CS 331

Spring 2016

A Runtime Environment for BPL

The stack: The most critical issue in implementing BPL is deciding how to use the stack. You have 13 student-weeks to implement your compiler; multiple professional-person-years are spent developing the typical commercial-grade compiler. We will make some simplifications as concessions to the lack of time during the semester. One of those concessions is to try to be as consistent as possible. The Intel coding conventions call for the first six arguments to a call to be passed in registers, after that arguments are passed on the stack. For simplicity we will pass everything on the stack. The arguments belong to the caller (for example, they might be based on local variables of the caller), so only the caller can push them onto the stack. The call pushes the return address on the stack on top of the arguments. The callee needs any number of local variables; the only place these can be allocated is the stack, and this must be done by the callee. So our stack during the call has local variables, on top of the return address, which is on top of the arguments.

Now think about this. We need to be able to find all of these things on the stack. We also need to be able to push and pop temporary values onto the stack, so the top of the stack is a moving target during the call. Life becomes much simpler if we make a fixed point on the stack and calculate all of the variable positions in terms of that. We call this fixed point the *frame pointer*. We will save it in a register, but of course when we make a call to a new function that functions frame pointer overrides the old one. This means that a call also needs to save the frame pointer on the stack. We will do this on top of the arguments. This gives the following picture for the frame during a call

temporary stuff
local var 4
local var 3
local var 2
local var 1
return address
old fp
arg 1
arg 2
arg 3

If we set the frame pointer to point at the return address, all of these locations are easy to find in terms of fp:

temporary stuff	← sp
local var 4	-32(fp)
local var 3	-24(fp)
local var 2	-16(fp)
local var 1	-8(fp)
return address	← fp
old fp	8(fp)
arg 1	16(fp)
arg 2	24(fp)
arg 3	32(fp)

Note that this picture assumes that everything on the stack is the same width. Here is another of our concessions to time. It is not difficult to keep track of the actual offset of variables that might need 32 or 64 bits of room, but you have enough to do. We will simply give each variable its index in the list of parameters or local variables and compute its offset from fp in terms of this index. If Microsoft refuses to buy our compiler because of this inefficiency, we'll just have a little less money to spend on bonbons.

This picture gives us the following protocol for function calls and returns:

Before the call the caller pushes the arguments onto the stack in reverse order (last argument first, first argument last) then pushes the fp onto the stack and makes the call.

At the start of the call the callee puts the current stack pointer, which points at the return address, into the frame pointer. The callee then decrements the stack pointer by 8 times the number of local variables, to allocate room for all of the local variables.

At the return the callee puts into eax or rax any return value, then pops off the stack any temporary data saved there, then increments the stack pointer to de-allocate the local variables. The callee then executes a `ret`.

The **return instruction** pops the return address off the stack.

At the return the caller pops the stack into the fp, restoring fp into what it was before the call. The caller then increments the stack by enough to pop the arguments that were pushed there. Any return value will be found in eax or rax.

For example, suppose a function f has a call

```
x = g(1, 2);
```

where g is defined as

```
int g(int, x, int y) {
        int A[3];
        int z;
        ....
}
```

The code for f pushes 2, then 1, then f's fp, then calls g. This is followed in f by code that pops the stack into fp, then takes the value in rax and assigns it to variable x.

The code for g first puts the stack pointer into the fp register,then decrements the stack by 32 bytes (for 4 integers, which we are treating as 8 bytes each). When ready to return g puts its return value into rax, then increments sp by 32 bytes and issues the return instruction.

Registers: You have lots of choices for registers. Naturally, you must use rsp for the stack pointer. I suggest using rbx for the frame pointer. The calls to printf and scanf require you to use rsi and rdi. I use rsi in other situations as a temporary variable. I use rax and eax as "accumulators" where the results of expressions are stored. I have a recursive function genCodeExp(exp) that takes an expression node and generates code to evaluate it and leave its value in rax. Integer division uses rbp and rdx, There are lots of other registers that could be used for temporary storage. Be careful in this. Since our code generator is highly recursive, if you have a temporary value stored in a register and make a recursive call, you probably need to save that register on the stack and restore it after the call to avoid having the register trashed by the recursive call.

