

Assignment 11: functions, calling conventions, and the stack

ECEN 4553 & 5013, CSCI 4555 & 5525
Prof. Jeremy G. Siek

November 30, 2007

The goal of this week's assignment is to remove function definitions and function calls, replacing them with operations that manipulate a stack, where each element of the stack is a chunk of data about a function invocation. These chunks are called *activation records* (or *frames*). They contain the arguments to a function, its local variables, the return address, and other information needed to execute the function. With the removal of function definitions, our IR (intermediate representation) will nearly be at the same abstraction-level as x86 assembly, so we will be able to emit assembly code to test this pass. After this week, the structure of the compiler will match what is shown in Figure 1.

1 Background

Before diving into the details of how a stack is used to implement functions, let us recall why a stack is needed in the first place and why it is the right data-structure for the job. Consider the following recursive function that computes the *n*th Fibonacci number.

```
int fib(int n) {  
    if (n == 0 || n == 1) return n;  
    else return fib(n - 2) + fib(n - 1);  
}
```

The calls made to the `fib` function can be visualized as a tree. Figure 2 shows the tree that results from calling `fib(6)`. The execution of `fib(6)` can be thought of as a depth-first search through this tree that visits the subtree on the left before visiting the subtree on the right. So `fib(6)` invokes `fib(4)`, which invokes `fib(2)`, which calls `fib(0)`. Once execution reaches a leaf in the tree, it backtracks to the previous node and continues with the subtree on the right, so from `fib(0)` we backtrack to `fib(2)` and invoke `fib(1)`. Once that is complete we again backtrack, but this time `fib(2)` is complete so we backtrack further to `fib(4)`.

Now, consider what is needed to implement this backtracking. We need to keep track of where we came from, that is, the sequence of nodes that led to the current node. Furthermore, for each node in the sequence, we need to keep track of what remains to be done and the values of the intermediate results. For example, when we return to `fib(4)` after computing `fib(3)`, we need to remember the result of `fib(2)` so that we can add it to the result of `fib(3)`. Also, note that as execution proceeds, we only add and remove nodes from the front of this sequence. Therefore, a stack data-structure is the appropriate choice, where each item in the stack contains information about intermediate results and what is left to be done.

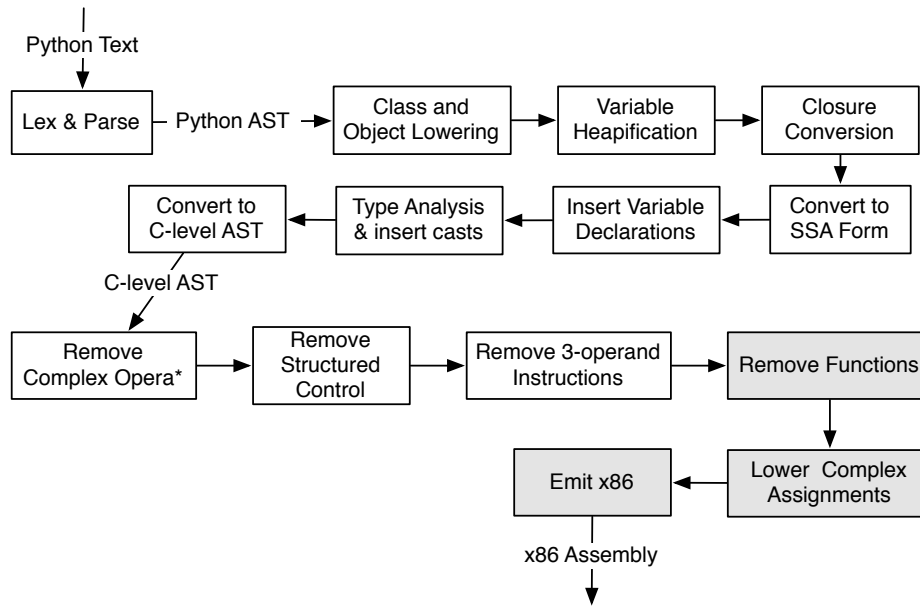


Figure 1: Structure of the compiler.

