



Functional Programming

Introduction to Scheme

Fall/Spring 2003
Copyright © 2003 Spelman College

Department of Computer and
Information Sciences (CIS)

CIS346



Functional Languages

- The design of the functional languages is based on mathematical functions
- A solid theoretical basis that is also closer to the user's abstract thinking
- Unconcerned with the architecture of the machines on which programs will run
- **Mathematical Functions** are mappings of members of one set, called the domain set, to another set, called the range set.

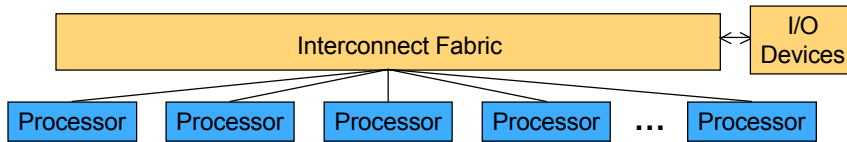
Fall/Spring 2003
Copyright © 2003 Spelman College

Department of Computer and
Information Sciences (CIS)

CIS346
(Lecture #7 Slide #2)



Functional Execution Model



- Based on the composition of functions to compute
 - **Functional mathematics**
 - $G(X) = X + 2$, $H(X) = X^2$
 - $F(X) = G(H(X))$
 - Each function is assigned to a processor
- Interconnect fabric connects results of one processor to operands of another
- Computation **does not** involve explicit storage of data
- Lisp, **Scheme**, and ML are example languages



Lambda Expressions

- A **lambda expression** specifies the parameter(s) and the mapping of a function in the following form
 $\lambda(x) x * x * x$
for the function $\text{cube}(x) = x * x * x$
- Lambda expressions describe nameless functions
- Lambda expressions are applied to parameter(s) by placing the parameter(s) after the expression
 $(\lambda(x) x * x * x)(3)$
which evaluates to 27
- Lambda expressions are used in Scheme/Lisp
 - **Nameless functions**
 - **Defining functions**
 - **Dynamically Generated Functions**



Functional Forms

- A functional form can take functions as parameters or produce functions as the result
- Some rules related to functional forms

given: $f(x) = x * x$ and $g(x) = x + 3$

The Form: $h = f \circ g$ (**composition**)

is translated as $h(x) = f(g(x))$ and $h(2) = f(g(2)) = 25$

The Form: $[f, g]$

is translated to $(f(x), g(x))$ such that $[f, g](4)$ is the list $(16, 7)$

The Form: \square (**apply-to-all**)

$\square(f, (x_1, x_2, \dots, x_n))$ is translated to $(f(x_1), f(x_2), \dots, f(x_n))$

such that $\square(g, (2, 4, 6))$ is the list $(5, 7, 9)$



Scheme

- A mid-1970s dialect of LISP
 - Designed to be cleaner
 - Modern
 - Simpler version
- Uses only static scoping
- Functions are first-class entities
 - They can be the values of expressions and elements of lists
 - They can be assigned to variables and passed as parameters



Scheme Syntax

- An operation in scheme is a function
 - (`<op> <opnd2> <opnd2> ... <opndn>`)
- An operand in scheme is
 - A value or identifier (called an Atom)
 - A list (a collection of atoms and/or lists)
 - (`<elem1> <elem2> ... <elemn>`)
 - A function
- A scheme program is a composition of functions

```
(define hello-world
  (lambda ()
    (begin
      (write 'Hello-World)
      (newline))))
```



Primitive Operations

Expression	Result
<code>(* 5 6)</code>	30
<code>(+ 10 5 20)</code>	35
<code>(sqrt (+ 20 5))</code>	5
<code>(+ (sqrt 16) (abs -5))</code>	9
<code>(QUOTE (* 5 6))</code>	<code>(* 5 6)</code>
<code>'(* 5 6)</code>	<code>(* 5 6)</code>

- Primitive operations are +, -, *, /, abs, sqrt, etc.
 - Uses **apply-to-all form semantics**
- The QUOTE operator takes one parameter, and does not allow evaluation
 - Useful when data is represented as a list
 - Can be abbreviated using the single-quote



CAR and CDR

- Scheme has a primitive datatype called a list
 '(a b c 1 2 3)
 '((charles 35 1968) (felicia 33 1970) (erin 5 1998) (cj 3 2000))
- List operations include **CAR** ("car") and **CDR** ("coulder")
 - **CAR** is 'First'
 - **CDR** is 'Rest'
- CAR and CDR are names with an interesting history
 - **CAR** (Contents of Address Register)
 - **CDR** (Contents of Decrement Register)

Expression	Result
(car '(a b c 1 2 3))	a
(cdr '(a b c 1 2 3))	(b c 1 2 3)
(car '((charles 35 1968)...(cj 3 2000)))	(charles 35 1968)
(cdr '((charles 35 1968)...(cj 3 2000)))	((felicia 33 1970) ... (cj 3 2000))
(car (cdr '(a b c 1 2 3)))	b

Fall/Spring 2003
 Copyright © 2003 Spelman College

Department of Computer and
 Information Sciences (CIS)

CIS346
 (Lecture #7 Slide #9)



Other List Operations

- **Cons** constructs a list from an atom/list and another list, where the atom/list is the CAR of the new list.
 (cons <atom or list> <list>)
- **List** constructs a list for an arbitrary number of atoms and/or lists that will be the list elements
 (list [atom | list]+)

