Project 10: Intermediate Code Generation (Part 3)

Due Date: Tuesday, February 28, 2006, Noon
Duration: Two weeks

Overview

In this project, we will finish generating the intermediate representation and will take care of some other tasks (such as laying out the activation records and assigning offsets to local variables).

File you are to modify:

Generator.java

Files

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

```
~harry/public_html/compilers/p10
```

IR.java
Changed to including additional printing.

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. This file just contains a "dummy" class, which you can modify if you have time after finishing the primary assignment. (The black box **Main.jar** uses this dummy code.)

Main.java
Altered to call **Peephole.optimize** and to print the variable offsets.

Lexer.java
Parser.java
Checker.java
Files you created last term.

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

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

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

tst
runAll
Same as before, but altered for this project. All the test programs from Project 9 are included, along with several new test files. (The output for the Project 9 test programs will be different, due to new code added for this project.)

Main.jar
The new "black box" code.

Reducing the Number of Temporaries

In projects 8 and 9 we generated a lot of temporary variables. Too many!

The first place that we are generating temporaries unnecessarily is in assignments to simple variables. Consider this source code:

```
i := j;
```

Previously, we generated this IR code:

```
! ASSIGNMENT STMT...
                t3 := &i
                *t3 := j
```

This can be "optimized" to the following IR code:

```
! ASSIGNMENT STMT...
                i := j
```

This optimization can occur only if the thing on the left-hand side is a simple variable. If it is something more complex (like an array or record dereference), then we will still need to generate code as we did in projects 8 and 9, with the "indirect store" instruction.

In the general case (which includes array and record dereferencing), the code for assignment statements, **genAssignStmt()**, will call **genLValue()** to deal with the left-hand side, which could be quite complex. You will call **genLValue()** to generate code for an L-value and it will return the name of a temporary into which we have stored an address. The code for **genAssignStmt()** can then store indirectly using that address.

You should implement the optimization in **genAssignStmt()**; change it to look and see what the left-hand side is. If it is a simple **Variable**, it should avoid calling **genLValue()** and generate the optimized version, using the **IRassign** instruction. Otherwise, you can call **genLValue()**, which returns a temporary. Then **genAssignStmt()** will generate a **IRstore** instruction, instead of the **IRassign** instruction.

Eliminating Temporaries for Constants

Another place that temporaries are created unnecessarily is for each constant in the source code. In the last project, the following source code:

```
     i := j + 1;
```

resulted in this IR sequence:

```
     ! ASSIGNMENT STMT...
                     t2 := &i
                     t3 := 1
                     t4 := j + t3            (integer)
                     *t2 := t4
```

Using the optimization discussed in the previous section, this code will become:

```
     ! ASSIGNMENT STMT...
                     t3 := 1
                     t4 := j + t3            (integer)
                     i := t4
```

In addition, we want to optimize the material that is underlined. In this project, you should modify the code generator to generate the following IR code:

```
     ! ASSIGNMENT STMT...
                     t1 := j + 1             (integer)
                     i := t1
```

In the last project, you were moving the constant into a temporary and then using the temporary. In this optimization, you will just use the constant directly. You should change **genExpr()** so that, when applied to constants, it will return a pointer to the AST node representing the value. This constant will then be used by whichever routine called **genExpr()**.

Up to now, we have said repeatedly that **genExpr()** will always return the “place” (i.e., the name of a variable) where the computed result will be found. That is, we said that **genExpr()** will return a pointer to either the **VarDecl** or the **Formal** node describing the variable.

To perform this optimization, this "invariant" will be relaxed a bit. Now **genExpr()** will return either a variable (i.e., a pointer to a **VarDecl** or a **Formal** node as before) or a constant (i.e., a pointer to an **IntegerConst** or **RealConst** node).

This optimization will affect these routines:

> **genIntegerConst()**
> **genRealConst()**
> **genBooleanConst()** and
> **genNilConst()**

(Some students may not have methods with these names. Instead, they may have put the code to deal with **IntegerConst**, **RealConst**, **BooleanConst**, and **NilConst** nodes directly into the **genExpr()** method.)

For the NIL constant, **genExpr()** should return a pointer to an **IntegerConst** node with **iValue** equal to zero.

Earlier, I suggested that you create three AST node of class **IntegerConst** at the beginning of **generateIR()** and fill them in with **iValues** of 0, 1, and 4. I suggested you call these **constant0**, **constant1**, and **constant4**. These nodes should be used as the return values of **genBooleanConst()** and **genNilConst()**.

For the "nil" value, you can just use the **IntegerConst** node with **iValue** of 0, since a null pointer in PCAT will be implemented with a 32-bit integer equal to zero. When you are generating the code for array dereferencing address calculations (as discussed later in this document), you'll need an **IntegerConst** with **iValue** equal to 4; you can use **constant4** at that time.

The processing for **BooleanConst**s should be improved in the same way as for integer and real constants, to eliminate unnecessary temporaries. For example, the following source:

```
b := true;
```

should now generate:

```
! ASSIGNMENT STMT...
           b := 1
```

instead of:

```
! ASSIGNMENT STMT...
           t2 := &b
           t3 := 1
           *t2 := t3
```

More complicated stuff, like "IF b THEN..." and "b := (i<j):" is still handled the same way with branches.

Runtime Errors and Boilerplate

Several things can go wrong during program execution. For the PCAT language, the following five errors may arise during runtime:

`runtimeError1: Allocation failed`
Memory allocation failed during array or record construction.

`runtimeError2: Pointer is NIL`
This may be detected during array dereferencing or during record dereferencing.

`runtimeError3: READ statement failed`
A call to the system routine to get input has failed.

`runtimeError4: Array index out of bounds`
During array dereferencing, the index expression was less than zero or too large.

`runtimeError5: Count is not positive in array constructor`
During array construction, the count expressions are evaluated. They must be >= 1.

