Lab 9: Nested and High-Order Functions

(adapted from Profs. Jones and Tolmach's earlier version)

Winter 2015

Nested Functions

- Sometimes it is useful to allow function definitions to be nested, one inside another

```
int f(int x, int y) {
  int square(int z) { return z*z; }
  return square(x) + square(y);
}
```

- This might, for example, be used to introduce a local function, square, without making it more widely visible
- Java, (standard) C, C++ do not allow this, but ML, Pascal, Haskell, and many other languages do ...

A More Complicated Example: Quicksort

```
void sort(File inp, File out) {
  int[] a;
  ... a ... readArray ... quicksort ... writeArray ...
}

void readArray(inp, a) { ... inp ... a ... }
void writeArray(a, out) { ... a ... out ... }

void quicksort(a, lo, hi) {
  int pivot = ...;
  ... a ... pivot ... partition ... quicksort ...
}

void partition(a, pivot, lo, hi) {
  ... a ... pivot ... swap ...
}

void swap(a, i, j) { ... a[i] ... a[j] ... }
```

- Now let's move `readArray` and `writeArray` in to `sort`

A More Complicated Example: Quicksort

```
void sort(File inp, File out) {
  int[] a;
  void readArray() { ... inp ... a ... }
  void writeArray() { ... a ... out ... }

  ... a ... readArray ... quicksort ... writeArray ...
}

void quicksort(a, lo, hi) {
  int pivot = ...;
  ... a ... pivot ... partition ... quicksort ...
}

void partition(a, pivot, lo, hi) {
  ... a ... pivot ... swap ...
}

void swap(a, i, j) { ... a[i] ... a[j] ... }
```

- parameters are no longer required!

A More Complicated Example: Quicksort

```
void sort(File inp, File out) {
  int[] a;
  void readArray() { ... inp ... a ... }
  void writeArray() { ... a ... out ... }

  ... a ... readArray ... quicksort ... writeArray ...
}

void quicksort(a, lo, hi) {
  int pivot = ...;
  void partition(a, pivot, lo, hi) {
    ... a ... pivot ... swap ...
  }
  void swap(a, i, j) { ... a[i] ... a[j] ... }

  ... a ... pivot ... partition ... quicksort ...
}
```

- Move `partition` and `swap` in to `quicksort`

A More Complicated Example: Quicksort

```
void sort(File inp, File out) {
  int[] a;
  void readArray() { ... inp ... a ... }
  void writeArray() { ... a ... out ... }

  ... a ... readArray ... quicksort ... writeArray ...
}

void quicksort(a, lo, hi) {
  int pivot = ...;
  void partition() {
    ... a ... pivot ... swap ...
  }
  void swap(i, j) { ... a[i] ... a[j] ... }

  ... a ... pivot ... partition ... quicksort ...
}
```

- again, fewer parameters are required!

A More Complicated Example: Quicksort

```
void sort(File inp, File out) {
  int[] a;
  void readArray() { ... inp ... a ... }
  void writeArray() { ... a ... out ... }

  ... a ... readArray ... quicksort ... writeArray ...
}

void quicksort(a, lo, hi) {
  int pivot = ...;
  void partition() {
    void swap(i, j) { ... a[i] ... a[j] ... }
    ... a ... pivot ... swap ...
  }

  ... a ... pivot ... partition ... quicksort ...
}
```

- Move `swap` in to `partition`

A More Complicated Example: Quicksort

```
void sort(File inp, File out) {
  int[] a;
  void readArray() { ... inp ... a ... }
  void writeArray() { ... a ... out ... }

  void quicksort(a, lo, hi) {
    int pivot = ...;
    void partition() {
      void swap(i, j) { ... a[i] ... a[j] ... }
      ... a ... pivot ... swap ...
    }

    ... a ... pivot ... partition ... quicksort ...
  }

  ... a ... readArray ... quicksort ... writeArray ...
}
```

- Move `quicksort` in to `sort`

A More Complicated Example: Quicksort

```
void sort(File inp, File out) {
  int[] a;
  void readArray() { ... inp ... a ... }
  void writeArray() { ... a ... out ... }

  void quicksort(lo, hi) {
    int pivot = ...;
    void partition() {
      void swap(i, j) { ... a[i] ... a[j] ... }
      ... a ... pivot ... swap ...
    }

    ... a ... pivot ... partition ... quicksort ...
  }

  ... a ... readArray ... quicksort ... writeArray ...
}
```

- yet again, fewer parameters are required!
- how can we compile code like this?

Exercise

Warning: These exercises require gcc extensions to C that are enabled by default on linux lab machines, but may not be available on Macs (where gcc is not actually really gcc ...)

