Functional Programming Languages

11.1-11.3

Functional languages

- Introduction
- Functional programming

Reference manual of common LISP

http://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html

Roots of functional programming

- Lambda calculus (1930s)
  - A computing model equivalent to *Turing Machine*
- Artificial Intelligence (1950s)
  - In one of the very first AI program by Simon and Newell, lists are heavily used
- Inventor of LISP – a functional language: John McCarthy
- Like Fortran, *LISP* (list processing) became the source of inspiration of language designers

- Lisp, Scheme, FP, ML, Miranda, Haskel

LISP

- LISP is the first language having
  - Conditionals - if-then-else constructs
  - A function type - functions are first-class objects (*orthogonality*)
  - Recursion
  - Typed values rather than typed variables
  - Garbage collection
  - Programs made entirely of functional expressions that return values
  - A symbol type
  - Built-in extensibility
  - The whole language always available -- programs can construct and execute other programs on the fly
  - Most of these features have gradually been added to other languages

LISP: applications

- LISP is effective for many AI problems
  - To develop (SHOP) – a tool for planning problems
    - Using LISP : a few weeks
    - Using Java: a few months
- LISP outside AI
  - AutoCAD
  - Emacs
  - …
  - Yahoo! Merchant Solutions - e-commerce software

- Eearly functional languages: dynamically scoped
- Scheme and Common Lisp statically scoped
- Pure Lisp is purely functional; all other Lisps have imperative features

Functional programming

- Informal language introduction
  - Application (in manipulating knowledge)
  - Implementation (constructs are based on lambda calculus, instead of a machine language)
- Syntax
- Semantics / execution (interpretation)

Functional programming – application

- Knowledge
  - (above redblock blueblock)
  - (above x y) and (above y z) implies (above x z)
- A list is used to represent objects and their relations
- We need manipulation of the lists

Simple programs

- A functional program is an expression that is either
  - an "object" as simple as a number

    1
  - Or, a list (whose element can be a simple object or a list)

    (+ 2 3)

    (> 2 1)

    (+ (- 2 3) 3)
- Every expression has a value (see next slides)

Numbers and functions on numbers

- Consider numbers (integer) first
  - Numbers: 1, 2, …
  - Functions on these numbers: +, -, …
  - Relations on these numbers: >, <, =, …
- Every expression has a value

```
1    ; => 1
(+ 2 3)            ;=> 5
(> 2 1)            :=> t
```

- Logical connectives (of expressions)
  - and

    ```
    (and (> x 3) (< x 5))
    ```
  - or

    ```
    (or (> x 3) (< x 3))
    ```
  - not

    ```
    (not (> 3 5))                    ; => t
    ```

- Condition

(if *test expr1 expr2*) ; if *test* is t, expr1, otherwise, expr2

(cond ((*test1 expr11 … expr1n)*
(*test2 expr21 … expr2m*)
…)

Find the first test that is evaluated to be t, evaluate the following expressions and value of the last expression is the value of the cond expression.

User defined function

- User defined function

```
(defun square (x)
    (* x x ))
```

- Exercise

define a function whose input is x output is x^3

Sequencing

- Sequencing

  (progn *expr1 … exprn*)

  evaluate the expressions in order and return the value of the *exprn*

  e.g.,

  ```
  (progn (+ 2 3) (- 1 1))                  ; => 0
  ```

Variables

- Variables
  - Define a function
    - input x y z
    - return the minimum of them

```
(defun min (x y z)
  ( let (m (if (> x y) y x))
    (if (> m z) z m) ))
```

Variables - continued

– Local variables:

```
( let ( (x1 value1)
        …
          (xn valuen))
  expr1 … exprn)
```

each variable xi is initially set to valuei and these variables can be used in expr1 to exprn

– Global variables: (setf *var value*) *var* will be global variable initialized by *value.*

Character and string

- Characters
  - Character g: #\g

- (equal *character character*)

Input/output/file

- Input from keyboard
  - (**read**)
  - To associate the input value to a variable
    - `(let ((x (read)) ……)`
    - `(setf x (read)) // global, try to avoid`
- Output to the console
  - `(format t "Hello World")`
  - Format directives: **~$** (float) **~d** (integer) **~A** (any type) **~S** (any type)
  - E.g., `(format t “This is ~A” “an example”)`
- Input from a file
  - `(open "a.txt")`
  - `(setf instream (open “a.txt”))`
  - `(read-char instream nil)`

- Input output

```
(progn
  (format t "Enter a number: ")            ; t: standard output
  (let (x (read))                          ; read a number
      (if (= (mod x 2) 0)
            (format t "Even number")
       (format t "odd number"))))
```

Forms (types of expressions)

- Forms
  - Self evaluating form: numbers, characters …
  - Special forms
    - (defun …), (if …), (cond …), ……
  - Function call

    (+ 2 3)

