

Functional Languages

Syllabus

Lecture Series (hours)	Topics
1-4	Introduction and Motivation, Paradigms
5-10	Syntax and Semantics, BNF, Compilation
11-18	Data Types, Constructs, Functions, Activation Records, Names and Bindings
19-28	Concurrency, Functional PLs, Logical PLs, Lambda Calculus, Event driven programming
29-36	Virtual Machines, Managed Languages, JIT, Case study

Lambda Expressions and Functions

◆ Expressions

$$x + y \qquad x + 2*y + z$$

◆ Functions

$$\lambda x. (x+y) \qquad \lambda z. (x + 2*y + z)$$

◆ Application

$$\begin{aligned} (\lambda x. (x+y)) 3 &= 3 + y \\ (\lambda z. (x + 2*y + z)) 5 &= x + 2*y + 5 \end{aligned}$$

Parsing: $\lambda x. f (f x) = \lambda x. (f (f (x)))$

Higher-Order Functions

◆ Given function f , return function $f \circ f$
 $\lambda f. \lambda x. f (f x)$

◆ How does this work?

$$\begin{aligned} & (\lambda f. \lambda x. f (f x)) (\lambda y. y+1) \\ &= \lambda x. (\lambda y. y+1) ((\lambda y. y+1) x) \\ &= \lambda x. (\lambda y. y+1) (x+1) \\ &= \lambda x. (x+1)+1 \end{aligned}$$

Same Procedure (JavaScript)

- ◆ Given function f , return function $f \circ f$

```
function (f) { return function (x) { return  
    f(f(x)); } ; }
```

- ◆ How does this work?

```
(function (f) { return function (x) { return f(f(x)); } ;  
})
```

```
function (x) { return (function (y) { return y + 1; })  
    (function (y) { return y + 1; })
```


```
    ((function (y) { return y + 1; })  
    (x))); }
```

```
function (x) { return ((x + 1) +  
    1); }
```

Declarations as “Syntactic Sugar”

```
function f(x) {  
    return x+2;  
}  
f(5);
```

$(\lambda f. f(5))$ $(\lambda x. x+2)$



block
body

declared
function

Free and Bound Variables

◆ Bound variable is a “placeholder”

- Variable x is bound in $\lambda x. (x+y)$
- Function $\lambda x. (x+y)$ is same function as $\lambda z. (z+y)$

◆ Compare

$$\int x+y \, dx = \int z+y \, dz \quad \forall x \, P(x) = \forall z \, P(z)$$

◆ Name of free (i.e., unbound) variable matters!

- Variable y is free in $\lambda x. (x+y)$
- Function $\lambda x. (x+y)$ is not same as $\lambda x. (x+z)$

◆ Occurrences

- y is free and bound in $\lambda x. ((\lambda y. y+2) x) + y$

Renaming Bound Variables

◆ Function application

$(\lambda f. \lambda x. f (f x)) (\lambda y. y+x)$

apply
twice

add x to
argument

◆ Substitute “blindly” – do you see the problem?

$\lambda x. [(\lambda y. y+x) ((\lambda y. y+x) x)] = \lambda x. x+x+x$

◆ Rename bound variables

$(\lambda f. \lambda z. f (f z)) (\lambda y. y+x)$

$= \lambda z. [(\lambda y. y+x) ((\lambda y. y+x) z))] = \lambda z. z+x+x$

Easy rule: always rename variables to be distinct

Main Points About Lambda Calculus

- ◆ λ captures the “essence” of variable binding
 - Function parameters
 - Declarations
 - Bound variables can be renamed
- ◆ Succinct function expressions
- ◆ Simple symbolic evaluator via substitution
- ◆ Can be extended with
 - Types, various functions, stores and side effects...

What is a Functional Language?

- ◆ “No side effects”
- ◆ Pure functional language: a language with functions, but without side effects or other imperative features

No-Side-Effects Language Test

Within the scope of specific declarations of x_1, x_2, \dots, x_n , all occurrences of an expression e containing only variables x_1, x_2, \dots, x_n , must have the same value.

begin

integer $x=3$; integer $y=4$;

$5*(x+y)-3$

... // no new declaration of x or y //

$4*(x+y)+1$

end

Reasoning About Programs

- ◆ To prove a program correct, must consider everything a program depends on
- ◆ In functional programs, dependence on any data structure is explicit
- ◆ Therefore, it's easier to reason about functional programs

Case Study

[Hudak and Jones, Yale TR,
1994]

- ◆ Naval Center programming experiment
 - Separate teams worked on separate languages

Language	Lines of code	Lines of documentation	Development time (hours)
(1) Haskell	85	465	10
(2) Ada	767	714	23
(3) Ada9X	800	200	28
(4) C++	1105	130	–
(5) Awk/Nawk	250	150	–
(6) Rapide	157	0	54
(7) Griffin	251	0	34
(8) Proteus	293	79	26
(9) Relational Lisp	274	12	3
(10) Haskell	156	112	8

some programs were incomplete or did not run

- Many evaluators didn't understand, when shown the code, that the Haskell program was complete. They thought it was a high-level partial specification.