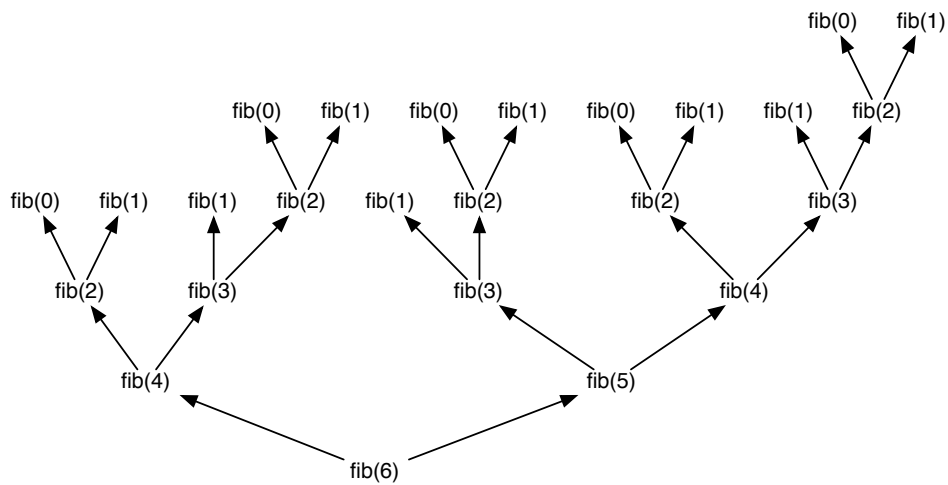


Figure 2: The call tree for fib(6).

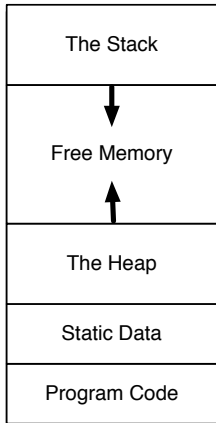


Figure 3: Memory Layout for a Linux Process.

When the program is about to be run, the operating system creates a process and dedicates a chunk of memory to it. In Linux, this memory is laid out as shown in Figure 3. Memory addresses increase from bottom to top. So *the heap*, which is the memory dedicated to object allocated with `malloc`, grows up, and *the stack*, used for activation frames, grows down. Having an upside-down stack is confusing because tradition refers to the “top” of the stack as the place where items are pushed and popped. However, because this stack is upside-down, the “top” is lower than the “bottom”. To avoid further confusion, we refer to the “front” of the stack as the place where items are pushed and popped. The machine code for the program itself is stored at the bottom. Static data, such as string constants, is stored directly above the program code.

The way in which a function should manipulate the stack is governed by *calling conventions*. By following an agree-to set of conventions, different functions, and even functions compiled by different compilers, can work together. Figure 4 is a diagram of the stack focusing on the current activation record at the front of the stack. The `%esp` register is the stack pointer and it should always point to the front of the stack. The `pushl` instruction pushes the contents of a 32-bit (double-word or 4-byte) register onto the stack, e.g., `pushl %eax` pushes the value of the `%eax` register. The `pushl` instruction does this by decrementing the `%esp` by 4 bytes so that it points to an empty memory location and then writing the contents of the register to the address in `%esp`. The instruction `pushl %eax` is equivalent to the following.

```
subl $4, %esp
movl %eax, (%esp)
```

The `popl` instruction does just the opposite. It removes an item from the front of the stack. The `popl` instruction accomplishes this by loading the value at the `%esp` address into a register and then incrementing the stack pointer by 4 bytes. The instruction `popl %eax` is equivalent to the following.

```
movl (%esp), %eax
addl $4, %esp
```

To call a function, you push the arguments for the function on the stack in reverse order and invoke the `call` instruction. Once the function is complete, control will be returned to the instruction following the `call`, which should pop the arguments from the stack by incrementing the stack pointer the appropriate number of bytes. For example, the call `fib(6)` corresponds to the following assembly code.

