Class Information

INFORMATION and REMINDERS

• Project 1 due date is Monday, March 27, at 11:59pm. Late policy: 20% grade reduction for every starting 24 hours. No credit if project is submitted more than four days late.

• Homework 5 has been posted. Due on Wednesday, March 29.

• Project 1: Your dead code eliminator should catch cases such as:

   a=1; b=a; a=2; #a.

   Please run the sample optimizer to see the result.
Functional Programming

Pure Functional Languages

Scott: Chapter 11

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 $f(a, b, c)$ depends only on the values of $f, a, b$ and $c$
   
   • It does not depend on the global state of computation
   
   $\Rightarrow$ all vars in function must be local (or parameters)
2. The concept of assignment is **not** part of functional programming

- no explicit assignment statements
- variables bound to values only through the association of actual parameters to formal parameters in function calls
- function calls have no side effects
- thus no need to consider global state

3. Control flow is governed by function calls and conditional expressions

⇒ no iteration
⇒ recursion is widely used
Pure Functional Languages

4. All storage management is implicit
   • needs garbage collection

5. Functions are *First Class Values*
   • Can be returned as the value of an expression
   • Can be passed as an argument
   • Can be put in a data structure as a value
   • (Unnamed) functions exist as values
Pure Functional Languages

A program includes:

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

E.g. in Scheme:

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

> (length '(A LIST OF 5 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-expressions”)
- Only five basic functions: list functions `cons`, `car`, `cdr`, `equal`, `atom` and one conditional construct: `cond`
- Useful for list-processing applications
- Programs and data have the same syntactic form: S-expressions
- Used in Artificial Intelligence
Lambda calculus

- formalism for studying ways in which functions can be formed, combined, and used for computation

- computation is defined as rewriting rules (operational semantics) ⇒ \( \beta \) reduction

- the syntactic notion of computation was developed first; a mathematical semantics followed much later

Examples:

\[
\begin{align*}
\text{f(x)} &= \text{x+2} & (\lambda x.\text{x+2}) & \text{different notation} \\
((\lambda x.\text{x+2}) \ 1) &= 1+2 = 3 & \text{function application and substitution} \\
((\lambda x.\text{x}) \ (\lambda y.y)) &= & \text{arguments and returned “values” can be functions} \\
(\lambda x.\text{xx}) &= & \text{untyped lambda calculus} \\
f(x) &= x(x)
\end{align*}
\]
SCHEME

- Developed in 1975 by G. Sussman and G. Steele
- A version of LISP
- Simple syntax, small language
- Closer to initial semantics of LISP as compared to COMMON LISP
- Provides basic list processing tools
- Allows functions to be first class objects
• Expressions are written in prefix, parenthesized form
  – (function arg_1 arg_2 ... arg_n)
  – (+ 4 5)
  – (+ (* 3 4 5) (- 5 3))

• Operational semantics: In order to evaluate an expression:
  1. evaluate function to a function value
  2. evaluate each arg_i in order to obtain its value
  3. apply the function value to these values
S-expressions

S-expression ::= Atom | ‘(’ S-expression ‘)’
Atom ::= Name | Number | #t | #f

#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.

Note: *(a.b)* is not a list!
Special (Primitive) Functions

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

• `()` is the empty list

• `(car '(a b c)) =

• `(car '((a) b (c d))) =

• `(cdr '(a b c)) =

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

• **car** and **cdr** can break up any list:
  
  – \((\text{car} \ (\text{cdr} \ (\text{cdr} \ '((a) \ b \ (c \ d)))))\) =

  – \((\text{caddr} \ '((a) \ b \ (c \ d)))\)

• **cons** can construct any list:
  
  – \((\text{cons} \ 'a \ '())\) =

  – \((\text{cons} \ 'd \ 'e)) =

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

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

• + - * / numeric operators, e.g.,
  (+ 5 3) = 8, (- 5 3) = 2
  (* 5 3) = 15, (/ 5 3) = 1.6666666

• = <> comparison operators for numbers

• Explicit type determination and test functions:
  ⇒ 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`
READ-EVAL-PRINT Loop

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 mzscheme, and you are in the READ-EVAL-PRINT loop. Use Control D to exit the interpreter.

READ: Read input from user: a function application

EVAL: Evaluate input: 

(f arg₁ arg₂ ... argₙ)

1. evaluate f to obtain a function
2. evaluate each argᵢ to obtain a value
3. apply function to argument values

PRINT: Print resulting value: the result of the function application

You can write your Scheme program in file <name>.ss and then read it into the Scheme interpreter by saying at the interpreter prompt: (load "<name>.ss")
READ-EVAL-PRINT Loop Example

> (cons 'a (cons 'b '(c d)))
(a b c d)

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)):
   (a) Evaluate cons to obtain a function
   (b) Evaluate 'b to obtain b itself
   (c) Evaluate '(c d) to obtain (c d) itself
   (d) Apply the cons function to b and (c d) to obtain (b c d)

5. Apply the 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)
Next Lecture

More on Scheme

Please see our website for an online Scheme textbook