    (square 2)
  - Variable
  - Macros

- Exercise
  - Define a function f(x,y) = x*x * y*y

Write recursive functions

- How to produce an algorithm to solve a problem
  - Define the input and output of the algorithm for this problem
  - Design the algo by decomposing the problem
- Exercises
  - Define the factorial function
    - Recursive function
  - Define Fibonacci function (1, 1, 2, 3, 5, 8, …)
    - Recursive function
  - Hanoi tower

List type

- In addition to ordinary data type like integer etc, we have ***symbols*** (not a string type)

  ‘above ; => above

  “above” ; => “above” is the same as above

- List type
  - E.g., to represent a list of numbers we use

    ‘(1 2 3 4 5)

    ‘(above redblock blueblock)

Functions (operations) on list

- list – Produce a list

```
(list 1 2 3 4)            ; => (1, 2, 3, 4)
(list `(+ 1 2) (+ 1 2))
```

- car – The first element of a list
  - input: a list; output: the first element of the list

```
(car `(1 2 3 4))          ; => 1
```

- cdr – The rest of a list
  - input: a list lis; output: a list same as lis except without first element from lis

```
(cdr `(1 2 3 4))          ; => (2 3 4)
```

- cons – Build a new list from an element and a list

```
(cons `a `(a b c))        ; => (a a b c)
```

- listp – returns t if its argument is a list

```
(listp 27)          ; => nil
(listp '(a b))      ; => t
```

- null – returns t if its argument is an empty list

```
(null nil)          ; => t
(null '(a))         ; => nil
```

- Example:
  - find the minimal value of a list
  - append two lists

```lisp
(defun min (list)
  (if (null list)
      nil
      (if (null (cdr list))
         (car x)
         (let ((m min((cdr x))))
           (if (> (car x) m)
               (car x)
               m))))))
```

```lisp
; append two lists
(defun append(lst1 lst2)
   (if (null lst1)
       lst2
      (cons (car lst1)
               (append
                   (cdr lst1)
                   lst2))))
```

The value of variables

- The value of arguments of a function **never** changes
  - > (setf lst ‘(a b c))
  - > (remove ‘a lst)   ;lst is still (a b c)
- Normally the value of a variable *does not* change. (Exception setf)

Function as first class object

- Given a, b, c, return a function ax^2 + bx + c
  – (note some lisp dialects has dynamic scope)

```
(defun bipoly (a b c)
  (lambda (x)                ; a function
     (+
      (* a x x)
      (* b x)
          c)))
(apply (bipoly 1 1 1) '(1))
```

Types

- Every value has a type

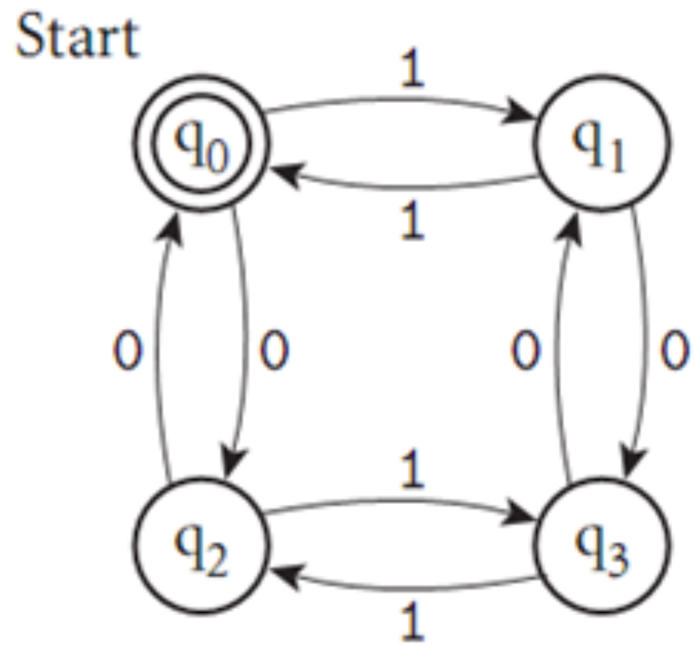
    (type-of 27)

    (type-of ‘a)   ; => symbol

- Common Lisp is a typed language! (done by type inferences we learned earlier)

Case study

- Mimic the execution of an automata
  - Represent an automata
    - (*startState transitionFunc -tion* (*final states*))
  - Algorithm to mimic the execution of an automata
  - Go through scanDFA-example



- Represent the automata

(q0 (q0) transition)

Note here we introduce two methods to define a function: set theory based, and input/output based

For the later, transition function is defined as

Input: a state *p*, and a symbol *s*

Output: state *q* which is next state of *p* given *s*

```
(defun transition (state symbol)
  …
)
```

- Represent the automata – an alternative way

Define the automata as a function: dfa

Input: ‘start, ‘final, or ‘transition (a component of the automata)

Output: initial state if input is ‘start

list of final states if input is ‘final

transition function if input is ‘transition

```
(defun dfa (component)
…
)
```

- Write the function to check if a string (as a list) is acceptable by an automata
  - A first version of a function to finish this is
    - Function name: scan
      - Input: automata dfa (a function) and a string i (as a list)
      - Output: t if i is accepted by dfa
        nil otherwise
    - When we try to decompose string i, we found that we need one more parameter for scan
      - Function name: scan
        - » Input: automata dfa, a string i and a state s
        - » Output: t if i is accepted by dfa with *current* state *s*
          nil otherwise.