Von Neumann Bottleneck

◆ Von Neumann

- Mathematician responsible for idea of stored program

◆ Von Neumann bottleneck

- Backus' term for limitation in CPU-memory transfer

◆ Related to sequentiality of imperative languages

- Code must be executed in specific order

```
function f(x) { if (x<y) then y = x; else x = y; }  
g( f(i), f(j) );
```

Eliminating VN Bottleneck

◆ No side effects

- Evaluate subexpressions independently
 - function `f(x) { return x < y ? 1 : 2; }`
 - `g(f(i), f(j), f(k), ...);`

◆ Good idea but ...

- Too much parallelism
- Little help in allocation of processors to processes
- ...

◆ Effective, easy concurrency is a **hard** problem

Introduction to Scheme

Scheme

- ◆ Impure functional language
- ◆ Dialect of Lisp
 - Key idea: symbolic programming using list expressions and recursive functions
 - Garbage-collected, heap-allocated
- ◆ Some ideas from Algol
 - Lexical scoping, block structure
- ◆ Some imperative features

Expressions and Lists

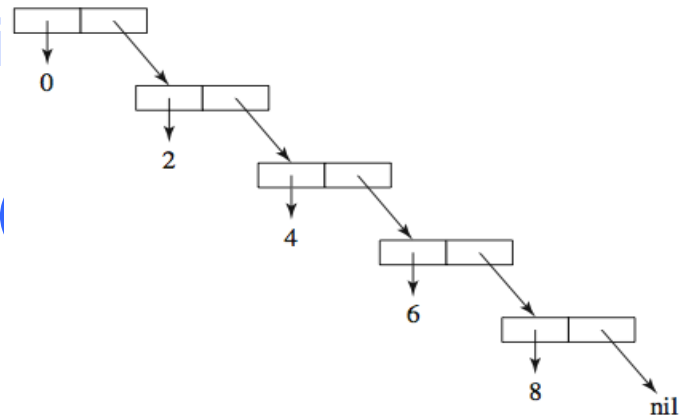
◆ Cambridge prefix notation: (f x1 x2 ... xn)

- (+ 2 2)
- (+ (* 5 4) (- 6 2)) means $5*4 + (6-2)$

◆ List = series of expressions enclosed in parentheses

- For example, (0 2 4 6 8) is a list of five numbers
- The empty list is written ()

◆ Lists represent both functions and data



Elementary Values

◆ Numbers

- Integers, floats, rationals

◆ Symbols

- Include special Boolean symbols #t and #f

◆ Characters

◆ Functions

◆ Strings

- “Hello, world”

◆ Predicate names end with ?

- (symbol? '(1 2 3)), (list? (1 2 3)), (string? “Yo!”)

Top-Level Bindings

- ◆ define establishes a mapping from a symbolic name to a value in the current scope
 - Think of a binding as a table: symbol \rightarrow value
 - (define size 2) ; size = 2
 - (define sum (+ 1 2 3 4 5)) ; sum = (+ 1 2 3 4 5)
- ◆ Lambda expressions
 - Similar to “anonymous” functions
 - Scheme: (define square (lambda (x) (* x x)))

Functions

- ◆ `(define (name arguments) function-body)`
 - `(define (factorial n)`
 `(if (< n 1) 1 (* n (factorial (- n 1)))))`
 - `(define (square x) (* x x))`
 - `(define (sumsquares x y)`
 `(+ (square x) (square y)))`
 - `(define abs (lambda (x) (if (< x 0) (- 0 x) x)))`
- ◆ Arguments are passed by value
 - **Eager evaluation:** argument expressions are always evaluated, even if the function never uses them
 - Alternative: lazy evaluation (e.g., in Haskell)

Expression Evaluation

- ◆ Read-eval-print loop
- ◆ Names are replaced by their current bindings
 - `x` ; evaluates to 5
- ◆ Lists are evaluated as function calls
 - `(+ (* x 4) (- 6 2))` ; evaluates to 24
- ◆ Constants evaluate to themselves.
 - `'red` ; evaluates to `'red`
- ◆ Innermost expressions are evaluated first
 - `(define (square x) (* x x))`
 - `(square (+ 1 2)) ⇒ (square 3) ⇒ (* 3 3) ⇒ 9`

Equality Predicates

- ◆ `=` - to check if two numbers are equal (if not numbers, error)

```
(= 2 3) => #f
```

```
(= 2.5 2.5) => #t
```

```
(= '() '()) => error
```

- ◆ `eq?` - check whether its two parameters represent the same object in memory

```
(define x '(2 3))
```

```
(define y '(2 3))
```

```
(eq? x y) => #f
```

```
(define y x)
```

```
(eq? x y) => #t
```