We will handle runtime errors as follows. When producing the target ".s" file, we will include some "boilerplate." This is canned material which will always be included in every output file produced by this compiler. It is "canned" in the sense that it is fixed and unchanging and could be produced by a single, giant print statement.

Among other things, the boilerplate will contain 5 labels: **runtimeError1**, **runtimeError2**, ... **runtimeError5**. After each label, the boilerplate will include code to print out a nasty little message (like "Allocation failed" or "Pointer is NIL") and terminate program execution.

During program execution, if anything goes wrong, we'll simply branch to the appropriate label. More precisely, the compiler will generate code to test and branch to labels such as **runtimeError1** if certain conditions are detected at runtime.

It is usually desirable to print out more information than just a fixed message when a runtime error arises. At a minimum, the source code line number should be printed. (At least our PCAT system will give a descriptive error message and terminate gracefully; I once heard of a language that would either keep on executing or print out something cryptic and uninformative like "core dumped." "Core" was an ancient form of digital memory using small, doughnut-shaped magnets, used back when this language was popular... I digress...)

When some unrecoverable error occurs during runtime execution, it is very helpful to the programmer to print out a source code line number. One technique to print the line number works as follows. A single register (or static storage location) is set aside to hold an integer. This word will always contain the current source code line number. Every time the compiler begins to generate code for a statement (e.g., in **genStmts()**), it generates an instruction or two to save the number of the current source line in this word. Happily, we have this information stored in the **pos** field of each AST node. Our boilerplate might then contain code that, when a runtime error occurs, prints the appropriate error message and also prints the value of this saved word under the rubric "Error at source code line %d..." There are many variants on this scheme. If several source code files are involved, additional words would be necessary to identify which source code file was involved at the time of the error.

We will not save or print source code line numbers during runtime errors; we'll just print a message.

Record Offsets and Sizes

Each field in a record will be 4 bytes in our implementation of PCAT.

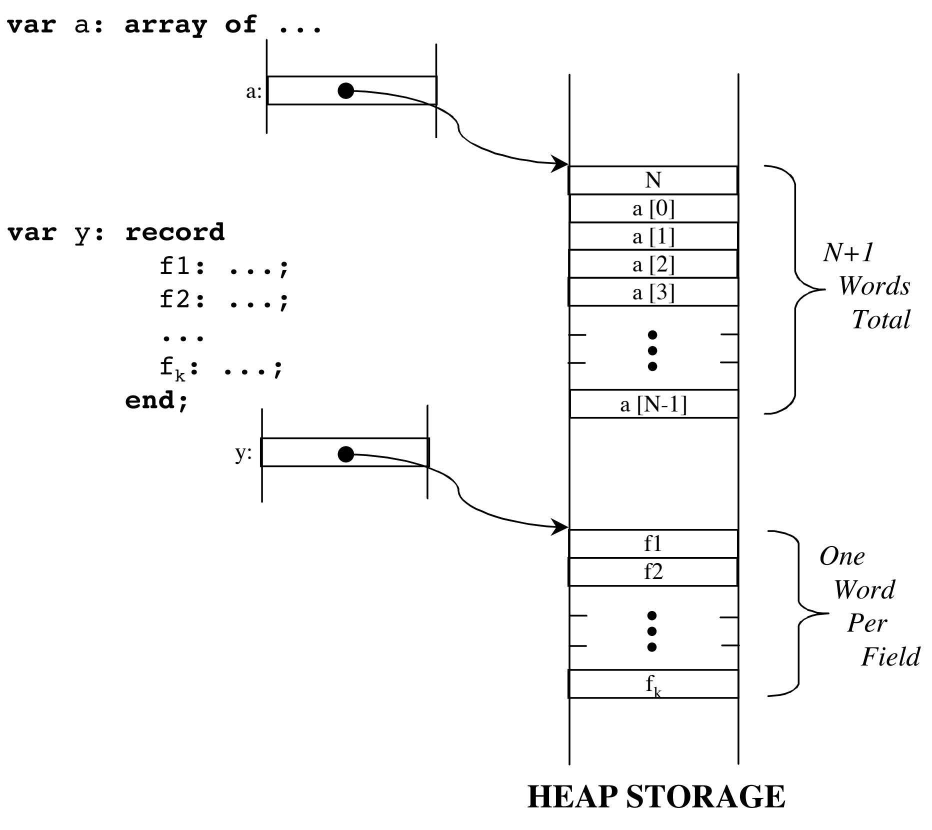
Recall that the **RecordType** node points to a linked list of **FieldDecl** nodes (with one **FieldDecl** node per field). A new field called **offset** has been added to each **FieldDecl** node and a field called **size** has been added to the **RecordType** node. You must set these appropriately, by walking the linked list of fields.

The **Body** node contains pointers to several linked lists: **typeDecls**, **varDecls**, **procDecls**, and **stmts**. Notice that you must process the **typeDecls** before the **varDecls** and **stmts** so that record offsets will be filled in before they are needed.

Representation of Arrays and Records

Consult the following diagram:

Layout of Arrays and Records



Consider the two definitions:

```
type MyArr is array of ...;
     MyRec is record
                 f1: ...;
                 f2: ...;
                 ...
                 fk: ...;
               end;
var x: MyArr := ...;
var y: MyRec := ...;
```

Array-valued variables (like **x**) and record-valued variables (like **y**) will each be allocated 4 bytes. In this example, the variables **x** and **y** are local variables, but we could have similar variables as formal variables or as elements in other record and arrays. In any case, each variable will be given 4 bytes, either in an activation record, or within another record or another array.

At runtime, **x** and **y** will either be NIL (i.e., zero) or will point to a block of words which have been allocated in the heap. The only way to allocate words in the heap is with an array constructor (using the **T {{ ... }}** syntax) or with a record constructor (using the **T { ... }** syntax).