Expressions: Our compiler needs to generate code for expressions of the form

        L op R

where L and R are themselves expressions. We will always have the value of expressions end up in eax (or rax if it needs 64 bits). The following algorithm does this:

- generate code to evaluate L, leaving its value in rax
- push rax onto the stack
- generate code to evaluate R, leaving its value in rax
- generate code to perform operation "op" between the top of the stack and rax, leaving its value in rax.
- pop the value of L off the stack.

This works as long as we are careful with the stack. Any part of your compiler that pushes something onto the stack needs to also remove it from the stack.

Note that the code this algorithm generates is very inefficient. For example, suppose variable x is stored at -8(fp) and we want to generate code for the assignment statement x = 5. Here is the code our algorithm generates:

```
movq %rbx, %rax    # I am using %rbx as the frame pointer
addq $-8, %rax     # rax now has the address of x
push  $rax         # that's our left operand.
movl $5, %eax
movq 0(%rsp), %rsi  # put the address of x into %rsi
movl $rax, 0(%rsi)   # this does the assignment.
```

Instead of this, the single instruction

```
movl $5, -8($%rbx)
```

would have been sufficient. The raw code generated by any compiler is very inefficient. Commercial compilers spend most of their time trying to make this code more efficient; we will talk about this at the end of the semester, but we won't have much time for you to implement the strategies we will discuss.

I/O: We will use C library functions to handle I/O. This uses a small bit of incantation code. We could instead use Linux interrupt calls to directly do our own I/O, but the problem is that the native I/O functions work only with strings. We would need to write our own functions (in assembly language) to translate between strings and integers. This is not at all difficult, but it is one more thing to do and one more place for potential bugs at a point when you need some kind of output (other than error messages) from your programs. Given our time limitations, C I/O seems like an easy choice.

Here is what is involved.

First, you need to have code at the top of your program for a ".rodata" section where your strings are defined. Here is what mine looks like:

```
.section   .rodata
.WriteIntString: .string "%d "  # note the space after the d
.WriteStringString: .string "%s "
.WritelnString: .string "\n"
.ReadIntString: .string "%d"
```

To write an integer , I first evaluate it into register eax. I then give code that does the following:

- move eax into esi
- move $.WriteIntString into rdi
- move 0 in eax.
- call printf

To write a string, do the same only move its address into rsi and move $.WriteStringString into rdi.

The writeln code is similar only there is no argument to put into rsi. Just move $WritelnString into rdi, move 0 into eax, and call printf.

To read an integer, you need a memory location where it can be temporarily stored. You could allocate a global variable (maybe an array) for this, but I use the stack. Now, here we get into some magic code. The C compiler expects the caller of scanf to allocate space on the stack for the scanf function to save all of the registers it needs to save. I think it would make more sense for scanf itself to do this, but they didn't ask me. The following sequence works:

- Decrement the stack pointer by 40 bytes.
- Put the address 24 bytes below the new stack pointer into rsi.   Why 24? Because. Just because.
- Put $.ReadIntString into rdi
- Call scanf
- Move 24(%rsp) into eax. read( ) is an expression in BPL, and all expressions leave their results in rax or eax.
- Increment the stack pointer by 40 bytes.

Runtime errors: The BPL language specification does not require any specific behavior on runtime errors, such as improper array subscripts.   An implementation may either take the C-approach: run until the operating system takes you down, or the Java approach: try to catch the problem and go down gracefully. For the latter approach, it is usually easy to add code to your compiler to catch the problem. Unfortunately, you might be nested deep within a sequence of function calls, which will try to finish before your program terminates. There are two similar ways to handle this. One is to call the C exit( ) routine. This expects to be called with an integer argument. Any non-zero value can be used for this. You can call this the same way you call the printf and scanf routines.

An alternative approach is to call the built-in Linux exit routine:

- Put a non-zero value into eax, as your error code.
- Put a 1 into ebx, as a signal that you want to exit.
- call syscall, a routine that generates various interrupts in the x86-64 linux system.

If you are going to do this, you should first print some kind of error message to tell the user why the program is shutting down.