Fundamentals of Artificial Intelligence

COMP221: Functional Programming in Scheme (and LISP)

Prof. Dekai Wu

Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Hong Kong, China

Fall 2009

Part I

Introduction

Models of Computation

In the 1930's, long before digital computers were invented, logicians studied abstract concepts of computation.

- Two simple models of computation:

Models of Computation

In the 1930's, long before digital computers were invented, logicians studied abstract concepts of computation.

- Two simple models of computation:
  1. Alan Turing's Turing machines — based on an updatable store (memory).

Models of Computation

In the 1930's, long before digital computers were invented, logicians studied abstract concepts of computation.

- Two simple models of computation:
  1. Alan Turing's Turing machines — based on an updatable store (memory).
  2. Alonzo Church's lambda calculus — based on the mathematical concept of functions.

Models of Computation

In the 1930's, long before digital computers were invented, logicians studied abstract concepts of computation.

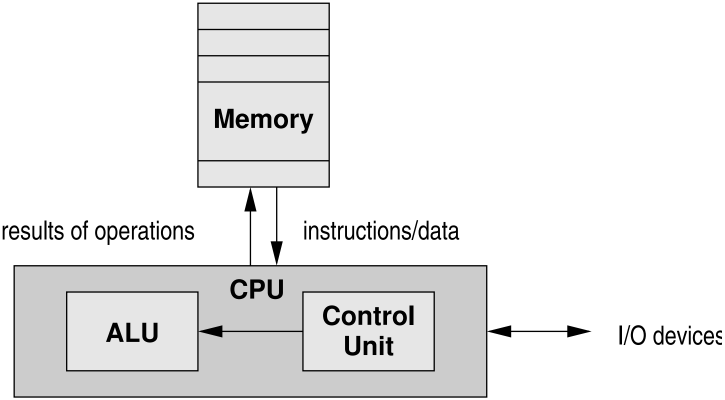
- Two simple models of computation:
  1. Alan Turing's Turing machines — based on an updatable store (memory).
  2. Alonzo Church's lambda calculus — based on the mathematical concept of functions.
- The Turing machine idea has great influence on the design of von Neumann computer architecture — used by most of today's common CPUs, and imperative programming languages are designed to compute efficiently on the architecture.

Models of Computation

In the 1930's, long before digital computers were invented, logicians studied abstract concepts of computation.

- Two simple models of computation:
  1. Alan Turing's Turing machines — based on an updatable store (memory).
  2. Alonzo Church's lambda calculus — based on the mathematical concept of functions.
- The Turing machine idea has great influence on the design of von Neumann computer architecture — used by most of today's common CPUs, and imperative programming languages are designed to compute efficiently on the architecture.
- Lambda calculus is the foundation of functional programming paradigm — closely tied to AI, as well as modern distributed programming, compilers, software verification, much more.

von Neumann Computer Architecture



Church-Turing Thesis

Any effective computation can be done in one of the two models.

Functional Programming Languages

- Functions are first-class objects: They may be (1) passed as arguments (2) returned as results (3) stored in variables. (c.f. variables in imperative languages.)

Functional Programming Languages

- Functions are first-class objects: They may be (1) passed as arguments (2) returned as results (3) stored in variables. (c.f. variables in imperative languages.)
- Basic mode of computation: construction and application of functions. (c.f. assignments in imperative languages.)

Functional Programming Languages

- Functions are first-class objects: They may be (1) passed as arguments (2) returned as results (3) stored in variables. (c.f. variables in imperative languages.)
- Basic mode of computation: construction and application of functions. (c.f. assignments in imperative languages.)
- Principal control mechanism: recursive function applications. (c.f. for-/while-loops in imperative languages.)

Functional Programming Languages

- Functions are first-class objects: They may be (1) passed as arguments (2) returned as results (3) stored in variables. (c.f. variables in imperative languages.)
- Basic mode of computation: construction and application of functions. (c.f. assignments in imperative languages.)
- Principal control mechanism: recursive function applications. (c.f. for-/while-loops in imperative languages.)
- Freed from side-effects (for "pure" FPL).
  Side-Effects: Operations which permanently change the value of a variable (by assignments) or other observable objects (e.g. by printing outputs).

Functional Programming Languages

- Functions are first-class objects: They may be (1) passed as arguments (2) returned as results (3) stored in variables. (c.f. variables in imperative languages.)
- Basic mode of computation: construction and application of functions. (c.f. assignments in imperative languages.)
- Principal control mechanism: recursive function applications. (c.f. for-/while-loops in imperative languages.)
- Freed from side-effects (for "pure" FPL).
  Side-Effects: Operations which permanently change the value of a variable (by assignments) or other observable objects (e.g. by printing outputs).
- Examples: LISP (LISt Processing), Scheme (a dialect of LISP), ML (Meta Language), Haskell, Miranda, etc.

Scheme, a dialect of LISP

Scheme supports:

- Static Scope: All identifier references can be resolved at compile time (unlike LISP, which uses dynamic scoping).

Scheme, a dialect of LISP

Scheme supports:

- Static Scope: All identifier references can be resolved at compile time (unlike LISP, which uses dynamic scoping).
- Dynamic or Latent Typing: All variables are dynamically or latently typed. Any variable can hold any type (imagine that all variables are of the type Object). There are no explicit type declarations. Many scripting languages take the same approach (e.g., Perl, Python, JavaScript, Tcl, etc.)

Scheme, a dialect of LISP

Scheme supports:

- Static Scope: All identifier references can be resolved at compile time (unlike LISP, which uses dynamic scoping).
- Dynamic or Latent Typing: All variables are dynamically or latently typed. Any variable can hold any type (imagine that all variables are of the type Object). There are no explicit type declarations. Many scripting languages take the same approach (e.g., Perl, Python, JavaScript, Tcl, etc.)
- Proper Tail-Recursion: Scheme (but not LISP) implementations are required to be properly tail-recursive, supporting an unbounded number of active tail calls.