At the time an array constructor is executed, we will know how many elements the array will have, but we may not know this number until runtime (in PCAT, anyway). With record constructors, we will know at compile-time how many bytes are to be allocated. It will be 4 times the number of fields.

An array of N elements will be stored in N+1 words. The extra word will store the N, the size of the array. This will be used in an "array index out-of-bounds check" every time the programmer tries to access one of the elements using the array dereferencing (i.e., a[i] or "array indexing").

The elements of an array are numbered from 0 to N-1. The diagram is drawn with higher (larger) addresses toward the bottom of the page.

A record with K fields will be stored in K*4 bytes.

L-Values and R-Values

There are only three kinds of node that can appear wherever an L-value is allowed: **Variable**, **ArrayDeref**, and **RecordDeref**. Here is an example of source code corresponding to each of these:

```
x
a[i]
r.f
```

Of course, each of these kinds of L-value can be used as an R-value. For example, in the following source code, we see array dereferencing used as an R-value:

```
x := a [ (i+1)*3 ];
```

In the following source code, we see record dereferencing occurring in an L-value:

```
person.address := y;
```

Of course a variable by itself can occur as either an L-value (like **x** in the first example) or as an R-value (like **y** in the second example).

If an L-value is used as an R-value, then the AST data structure will contain a **ValueOf** node. The **ValueOf** node has an **lValue** field which will point to either a **Variable**, an **ArrayDeref**, or a **RecordDeref** node.

The code generation for L-values is handled in **genLValue()**. The code generation for R-values is handled in **genValueOf()**.

In my own version of **Generator.java**, I did not have routines named **genArrayDeref()** or **genRecordDeref()** or **genVariable()**. (Actually, I got rid of them.) Instead, it seemed clearer to include the necessary code directly in **genLValue()** and **genValueOf()**.

The simplest approach to **genValueOf()** is to do the following:

(1) Call **genLValue()** on the **lValue** field. It will return a temporary (call it **temp1**) after generating the IR instructions to move an address into it.
(2) Create a new temporary. Call it **temp2**.
(3) Generate a **loadIndirect** instruction to move the data at the address in **temp1** into **temp2**.

However, you should make an optimization when the L-value in question is a simple variable.

To accomplish this, put a switch statement into **genValueOf()** and take a look at what kind of node the **ValueOf** is pointing to. There are three cases: **Variable**, **ArrayDeref**, and **RecordDeref**.

For the **ArrayDeref** and **RecordDeref** cases, you'll need to do steps (1) (2) (3) above.

For the **Variable** case, simply return the variable (as represented by its **VarDecl** or **Formal** node), avoiding the call to **genLValue()** and the **loadIndirect** instruction.

Record Dereferencing

Next, look at a record dereference used as an L-value. Consider the following PCAT source code:

```
r.f3 := i + 123;
```

The variable **r** will be a word containing a pointer to a record stored in the heap, or possibly NIL. The fields in a record are at offsets 0, 4, 8, 12, 16, ... For this record type, assume that **f1** has been assigned to offset 0, **f2** to offset 4, **f3** to offset 8, etc.

You should generate the following code:

```
! ASSIGNMENT STMT...
                if r = 0 then goto runtimeError2    (integer)
                t1 := r + 8                         (integer)
                t2 := i + 123                       (integer)
                *t1 := t2
```

The test and branch checks to make sure the programmer is not trying to dereference a NIL pointer.

The second instruction computes the address of the word in question and puts it in temporary **t1**. The third instruction computes the right-hand side of the assignment and moves the result into temporary **t2**. The final instruction completes the assignment by moving the value into the word pointed to by the address in **t1**.

The first two instructions are generated by **genLValue()**. The third instruction is generated by the call to **genExpr()**. The final instruction is added by **genAssignStmt()**.

Notice that a new **IntegerConst** node (in this case with **iValue** = 8) was needed to use as **arg2** of the **add** instruction. As you generate these instructions, you'll have to call **new Ast.IntegerConst()** to allocate a new **IntegerConst** node and set its **iValue** field to whatever the offset is. (In this case, it was 8 but other fields will have different offsets.)

One optimization you need to perform is to watch for offset 0 and deal with it separately. In the source code:

```
r.f1 := i + 123;
```

we see a field **f1** whose offset is zero. You should eliminate the **add** instruction since it would just be adding zero. Instead, generate this code:

```
! ASSIGNMENT STMT...
                if r = 0 then goto runtimeError2     (integer)
                t1 := i + 123                        (integer)
                *r := t1
```

Notice that **RecordDeref** and **ArrayDeref** nodes themselves contain an **lValue** field. In general, the thing being dereferenced could be something besides a simple **Variable**. So, when dealing with a **RecordDeref** in **genLValue()**, we will first need to call **genLValue()** to generate code for the L-value. This recursive call to **genLValue()** will return the name of the variable containing an address. Then you can generate the test for zero and so on.

For example, in the following source code, **r2** is a record whose **f2** field is itself a record.

```
r2.f2.f3 := i + 123;
```

In processing this, **genLValue** will call itself recursively. We can show the calling history (the "activation tree") for this example as follows:

```
genAssignStmt ("r2.f2.f3 := i + 123;") {
   genLValue ("r2.f2.f3") {
      genLValue ("r2.f2") {
      }
   }
   genExpr ("i + 123") {
   }
}
```

As these methods are executed, they will generate these IR instructions.

```
1.    if r2 = 0 then goto runtimeError2    (integer)
2.    t1 := r2 + 4                         (integer)
3.    t2 := *t1
4.    if t2 = 0 then goto runtimeError2    (integer)
5.    t3 := t2 + 8                         (integer)
6.    t4 := i + 123                        (integer)
7.    *t3 := t4
```

Lines 1-2 compute the address of a word containing a pointer to a record containing an **f3** field and store it in temporary **t1**. They are the result of the inner invocation of **genLValue()**.

