CS 314 Principles of Programming Languages

Lecture 16: Functional Programming

Zheng (Eddy) Zhang

Rutgers University

April 2, 2018
Functional:

Composition of operations on data.
- No named memory locations
- Value binding through parameter passing
- Key operations: *Function application* and *Function abstraction*
- Basis in *lambda calculus*
Review: Pure Functional Languages

Fundamental concept: application of (mathematical) functions to values

1. Referential transparency: the value of a function application is independent of the context in which it occurs
   - value of $foo(a, b, c)$ depends only on the values of $foo$, $a$, $b$ and $c$
   - it does not depend on the global state of the computation
   $\Rightarrow$ all vars in function must be local (or parameters)

2. The concept of assignment is NOT part of function programming
   - no explicit assignment statements
   - variables bound to values only through the association of actual parameters to formal parameters in function calls
   - thus no need to consider global states
3. **Control flow is governed by function calls and conditional expressions**

⇒ no loop

⇒ recursion is widely used

4. **All storage management is implicit**

• needs garbage collection

5. **Functions are** *First Class Values*

• can be returned from a subroutine
• can be passed as a parameter
• can be bound to a variable
Review: Pure Functional Languages

A program includes:

1. A set of function definitions
2. An expression to be evaluated

E.g. in scheme,

```scheme
> (define length
    (lambda (x)
      (if (null? x)
          0
          (+ 1 (length (rest x))))))

> (length '(A LIST OF 7 THINGS))
5
```
LISP

• Functional language developed by John McCarthy in the mid 50’s
• Semantics based on *Lambda Calculus*
• All functions operate on lists or symbols called: “S-expression”
• Only five basic functions:
  list functions *con, car, cdr, equal, atom*,
  & one conditional construct: *cond*
• Useful for list-processing (LISP) applications
• Program and data have the same syntactic form “S-expression”
• Originally used in Artificial Intelligence
SCHEME

• Developed in 1975 by Gerald J. Sussman and Guy L. Steele
• A dialect of LISP
• Simple syntax, small language
• Closer to initial semantics of LISP as compared to COMMON LISP
• Provide basic list processing tools
• Allows functions to be first class objects
• Expressions are written in prefix, parenthesized form

\[(function \ arg_1 \ arg_2 \ldots \ arg_n)\]

\[(+ \ 4 \ 5)\]
\[(+ \ (* \ 3 \ 4) \ (- \ 5 \ 3))\]

• Operational semantics:

In order to evaluate an expression
1. Evaluate function to a function value
2. Evaluate each \( \text{arg}_i \) in order to obtain its value
3. Apply function value to these values
S-expression

S-expression ::= Atom | ( S-expression ) | S-expression S-expression
Atom ::= Name | Number | #t | #f | ε

#t
()
(a b c)
(a (b c) d)
((a b c) (d e (f)))
(1 (b) 2)

Lists have nested structure!
Lists in Scheme

The building blocks for lists are pairs or cons-cells.
Lists use the empty list “( )” as an “end-of-list” marker.

(a b)

(a (b c) (d e) )

(a . b)
Special (Primitive) Functions

- **eq?**: identity on names (atoms)
- **null?**: is list empty?
- **car**: select first element of the list
  (contents of address part of register)
- **cdr**: select rest of the list
  (contents of decrement part of register)
- **(cons element list)**: constructs lists by adding **element** to the front of **list**
- **quote or '**: produces constants

*Do not evaluate the ' the content after '. Treat them as list of literals.*
Special (Primitive) Functions

• '() is an empty list

• (car ' (a b c)) = a

• (car ' ((a) b (c d))) = ( a )

• (cdr ' (a b c)) = (b c)

• (cdr ' ((a) b (c d))) = (b (c d))
Special (Primitive) Functions

• **car** and **cdr** can break up any list:

  \[(\text{car} \ (\text{cdr} \ (\text{cdr} \ '((a) \ b \ (c \ d)))))) = \ (c \ d)\]

  \[(\text{cdr} \ '((a) \ b \ (c \ d))) = \ (b \ (c \ d))\]

• **cons** can construct any list:

  \[(\text{cons} \ 'a \ '() \ ) = \ (a)\]

  \[(\text{cons} \ 'd \ '(e)) = \ (d \ e)\]

  \[(\text{cons} \ '((a \ b) \ '(c \ d))) = \ ((a \ b) \ c \ d)\]

  \[(\text{cons} \ '((a \ b \ c) \ '((a) \ b))) = \ ((a \ b \ c) \ (a) \ b)\]
Other Functions

- +, -, *, / numeric operators, e.g.,
  - \((+\ 5\ 3) = 8,\ (-\ 5\ 3) = 2\)
  - \((\ast\ 5\ 3) = 15,\ (/\ 5\ 3) = 1.6666666\)
- = <> comparisons for numbers
- Explicit type determination and type functions:
  \Rightarrow All return Boolean values: #f and #t
  - (number? 5) evaluates to #t
  - (zero? 0) evaluates to #t
  - (symbol? 'sam) evaluates to #t
  - (list? '(a b)) evaluates to #t
  - (null? '( )) evaluates to #t

Note: SCHEME is a strongly typed language.
Other Functions

• (number? 'sam) evaluates to #f
• (null? '(a) ) evaluates to #f
• (zero? (- 3 3)) evaluates to #t
• (zero? '(- 3 3)) ⇒ type error
• (list? (+ 3 4)) evaluates to #f
• (list? '(+ 3 4)) evaluates to #t
The Scheme interpreters on the ilab machines are called *mzscheme*, *racket*, and *DrRacket*. “drracket” is an interactive environment, the others are command-line based.