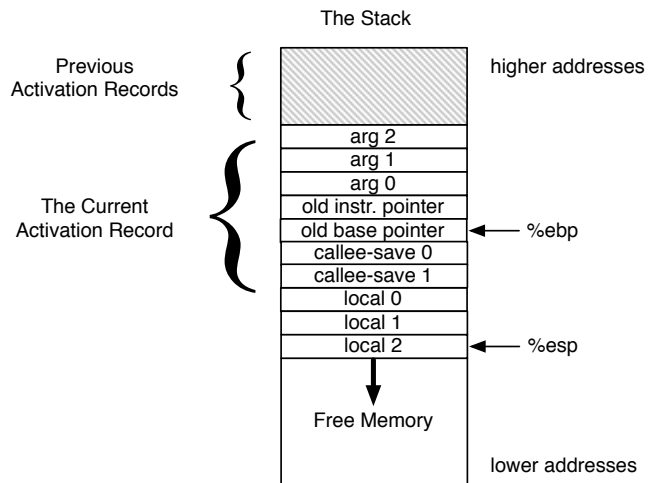


Figure 4: An activation record following the Linux x86 calling conventions.

```

pushl   $6
call    fib
addl    $4, %esp

```

The `call` instruction does two things: it pushes return address on the top of the stack and it jumps to the address indicated by the label, in this case `fib`. In Figure 4, the return address is in the slot labeled “old instr. pointer”.

```

fib:
    pushl   %ebp
    movl    %esp, %ebp
    pushl   %ebx
    subl   $4, %esp
    cmpl   $0, 8(%ebp)
    je     .L3
    cmpl   $1, 8(%ebp)
    je     .L3
    jmp    .L2
.L3:
    movl   8(%ebp), %eax
    movl   %eax, -8(%ebp)
    jmp    .L1
.L2:
    movl   8(%ebp), %eax
    subl   $2, %eax
    pushl   %eax
    call   fib
    addl   $4, %esp
    movl   %eax, %ebx
    movl   8(%ebp), %eax
    decl   %eax
    pushl   %eax
    call   fib
    addl   $4, %esp
    addl   %eax, %ebx
    movl   %ebx, -8(%ebp)
.L1:
    movl   -8(%ebp), %eax
    movl   -4(%ebp), %ebx
    leave
    ret

```

Figure 5: x86 assembly code for the `fib` function.

Figure 5 shows the assembly code produced by the GNU C compiler for the `fib` function. The first thing a function needs to do is save the base pointer and then change it to point to the same place as the current stack pointer. The base pointer is used to keep track of where the function arguments and local variables are located on the stack. For example, `8(%ebp)` refers to argument 0 of the function and `12(%ebp)` refers to argument 1 (assuming that argument 0 takes up 4 bytes). The base pointer is also used by the `leave` instruction (more about this later).

The next thing a function needs to do is to save the values of any callee-save registers that the function plans to use. The GNU C convention is that `%ebx`, `%esi`, `%edi`, and `%ebp` are designated as callee-save. The rest of the register are caller-save: `%eax`, `%ecx`, and `%edx`. In Figure 5 we see the `%ebx` register being pushed onto the stack just after the base pointer. The value for `%ebx` is retrieved from the stack at the very end of the `fib` function.

After the callee-save registers are saved, the function makes room for any local variables that need to be stored on the stack. For the `fib` function, there is just one local variable stored on the stack; it is used to store the result of the function. To make room for this variable, 4 bytes are subtracted from the stack pointer. Later in the function, this variable is referred to as `-8(%ebp)`.

Scanning towards the middle of the `fib` function we see two calls to `fib`. In each case the argument is pushed on the stack, the call to `fib` is made, and then the argument is removed by adding 4 to the stack pointer.

At the end of the function are two instructions: `leave` followed by `ret`. The `leave` instruction copies the contents of the `%ebp` register into the `%esp` register, so `%esp` points to the old base pointer. It then pops the old base pointer into the `%ebp` register. The `ret` instruction pops the return address from the stack and jumps back to the calling function.

Exercise 1.1. Read chapter 6 through 6.3.4 of the Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 1. The advice given in this chapter is quite general, whereas the above description is specific to GNU C calling conventions.