Scheme, a dialect of LISP

Scheme supports:

- Static Scope: All identifier references can be resolved at compile time (unlike LISP, which uses dynamic scoping).
- Dynamic or Latent Typing: All variables are dynamically or latently typed. Any variable can hold any type (imagine that all variables are of the type Object). There are no explicit type declarations. Many scripting languages take the same approach (e.g., Perl, Python, JavaScript, Tcl, etc.)
- Proper Tail-Recursion: Scheme (but not LISP) implementations are required to be properly tail-recursive, supporting an unbounded number of active tail calls.
- Pure and Impure Functional Programming: Scheme/LISP encourage pure functional programming, but impure functional programming using some side-effects can also be done.

Scheme, a dialect of LISP (cont'd)

Scheme supports:

- List-Oriented Programming: The main data structure in Scheme/LISP is the list, implemented as a cons data structure or cons tree. This versatile data structure can easily be applied in many ways, and is easy to manipulate.

Scheme, a dialect of LISP (cont'd)

Scheme supports:

- List-Oriented Programming: The main data structure in Scheme/LISP is the list, implemented as a cons data structure or cons tree. This versatile data structure can easily be applied in many ways, and is easy to manipulate.
- Programs as Data: The concrete syntax of Scheme/LISP is almost the abstract syntax. All expressions are in prefix notation and fully parenthesized. A big advantage is that programs look just like cons list data structures, which makes it easy to manipulate programs as if they were just data (so programs can write and execute programs).

Scheme, a dialect of LISP (cont'd)

Scheme supports:

- List-Oriented Programming: The main data structure in Scheme/LISP is the list, implemented as a cons data structure or cons tree. This versatile data structure can easily be applied in many ways, and is easy to manipulate.
- Programs as Data: The concrete syntax of Scheme/LISP is almost the abstract syntax. All expressions are in prefix notation and fully parenthesized. A big advantage is that programs look just like cons list data structures, which makes it easy to manipulate programs as if they were just data (so programs can write and execute programs).
- Interpreted: Traditionally, Scheme/LISP implementations all provide an interpreter (and sometimes also a compiler), making interactive debugging easy.

Scheme, a dialect of LISP (cont'd)

Scheme supports:

- List-Oriented Programming: The main data structure in Scheme/LISP is the list, implemented as a cons data structure or cons tree. This versatile data structure can easily be applied in many ways, and is easy to manipulate.
- Programs as Data: The concrete syntax of Scheme/LISP is almost the abstract syntax. All expressions are in prefix notation and fully parenthesized. A big advantage is that programs look just like cons list data structures, which makes it easy to manipulate programs as if they were just data (so programs can write and execute programs).
- Interpreted: Traditionally, Scheme/LISP implementations all provide an interpreter (and sometimes also a compiler), making interactive debugging easy.
- Automatic Garbage Collection: All implementations all provide this feature, which Java inherits.

Part II

Types and Values

8 Basic Types, 2 Composite Types

TYPE	EXAMPLE	COMMON OPERATIONS
boolean	#t, #f	boolean?, not, and, or
integer	2, 0, 87	+, -, *, quotient, remainder, modulo
rational	(/ 4 6)	+, -, *, /, numerator, denominator
real	1.3, 300.1	real?, +, -, *, /, floor
complex	3+4i	complex?, +, -, *, /
character	#\a, #\space, #\newline	char?
string	"hello"	string?, string-ref, string-set!
symbol	hello	symbol?, eqv?, symbol->string
dotted pair	(1 . "yes")	list?, null?, cons, car, cdr
list	(), (1 "yes" 2.5) (1 . ("yes" . (2.5 . ())))	
vector	#(1 "yes" 2.5)	vector?, vector-ref, vector-set!, vector

Quoting

Syntax:

> (**quote** < *expression* >)
> **'**< *expression* >

- Quoting is needed to treat expressions as data.

```
> (define pi 3.14159)
> pi
3.14159
```

Quoting

Syntax:

> (**quote** < *expression* >)
> **'**< *expression* >

- Quoting is needed to treat expressions as data.

```
> (define pi 3.14159)
> pi
3.14159
```

- A quoted item evaluates to itself. quote prevents eval from being called:

```
> (quote pi)
pi
```

Quoting

Syntax:

(**quote** $<$ *expression* $>$)
'$<$ *expression* $>$

- Quoting is needed to treat expressions as data.

```
> (define pi 3.14159)
> pi
3.14159
```

- A quoted item evaluates to itself. quote prevents eval from being called:

```
> (quote pi)
pi
```

- An equivalent convenient shorthand is provided using ' :

```
> 'pi
pi
```

Quoting (cont'd)

- Unquoted, * represents the multiplication function:

```
> (define f *)
> (f 2 3)
6
```

Quoting (cont'd)

- Unquoted, * represents the multiplication function:

```
> (define f *)
> (f 2 3)
6
```

- Quoted, '* represents the symbol with spelling * :

```
> (define f '*)
> (f 2 3)
ERROR: Wrong type to apply: *
```

List

- Empty list: '()

List

- Empty list: '()
- '($e_1 e_2 \ldots e_n$) is an abbreviation for
  (cons e_1 (cons e_2 (cons ... (cons e_n '())))))

List

- Empty list: '()
- '($e_1 e_2 \ldots e_n$) is an abbreviation for (cons e_1 (cons e_2 (cons ... (cons e_n '())))))
- cons is the list constructor

List

- Empty list: '()
- '($e_1 e_2 \ldots e_n$) is an abbreviation for (cons e_1 (cons e_2 (cons $\ldots$ (cons e_n '())))))
- cons is the list constructor
- <new-list> = (cons <item> <list>)

```
  (cons 1 (cons 2 (cons 3 '())))
= (cons 1 (cons 2 '(3)))
= (cons 1 '(2 3))
= '(1 2 3)
```

List

- Empty list: '()
- '($e_1 e_2 \ldots e_n$) is an abbreviation for (cons e_1 (cons e_2 (cons ... (cons e_n '())))))
- cons is the list constructor
- <new-list> = (cons <item> <list>)

```
  (cons 1 (cons 2 (cons 3 '())))
= (cons 1 (cons 2 '(3)))
= (cons 1 '(2 3))
= '(1 2 3)
```

- Equality on 2 lists is item-by-item.

