CSE 341: Programming Languages

Spring 2007

Lecture 8 — First Class Functions, Closures, ...

Today

- Functions as first-class citizens
- Examples of functions taking and returning other functions
- Discuss *free variables* in function bodies
- In general, discuss environments and lexical scope
- See key idioms using first-class functions

A (Partial) Motivating Example—Sorting

Sorting is useful in many contexts, for many kinds of data.

Don't want specialized sort routine for each
(`sort(int list), sort(string list)...`)

Polymorphism, classes, etc. only handle part of the problem:

- `sort('a list) -> 'a list` is good,...
- but in what order? based on what part of the data?

Partial answer: write a function to compare two records, pass it to `sort` along with data

What if you don't know at "compile time"?

Fuller answer: write a function that *dynamically builds* (e.g., based on user input) a function to compare two records, pass it to `sort` ...

First-class functions

Want: ability to treat functions "just like" (other) data—assign to variables, pass as values, return as results, etc.

While "call-backs" like record comparison in sorting are one motivation, and a commonly occuring case, more general treatment of functions enables a very different style of programming, because it enables new styles of control structure.

Need: a very precise understanding of the meaning ("semantics") of functions, function definitions, function applications (calls), etc.

Another Example: Functions for Control

Problem: Sum the squares of the even elements in a list

More Generally: Pick some list elements, munge each, combine

Imperative (C/Java/...) solution:

```
temp = 0;
for(i=1;i<n;i++){
    if (List[i]%2==0) {temp = temp+List[i]^2}
}
```

Functional solution: reduce, map, filter

```
reduce(sum,mymap(square,(filter(even,list)));

fun even x = x mod 2 = 0;
fun square x = x * x;
fun sum(x,y) = x + y;
```

```
fun filter(f,[]) = []
  | filter(f,x::xs) = if f(x)
                      then x :: filter(f,xs)
                      else filter(f,xs)

fun mymap(f,[]) = []
  | mymap(f,x:xs) = f(x) :: mymap(f,xs);

fun reduce(f,[x,y]) = f(x+y)
  | reduce(f,x::xs) = f(x,reduce(f,xs));
```

Semantics of First-class Functions

- Functions are values. (Names in the environment are bound to them.)
- We can pass functions to other functions.
  - *Factor* common parts and *abstract* different parts.
- We can return functions as values from other functions.

Anonymous Functions

As usual, we can write functions anywhere we write expressions.

- We already could:

  `(let fun f x = e in f end)`

  Note: `(let fun f x = e in f(42) end)`

  vs: `(let fun f x = e in f end)(42)`

- Here is a more concise way (better style when possible):

  `(fn x => e)`

  E.g. `(fn x => e)(42)`

- Cannot do this for recursive functions (why?)

Returning Functions

The following has type `int->int->int`:

```
fun f x = fn y => x + y
```

Syntax note: `->` "associates to the right"

`t1->t2->t3` means `t1->(t2->t3)`

The key question: What about *free variables* in a function value?
What *environment* do we use to *evaluate* them?

Are such free variables useful?

If you remember one thing...

We evaluate expressions in an evironment, and function bodies in <u>an environment</u> extended to map arguments to values.

But which one? The environment in which the function was defined!

An equivalent description:

- Functions are values, but they're not just code.
- `fun f p = e` and `fn p => e` evaluate to values with two parts (a "pair"): the code and the current environment
- Function application evaluates the "pair"'s function body in the "pair"'s environment (extended by args)
- This "pair" is called a *(function) closure*.

There are *lots* of good reasons for this semantics.

Get your yellow highlighter—this is a very important concept!

Other Environmental Effects

Even the type of a function can change depending on its environment.

```
val y = "foo"
fun equals_y x =
  if x = y
  then "same"
  else "diff"
```

vs.

```
val y = 3
fun equals_y x =
  if x = y
  then "same"
  else "diff"
```

Example 1

```
val x = 1
fun f y = x + y
val x = 2
val y = 3
f (x+y)
```

Example 2

```
val x = 1
fun f y = let val x = 2 in fn z => x + y + z end
val x = 3
val g = f 4
val y = 5
g 6
```

Example 3

```
fun f g = let val x = 3 in g 2 end

val x = 4

fun h y = x + y

f h
```