Look at how the outer invocation of **genLValue()** works. It begins by making the recursive call (resulting in lines 1 and 2). Then, it needs to generate a **loadIndirect** instruction in line 3. Now **t2** contains a pointer to the record containing the **f3** field. Then **genLValue()** generates instructions (in lines 4-5) to compute the address of the **f3** field in that record. Finally, **genLValue()** returns this address in temporary **t3**.

The remaining 2 instructions are produced by **genExpr()** and **genAssignStmt()**.

Now consider the straightforward approach to generating code for following source code:

```
r.f3 := i + 123;
```

The method **genAssignStmt()** begins by calling **genLValue()** to deal with the left-hand side ("**r.f3**"). In the call to **genLValue()**, we see that we are dealing with a **RecordDeref**. To deal with the **lValue** field in that node ("**r**"), the obvious thing to do is to make a recursive call to **genLValue()**. But note that **genLValue()** will always load the address of the thing in question into a temporary and will return that temporary, even if it is a simple variable like in this example. Such a recursive invocation of **genLValue()** would move the address of **r** into a temporary, (**t1** in the code below). Then the main call to **genLValue()** would have to generate a **loadIndirect** instruction. This would produce the following IR sequence:

```
t1 := &r
t2 := *t1
if t2 = 0 then goto runtimeError2   (integer)
t3 := t2 + 8                        (integer)
t4 := i + 123                       (integer)
*t3 := t4
```

The first two instructions are pointless and you must avoid generating them. The desired code (shown at the beginning of this section) just used **r** directly in the if-equal-goto instruction. You can avoid generating these two unnecessary instructions by inserting a test in **genLValue()** when handling a **RecordDeref**. Look to see if you have a simple **Variable** for the **lValue**. If so, just use it; if not, you'll need to call **genLValue()** recursively to handle whatever the **lValue** is. Then you will have to generate a **loadIndirect** instruction.

Array Dereferencing

Now look at an array dereference used as an L-Value. Consider the following source code:

```
a [i*3] := z;
```

First, **genAssignStmt()** will call **genLValue()** to handle the lefthand-side, which is an **ArrayDeref** node. **GenLValue()** will need to call **genExpr()** to deal with the index expression ("i*3").

Here is the code you should generate for this source:

```
1.     ! ASSIGNMENT STMT...
2.                 if a = 0 then goto runtimeError2       (integer)
3.                 t1 := i * 3                            (integer)
4.                 if t1 < 0 then goto runtimeError4      (integer)
5.                 t2 := *a
6.                 if t1 >= t2 then goto runtimeError4    (integer)
7.                 t2 := t1 * 4                           (integer)
8.                 t2 := t2 + 4                           (integer)
9.                 t2 := a + t2                           (integer)
10.                *t2 := z
```

First, in **genLValue()** we will generate a test to see if the current value of the variable **a** is NIL (i.e. zero). If so, we must branch into the boilerplate code to deal with the error. Next, **genLValue()** will call **genExpr()** to deal with the index expression ("**i*3**" in this example). **genExpr()** will generate the assignment to **t1** (line 3) and will return **t1**. Back in **genLValue()**, we will then generate the instructions in lines 4-6 which test to make sure the computed index falls within $0 \leq \text{index} < N$.

Next, we need to generate instructions to compute the address of the correct word. At this point (before line 7), **t1** contains the index. We need to multiply it by 4 and add 4 to bump past the initial word which stores the array size. Then, we add it to the address of the array in the heap in line 9. Finally, we can return the temp containing the computed address from **genLValue()**.

Also notice that **t1** is whatever temporary got returned from **genLValue()**'s call to **genExpr()**. Due to our optimization of temporaries, as discussed earlier, the thing that gets returned from **genExpr()** might also, in general, be an **IntegerConst** instead of a variable. This is why we created a new temporary (**t2**) in **genLValue()**, instead of performing the computation directly in the temporary returned from **genExpr()** (which was **t1**).

After returning from **genLValue()**, **genAssignStmt()** will generate the instructions to evaluate the righthand-side (there are none needed in this example) and the finally **store** instruction in line 10.

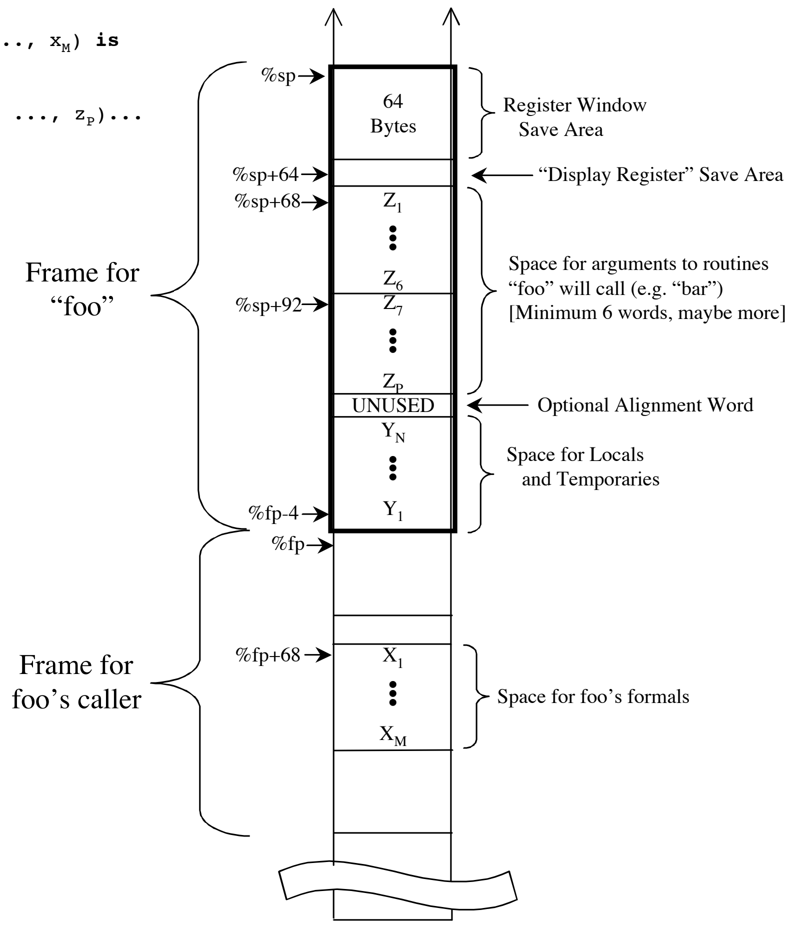
An **ArrayDeref** node contains an **lValue** field, so in general you need to call **genLValue()** to deal with it. Then you need to generate a **loadIndirect** instruction. However, it is often the case that the **lValue** is a simple **Variable** (like "**a**" in this example). To avoid generating poor code, you should include a special case test to see if the **lValue** field happens to point to a **Variable** node. If so, you can use the variable directly, otherwise call **genLValue()** recursively and generate a **loadIndirect** into a new temporary. This special case test is analogous to the test for **RecordDeref**s discussed in the previous section.