For example: Type racket, and you are in the READ-EVAL PRINT loop. Use “*Control D*” to exit the interpreter.

```
zz124@vi ~> racket
Welcome to Racket v6.5.
> (define length
(lambda (x)
  (if (null? x)
      0
      (+ 1 (length (rest x))))))
> (length '(A LIST OF 6 THINGS))
5
> }
```
The Scheme interpreters on the ilab machines are called *mzscheme*, *racket*, and *drracket*. “drracket” is an interactive environment, the others are command-line based.

**READ**: Read input from user:
A function application

**EVAL**: Evaluate input:
(f arg₁ arg₂ … argₙ)
1. evaluate function to a function value
2. evaluate each argᵢ in order to obtain its value
3. apply function value to these values

**PRINT**: Print resulting value:
The result of function application

You can write your Scheme program in file `<name>.rkts` and then read it into the Scheme interpreter by saying at the interpreter prompt:

```
(load "<name>.rkts")
```
1. Read the function application
   (cons 'a (cons 'b '(c d)))
2. Evaluate cons to obtain a function
3. Evaluate 'a to obtain a itself
4. Evaluate (cons 'b '(c d))
   (i) Evaluate cons to obtain a function
   (ii) Evaluate 'b to obtain b itself
   (iii) Evaluate '(c d) to obtain (c d) itself
   (iv) Apply cons function to b and (c d) to obtain (b c d)
5. Apply cons function to 'a and (b c d) to obtain (a b c d)
6. Print the result of the application: (a b c d)
Defining Global Variables

The `define` constructs extends the current interpreter environment by the new defined (name, value) association

```lisp
> (define foo '(a b c))
#<unspecified>

> (define bar '(d e f))
#<unspecified>

> (append foo bar)
(a b c d e f)

> (cons foo bar)
((a b c) d e f)

> (cons 'foo bar)
(foo d e f)
```
Defining Scheme Functions

(define <fcn-name>   (lambda (<fcn-params>) <expression>) )

Example: Given function pair? (true for non-empty lists, false o/w) and function not (boolean negation):

Evaluating (atom? '(a)):
1. Obtain function value for atom?
2. Evaluate '(a) obtaining (a)
3. Evaluate (not (pair? object))
   a) Obtain function value for not
   b) Evaluate (pair? object)
      i. Obtain function value for pair?
      ii. Evaluate object obtaining (a)
      iii. Evaluates to #t
   c) Evaluates to #f
4. Evaluates to #f
Conditional Execution: if

(if <condition> <result1> <result2>)

1. Evaluate <condition>
2. If the result is a “true value” (i.e., anything but #f), then evaluate and return <result1>
3. Otherwise, evaluate and return <result2>

(define abs-val
  (lambda (x)
    (if (>= x 0) x (- x))
  )
)

(define rest-if-first
  (lambda (e l)
    (if (eq? e (car l)) (cdr l) '())
  )
)
Conditional Execution: cond

(cond (<condition1> <result1>))
  (<condition2> <result2>)
...
  (<conditionN> <resultN>))
(else <else-result>) ; optional else clause

1. Evaluate conditions in order until obtaining one that returns a #t value
2. Evaluate and return the corresponding result
3. If none of the conditions returns a true value, evaluate and return <else-result>
(define abs-val
  (lambda (x)
    (cond ((>= x 0) x)
          (else (- x)))))

(define rest-if-first
  (lambda (e l)
    (cond ((null? l) '())
          ((eq? e (car l)) (cdr l))
          (else '()))))
Recursive Scheme Functions: abs-List

(define abs-list
  (lambda (l)
    (if (null? l) '()
        (cons (abs (car l)) (abs-list (cdr l)))
    )
  )
)

• (abs-list '((1 -2 -3 4 0)) ⇒ (1 2 3 4 0)
• (abs-list '()) ⇒ ()
Recursive Scheme Functions: Append

- \((\text{append } '(1 2) '(3 4 5)) \Rightarrow (1 2 3 4 5)\)
- \((\text{append } '(1 2) '(3 (4) 5)) \Rightarrow (1 2 3 (4) 5)\)
- \((\text{append } '() '(1 4 5)) \Rightarrow (1 4 5)\)
- \((\text{append } '(1 4 5) '()) \Rightarrow (1 4 5)\)
- \((\text{append } '() '()) \Rightarrow ()\)

\[(\text{define append})\]
\[\text{(lambda} (x y)\]
\[\quad (\text{cond} ((\text{null?} \ x) \ y)\]
\[\quad \quad ((\text{null?} \ y) \ x)\]
\[\quad \quad (\text{else} (\text{cons} \ (\text{car} \ x) \ (\text{append} \ (\text{cdr} \ x) \ y))))\]
\[\)]
\[\)]
Next Lecture

Things to do: