CS4613 Lecture 4

David Bremner

January 15, 2024

Defining Functions

p. 58

Almost every language supports functions

```
(deffun (f x) (+ x x))
(f 3)
```

stacker

Most modern languages support inner functions

```
(deffun (f x)
  (deffun (sq y) (* y y))
  (+ (sq x) (sq x)))
```

stacker

Often via anonymous functions

```
(deffun (f x)
  (defvar sq (lambda (y) (* y y)))
  (+ (sq x) (sq x)))
```

stacker

Anonymous functions are all we need

Let's once again prototype things by slightly modifying plait syntax to match the book.

```
(define-syntax-rule (lam x body) (lambda (x) body))
```

Now try some examples.

p. 59

lam1

```
{let1 {f {lam x {+ x x}}}
      {f 3}}
```

lam2

```
{let1 {x 3}
      {let1 {f {lam y {+ x y}}}
            {f 3}}}
```

Syntax for defining and using functions

Abstract

```
[lamE (var : Symbol) (body : Exp)]
[appE (fun : Exp) (arg : Exp)]
```

Parser

```
[(? `(lam SYMBOL ANY))
 (lamE (s-exp->symbol (sx 1)) (px 2))]
[(? `(ANY ANY)) (appE (px 0) (px 1))]
```

parse

```
(test (parse `{let1 {f {lam x {+ x x}}} {f 3}})
      (let1E 'f (lamE 'x (plusE (varE 'x) (varE 'x)))
             (appE (varE 'f) (numE 3))))
```

Function values: 1/2

p. 60

```
print(lambda x: x + 1)    # Python
```

```
<function <lambda> at 0x7fef0b67c4a0>
```

```
console.log((x) => (x + 1))  # JavaScript
```

```
[Function (anonymous)]
```

```
lambda { |x| x + 1 };          # Ruby
```

```
#<Proc:0x00007f6eed050d08 -:3 (lambda)>
```

Function values: 2/2

```
(lambda (x) (+ x 1)) ;;  Common Lisp

#<FUNCTION (LAMBDA (X)) {535578CB}>

(lambda (x) (+ x 1))  ;; Emacs Lisp

(lambda (x) (+ x 1))

(lambda (x) (+ x 1))  ;; Plait

- (Number -> Number)
#<procedure>
```

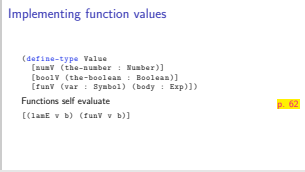
Implementing function values

```
(define-type Value
  [numV (the-number : Number)]
  [boolV (the-boolean : Boolean)]
  [funV (var : Symbol) (body : Exp)])
```

Functions self evaluate

```
[(lamE v b) (funV v b)]
```

p. 62



1. As the book notes, this looks like nothing. In practice it acts is a special kind of delayed evaluation.
2. It turns out this simple evaluation won't quite be enough. The reason it was for Emacs Lisp was that it originally had dynamic scope

Strategy for evaluation

example eval

```
(let ([f (lambda (x) (+ x 1))])
  (f 7))
```

stacker

p. 62

```
(interp (appE f a) env)
```

1. Evaluate `f`
2. Evaluate `a`
3. Check that `f` evaluates to a function
4. Evaluate the body of `f` in a new env
5. with the formal bound to evaluated `a`

Implementing application 1/2

p. 63

Eval function and argument

```
[(appE f a) (let ([fv (interp f nv)]
                  [av (interp a nv)])
              ....)]
```

Check for a function

```
[(appE f a) (let ([fv (interp f nv)]
                  [av (interp a nv)])
              (type-case Value fv
                [(funV v b) ....]
                [else (error 'app "not a function")]))]
```

Implementing application 2/2

p. 63

Interpret body

```
[(appE f a) (let ([fv (interp f nv)]
                  [av (interp a nv)])
              (type-case Value fv
                [(funV v b) (interp b ....)]
                [else (error 'app "not a function")]))]
```

in a new environment

```
[(appE f a) (let ([fv (interp f nv)]
                  [av (interp a nv)])
              (type-case Value fv
                [(funV v b) (interp b (extend nv v av))]
                [else (error 'app "not a function")]))]
```

