CSE 341: Programming Languages

Spring 2006

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.

Semantics of First-class Functions

- Functions are values. (Variables 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)
```

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

```
(fn x => e)
```

- 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)`

Again, there is nothing new here.

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 an environment 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)
- This "pair" is called a *(function) closure*.

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

For hw, exams, and competent programming, you must "get this"!

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
```