List Operators

- cons operator: creates a dotted pair of the two operands

List Operators

- cons operator: creates a dotted pair of the two operands
- car operator: returns the first element of a dotted pair (cons cell)

List Operators

- cons operator: creates a dotted pair of the two operands
- car operator: returns the first element of a dotted pair (cons cell)
- cdr operator: returns the second element of a dotted pair (cons cell)

List Operators

- cons operator: creates a dotted pair of the two operands
- car operator: returns the first element of a dotted pair (cons cell)
- cdr operator: returns the second element of a dotted pair (cons cell)
- append operator: returns a list consisting of the elements of the first list followed by the elements of the other lists

List Examples

List Examples

```
> (car '(1 2 3 4))
1
```

List Examples

```
> (car '(1 2 3 4))
1

> (cdr '(1 2 3 4))
(2 3 4)
```

List Examples

```
> (car '(1 2 3 4))
1

> (cdr '(1 2 3 4))
(2 3 4)

> (cons (car '(1 2 3 4)) (cdr '(1 2 3 4)))
(1 2 3 4)
```

List Examples

```
> (car '(1 2 3 4))
1

> (cdr '(1 2 3 4))
(2 3 4)

> (cons (car '(1 2 3 4)) (cdr '(1 2 3 4)))
(1 2 3 4)

> (append '(5 6) (cdr '(1 2 3 4)))
(5 6 2 3 4)
```

Composite Type: Vector

Composite Type: Vector

```
> (vector 4 true cat)
#(4 true cat)
> (if (equal? 3 8) "X" (vector "Y" 9.5/0.5 (quotient 5 2)))
#("Y" 19.0 2)
> (equal? (vector (modulo 14 3) (not #f))
          (vector (+ 1 1) #t))
#t
```

Composite Type: Vector

```
> (vector 4 true cat)
#(4 true cat)
> (if (equal? 3 8) "X" (vector "Y" 9.5/0.5 (quotient 5 2)))
#("Y" 19.0 2)
> (equal? (vector (modulo 14 3) (not #f))
          (vector (+ 1 1) #t))
#t

> (vector-ref (vector "for" "your" "info") 1)
"your"
```

Composite Type: Vector

```
> (vector 4 true cat)
#(4 true cat)
> (if (equal? 3 8) "X" (vector "Y" 9.5/0.5 (quotient 5 2)))
#("Y" 19.0 2)
> (equal? (vector (modulo 14 3) (not #f))
          (vector (+ 1 1) #t))
#t

> (vector-ref (vector "for" "your" "info") 1)
"your"
```

- Ordered *n*-tuple: $\#(e_1 e_2 \ldots e_n)$.
- The *n* expressions may be of mixed types.
- 2 *n*-tuples are equal if their corresponding components are equal.

Composite Type: Vector ..

- "(vector-ref myvector k)" is the item selection operator.

Composite Type: Vector ..

- "(vector-ref myvector k)" is the item selection operator.
- "(vector-length myvector)" returns n.

- "(vector-ref myvector k)" is the item selection operator.
- "(vector-length myvector)" returns n.
- "(vector $\#(e_1 e_2 \ldots e_n)$)" returns a newly constructed vector containing the given elements.

- "(vector-ref myvector k)" is the item selection operator.
- "(vector-length myvector)" returns n.
- "(vector #($e_1 e_2 \ldots e_n$))" returns a newly constructed vector containing the given elements.
- "(make-vector n fill)" returns a newly allocated vector of n elements. If the optional second argument is given, then each element is initialized to fill.

Identifiers

Most identifiers allowed by other programming languages are also acceptable to Scheme. The precise rules for forming identifiers vary among implementations of Scheme, but in all implementations a sequence of letters, digits, and "extended alphabetic characters" that begins with a character that cannot begin a number is an identifier. In addition, +, -, and ... are identifiers. Here are some examples of identifiers:

Extended alphabetic characters may be used within identifiers as if they were letters. The following are extended alphabetic characters:

Identifiers

Most identifiers allowed by other programming languages are also acceptable to Scheme. The precise rules for forming identifiers vary among implementations of Scheme, but in all implementations a sequence of letters, digits, and "extended alphabetic characters" that begins with a character that cannot begin a number is an identifier. In addition, +, -, and ... are identifiers. Here are some examples of identifiers:

```
lambda                        q
list->vector                  soup
+                             V17a
<=?                           a34kTMNs
the-word-recursion-has-many-meanings
```

Extended alphabetic characters may be used within identifiers as if they were letters. The following are extended alphabetic characters:

Identifiers

Most identifiers allowed by other programming languages are also acceptable to Scheme. The precise rules for forming identifiers vary among implementations of Scheme, but in all implementations a sequence of letters, digits, and "extended alphabetic characters" that begins with a character that cannot begin a number is an identifier. In addition, +, -, and ... are identifiers. Here are some examples of identifiers:

```
lambda                        q
list->vector                  soup
+                             V17a
<=?                           a34kTMNs
the-word-recursion-has-many-meanings
```

Extended alphabetic characters may be used within identifiers as if they were letters. The following are extended alphabetic characters:

```
! $ % & * + - . / : < = > ? @ ^ _ ~
```

Identifiers (cont'd)

Identifiers have two uses within Scheme programs:

- Any identifier may be used as a variable or as a syntactic keyword.