- Write the function to check if a string (as a list) is acceptable by an automata (continued)

Here is the pseudo-code

```
(defun scan (dfa curState input)
  if input is empty
   if curState is in (dfa ‘final)
             t
    else nil
  else
      let s be (apply (dfa ‘transition) ‘(car input)) // next state
         (scan dfa s (cdr input)) // scan the rest of the input
)
```

Summary

- Basics of functional programming
  - Simple expressions
  - Condition
  - Define a function / application a function to expressions
  - Recursive programming
  - Data structure: list

Appendix

- Project 2 (scanner in lisp program – future)
  - Go through the project description
  - To define scan which return a state where the farthest the dfa can go with the input
  - Open file whose handle is a golobal variable
    - (setf instream (open "test-data.txt"))
    - You can use instream anywhere in your program
  - Character constant
    - #/g   ;=> letter g
  - Demo the execution of a program
  - Print to the display
    - (format t “ERROR”)
    - (format “~S” ‘(1 2 3))

Example: a function returns the list all tokens. *Here we assume no peeking is needed.*

```
(defun tokens (dfa)
   (if (equal (read-char nil instream) nil)
       ; the end of file
      nil ; return empty list or $$ in the future
     (let ((state (scan dfa (dfa 'start))))
       (if (isFinal dfa state) ; the state returned by
                               ; scan is final
         (cons (token state) (tokens dfa)) ; get the token
       '(error)))))
```

Atom and lists

- Every list object is either an *atom* or *object*
- Atom example – number, variable (identifier)
    - numbers: 235.4 2e10 #x16 2/3
    - variables: foo
    - constants: pi t
    - strings, chars: "Hello!"
- List example (a1, …, an)
    - (a b c d) (a (b c) d) [note orthogonality]
    - () or nil – empty list,

Functions

+, *, /	plus, times, divide	(/ (* 2 3 4) (+ 3 1)) => 6
-	minus	(- (- 3) 2) => -5
sqrt	square root	(sqrt 9) => 3
exp, expt	exponentiation	(exp 2) => e^2, (expt 3 4) => 81 [3^4]
log	logarithm	(log x) => ln x
min, max	minimum, maximum	(min -1 2 -3 4 -5 6) => -5
abs, round	absolute val, round	(abs (round -2.4)) => 2
truncate	integer part	(truncate 3.2) => 3
mod	remainder	(mod 5.6 5) => 0.6
sin, ...	trig funcs (radians)	(sin (/ pi 2) => 1.0

Lisp

- A lisp program is organized as forms and functions

Forms

- A form is
  - A “simple object”: number, string etc
  - A variable
  - A list
    - Special form: first element of the list is
      - One of the keywords: if, let, quote, …
      - The list will be processed by a special code corresponding to the first element
    - Macro form: the first element of the list is a macro
      - A user may define a macro by defmacro
    - Function call: if the list is not of special or macro form

Evaluation of a form

- Evaluation of “simple objects”
  - Numbers etc
  - Variable
- Evaluation of lists
  - Special form / macros: (skip)
    - Evaluation of defun form: the function name becomes a global name
  - Evaluation of a function call
    - (+ 1 (* 3 4))
    - The first element: function name
    - Each element of the rest of the list will be evaluated and the values obtained are the arguments of the function

Execution of a program

- A LISP interpreter
- It accepts any form and evaluate it
- Top level forms
  - Define globally named functions, macros, …

Special forms

- (if test expr1 [expr2])
  - if test is non-NIL then return expr1 else return expr2 (or NIL)
- (cond (test1 expr11)

  (test2 expr21)

  ......)

```
(cond ((evenp a) a)      ;if a is even return a
      ((> a 7) (/ a 2))  ;else if a is bigger than 7 return a/2
      ((< a 5) (- a 1))  ;else if a is smaller than 5 return a-1
      (t 17))            ;else return 17
```

LISP: problems

- Problems
  - difficult (but not impossible!) to implement efficiently on von Neumann machines
    - lots of copying of data through parameters
    - (apparent) need to create a whole new array in order to change one element
    - heavy use of pointers (space/time and locality problem)
    - frequent procedure calls
    - heavy space use for recursion
    - requires garbage collection
    - requires a different mode of thinking by the programmer
    - difficult to integrate I/O into purely functional model