Testing our evaluator

p. 65

```
[(appE f a)
 (let ([fv (interp f nv)]
       [av (interp a nv)])
   (type-case Value fv
     [(funV v b)
      (interp b (extend nv v av))]
     [else (error 'app "not a function")])))]
```

interp1

```
(test (run `{let1 {f {lam x {+ x 1}}} {f 8}}) (numV 9))
(test (run `{let1 {y 1} {let1 {f {lam x {+ x y}}}
                              {f 8}}})
      (numV 9))
```

Oops we made dynamic scope

interp1
```
(test (run `{let1 {y 1} {let1 {f {lam x {+ x y}}}
                               {let1 {y 2} {f 8}}}})
      (numV 9))
```

Test with plait

p. 66

scope1
```
(test
 {let1 {y 1} {let1 {f {lam x {+ x y}}}
                   {let1 {y 2} {f 8}}}}
 9)
```

Evaluation via substitution

p. 66

```
(let ([y 1])
  (let ([f (lambda (x) (+ x y))])
    (let ([y 2]) (f 8))))
```

stacker

subst. 1 for x

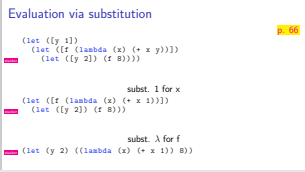
```
(let ([f (lambda (x) (+ x 1))])
  (let ([y 2]) (f 8)))
```

stacker

subst. λ for f

```
(let (y 2) ((lambda (x) (+ x 1)) 8))
```

stacker



1. Our example is slightly different than the one in the book, but the idea is the same
2. In fact we have to be a bit careful about how we do substitution here, as the second y "obviously" should not be replaced

Substitution and environments

p. 67

1. Substitution enforces static scope: by execution time the variable is gone
2. Substitution replaces variables according to the defining environment
3. To get the equivalent of substitution, we need to remember the defininining environment.

Substitution and environments

p. 67

1. Substitution enforces static scope: by execution time the variable is gone
2. Substitution replaces variables according to the defining environment
3. To get the equivalent of substitution, we need to remember the defining environment.

1. As mentioned above, it is actually not completely trivial to get the rules for substitution correct. One needs to define *free* and *bound* variables, for a start
2. The book talks about environments as delayed substitutions. This is true, although maybe more relevant in the context of environments as a faster replacement for substitution

Remembering the defining environment

p. 67

closure constructor

```
(define-type Value
  [numV (the-number : Number)]
  [boolV (the-boolean : Boolean)]
  [funV (var : Symbol) (body : Exp) (nv : Env)])
```

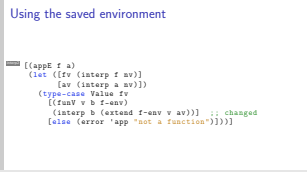
save environment

```
[(lamE v b) (funV v b nv)]
```

Using the saved environment

interp2

```
[(appE f a)
 (let ([fv (interp f nv)]
       [av (interp a nv)])
   (type-case Value fv
     [(funV v b f-env)
      (interp b (extend f-env v av))]  ;; changed
     [else (error 'app "not a function")]))]
```



1. The book uses shadowing to bind a new `nv`; I find it less confusing to use a new name.

Testing the revised interpreter

p. 68

interp2

previously failing

```
(test (run `{let1 {y 1} {let1 {f {lam x {+ x y}}}
                              {let1 {y 2} {f 8}}}})
      (numV 9))
```

interp2

a new pattern

```
(test (run `{{let1 {x 3} {lam y {+ x y}}} 4})
      (numV 7))
(test (run `{{let1 {y 3} {lam y {+ y 1}}} 5})
      (numV 6)) ;; maybe 4?
```

Understanding one of our tests

p. 68

try in stacker

```
((let ([y 3])
   (lambda (y) (+ y 1)))
 5)
```

stacker

subst. y does nothing

```
((lambda (y) (+ y 1))
 5)
```

stacker