Passing Large Objects If your `pyobj` type is larger than 32 bits, then special care is needed for passing `pyobjs` as function arguments and returning then from a function. Let’s see how the GNU C compiler handles the following program.

```

#include <stdio.h>

struct pyobj {
    int tag;
    union { int i; double f; } u;
};

struct pyobj add(struct pyobj a, struct pyobj b) {
    struct pyobj r;
    r.tag = 0; r.u.i = a.u.i + b.u.i;
    return r;
}

int main() {
    struct pyobj x, y, z;
    y.tag = 0; y.u.i = 40;
    z.tag = 0; z.u.i = 2;
    x = add(y, z);
    printf("%d\n", x.u.i);
    return 0;
}

```

Here's the interesting portion of the assembly code for the main function. The object x is stored at $-16(\%ebp)$ through $-24(\%ebp)$, y is stored at $-32(\%ebp)$ through $-40(\%ebp)$, and z is stored at $-48(\%ebp)$ through $-56(\%ebp)$. The sequences of pushes leading up to call to the add function push z, then y, then the *address* of x.

```

main:
    ...
    leal    -24(%ebp), %eax
    pushl  -48(%ebp)
    pushl  -52(%ebp)
    pushl  -56(%ebp)
    pushl  -32(%ebp)
    pushl  -36(%ebp)
    pushl  -40(%ebp)
    pushl  %eax
    call   add
    addl   $28, %esp
    subl   $8, %esp
    pushl  -20(%ebp)
    pushl  $.LCO
    call   printf
    ...

```

The assembly code for the add function is shown below.

```

add:
    pushl  %ebp
    movl   %esp, %ebp
    subl   $24, %esp
    movl   8(%ebp), %edx
    movl   $0, -24(%ebp)
    movl   28(%ebp), %eax
    addl   16(%ebp), %eax

```

```

movl    %eax, -20(%ebp)
movl    -24(%ebp), %eax
movl    %eax, (%edx)
movl    -20(%ebp), %eax
movl    %eax, 4(%edx)
movl    -16(%ebp), %eax
movl    %eax, 8(%edx)
movl    %edx, %eax
leave
ret     $4

```

The parameter `b` is stored in `24(%ebp)` through `32(%ebp)` and `a` is stored in `12(%ebp)` through `20(%ebp)` (which is where `main` put them when it pushed the arguments `z` and `y`). The address for the return value is in `8(%ebp)`. The local variable `r` is stored in `-24(%ebp)` through `-16(%ebp)`. Starting with the instruction `movl -24(%ebp), %eax`, the assembly code is copying `r` into the return value. This is a memory-to-memory copy and x86 does not have a general purpose instruction that copies memory to memory. Instead one 32-bit chunk at a time of `r` is copied into the `%eax` register and then copied into the return value, where the `%edx` register holds the address of the return value.

2 Remove Functions

In this pass we remove the `CallFunc` and `Return` AST nodes and add the following new kinds of nodes.

```

class Call:
    def __init__(self, label):
        self.label = label

class Push:
    def __init__(self, arg):
        self.arg = arg

class Register:
    def __init__(self, name):
        self.name = name

class Address:
    def __init__(self, arg):
        self.arg = arg

class Offset:
    def __init__(self, arg, offset):
        self.arg = arg
        self.offset = offset

class Leave:
    pass

class Ret:
    pass

```

The `Push` AST node corresponds to the x86 `pushl` instruction that pushes an item on the stack. The `Call` AST node corresponds to the x86 `call` instruction.

2.1 Translating CallFunc

For the simple case when all of the arguments and the return value are small (32 bits), the CallFunc node is translated as follows, where m is the sum of the sizes of the arguments.

```
CallFunc(Name( $f$ ), [ $arg_1, \dots, arg_n$ ])  
⇒  
Push( $arg_n$ )  
...  
Push( $arg_1$ )  
Call( $f$ )  
Register('esp') +=  $m$ 
```

If an argument is large, then instead of a single Push you'll take the address of the argument and then use several Pushes. Suppose that arg_i is 12 bytes. Then you'll need three Pushes to copy it onto the stack.

```
Push(Offset(Name( $arg_i$ ), 8))  
Push(Offset(Name( $arg_i$ ), 4))  
Push(Offset(Name( $arg_i$ ), 0))
```