Identifiers (cont'd)

Identifiers have two uses within Scheme programs:

- Any identifier may be used as a variable or as a syntactic keyword.
- When an identifier appears as a literal or within a literal, it is being used to denote a symbol.

Identifiers: Value Binding

Syntax: **(define** $<$ *identifier* $>$ $<$ *expression* $>$**)**

Identifiers: Value Binding

Syntax: **(define** *< identifier > < expression >***)**

```
> (define a_df (+ 3 2))  ; c.f. int a_df = 3+2; in C++
> a_df
5
```

Identifiers: Value Binding

Syntax: **(define** *< identifier > < expression >* **)**

```
> (define a_df (+ 3 2))  ; c.f. int a_df = 3+2; in C++
> a_df
5

> (define a'a (string-append "Albert" " " "Einstein"))
> a'a
"Albert Einstein"
```

Identifiers: Value Binding

Syntax: **(define** < *identifier* > < *expression* >**)**

```
> (define a_df (+ 3 2))   ; c.f. int a_df = 3+2; in C++
> a_df
5

> (define a'a (string-append "Albert" " " "Einstein"))
> a'a
"Albert Einstein"

> (define a1b2 2)
> a1b2
2
```

Identifiers: Value Binding

Syntax: **(define** *< identifier > < expression >* **)**

```
> (define a_df (+ 3 2))  ; c.f. int a_df = 3+2; in C++
> a_df
5

> (define a'a (string-append "Albert" " " "Einstein"))
> a'a
"Albert Einstein"

> (define a1b2 2)
> a1b2
2

> (define +++$$$ (* 9 3))  ; may hold integral value
> +++$$$
27
```

Identifiers: Value Binding

Syntax: **(define** *< identifier > < expression >***)**

```
> (define a_df (+ 3 2))  ; c.f. int a_df = 3+2; in C++
> a_df
5

> (define a'a (string-append "Albert" " " "Einstein"))
> a'a
"Albert Einstein"

> (define a1b2 2)
> a1b2
2

> (define +++$$$ (* 9 3))  ; may hold integral value
> +++$$$
27

> (+ +++$$$ +++$$$)  ; Though you don't want to do that
54
```

Part III

Scheme Functions

Lambda: Constructing Anonymous Functions

Syntax: (**lambda** (<formal parameters>) <body>)

- An *anonymous function* is a function without a name.
- lambda returnes a newly constructed anonymous parameterized function *value*.
- <formal parameters> is a sequence of parameter names.
- <body> is an expression, possibly containing occurrences of the parameter names.
- Used when only a locally defined function is needed.

Lambda: Constructing Anonymous Functions

Syntax: (**lambda** (<formal parameters>) <body>)

- An *anonymous function* is a function without a name.
- lambda returnes a newly constructed anonymous parameterized function *value*.
- <formal parameters> is a sequence of parameter names.
- <body> is an expression, possibly containing occurrences of the parameter names.
- Used when only a locally defined function is needed.

```
> (lambda (x) (* x x))   ; constructing anonymous function
#<procedure #f (x)>
```

Lambda: Constructing Anonymous Functions

Syntax: **(lambda** (<formal parameters>) <body>)

- An *anonymous function* is a function without a name.
- lambda returnes a newly constructed anonymous parameterized function *value*.
- <formal parameters> is a sequence of parameter names.
- <body> is an expression, possibly containing occurrences of the parameter names.
- Used when only a locally defined function is needed.

```
> (lambda (x) (* x x))   ; constructing anonymous function
#<procedure #f (x)>


> ((lambda (x) (* x x)) 4)   ; applying anonymous function
16
```

Defining Named Functions

- To define a named function, you simply have to bind some identifier (which will be the function name) to a function value.
- Parameter passing method: Call-By-Value.

Defining Named Functions

- To define a named function, you simply have to bind some identifier (which will be the function name) to a function value.
- Parameter passing method: Call-By-Value.

```
> (define square (lambda (x) (* x x)))
> square
#<procedure square (x)>
```

Defining Named Functions

- To define a named function, you simply have to bind some identifier (which will be the function name) to a function value.
- Parameter passing method: Call-By-Value.

```
> (define square (lambda (x) (* x x)))
> square
#<procedure square (x)>

> (square 4)
16
```

Defining Named Functions

- To define a named function, you simply have to bind some identifier (which will be the function name) to a function value.
- Parameter passing method: Call-By-Value.

```
> (define square (lambda (x) (* x x)))
> square
#<procedure square (x)>

> (square 4)
16

> (define first (lambda (x y) x))
> (first 3 "man")
3
```

Convenient “Syntactic Sugar” for Named Functions

- Syntax: (**define** (<identifier> <formal-parameters>) <body>)

Convenient “Syntactic Sugar” for Named Functions

- Syntax: (**define** (<identifier> <formal-parameters>) <body>)

```
> (define (square x) (* x x))
> square
#<procedure square (x)>
```

Convenient "Syntactic Sugar" for Named Functions

- Syntax: (**define** (<identifier> <formal-parameters>) <body>)

```
> (define (square x) (* x x))
> square
#<procedure square (x)>

> (square 4)
16
```

Convenient “Syntactic Sugar” for Named Functions

- Syntax: (**define** (<identifier> <formal-parameters>) <body>)

```
> (define (square x) (* x x))
> square
#<procedure square (x)>

> (square 4)
16

> (define (first x y) x)
> (first 3 "man")
3
```