Assigning Offsets to Variables and Formals

The diagram titled "Activation Record Layout" shows where in the activation record we will store the local variables and the formal parameters. Normal local variables (i.e., user-defined and temporary variables) will be stored at offsets: -4, -8, -12, -16, ... Formal variables will be stored at offsets: 68, 72, 76, 80, ...

Activation Record Layout

```
procedure foo (x1, x2, ..., xM) is
    var y1, y2, ..., yN
    begin
        ... bar (z1, z2, ..., zP)...
    end
```


%sp
64 Bytes
Register Window Save Area
%sp+64
"Display Register" Save Area
%sp+68
Z1
Z6
Space for arguments to routines "foo" will call (e.g. "bar") [Minimum 6 words, maybe more]
%sp+92
Z7
ZP
UNUSED
Optional Alignment Word
YN
Space for Locals and Temporaries
%fp-4
Y1
Frame for "foo"
%fp
Frame for foo's caller
%fp+68
X1
XM
Space for foo's formals

A new field, **offset**, has been added to **VarDecl** and **Formal** nodes. You must include the code to set these **offset** fields.

The following lines where placed in the starter file, **Generator0.java**, to aid in this process:

```
    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;
```

A new method called **printOffsets()** has been added to **IR.java** to print out all the offset information. The **main()** method has been modified to call **printOffsets()** right after it calls **generateIR()**. The result is a printed listing of all variables and formals along with their offsets. For example:

```
Printing Offset Information...
  MAIN BODY:
    FrameSize = 128
    Offsets of local variables:
      -4        w
      -8        x
      -12       y
      -16       z
      -20       t1
      -24       t2
      -28       t3
      -32       t4
  PROCEDURE p1_foo1:
    FrameSize = 112
    Lex Level = 1
    Offsets of formals:
      68        a
      72        b
      76        c
    Offsets of local variables:
      -4        i
      -8        j
      -12       k
      -16       t5
      -20       t6
  PROCEDURE p2_foo2:
    ...
```

Computing the Size of Activation Records

A field called **frameSize** has been added to the **Body** node and you need to set it correctly. This field will be used when you generate target code for **procEntry** and **mainEntry** in a later project. At that time, you will take the **frameSize** and plug it directly into the "save" SPARC instruction that you will generate.

You can perform the **frameSize** computation in **genBody()** using the global variable called **maxNumberOfArgsUsed**. While processing the **VarDecl**s and the statements in a **Body**, you will occasionally run into a **CallStmt** or a **FunctionCall**. You'll need to count the arguments and update **maxNumberOfArgsUsed** accordingly. When computing the **frameSize**, you might also want to count the number of local variables (after the statements have been processed and all temporaries have been created. You should also use the above constants.

Assume that routine **foo** calls **bar**. The activation record for **foo** must include enough space for the arguments to **bar**. But the activation record for **foo** does not need to contain any room for the formal parameters to **foo** since these will be stored in the frame of the routine that calls **foo**. After processing all of the **Body**, **maxNumberOfArgsUsed** should tell how much space to allocate in **foo**'s frame for arguments to routines it calls. According to the calling conventions, you must allocate at least 6 words here. (Since our target code will call **scanf**, **printf**, and **calloc**, we must follow the "C" calling conventions.) [Clever hint: You can achieve this "max" function by initializing **maxNumberOfArgsUsed** to 6 instead of 0!]

The optional alignment word will be either zero bytes or four bytes, as necessary to ensure that the frame size is an even multiple of 8.

Unique Procedure Names

Each PCAT source-level procedure will need a label to act as the target of a SPARC **call** instruction. We might consider using the name given by the programmer — this is what the “C” language does. This approach works for “C” since routines are never nested and therefore must each have a different name. However, in PCAT, routines may be nested. We might have several routines named “foo” in the same program!

We also have another problem. Our final SPARC program will have a number of symbols, such as:

```
runtimeError1
runtimeError2
runtimeError3
...
Label_1
Label_2
Label_3
...
str1
str2
str3
...
float1
float2
float3
...
```

What if the PCAT programmer happens to name one of his or her procedures “Label_4” or “runtimeError2”? According to the language definition, these are legal procedure names, yet we do not want to have a conflict with one of the labels our compiler is introducing.

One approach is to just rename all procedures with names like

```
proc1
proc2
proc3
...
```

This guarantees that every name will be unique and will not conflict with other names, but it makes the resulting SPARC code hard to read and/or debug. If the programmer tries to debug the compiled code (say with “gdb” or “ddd”), the source level names would be nice to have.