Equality Predicates

`(eq? 2 2)` \Rightarrow depends upon the implementation

`(eq? "a" "a")` \Rightarrow depends upon the implementation

◆ `eqv?` – same as `eq?` except return `#t` for primitives

`(eqv? 2 2)` \Rightarrow `#t`

`(eqv? "a" "a")` \Rightarrow depends upon the implementation

◆ `equal?` – are two values structurally equivalent? Works with list, vectors etc

`(equal "abc" "abc")` \Rightarrow `#t`

Operations on Lists

◆ car, cdr, cons

- (define evens '(0 2 4 6 8))
- (car evens) ; gives 0
- (cdr evens) ; gives (2 4 6 8)
- (cons 1 (cdr evens)) ; gives (1 2 4 6 8)

◆ Other operations on lists

- (null? '()) ; gives #t, or true
- (equal? 5 '(5)) ; gives #f, or false
- (append '(1 3 5) evens) ; gives (1 3 5 0 2 4 6 8)
- (cons '(1 3 5) evens) ; gives ((1 3 5) 0 2 4 6 8)
 - Are the last two lists same or different?

Conditionals

◆ General form

`(cond (p1 e1) (p2 e2) ... (pN eN))`

- Evaluate p_i in order; each p_i evaluates to `#t` or `#f`
- Value = value of e_i for the first p_i that evaluates to `#t` or e_N if p_N is “else” and all $p_1 \dots p_{N-1}$ evaluate to `#f`

```
(cond ((> 3 3) 'greater)
      ((< 3 3) 'less)
      (else 'equal))    => equal
```

Conditionals

◆ Simplified form

- (if (< x 0) (- 0 x)) ; if-then
- (if (< x y) x y) ; if-then-else

◆ Boolean predicates:

(and (e1) ... (eN)), (or (e1) ... (eN)), (not e)

Other Control Flow Constructs

◆ Case selection

- (case month
 ((sep apr jun nov) 30)
 ((feb) 28)
 (else 31)
)

◆ What about loops?

- Iteration \leftrightarrow Tail recursion
- Scheme implementations must implement tail-recursive functions as iteration

Delayed Evaluation

- ◆ Bind the expression to the name as a literal...
 - `(define sum '(+ 1 2 3))`
 - `sum` \Rightarrow `(+ 1 2 3)`
 - Evaluated as a symbol, not a function
- ◆ Evaluate as a function
 - `(eval sum)` \Rightarrow 6
- ◆ No distinction between code (i.e., functions) and data – both are represented as lists!

Imperative Features

Scheme allows imperative changes to values of variable bindings

- `(define x `(1 2 3))`
- `(set! x 5)`

```
(define x 3)
```

```
(define (foo)
```

```
  (define x 4)  
  x)
```

Imperative Features

```
(define (bar)
  (set! x 4)
  x)
```

(foo) ; returns 4

x ; still 3

(bar) ; returns 4

x ; is now 4

Let Expressions

◆ Nested static scope

◆ `(let ((var1 exp1) ... (varN expN)) body)`

`(define (subst y x alist)`

`(if (null? alist) '()`

`(let ((head (car alist)) (tail (cdr alist)))`

`(if (equal? x head)`

`(cons y (subst y x tail))`

`(cons head (subst y x tail))))))`

Let*

- ◆ `(let* ((var1 exp1) ... (varN expN)) body)`
 - Bindings are applied sequentially, so var_i is bound in $\text{exp}_{i+1} \dots \text{exp}_N$
- ◆ This is also syntactic sugar for a (different) lambda application
 - `(lambda (var1) (
 (lambda (var2) (
 (lambda (varN) (body)) expN) ...) exp1`

Let and Let*

```
(let ((x 10)
      (y (+ x 6))) ; error! unbound identifier in
                  module in: x
  y)
```

```
(let* ((x 10)
       (y (+ x 6))) ; works fine
  y)
=> 16
```

Functions as Arguments

```
(define (mapcar fun alist)
  (if (null? alist) '()
      (cons (fun (car alist))
              (mapcar fun (cdr alist)))
  ))
```

```
(define (square x) (* x x))
```

What does `(mapcar square '(2 3 5 7 9))` return?
`(4 9 25 49 81)`

Using Recursion

- ◆ Compute length of the list recursively
 - (define length
 (lambda(list)
 (if (null? list) 0 (+ 1 (length (cdr list))))))

Example Programs

```
(define (contains? item lst)
  (cond
    [(empty? lst) false]
    [(= item (first lst)) true]
    [else (contains? item (rest lst))]))
```

Key Features of Scheme

- ◆ Scoping: static
- ◆ Typing: dynamic
- ◆ No distinction between code and data
 - Both functions and data are represented as lists
 - Lists are first-class objects
 - Can be created dynamically, passed as arguments to functions, returned as results of functions and expressions
 - This requires heap allocation and garbage collection
 - Self-evolving programs