Global Variables, Arrays, and Pointers

I. Global Variables

We allocate local variables as part of the stack frame for the function that declares the variable. Global variables, on the other hand, will be allocated in the main data segment. At the start of the assembly language file, before the .rodata segment, we will have a set of .comm directives that declare global variables and global arrays. The format of these is

.comm <symbol>, <num bytes>, <alignment>

This creates a memory location referenced by the symbol that holds the specified number of bytes. The alignment field is optional; if used it guarantees that the address of this location is a multiple of the alignment. I use 32 for the alignment because the gcc compiler does, but this is optional and you can just omit the alignment field.

For example, if a BPL program declares a global integer variable X, I write into the code file

```
.comm X, 8, 32
```

It is easy to do this by walking along the top-level declarations of the program.

If you have the declaration above and a line in the program assigns X the value 23, the code you need to generate is

```
movq X, %rax  # move the l-value of X into %rax
push %rax
movl $23, %eax
pop %rsi        # pop the address of X into a temp register
movl %eax, 0(%rsi)
```

In other words, we use the symbol X as the address of this location. If you groove on terminology, movq X, %rax uses "absolute address mode" while movl $23, %eax uses "immediate address mode"

II. Strings

BPL only allows string constants. We want to allocate them in the .rodata section, along with the strings we use for I/O. To achieve this I make a pass through the complete program building up a *string table* of all of the strings that appear in the program. The string table is a hashmap where each string is mapped to a symbol: .S1, .S2, etc. The keys of a hashmap are unique, so this means that any particular string appears in the table only once. We could use the same labels for strings as we do for branch destinations in the code, but I find that having separate labels for strings makes the code file easier to read. If we need to load a string into the accumulator, as we would with the code

```
write( "Hi, Mom!" )
```

we can look up the string in the string table to gets its label. If that label is .S3 we would generate the code

```
movq .S3, %rax
```

III. Arrays

The BPL specification calls for bounds checking on all indexing of a arrays. Let's skip over that for the time being. You can think of an array of length n as a block of n consecutive integer or string variables. To make indexing a little more uniform, I allocate 8 bytes for each entry, whether the base type is int or string. For a global array A of length 10 I use a .comm directive:

```
.comm A, 80, 32
```

For a local array of length 10 declared within a function I just decrement the stack pointer by 80 bytes when I enter the function.

The two main actions we have for arrays are

a) indexing them, as in A[i]

b) passing them as arguments in a function

Suppose we want to put the value of A[i+1] into the accumulator. We do the following steps:

a. put the value of A, the starting address of the array, into %rax.
b. push %rax
c. generate code to put the value of i+1 into %eax.
d. multiply %eax by 8
e. pop the stack (with the starting address of A) into a temporary register such as %rsi
f. Add %rax onto %rsi. This gives the address of A[i]
g. movl 0(%rsi), %eax

Step (a) depends on what kind of declaration we have for A. If it is a global array this will be

```
movq A, %rax
```

If A is the 1st parameter to the current function the starting address of the array will be on the stack.

```
movq -16(%rbx), %rax
```

Finally, if A is a local variable, perhaps with position 2, we get its starting address as an offset of -24 from the frame pointer:

```
movq %rbx, %rax
addq -24, %rax
```

Obtaining the value of an array A to use as an argument is the same as the first step in the sequence above.

If you want to implement bounds checking, arrays need to carry around their lengths. One way to achieve this is to allocate n+1 values for an array of size n, and to write into the starting address the length (which is known at compile time). We find the value of A[i] at offset 8*(i+1) from the starting address of A. To do bounds checking, put the length of A into a temporary register such as %esi by first putting the starting address of A into %rax, then

```
movl 0(%rax), %esi
```

You will need the starting address of A, so push %rax onto the stack.

Next, generate code to put the index into %eax. You need to do 2 comparisons on %eax. If it is negative, you have a bad index. If the value in %eax is larger than or equal to the value you saved in %esi (the length of A), you have a bad index. In either bad situation print an error message and use the C *exit()* routine to leave the program. If you pass both comparisons add 1 to %eax, multiply it by 8, and add the value at the top of the stack (the starting address of A) onto %rax. This is the address of A[i]. You dereference this

```
movl 0(%rax), %eax
```

to get the value of A[i]. Of course, if A is an array of strings you would need

```
movq 0(%rax), %rax
```

IV. Pointers

By this point you have already done all of the steps you need for the two pointer operations &x and *p. There are only a few things you can apply the & operator to: variables and array elements. Your tree structure should tell you which of those cases you are in. Finding the address of a variable depends on how it was declared: global, parameter, or local. Finding the address of an array element again depends on how the array was declared. Since you need exactly the same code for this as for finding L-values for assignment expressions, you might want to write a function for generating L-values and leaving them in the accumulator.

The dereferencing operator *p is even easier. We generate code to put the value of p into %rax, then dereference it with

```
movl 0(%rax), %rax
```