The approach we will take is to combine the source name with an automatically generated name. We’ll end up with names like:

```
   p1_foo
   p2_readArray
   p3_quickSort
   p4_foo
   p5_partition
   ...
```

This approach allows the source name to show through, but also ensures that every procedure will have a unique name which will not conflict with other names in the SPARC program. We have chosen "p" for "procedure" to make it clear that the name refers to a procedure and not, say a label.

To implement this approach, you might create and add a new method to **Generator.java:**

```
   String newProcName (String sourceName);
```

When passed a procedure name (like "foo"), this method will create a String such as "p47_foo", where the "47" will be a sequentially generated unique integer. This method would maintain a static counter which it increments. If called again, with "bar" as the argument, it would return "`p48_bar`".

The programmer-defined routine names show through, even though now all names will be unique. Because of the "p" prefix, these names will be recognizable as procedure names and will never conflict with other names occurring in the SPARC output (like "Label_5," "runtimeError3," "str4," or "float7").

[Existing compilers often do something similar, but the prefixes they choose are sometimes a little obscure. Here are some labels generated by a C++ compiler:

```
   LC0
   LC1
   LC2
   _main
   LFB1
   LCFI0
   LCFI1
   LCFI2
   L1$pb
   LFE1
   L_printf$stub
   L0$_printf
   L_printf$lazy_ptr
```

]

As each **ProcDecl** is processed, you should alter its **id** field to point to the new name:

```
   procDecl.id = newProcName (procDecl.id);
```

I don't really like altering a field that has been previously set, but I feel that, in this case, it is preferable to adding yet another field to the AST node data structures. Note that in **Generator.java**, we are never using the **procDecl.id** field, except possibly during printing. The lookup and linking of **CallStmt**s and **FunctionCall**s to their **ProcDecl**s was done earlier during type checking in **Checker.java**.

The "alloc" IR Instruction

The following IR instruction:

```
x := alloc (n)
```

will cause **n** bytes to be allocated on the heap and a pointer to the bytes to be stored in variable **x**. If the allocation fails for any reason (such as "heap full"), zero will be stored in **x**.

You can create this instruction with the following:

```
IR.alloc (x, n);
```

The **result** field (e.g., "x") should be a **VarDecl** node. The **arg1** field (e.g., "n") should be a **VarDecl** or **IntegerConst** node.

This instruction will be used by array constructors and record constructors.

When we generate SPARC code, the **alloc** IR instruction will be translated into a call to the "malloc" or "calloc" routines, which should be familiar to "C" programmers.

By the way, PCAT has no facility for explicitly "free"ing arrays or records, although you might envision adding something like this to the language. More likely, we would run PCAT in an environment with an automatic garbage collector.

In this course project, we will not worry about freeing storage. We will allocate and allocate, in the hope that the heap will always be big enough. If any allocation should fail, we will at least get a reasonable error message (runtimeError1: "Allocation failed").

Record Construction

Consider the following source statement, involving a **RecordConstructor** expression:

```
rec := RecType { f1 := 123; f2 := 23.45; f3 := false };
```

For this, you should generate the following IR code:

```
! ASSIGNMENT STMT...
        t1 := alloc (12)
        if t1 = 0 then goto runtimeError1   (integer)
        t2 := t1 + 0                        (integer)
        *t2 := 123
        t2 := t1 + 4                        (integer)
        *t2 := 23.45
        t2 := t1 + 8                        (integer)
        *t2 := 0
        rec := t1
```

The first two instructions allocate a block of memory on the heap and check to make sure that the allocation succeeded. From the **RecordConstructor** node, you can get to the **RecordType** node describing the record to be allocated (via **myDef->TypeDecl.type**). The number of bytes to allocate comes from the **size** field in this **RecordType** node.

The **RecordType** node also contains a pointer to a linked list of **FieldDecl** nodes, one for every field in the type definition describing the record. Likewise, a **RecordConstructor** node points to a linked list of **FieldInit** nodes, one for each field assignment in the record constructing expression.

There will be exactly one field assignment for every field in the record. (Fortunately, we checked that during type checking!) Each **FieldInit** node therefore will correspond to exactly one **FieldDecl** node. (Note that the two linked lists may be in different orders.)

Fortunately, during type checking, we saved a pointer from the **FieldInit** node to the corresponding **FieldDecl** node in a field called **myFieldDecl**, in the **FieldInit** node. The **myFieldDecl** field of each **FieldInit** should point to the **FieldDecl** node with the same field id.

Given the **myFieldDecl** field, generating the above instructions becomes easy!

In a loop, you'll need to walk the list of **FieldInit**s. For each, first call **genExpr()** to generate the code for the expression. Then generate the **IRiadd** instruction and the **IRstore** instruction.

Finally, **genRecordConstructor**() will need to return a temporary containing a pointer to the newly allocated record.

Array Construction

Here are two examples showing array constructor usage in PCAT:

```
a1 := MyArray1 {{ 100 OF –1 }};
a2 := MyArray2 {{ 4.5, 6.1, 3.0, 2.3, 4.0 }};
```

In general, an array constructor expression will look like this:

Type `{{` $count_1$ `OF` $expr_1$, $count_2$ `OF` $expr_2$, ..., $count_K$ `OF` $expr_K$ `}}`

Each of the $count_i$ and $expr_i$ are nested sub-expressions. As such, they may contain arbitrary computation, including calls to functions, and so on. The PCAT definition specifies that they must be evaluated in the lexical order given. So you will need to call **genExpr()** for each of these 2K expressions in exactly the order the corresponding sub-expressions appear in the source.

```
genExpr (count1);
genExpr (expr1);
genExpr (count2);
genExpr (expr2);
...
genExpr (countK);
genExpr (exprK);
```

Recall that an array constructor is represented by an **ArrayConstructor** node which points to a linked list of **ArrayValue** nodes. Each **ArrayValue** node has pointer to a **countExpr** and a

valueExpr. The **countExpr** is optional, but for now assume that it is present in each of the K **ArrayValue** nodes.