Expression	Result
(cons 'a '(b c d))	(a b c d)
(cons '(a b) '(b c d))	((a b) b c d)
(list 'a 'b '(c d) 'a)	(a b (c d) a)
(list 'a '(b c d))	(a (b c d))

Fall/Spring 2003
 Copyright © 2003 Spelman College

Department of Computer and
 Information Sciences (CIS)

CIS346
 (Lecture #7 Slide #10)



Predicates

- True is #T and False is () “null list”

Expression	Result	Meaning
(eq? 'a 'a)	#T	symbolic comparison
(eq? 'a '(a b))	()	
(list? 'a)	()	determine if arg is a list
(list? '(a b c))	#T	
(null? '(a b c))	()	determine if arg is null
(null? (cdr '(a)))	#T	
(= 5 6)	()	comparing numeric arguments (=, <>, <, >, <=, >=, even?, odd?, zero?, positive?, negative?)
(< 5 6)	#T	



Control-Flow: Selection

- Selection: IF-FORM
(if <predicate> <then_exp> <else_exp>)
- Multiple Selection: COND-FORM (like C/C++ “switch”)
(cond
 <predicate> <expr> [<expr>]
 <predicate> <expr> [<expr>]
 ...
 (else <expr> [<expr>])) ;; C/C++ default

Expression	Result
(if (<> count 0) (/ sum count) 0)	if count is not equal to 0 then evaluate to sum/count otherwise evaluate to 0
(define (member atm lis) (cond ((null? lis) '()) ((eq? atm (car list)) #T) (else (member atm (cdr list))))))	if the lis is null then stop searching if the atm equals the first of the lis then its found otherwise search the “rest” of the list



Control-Flow: Iteration

- Iteration in functional languages is accomplished by recursion
- Iteration in imperative languages requires maintaining state information
 - **loop variable**
 - **sentinel value**

Iteration (imperative loop)	Iteration (functional recursion)
<pre>j = 0; found=false; while ((j < MAX_SIZE) && (!found)) { if (atm == lis[j]) found = true; j++; }</pre>	<pre>(define (member atm lis) (cond ((null? lis) '()) ((eq? atm (car list)) #T) ((else (member atm (cdr list))))))</pre>

Fall/Spring 2003
Copyright © 2003 Spelman College

Department of Computer and
Information Sciences (CIS)

CIS346
(Lecture #7 Slide #13)



Binding Symbols

- Binding symbols to expressions
 - **Defining identifiers**
- Binding symbols to lambda expressions
 - **Defining functions**

Fall/Spring 2003
Copyright © 2003 Spelman College

Department of Computer and
Information Sciences (CIS)

CIS346
(Lecture #7 Slide #14)



Binding Symbols: Identifier Defs

- The *define* operation is used to bind symbols to expressions to create “identifiers”

```
(define pi 3.141593)
```

```
(define two_pi (* 2 pi))
```



Binding Symbols: Function Defs

- Binding symbols to lambda expressions defines new functions

```
(lambda (L) (car (car L))) ;; defines a lambda function
```

```
((lambda (L) (car (car L))) '((a b) c d)) ;; applies a lambda function
```

Result: a

```
(define (two-car L) (car (car L))) ;; binds (two-car L) to lambda func
```

```
(two-car '((a b) c d)) ;; applies (two-car L) function
```

```
(define (cube x) (* x x x)) ;; binds (cube x) to lambda function
```



Function Evaluation Process: Step 1

- Consider the example
(hourly-wages (* 5 2) 10.2)
 - 1. Parameters are evaluated in any order
(hourly-wages 10 10.2)
- ```
(define hourly-wages
 (lambda (no-hours hourly-rate)
 (if (<= no-hours 40)
 (* no-hours hourly-rate)
 (+(* 40 hourly-rate)
 (*(- no-hours 40)
 hourly-rate
 1.5))))))
```



## Function Evaluation Process: Step 2

- Consider the example  
(hourly-wages (\* 5 2) 10.2)
- 1. Parameters are evaluated in any order  
(hourly-wages 10 10.2)
- 2. Value of parameters are substituted in the function body  
(if (<= no-hours 40)  
 (\* no-hours hourly-rate)  
 (+(\* 40 hourly-rate)  
 (\*(- no-hours 40)  
 hourly-rate  
 1.5))))))



## Function Evaluation Process: Step 3

- Consider the example  
(**hourly-wages** (\* 5 2) 10.2)
- 1. Parameters are evaluated in any order
- 2. Value of parameters are substituted in the function body
- 3. The function body is evaluated

```
(define hourly-wages
 (lambda (no-hours hourly-rate)
 (if (<= 10 40) <-- #T
 (* 10 10.2) <-- 102 } Then
 (+(* 40 10.2)
 (*(- 10 40)
 10.2
 1.5)))) } Else
```



## Function Evaluation Process: Step 4

- Consider the example  
(**hourly-wages** (\* 5 2) 10.2)
- 1. Parameters are evaluated in any order
- 2. Value of parameters are substituted in the function body
- 3. The function body is evaluated
- 4. Evaluation of the last expression in the body is the value

**102 is value of the last expression**

```
(define hourly-wages
 (lambda (no-hours hourly-rate)
 (if (<= no-hours 40)
 (* 10 10.2) --> 102
 (+(* 40 hourly-rate)
 (*(- no-hours 40)
 hourly-rate
 1.5))))
```