Code Generation
Statements and Expresssions

I. Function calls

Suppose a node in your tree says to call function f with a particular list of arguments. Since any program you are generating code for has passed your type checker, you should have a declaration of function f, so there should be a section in your assembly code file with label f.

Here are the pieces of code you need to generate for the call:

a) Code to evaluate each argument (as always, leavings its value in the accumulator), then pushing the argument onto the stack.
b) Code to push the frame pointer

```
push %rbx
```

c) Code to call the function:

```
call f
```

d) The next line of code will be executed when you return from the call:

```
pop %rbx          # restore the frame pointer
```

e) Finally, add 8 times the number of arguments to %rsp to pop the args

I always push the arguments to the call in reverse order, so the first argument is highest on the stack. I do this with a little recursive function pushArgs(argList).

pushArgs(argList) recurses on the tail of the argList, then generates code for the current element in argList, leaving its value in the accumulator, and then push the accumulator onto the stack.

II. Return statements

There are 2 types of return statements in BPL:

return

and

return <exp>

If you are returning a value, generate code for computing that value, leaving it in the accumulator.

Whether you are returning a value or not, you need to deallocate the function's local variables before you return.

One way to do this is to just move the frame pointer into the stack pointer:

```
movq %rbx, %rsp
```

I find that it helps me to be very finicky about stack discipline, so instead of moving sp into fp, I add enough to %rsp to pop off the local variables I pushed on when I entered the function. This requires knowing what function the return statement is in, so in one of my passes through each function I make a link from each return statement back to the declaration node for the function that contains it

After deallocating the temporary variables, the only thing left to do is to return:

ret

III While loops

The tree node for a while loop has a condition child and a body child. The code starts by generating two labels, which you need to save in variables of your code generator; I'll call these varibles Label1 and Label2.

You emit Label1, then recursively emit code for the condition expression. This code results in the value of the condition being put into the accumulator. BPL regards 0 as *False* and anything non-zero as *True*. After generating code for the condition emit a comparison and a conditional branch:

```
        cmpl $0, %eax
        je <Label2>
```

This is followed by the code for the body of the loop. At the end there are two more instructions:

```
        jmp <Label1>
<Label2>:
```

IV Comparison Operators

As we have said before, you generate code for binary operators with

- code to evaluate the left operand, leaving it in the accumulator
- push the accumulator
- code to evaluate the right operand, leaving it in the accumulator
- code to perform the operation, leaving the value in the accumulator
- pop the stack

This is straightforward for arithmetic operations. Comparisons need a bit more work. Consider the comparison x < 10; suppose we have already generated code to evaluate x and push its value onto the stack, and that we have moved 10 into the accumulator. We need to put either 1 or 0 into the accumulator. The following code does this:

```
        cmpl %eax, 0(%rsp)
        jl Label1
        movl $0, %eax
        jmp Label2
Label1:
        movl $1, %eax
Label2:
```

V. Write Statements

We are using the C *printf* function for output.

The BPL write(exp) statement is converted to

printf("%d ", <exp>) or printf("%s ", <exp>)

and the writeln( ) stataement is converted to

printf("\n")

For these to work we need three strings defined in the .rodata section:

```
.WriteIntString: .string "%d "
.WriteStringString: .string "%s "
.WriteInString: .string "\n"
.
```

Naturally, you can call those strings something else, but I'll refer to them with those names.

The code for write(exp) where exp is an integer expression, starts with generating code for exp, leaving its value in the accumulator %eax. This is followed with

- movl %eax, %esi
- movq $.WriteIntString, %rdi
- movl $0, %eax
- call printf

If exp is a string the code changes only in that the value of the string would be an address, so it would go into %rax and then into %rsi using movq instead of movl.

The writeln( ) statement is almost identical; we just omit the second argument:

- movq $.WritelnString, %rdi
- movl $0, %eax
- call printf

VI. read expressions

The read( ) expression only reads an integer value and places it in %eax. We do this via scanf("%d", &x). I define a string in the .rodata section

```
.ReadIntString: .string "%d"
```

Here is the code you to generate for the read( ) expression:

- Decrement the stack pointer by 40 bytes.
- Put the address 24 bytes below the new stack pointer into %rsi.
- Put $.ReadIntString into %rdi
- Call scanf
- Move 24(%rsp) into eax.
- Increment the stack pointer by 40 bytes.