You cannot allocate the block of memory on the heap (where the array will be stored) until after you have evaluated all of the count expressions, since you'll need that information to compute how many bytes to allocate.

To help out, I have added two new fields (called **tempCount** and **tempValue**) to each **ArrayValue** node. One approach is to walk the linked list of **ArrayValue** nodes twice. During the first list traversal, you will call **genExpr()** twice for each node. You'll need to save the results (i.e., the temporaries into which the results were placed) in the two new fields for use later during the second list traversal. Also, as you walk this list the first time, you can generate the code to compute the size of the array.

Here is an attempt to show the code you should generate in **genArrayConstructor()**. It involves four new temporaries (called **t1**, **t2**, **t3**, and **t4** here) in addition to whatever temporaries are created by calls to **genExpr()**. The code in the two boxes is somewhat schematic and will be repeated several times. In particular, the code in the first box will be repeated K times, once for each count-value pair. The code in the second box will also be repeated K times.

```
        t1 := 0
        tcnt := ...genExpr(countExpri)...
        if tcnt ≤ 0 then goto runtimeError5
        t1 := t1 + tcnt
        tval := ...genExpr(valueExpri)...
        t2 := t1 * 4
        t2 := t2 + 4
        t2 := alloc (t2)
        if t2 = 0 then goto runtimeError1
        t3 := t2
        *t2 := t1
        t4 := tcnt
label:  t2 := t2 + 4
        *t2 := tval
        t4 = t4 - 1
        if t4 > 0 then goto label
        ... := t3
```

Your **genArrayConstructor()** routine will proceed as follows. After generating the first IR instruction, you should walk the linked list of the **ArrayValue**s. (This is the "first walk" of this list.) The instructions in the first box are generated once for each **ArrayValue**, so these 4 instructions will be repeated K times.

For each **ArrayValue** node, you'll call **genExpr()** twice and save the returned temporaries' names in **tempCount** and **tempValue**. As you go through the list, you'll also generate the instructions to compute the number of elements in the array. These instructions will add to and store the result into temp **t1**.

Next, you'll generate code to compute the size of the array in bytes into **t2**. Then you'll call **alloc()** and test the returned pointer. Then save a copy of the pointer (using **t3**) so that you'll have a pointer to the new array after you are done initializing it. Your **genArrayConstructor()** will end up returning **t3** to be used by the surrounding expression. Then you'll generate the instruction to save N, the number of elements in the array, in the first word. The pointer **t2** will be used to index through the array as you walk through it initializing all the elements. Just before you hit the second box, **t2** is pointing to the word containing N. You'll need to increment **t2** by 4 before using it each time. (Alternatively, one could put the **add** instruction *after* the store, but this would result in an extra **add** at the beginning, to get past the word containing N, and an unnecessary **add** at the end.)

Next, walk the linked list of **ArrayValue** nodes a second time. For each, you'll generate the code in the second box. Thus, the code in the second box (shown only once) will actually be repeated K times. This means that you will create K new labels. For each count-value pair, you will generate a small loop to initialize the right number of words in the array.

Notice that the first box is repeated for each of the K nodes in the **ArrayValue**s linked list. This means that you'll create K temporaries called t_{cnt} and K temporaries called t_{val}, during the first walk. Next, you walked the linked list a second time and generated the instructions in the second box K times. You'll need to remember all these temporaries between their creation (during the first walk) and use in the instructions in the second box (during the second walk).

The **ArrayValue** node has 2 fields (called **tempCount** and **tempValue**) which are these just for that purpose. In these fields, you may save the temporaries you create during the first walk of the linked list for use during the second walk of the linked list.

Finally, **genArrayConstructor**() can return the pointer to the array (**t3**).

Matters are somewhat complicated by the fact that the "count" expression may be missing. If so, a value of 1 is assumed.

The code in the first box is modified to simply increment **t1** by 1:

```
        t1 := t1 + 1
        tval := ...genExpr(valueExpri)...
```

The code in the second box is modified to:

```
        t2 := t2 + 4
        *t2 := tval
```

As an example, consider the following **ArrayConstructor** expression:

```
a1 := T1 {{ 11, 12, 13, 14, 15 }};
```

The above scheme will generate the following IR code:

```
! ASSIGNMENT STMT...
      t1 := 0
      t1 := t1 + 1                       (integer)
      t1 := t1 + 1                       (integer)
      t1 := t1 + 1                       (integer)
      t1 := t1 + 1                       (integer)
      t1 := t1 + 1                       (integer)
      t2 := t1 * 4                       (integer)
      t2 := t2 + 4                       (integer)
      t2 := alloc (t2)
      if t2 = 0 then goto runtimeError1  (integer)
      t3 := t2
      *t2 := t1
      t2 := t2 + 4                       (integer)
      *t2 := 11
      t2 := t2 + 4                       (integer)
      *t2 := 12
      t2 := t2 + 4                       (integer)
      *t2 := 13
      t2 := t2 + 4                       (integer)
      *t2 := 14
      t2 := t2 + 4                       (integer)
      *t2 := 15
      a1 := t3
```

Obviously, there is room for optimization when the count expressions are missing, but we will not be doing it here. At least this code will execute correctly. The optimizations we will discuss in class should be able to clean this code up.

String Constants

String constants can appear only in WRITE statements. Consider this source:

```
write ("Hello, world!");
write ("i = ", i);
```

When you generate the boilerplate, you'll need to generate something like this:

```
str1:   .asciz   "Hello, world!"
str2:   .asciz   "i = "
```

In order to facilitate this later, I have added two new fields to **StringConst** called **next** and **nameOfConstant**, which you must fill in.