Higher-Order Functions (I)

Functions taking functions as arguments:

Higher-Order Functions (I)

Functions taking functions as arguments:

```
> (define (square x) (* x x))
> (define (twice x) (* 2 x))
```

Higher-Order Functions (I)

Functions taking functions as arguments:

```
> (define (square x) (* x x))
> (define (twice x) (* 2 x))

> (define (apply5 f) (apply f '(5)))
> (apply5 square)
25
```

Higher-Order Functions (I)

Functions taking functions as arguments:

```
> (define (square x) (* x x))
> (define (twice x) (* 2 x))

> (define (apply5 f) (apply f '(5)))
> (apply5 square)
25

> (define (apply-to-twice-x f x) (apply f (list (twice x))))
> (apply-to-twice-x square 3)
36
```

Higher-Order Functions (II)

Functions returning function:

Higher-Order Functions (II)

Functions returning function:

```
> (define (sq_or_twice x) (if (> x 0) square twice))
```

Higher-Order Functions (II)

Functions returning function:

```
> (define (sq_or_twice x) (if (> x 0) square twice))

> (apply (sq_or_twice 2) '(5))
25
```

Higher-Order Functions (II)

Functions returning function:

```
> (define (sq_or_twice x) (if (> x 0) square twice))

> (apply (sq_or_twice 2) '(5))
25

> (sq_or_twice 2)
#<procedure square (x)>
```

Lambda Calculus

Recall this example:

```
> (lambda (x) (* x x))   ; constructing anonymous function
#<procedure #f (x)>


> ((lambda (x) (* x x)) 4)   ; applying anonymous function
16
```

Lambda Calculus

Recall this example:

```
> (lambda (x) (* x x))   ; constructing anonymous function
#<procedure #f (x)>


> ((lambda (x) (* x x)) 4)   ; applying anonymous function
16
```

In the lambda calculus, which was the origin of lambda in LISP/Scheme, the same two lines would be written:

Lambda Calculus

Recall this example:

```
> (lambda (x) (* x x))   ; constructing anonymous function
#<procedure #f (x)>


> ((lambda (x) (* x x)) 4)   ; applying anonymous function
16
```

In the lambda calculus, which was the origin of lambda in LISP/Scheme, the same two lines would be written:

$(\lambda x.\ (x * x))$

Recall this example:

```
> (lambda (x) (* x x))   ; constructing anonymous function
#<procedure #f (x)>


> ((lambda (x) (* x x)) 4)   ; applying anonymous function
16
```

In the lambda calculus, which was the origin of lambda in LISP/Scheme, the same two lines would be written:

$(\lambda x.\ (x * x))$

$(\lambda x.\ (x * x))\ 4$

Recall this example:

```
> (lambda (x) (* x x))  ; constructing anonymous function
#<procedure #f (x)>


> ((lambda (x) (* x x)) 4)  ; applying anonymous function
16
```

In the lambda calculus, which was the origin of lambda in LISP/Scheme, the same two lines would be written:

$(\lambda x.\ (x * x))$

$(\lambda x.\ (x * x))\ 4$

We say that applying λx to the expression $x * x$ performs a lambda abstraction.

Lambda Calculus: Syntactic Conventions (I)

A couple conventions for the lambda calculus make it much more readable.

First, parentheses may be dropped from (MN) and $(\lambda x.\ M)$.

Notice that function application is an operator; the operator is left associative. E.g., xyz is an abbreviation for $((xy)\ z)$. Also, function application has higher precedence than lambda abstraction, so

$$\lambda x.\ x * x$$

is an abbreviation for

$$(\lambda x.\ (x * x))$$

Quiz: What is the abbreviated syntax for this?

$((\lambda x.\ (x * x))\ 4)$

Quiz: What is the abbreviated syntax for this?

$((\lambda x.\ (x * x))\ 4)$

Answer:

$(\lambda x.\ x * x)\ 4$

Lambda Calculus: Syntactic Conventions (II)

Second, a sequence of consecutive lambda abstractions, as in

$\lambda x.\ \lambda y.\ \lambda z.\ M$

can be written with a single lambda, as in

$\lambda xyz.\ M$

Quiz: Note that $\lambda xyz.\ M$ corresponds to what we normally think of as a function with three parameters. If you instead write it as $\lambda x.\ \lambda y.\ \lambda z.\ M$, then what is the meaning of this:

Lambda Calculus: Syntactic Conventions (II..)

Quiz: Note that $\lambda xyz.\ M$ corresponds to what we normally think of as a function with three parameters. If you instead write it as $\lambda x.\ \lambda y.\ \lambda z.\ M$, then what is the meaning of this:

$(\lambda x.\ \lambda y.\ \lambda z.\ M)\ 8$

Quiz: Note that $\lambda xyz.\ M$ corresponds to what we normally think of as a function with three parameters. If you instead write it as $\lambda x.\ \lambda y.\ \lambda z.\ M$, then what is the meaning of this:

$(\lambda x.\ \lambda y.\ \lambda z.\ M)\ 8$

Answer: The value of this expression is a function of two parameters, where y and z remain free variables, but x has already been bound to the value 8.

Lambda Calculus: Syntactic Conventions (II..)

Quiz: Note that $\lambda xyz.\ M$ corresponds to what we normally think of as a function with three parameters. If you instead write it as $\lambda x.\ \lambda y.\ \lambda z.\ M$, then what is the meaning of this:

$(\lambda x.\ \lambda y.\ \lambda z.\ M)\ 8$

Answer: The value of this expression is a function of two parameters, where y and z remain free variables, but x has already been bound to the value 8.