If the return type of the function is larger than 32 bits, then you need to push the address of the return value onto the stack just before the Call. The temporary variable will need a VarDecl at the beginning of the function.

```
Assign([AssName( $x$ , 'OP_ASSIGN')], CallFunc(Name( $f$ ), [ $arg_1, \dots, arg_n$ ]))  
⇒  
Push( $arg_n$ )  
...  
Push( $arg_1$ )  
tmp = Address( $x$ )  
Push(tmp)  
Call( $f$ )  
Register('esp') +=  $m$ 
```

If the CallFunc did not appear on the RHS of an assignment, then you must first create a new temporary variable and an assignment statement where the temporary is the LHS and the CallFunc is on the RHS. Then proceed as shown above.

2.2 Translating Return

If the argument to return is small, then simply assign the argument to the %eax register and then issue the Leave and Ret instructions.

```
Return( $e$ )  
⇒  
Register('eax') =  $e$   
Leave()  
Ret()
```

If the argument is large, then copy the value into the return value. We'll introduce a special variable named `__ret` to represent the return value.

```
Return( $x$ )  
⇒  
__ret =  $x$   
Leave()  
Ret()
```

2.3 Assign Stack Locations to Parameters and Locals

We assign a location on the stack to each parameter and local variable of a function. The location will be specified by an offset relative to the base pointer. The parameters get positive offsets and the local variables get negative offsets. The first parameter gets offset 8. Each parameter after the first gets an offset that is equal to the offset for the previous parameter plus the size of the previous parameter. So if the first parameter is 4 bytes, the second parameter gets offset 12. The first local variable is at offset -4 and each local variable after the first gets an even lower offset according to the size of the preceding local.

If the return type of a function is large, then the special variable `_ret` has been introduced and it should be assigned an offset of 8, which bumps the first parameter up to offset 12.

When we emit the x86 code, we'll need to access the assigned stack location when we come upon a Name AST node. There are several ways one could accomplish this. One way is to attach a dictionary to every function that maps variable names to their assigned location. When you emit the assembly code, pass the dictionary as a parameter to the emit function.

3 Lower Complex Assignments

If an assignment statement is between large objects (greater than 32 bits) then you have to expand the assignment into a sequence of assignments. Suppose that x and y are 12 bytes. Then an assignment of x to y translates to three assignment statements.

```
y = x;  
=>  
Offset(y, 0) = Offset(x, 0)  
Offset(y, 4) = Offset(x, 4)  
Offset(y, 8) = Offset(x, 8)
```

4 Emit x86 Assembly Code

To emit x86 assembly from the AST, we write a recursive function that traverses the AST and produces a string containing the x86 instructions. Several cases are discussed below.

Assignment `x = e;`

Recursively apply the emit function to get x' and e' . Then emit two `movl` instructions as shown below.

```
x = e;  
=>  
movl e', %eax  
movl %eax, x'
```

Add `x += e;`

For integer addition, recursively apply the emit function to get x' and e' . Emit a `movl` instruction to load e' into a register then emit an `addl` instruction.

```

x += e
⇒
movl e', %eax
addl %eax, x'

```

Dealing with floating point arithmetic is extra credit.

Conditional Goto if ($x == 0$) goto *label*;

Recursively apply the emit function to get x' . Then emit a `cmp` instruction followed by a `jne` instruction.

```

if (x == 0) goto label;
⇒
cmpl $0, x'
je label

```

Variable

Look up the offset n that you've assigned to x and then emit

```
n(%ebp)
```

Constant

If the constant is an integer (or Boolean represented as 0 or 1) then emit

```
$c
```

Dealing with floating point constants is extra credit.

Offset (x, m)

Look up the offset n that you've assigned to x and let $n' = n + m$. Emit

```
n'(%ebp)
```

Address $x = \&y$

Recursively apply the emit function to get x' and y' then emit a `leal` instruction.

```

x = Address(y)
⇒
leal y', x'

```

Exercise 4.1. Add the three passes described above so that your compiler outputs x86 assembly code.