In this project, you must assign a unique name to each string constant in the PCAT source by storing a Java String in the **nameOfConstant** field of each **StringConst** node. You might create a new method (perhaps called **newStringName()**) by copying and modifying the code from **newLabel()**. Each string name should be of the form "str**N**" where **N** is an integer.

In order to access the strings, you must build a linked list of **StringConst** nodes, using the **next** field. The static variable **stringList** will point to the head of this list. (This variable was included in the starter file, so you should have it in **Generator.java**.)

You need to set **stringList** and build the linked list as you encounter string constants.

I have modified **printIR()** to print out this linked list. It produces output like this. In my own **Generator.java**, I add newly encountered strings to the head of the list, instead of the tail, since it is easier and faster. As a result, the string list ends up in reverse order. (The assembler will place the strings into memory in this backwards order, but of course that will not matter.)

```
=====  String List Follows  =====
   str2:  "i = "
   str1:  "Hello, world!"
```

Real Constants

We will deal with floating point constants (**RealConst**) nodes much the same way we deal with **StringConst** nodes. Like **stringList**, there is a global variable called **floatList**. Each **RealConst** has a **next** field. You must build the linked list of **RealConst** nodes. Every time you encounter a **RealConst**, you need to add it to this list.

In addition, each **RealConst** has a field called **nameOfConstant**, which you must set to point to a name. You can create a routine (call it **newFloatName()**) that works like **newStringName()**. Every time it is called, it will return a name like “float1”, “float2”, “float3”, and so on. Then you can use it to set the **nameOfConstant**field whenever you add the **RealConst** to the growing **floatList**.

I have modified **printIR()** to print out the **floatList**.

The code that adds a **StringConst** or **RealConst** to the linked list can be short and clean, perhaps like this:

```
r.next = floatList;
floatList = r;
```

This code makes the assumption that the constant is not already on the list. Is this really true? Will we visit every **StringConst** node and **RealConst** node only once? The answer should be “yes” if our AST is really a “tree.” However, our AST does contain some cycles. Also, some nodes are shared, making it DAG-ish. Nevertheless, if you do the “natural thing”, you will visit each **StringConst** and **RealConst** only once and there will be no problem with the above code.

[WARNING: What happens if your code does something “unnatural” and visits some **StringConst** or **RealConst** more than once? It is very likely that the above code will create a circular linked list, instead of a finite linked list. There will be no noticeable problem until my **printIR** code tries to print the list out. Then, **printIR** will infinite loop. Since the loop will not occur until after the code generation is complete, every single print statement you add to your code to find the loop with get printed! So, if you have problems looping in a program with **StringConst**s or **RealConst**s, remember this paragraph!]

Take a look at this source code:

```
var x, y, z: real := 12.34;
```

One way to deal with initializing expressions would be to treat this code like:

```
var x: real := 12.34;
    y: real := 12.34;
    z: real := 12.34;
```

Another approach is to treat it like:

```
var x: real := 12.34;
    y: real := x;
    z: real := x;
```

We have chosen the second approach because we only want the initializing expression to be evaluated once. If we had chosen the first approach, we would have created an AST with a shared **RealConst** node! Using the second approach, we only have shared **ValueOf** and **Variable** nodes, which turns out to be safe.

The Read Statement

There are 2 new IR instructions concerned with input: **readInt** and **readFloat**.

Consider the following source code:

```
var i,j: integer := ...;
var x,y: real := ...;
...
read (i,j,x,y);
```

You should generate the following:

```
! READ STMT...
                t3 := &i
                readInt t3
                t4 := &j
                readInt t4
                t5 := &x
                readFloat t5
                t6 := &y
                readFloat t6
```

The **readInt** and **readFloat** instructions require (in their **result** fields) a variable containing a pointer to the target word. In other words, this variable should contain the address to be stored into. The code to compute a temporary containing this address is naturally generated by a call to **genLValue**().

The Write Statement

There are five new IR instructions concerned with output: **writeInt**, **writeFloat**, **writeString**, **writeBoolean**, and **writeNewline**.

Consider the following source code:

```
var x: real := ...;
var i: integer := ...;
var b: boolean := ...;
...
write ("Hello, world!");
write ("x = ", x*2.3, "i = ", i-45, "b = ", b);
```

You should generate the following:

```
! WRITE STMT...
                writeString str1
                writeNewline
! WRITE STMT...
                writeString str2
                t1 := x * 2.3           (float)
                writeFloat t1
                writeString str3
                t2 := i - 45            (integer)
                writeInt t2
                writeString str4
                writeBoolean b
                writeNewline
```

The **writeString** instruction requires a Java String in its **result** field. This should be the name of the string constant; see the discussion of **StringConst**s above.

The **writeInt**, **writeFloat**, and **writeBoolean** instructions require a pointer to a variable in their **result** field (or possibly an **IntegerConst** or **RealConst** — in other words, whatever gets returned from **genExpr()**).

The **writeNewline** instruction expects no arguments. It must be generated once at the end of every WRITE statement.

Optional Extension: A Peephole Optimizer

In class, I will discuss the general idea behind peephole optimization.

As an optional extension for this project, you may implement a class that makes one (or possibly several) passes over the list of IR instructions. This class should be called **Peephole()** and should contain a method called **optimize**(), which is called from **main()** after the call to **generateIR**().

I am providing a file called **Peephole.java**, which contains a dummy ("stub") **optimize** method; if you have time, you can modify this file to perform the peephole optimization an optional extension.

The black box code **Main.jar** does not perform any peephole optimization; it just uses the dummy file.

Standard Boilerplate...

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

[IGNORE THE NEXT PARAGRAPH]

I encourage you to use your own files 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.

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 10 - 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**.

Work independently: you must write this program by yourself.