The practice of breaking up multi-parameter functions into single-parameter functions like this in programming is called currying a function (after Tim Curry who promoted this practice, widely found in functional languages like ML and Haskell).

Lambda Calculus: Substitution

- The result of applying an abstraction $(\lambda x.\ M)$ to an argument N is formalized as substitution of N for x in M, written $\{N/x\}\ M$.

Lambda Calculus: Substitution

- The result of applying an abstraction $(\lambda x.\ M)$ to an argument N is formalized as substitution of N for x in M, written $\{N/x\}\ M$.
- Informally, N replaces all free occurrences of x in M.

Lambda Calculus: Substitution

- The result of applying an abstraction ($\lambda x.\ M$) to an argument N is formalized as substitution of N for x in M, written $\{N/x\}\ M$.
- Informally, N replaces all free occurrences of x in M.
- Caution! A fully correct definition of substitution is tricky (there was a long history of not-quite-adequate definitions). This is beyond our scope here.

Lambda Calculus: Substitution

- The result of applying an abstraction $(\lambda x.\ M)$ to an argument N is formalized as substitution of N for x in M, written $\{N/x\}\ M$.
- Informally, N replaces all free occurrences of x in M.
- Caution! A fully correct definition of substitution is tricky (there was a long history of not-quite-adequate definitions). This is beyond our scope here.
- Here is an almost-correct definition:

Lambda Calculus: Substitution

- The result of applying an abstraction $(\lambda x.\ M)$ to an argument N is formalized as substitution of N for x in M, written $\{N/x\}\ M$.
- Informally, N replaces all free occurrences of x in M.
- Caution! A fully correct definition of substitution is tricky (there was a long history of not-quite-adequate definitions). This is beyond our scope here.
- Here is an almost-correct definition:
    1. Suppose the free variables of N have no bound occurrences in M. Then the term $\{N/x\}\ M$ is formed by replacing all free occurrences of x in M by N.

- The result of applying an abstraction $(\lambda x.\ M)$ to an argument N is formalized as substitution of N for x in M, written $\{N/x\}\ M$.
- Informally, N replaces all free occurrences of x in M.
- Caution! A fully correct definition of substitution is tricky (there was a long history of not-quite-adequate definitions). This is beyond our scope here.
- Here is an almost-correct definition:
  1. Suppose the free variables of N have no bound occurrences in M. Then the term $\{N/x\}\ M$ is formed by replacing all free occurrences of x in M by N.
  2. Otherwise, suppose variable y is free in N and bound in M. Consistently replace the binding and corresponding bound occurrences of y in M by some fresh variable z. Repeat the renaming of bound variables in M until case 1 applies, then proceed as in case 1.

Lambda Calculus: Substitution examples

In the following examples, M has no bound occurrences, so N replaces all occurrences of x in M to form $\{N/x\}\ M$:

$$
\begin{aligned}
\{u/x\}\ x &= u \\
\{u/x\}\ (x\ x) &= (u\ u) \\
\{u/x\}\ (x\ y) &= (u\ y) \\
\{u/x\}\ (x\ u) &= (u\ u) \\
\{(\lambda x.\ x)/x\}\ (x\ u) &= (\lambda x.\ u)
\end{aligned}
$$

Lambda Calculus: Substitution examples..

In the following examples, M has no free occurrences, so $\{N/x\}\ M$ is M itself:

$$
\begin{array}{rcl}
\{u/x\}\ y & = & y \\
\{u/x\}\ (y\ z) & = & (y\ x) \\
\{u/x\}\ (\lambda y.\ y) & = & (\lambda y.\ y) \\
\{u/x\}\ (\lambda x.\ x) & = & (\lambda x.\ x) \\
\{(\lambda x.\ x)/x\}\ y & = & y
\end{array}
$$

In the following examples, free variable u in M has bound occurrences in M, so $\{N/x\}\ M$ is formed by first renaming the bound occurrences of u in M:

$$
\begin{aligned}
\{u/x\}\ (\lambda u.\ x) = \{u/x\}\ (\lambda z.\ x) &= (\lambda z.\ u) \\
\{u/x\}\ (\lambda u.\ u) = \{u/x\}\ (\lambda z.\ z) &= (\lambda z.\ z)
\end{aligned}
$$

List Function: map

- The built-in library function map() has 2 or more arguments: a function <func> and one or more lists.

List Function: map

- The built-in library function map() has 2 or more arguments: a function <func> and one or more lists.
- It applies function <func> to the elements of the lists as follows.

  (map <func> $<\text{list}_1> \; <\text{list}_2>$ ...)

  <func> must be a function taking as many arguments as there are lists and returning a single value. If more than one list is given, then they must all be the same length. Map applies <func> element-wise to the elements of the lists and returns a list of the results, in order. The dynamic order in which <func> is applied to the elements of the lists is unspecified.

map: Examples

map: Examples

```
> (define (odd x) (equal? (modulo x 2) 1))
```

map: Examples

```
> (define (odd x) (equal? (modulo x 2) 1))

> (map odd '(1 2 3))
(#t #f #t)
```

map: Examples

```
> (define (odd x) (equal? (modulo x 2) 1))

> (map odd '(1 2 3))
(#t #f #t)

> (map cadr '((a b) (d e) (g h)))
(b e h)
```

map: Examples

```
> (define (odd x) (equal? (modulo x 2) 1))

> (map odd '(1 2 3))
(#t #f #t)

> (map cadr '((a b) (d e) (g h)))
(b e h)

> (map (lambda (n) (expt n n))
       '(1 2 3 4 5))
(1 4 27 256 3125)
```

map: Examples

```
> (define (odd x) (equal? (modulo x 2) 1))

> (map odd '(1 2 3))
(#t #f #t)

> (map cadr '((a b) (d e) (g h)))
(b e h)

> (map (lambda (n) (expt n n))
       '(1 2 3 4 5))
(1 4 27 256 3125)

> (map + '(1 2 3) '(4 5 6))
(5 7 9)
```