- Compile and run the program `example1.c`
- Rewrite it to use nested functions as much as possible, but without changing the parameters to any function. Name this program `nested1.c`
- Test this new program by compiling and running it
- Now attempt to drop as many parameters as possible from each nested function, but without changing the sequence of calls made. Name this program `dropped1.c`
- Test this new program by compiling and running it

Implementation: Free Variables

- The challenge here is in dealing with nested functions that access variables that are defined in enclosing functions

```
int f(int x, int y) {
  int g(int z) { return x+z; }
  return g(x+y);
}
```

- For example, `x` is said to be "bound" in the definition of `f`, but "free" in the definition of `g`
- The code for `g` refers to a variable that is not in its stack frame

Static Links

- To support calls to nested functions like this, we can give the callee a pointer to the stack frame of the "lexically enclosing" function.
- This is known as a *static link* (or *access link*)
- For example, if arguments are passed on stack, we might include the static length as a special "zeroeth" argument

	arg_n (enclosing stack frame)
	...
	arg_1
24	static link
16	ret addr
+8	dynamic link
`rbp`	local_1
-8	...
	local_m
`rsp`	

Example

Given the earlier definition:

```
int f(int x, int y) {
  int g(int z) { return x+z; }
  return g(x+y);
}
```

and a call `f(4,2)`, the stack might look something like the following during the call to g

In particular, the value for `x` can be found by following the static link $_g$ and taking the usual offset for `x`

	y=2
stack frame for f	x=4
	static link $_f$
	ret addr $_f$
	dyn link $_f$
	z=6
stack frame for g	static link $_g$
	ret addr $_g$
	dyn link $_g$

Static Link $\neq$ Base Pointer

- Suppose we have the definition:

```
int f(int x, int y) {
  int g(int z) { return h(x+h(z)); }
  int h(int u) { return y*u; }
  return g(x+y);
}
```

- The static link that g uses in calls to `h` points to the stack frame for `f`, not the stack frame for g

	y
stack frame for f	x
	static link $_f$
	ret addr $_f$
	dyn link $_f$
	z
stack frame for g	static link $_g$
	ret addr $_g$
	dyn link $_g$
	u
stack frame for h	static link $_h$
	ret addr $_h$
	dyn link $_h$

Lexical Depth

- A function at the top level (i.e., with no enclosing function) has lexical depth 0 (and does not need a static link)
- A function that appears inside the definition of a function with depth n has depth n+1
- To access a variable/call a function at depth n, from a function at depth m (note that $n \leq m$), then we have to follow the static link in the current frame $(m-n)$ times

Lexical Depth for Quicksort

```
/*0*/  void sort(File inp, File out) {
          int[] a;
 /*1*/    void readArray() { ... inp ... a ... }
 /*1*/    void writeArray() { ... a ... out ... }

 /*1*/    void quicksort(a, lo, hi) {
            int pivot = ...;
  /*2*/     void partition() {
   /*3*/      void swap(i, j) { ... a[i] ... a[j] ... }
              ... a ... pivot ... swap ...
            }

            ... a ... pivot ... partition ... quicksort ...
          }

          ... a ... readArray ... quicksort ... writeArray ...
       }
```

Static Links in gcc for X86-64

- The X86-64 ABI tries to avoid passing arguments on the stack, and the same philosophy applies to the static link
- So the static link is passed in `%r10` (an otherwise unused caller-save register)
- Even when compiling with optimization on, any variable accessed from a nested function *must* be stored in a stack frame!
  - But still typically no need for `%rbp`; static link points to bottom address of this frame

Worked Example

- Compile the program `example2.c` to a `.s` file using

  ```
  gcc -O1 -S -o example2.s example2.c
  ```

- Walk step-by-step through the behavior of the assembly code, annotating it
- Show the contents of the stack and key registers at each point

Exercise

Repeat the same steps for `example3.c`

- Annotate the assembly code.
- Identify the lexical depth of each function
- Clearly identify the code that implements a variable reference which spans a lexical depth difference of 2

Higher Order Functions

Functions that take other functions as arguments or return other functions as results are sometimes called *higher order functions*:

```
int(int) f(int x, int y) {
  int g(int z) { return x+z; }
  return g;
}
```

`int(int)` represents a type for functions that take an `int` argument and return an `int` result

- In this case, g may be called after f has returned
- ... so g may need to access `f`'s `x` parameter after `f` has returned
- ... so we can't just dispose of the activation record as soon as a function exits.

Moving from the Stack to the Heap ...

- A solution in this case is to allocate the activation record for `f` on the heap, and not on the stack
- Note that the term "stack frame" is no longer appropriate!
- Unused activation records can be recovered by garbage collection instead of by popping them off the stack
- Alternatively, if we don't keep the stack frames in the heap, then we will need to save a copy of `x`'s value in the representation for g before `f` returns
- Variants of these schemes are used in many functional language implementations (and, increasingly now, also in other settings such as C++, Python, and Javascript implementations)

Lambda Expressions

Lambda expressions (anonymous functions) provide a way to write functions without giving them a name.

$$\mathtt{x} \longrightarrow \boxed{\lambda\mathtt{x.e}} \longrightarrow \mathtt{e}$$

