work independently: you must write this program by yourself.