Part IV

Static Scope: let Expression

let Expression

(**let**
((<1st-identifier> $< E_1 >$)
(<2nd-identifier> $< E_2 >$)
. . .)
<body-expression>)

(**let**

((<1st-identifier> $< E_1 >$)

(<2nd-identifier> $< E_2 >$)

$\ldots$)

<body-expression>)

- c.f. Declaration of local variables in C++

let Example

```
> (define z
     (let ((x 3)
           (y 5))
        (+ (* x x) (* 3 y))))
> z
24
```

let Example

```
> (define z
      (let ((x 3)
            (y 5))
         (+ (* x x) (* 3 y))))
> z
24
```

- As spaces are immaterial, the statement may as well be written all in one single line as follows:

```
> (define z (let ((x 3) (y 5)) (+ (* x x) (* 3 y))))
```

let Example

```
> (define z
      (let ((x 3)
            (y 5))
        (+ (* x x) (* 3 y))))
> z
24
```

- As spaces are immaterial, the statement may as well be written all in one single line as follows:

```
> (define z (let ((x 3) (y 5)) (+ (* x x) (* 3 y))))
```

- Quiz: What is the relationship between let and lambda?

```
> (define z
      ((lambda (x y) (+ (* x x) (* 3 y)))
       3 5))
> z
24
```

Nested let Example

```
> (let ((x 3.0)
        (y 5.0))
     (let ((a (+ x y))
           (b (- x y)))
        (let ((f (* a b x))
              (g (/ a b y)))
           (/ f g))))
```

Quiz: What is the output?

Part V

Misc: Different Notions of Equality and Equivalence

Equality and Equivalence Predicates

Scheme (like LISP) offers various notions of equality and equivalence, to support reference vs. value comparisons over different types, with different efficiency tradeoffs.

Equality and Equivalence Predicates

Scheme (like LISP) offers various notions of equality and equivalence, to support reference vs. value comparisons over different types, with different efficiency tradeoffs.

- $=$: Only applies to numeric values. (Very efficient.)

Equality and Equivalence Predicates

Scheme (like LISP) offers various notions of equality and equivalence, to support reference vs. value comparisons over different types, with different efficiency tradeoffs.

- =: Only applies to numeric values. (Very efficient.)
- char=: Only applies to character values. (Very efficient.)

Equality and Equivalence Predicates

Scheme (like LISP) offers various notions of equality and equivalence, to support reference vs. value comparisons over different types, with different efficiency tradeoffs.

- =: Only applies to numeric values. (Very efficient.)
- char=: Only applies to character values. (Very efficient.)
- string=: Only applies to string values.

Equality and Equivalence Predicates

Scheme (like LISP) offers various notions of equality and equivalence, to support reference vs. value comparisons over different types, with different efficiency tradeoffs.

- =: Only applies to numeric values. (Very efficient.)
- char=: Only applies to character values. (Very efficient.)
- string=: Only applies to string values.
- eq?: Merely compares references and booleans. (Very efficient; essentially just pointer comparison.)

Equality and Equivalence Predicates

Scheme (like LISP) offers various notions of equality and equivalence, to support reference vs. value comparisons over different types, with different efficiency tradeoffs.

- $=$: Only applies to numeric values. (Very efficient.)
- char$=$: Only applies to character values. (Very efficient.)
- string$=$: Only applies to string values.
- eq?: Merely compares references and booleans. (Very efficient; essentially just pointer comparison.)
- eqv?: Like eq? combined with $=$ plus handles numeric and character values. (Still efficient but slightly less so.)

Equality and Equivalence Predicates

Scheme (like LISP) offers various notions of equality and equivalence, to support reference vs. value comparisons over different types, with different efficiency tradeoffs.

- $=$: Only applies to numeric values. (Very efficient.)
- char$=$: Only applies to character values. (Very efficient.)
- string$=$: Only applies to string values.
- eq?: Merely compares references and booleans. (Very efficient; essentially just pointer comparison.)
- eqv?: Like eq? combined with $=$ plus handles numeric and character values. (Still efficient but slightly less so.)
- equal?: Recursively compares the contents of pairs, vectors, and strings, applying eqv? on other objects such as numbers and symbols. A rule of thumb is that objects are generally equal? if they print the same. (Least efficient. *Why?*)

eq? vs. eqv? vs. equal?

	eq?	eqv?	equal?
(eq? '() #f) ; *cf LISP's* nil	#f	#f	#f
(eq? 'a 'a)	#t	#t	#t
(eq? 'a 'b)	#f	#f	#f
(eq? 2 2)	#t only if eqv?	#t	#t
(eq? #\a #\a)	#t only if eqv?	#t	#t
(eq? "" "")	unspecified	unspecified	#t
(eq? "a" "a")	#t iff eqv?	unspecified	#t
(eq? '#() '#())	unspecified	unspecified	#t
(eq? '#(1 2) '#(1 2))	#t iff eqv?	unspecified	#t
(eq? '() '())	#t	#t	#t
(eq? '(a) '(a))	unspecified	unspecified	#t
(eq? '(b) (cdr '(a b)))	unspecified	unspecified	#t
(eq? (cons 1 2) (cons 1 2))	#f	#f	#t
(let ((x '(a))) (eq? x x))	#t	#t	#t
(eq? (lambda () 'a) (lambda () 'b))	#f	#f	#f
(eq? (lambda (x) x) (lambda (x) x))	unspecified	unspecified	unspecified
(let ((p (lambda (x) x))) (eq? p p))	#t	#t	#t