Haskell	`\x -> x + 1`
LISP	`(lambda (x) (+ x 1))`
Python	`lambda x:  x + 1`
Javascript	`function (x) x + 1`
C++11	`[](int x) -> int { return x + 1; }`
Java 8	`(int x) -> x + 1`

Example

- The previous example:

```
int(int) f(int x, int y) {
  int g(int z) { return x+z; }
  return g;
}
```

- Rewritten using a lambda expression:

```
int(int) f(int x, int y) {
  return (\z -> x+z);
}
```


Using Function Values

- A general purpose "mapping" primitive:

```
void mapArray(int(int) f, int[] arr) {
  for (int i=0; i<arr.length; i++) {
    arr[i] = f(arr[i]);
  }
}
```

  Note that the `f` in the body of the `for` loop is a parameter of `mapArray`, not a known function

- To increment every element in an array, `arr`:

```
mapArray(\x -> x + 1, arr);
```

- To double every element in an array, `arr`:

```
mapArray(\x -> x * 2, arr);
```

- Etc...

Composing Function Values

- A general purpose “composition” primitive:

```
(int)int compose(int(int) f, int(int) g) {
  return \x -> f (g x);
}
```

- Using `compose`, we can combine two separate mapping operations:

```
mapArray(g, arr);
mapArray(f, arr);
```

- Into a single iteration across the array:

```
mapArray(compose(f, g), arr);
```

- Etc...

Representing Function Values

How should we represent values of type `int(int)`?

- There are many different values, including:

  `(\z -> x+z), (\x -> x+1), (\x -> x*2), (\x -> f(g(x))), ...`

- ... any of which could be passed as arguments to functions like `mapArray` or compose, ...
- ... so we need a uniform, but flexible way to represent them

A common answer is to represent functions like these by a pointer to a “*closure*”, a heap allocated record that contains:

- a code pointer (i.e., the code for the function)
- the values of its free variables

Because we're making copies of the free variables, we usually require them to be immutable

Closures

- Every function of type int(int) will be represented using the same basic structure:

`codeptr`	...

- The code pointer and list of variables vary from one function value to the next:

`(\z -> x+z)`	`codeptr1`	`x`	
`(\x -> x+1)`	`codeptr2`		
`(\x -> x*2)`	`codeptr3`		
`(\x -> f(g(x)))`	`codeptr4`	`f`	`g`

Constructing and Calling Closures

We can encode closure construction in standard IR code:

- make an RTS call to allocate the closure record
- store the values of the free variables into the record

To call a function via a closure:

- add a pointer to the closure as an extra initial argument (to provide access to any free variables)
- make an indirect call to the code pointed to by the first field of the closure

Within the function code, free variables are referenced via the closure argument

- Known functions without free variables don't need a closure argument and can be called directly

Implementation of (\z -> x+z)

```
  ...
  t1 = call _malloc(12)
  0[t1]:P = _code1
  8[t1]:I = x
  ...

_code1(clo,z)
(x)
{
  x = 8[clo]:I
  t1 = x+z
  return t1
}
```


Implementation of (\x -> x+1)

```
  ...
  t1 = call _malloc(8)
  0[t1]:P = _code2
  ...

_code2(clo,x)
{
  t1 = x+1
  return t1
}
```

Implementation of (\x -> f(g(x)))

```
  ...
  t1 = call _malloc(24)
  0[t1]:P = _code4
  8[t1]:P = f
  16[t1]:P = g
  ...

_code4(clo,x)
(f,g)
{
  f = 8[clo]:P
  g = 16[clo]:P
  t1 = 0[g]:P
  t2 = call *t1(g,x)
  t3 = 0[f]:P
  t4 = call *t3(f,t2)
  return t4
}
```


Exercises

- Fill in the code for these three functions in `example4.ir`. For simplicity, assume that they are always called directly, rather than via closures, so they have no closure argument

```
(int)int compose(int(int) f, int(int) g) {
  return \x -> f (g x);
}
(int)int h(int x) {
  return compose(\z -> x + z,
                 \x -> x + 1);
}
int top() {
  return (h(42))(0);
}
```

- Test your code using the IR interpreter

Closures vs. Objects

Invoking an unknown function through a closure is very similar to invoking a method of an object ...

- Recall that method invocations pass the object itself as an implicit argument
- Closures are like objects with a single method
- Free variables correspond to object fields

In Java 8, lambda expressions are "just" a convenient way to write (local, anonymous) definitions of single-method classes

Very useful for GUI call-backs, aggregate operations, etc.

Summary

- More sophisticated forms of function can be supported by modifying or generalizing how activation records are used
- For a language with higher-order functions, we need to allocate activation records on the heap, or copy data to other heapallocated objects, because a function value may have a longer lifetime than the function that created it
- Function values can be represented by closure records that pair a code pointer with a list of variable values
- Invoking an unknown function through a closure is very similar to invoking a method of an object ...
- The techniques shown here are key tools in the implementation of functional programming languages and are gradually becoming common in OO languages too