Part VI

Misc: Value Binding

Impure FP: Imperative-style Side-effects

- (set! <variable> <expression>)
  <expression> is evaluated, and the resulting value is stored in the location to which <variable> is bound.

```
> (define x '(b c))              |  > (set! x '(b c))
> (define y x)                   |  > (eq? x y)
> (cadr x)                       |  #f
c                                |  > (equal? x y)
> (eq? x y)                      |  #t
#t                               |  > (set! y x)
> (set! x '(d e f))              |  > (eq? x y)
> (cadr x)                       |  #t
e                                |  > (set-cdr! x '(g h))
> (eq? x y)                      |  > y
#f                               |  (b g h)
```

Impure FP: Imperative-style Side-effects

- (set! <variable> <expression>)
  <expression> is evaluated, and the resulting value is stored in the location to which <variable> is bound.
- (set-car! <pair> <expression>)
  Stores <expression> in the car field of <pair>.

```
> (define x '(b c))           |  > (set! x '(b c))
> (define y x)                |  > (eq? x y)
> (cadr x)                    |  #f
c                             |  > (equal? x y)
> (eq? x y)                   |  #t
#t                            |  > (set! y x)
> (set! x '(d e f))           |  > (eq? x y)
> (cadr x)                    |  #t
e                             |  > (set-cdr! x '(g h))
> (eq? x y)                   |  > y
#f                            |  (b g h)
```

Impure FP: Imperative-style Side-effects

- (set! <variable> <expression>)

  <expression> is evaluated, and the resulting value is stored in the location to which <variable> is bound.
- (set-car! <pair> <expression>)

  Stores <expression> in the car field of <pair>.
- (set-cdr! <pair> <expression>)

  Stores <expression> in the cdr field of <pair>.

```
> (define x '(b c))               |  > (set! x '(b c))
> (define y x)                    |  > (eq? x y)
> (cadr x)                        |  #f
c                                 |  > (equal? x y)
> (eq? x y)                       |  #t
#t                                |  > (set! y x)
> (set! x '(d e f))               |  > (eq? x y)
> (cadr x)                        |  #t
e                                 |  > (set-cdr! x '(g h))
> (eq? x y)                       |  > y
#f                                |  (b g h)
```

Value Binding and Environment

- The phrase: "(**define** x '(b c))" is called a binding; it binds the variable x to a value '(b c).
- Can occur at top-level (global) or at the beginning of a lambda or let body (local).
- Don't re-define variables; think of them as aliases or constants. You can re-define existing variables at the top-level, for convenience. Whenever an identifier is defined, it's as if a new identifier is "created" — it has nothing whatever to do with any previously existing identifier of the same name.
- The phrase: "(**set!** x '(d e f))" rebinds the variable x to another value '(d e f).
- Don't use **set!** unless you are intentionally violating pure functional programming.
- Environment: the current set of ordered pairs (identifier, value) that are visible.

Environment: Example (using SML syntax)

env:

- val x = 17;

val x = 17 : int

x = 17

- val y = x;

val y = 17 : int

y = 17
x = 17

- val x = true;

val x = true : bool

x = true
y = 17
x = 17

- val z = x;

val z = true : bool

z = true
x = true
y = 17
x = 17

Alias and Side Effects

Alias: When a data object is visible through more than one name in a single referencing environment, each name is termed an alias.

- Examples: passed parameters by reference in a function, several pointers to the same object.
- Pitfall: programs are harder to understand.

Side Effects: An operation has side effects if it makes changes which persist after it returns.

- Examples : A function changes its parameters or modifies global variables (through assignments); printouts.
- Pitfall : programs are harder to understand, evaluation order of expressions becomes important.

Alias and Side Effects: Example

```
int x = 2, y = 5;
int Bad(int m) { return x+=m; }

void Swap(int* a, int* b)
{
    int temp = *a; *a = *b; *b = temp;
    x = 4;
}

int main()
{
    int* z = &x;
    int k = x * Bad(7) + x;

    printf("k = %d\n", k);
    Swap(&x, &y);
    printf("(x,y) = (%d,%d)\n", x, y);
}
```

Part VII

Summary

Summary

- $\surd$ A task is achieved through applications of functions.
- $\surd$ No pointers!
- $\surd$ No coercion!
- $\surd$ No side-effects!
- $\surd$ Assignment is replaced by value binding.
- $\surd$ Implicit type inference.
- $\surd$ Implicit memory management: Objects are allocated as needed, and deallocated when they become inaccessible.
- $\surd$ Pattern matching $\Rightarrow$ program by examples.
- $\surd$ Allow recursive definition of polymorphic datatypes.
- $\surd$ Simple exception handling.

Summary: FP vs. IP

IP:	Since IP languages are based on the von Neumann architecture, programmers must deal with the management of variables, assignment of values to them, memory locations, and sometimes even memory allocations. • Adv: efficient computation • Disadv: laborious construction of programs
FP:	Do not manipulate memory directly; no variables, no assignments. Instead they work on values that are independent of an underlying machine. • Adv: compact language, simple syntax, higher level of programming • Disadv: efficiency is sacrificed

Summary: FP vs. IP ..

IP:	Due to aliases and side effects, the effects of a subprogram or a block cannot be determined in isolation from the entire program.
FP:	Since they only manipulate values, there are no aliases nor side effects.
IP:	Explicit memory management.
FP:	Storage is allocated as necessary; and storage that becomes inaccessible is automatically deallocated and reclaimed during garbage collection.
IP:	The power comes from mimicking operations on the underlying computer architecture with assignments, loops, and jumps.
FP:	The power comes from recursion and treating functions as “first-class” values.