CS342: Organization of Prog. Languages

Topic 3: [Language] An Introduction to Scheme

- The Lisp family of languages
- Some general characteristics of Lisps
- Scheme:
- An implementation
- Basic syntax (informal)
- Program structure
- Basic types
- Pairs as lists
- Recursive functions on lists
- Quoted data
- `begin` for multiple expressions

- Parameters as variables
- Local bindings, `let`
- Nested functions
- Creating functions dynamically; Closures
- N-ary functions
- Lexical binding forms
- Loops

© 2000-2007 Stephen M. Watt.

The Lisp Family of Languages

- Originally a language for working with dynamic data structures.
- The primitive allocated unit is the "cons cell" (or "pair") from which linked lists, binary trees etc. are built.
- Many dialects over the 1960s and 1970s (MacLisp, InterLisp, ...)
- Simplified uniform/orthogonal version Scheme later in 1970s.
- Lisps were and continue to be used
  - in artificial intelligence
  - in symbolic mathematical computation
  - as an extension language to software systems (Emacs, Autocad, ...)
  - ...
- Standardization with Common Lisp in the 1980s.
- CLOS — Common Lisp Object System

General Characteristics of Lisps

- Parenthesized syntax
- Easy allocation and garbage collection
- All have pairs (cons cells), most have arrays, strings, etc.
- Use various scoping mechanisms
- All support functional programming style.

- We shall learn the basic constructs of Scheme, and later review them from a more formal point of view.

A Scheme Implementation

- Two programs are available on Gaul to use for the Scheme programs on the assignment:
- mzscheme — a command-line interface to an interactive Scheme
- drscheme — a GUI with editor and execution windows
- You can download these for use on your own PC (Google PLT Scheme or drscheme)
- NOTE: if you use drscheme, use the menus to set the language to “PLT, Textual”

Scheme — Basic syntax (informal)

- White space and comments,
  e.g. `; until end of line`
- Simple elements
  - Boolean literals, e.g. `#t #f`
  - Numeric literals, e.g. `3  3/2 3+2i  3.0`
  - Character literals, e.g. `#\x  #\space`
  - String literals, e.g. `"hello"`
  - Identifier, e.g. `hello  get-lost  boo!  huh?  +`
- Compound elements
  - List, e.g. `(A 1 9)`
  - Vector, e.g. `#(A 1 9)`
  - Abbreviation, e.g. `'H     ` `` `(A 1 9) ``

Program Structure
Expressions

- Expressions are given in list syntax.
- Operator is the first element, operands follow.
- E.g.

```
(+ 1 2)   (+ 1 2 3 4)  (+ 3)   (+)

(- 4 5)   (- 5)



(gcd 12 16)

(call-with-current-continuation F)



(+ (* 3 n) 1)          ; 3*n + 1
```

Program Structure
Definition and Update

- The special form (`define` *id expr*) introduces a new name with an initial value.

```
(define n 4)
(define s "hello")
```

- The special form (`set!` *id expr*) updates the value associated with a name.

```
(set! x 7)
(set! x (+ x 1))
```

- Names are visible in certain regions of the program according to *scope rules*, discussed later.

Program Structure
Conditional Evaluation

- `#f` is false. Everything else counts as a "true" value.
- If test.

  (`if` *test* $expr_1$)
  (`if` *test* $expr_1$ $expr_2$)

- Cond test.

```
(cond (test1 expr ...  )
      (test2 expr ...  )
      ...
      ; optional
      (else expr ...)  )
```

- Cond `=>`

```
(cond (test => fn-expr)
      ...)
```

Program Structure
Functions

- The special form `(lambda` *param-list expr expr* `..)` creates a function.

```
(lambda (n m) (write "Hello!)  (+ n m 3))
```

- A function value may be saved in a variable or be used directly (just as any other value).

```
(define mycalc (lambda (n) (+ (* 3 n) 1)))
(mycalc 14)

(    (lambda (n) (+ (* 3 n) 1))    14    )
```

Basic types

- Data values belong to a fixed set of built-in types.
- A boolean valued function ("predicate") is available to test for values in each type.

```
boolean?        #t #f
number?         3  3/2 3+2i  3.0 #b10101   #i54
char?           #\x  #\space
string?         "hello"
symbol?         'hello
null?           '()
pair?           '(2 . 3)
vector?         #(3 4 5)
procedure?      (lambda vlist ...)
port?
```

Pairs as Lists

```
;;; A pair is created with the cons function

(define mypair (cons 3 4))


;;; The parts are accessed by the functions car and cdr

(car mypair)   ; 3
(cdr mypair)   ; 4


;;; The ``null'' value can be given as  '()

'()


;;; From these one can make binary trees

(cons (cons 1 2) (cons 3 4))


;; By convention a ``list'' is a binary tree
;; with the elements in successive cars
;; and the tails in successive cdrs.

(define mylist (cons 1 (cons 2 (cons 3 (cons 4 '())))))
```

Recursive Functions on Lists

- It is most natural to define recursive functions to operate on lists.
- **Pattern:** Use `null?` to *terminate*, and `cdr` in the *recursive call*.

```
;; Add up the elements of a list
(define addlist (lambda (l)
    (if (null? l)
        0
        (+ (car l) (addlist (cdr l))) ) ))

;; Print out the elements of a list
(define write-list (lambda (l)
    (cond ( (not (null? l))
              (write (car l))
              (write-list (cdr l)) )) ))

;; Interleave lists
(define mix-lists (lambda (la lb)
    (cond ((null? la) lb)
          ((null? lb) la)
          (else
             (cons (car la)
               (cons (car lb)
                 (mix-lists (cdr la) (cdr lb)) ) ) ) ) ))
```

Quoted Data

- The syntax `(quote X)` means
  "X is data, not an expression to evaluate"

```
(gcd 12 16)             ; computes a number

(quote (gcd 12 16))     ; gives a list of 3 elements:
                        ; the symbol ``gcd'',
                        ; and the numbers 12 and 16.
```

- `'X` is an abbreviation for `(quote X)`

```
'(a . b)                  ; Like  (cons 'a 'b)

'(a . (b . (c . d)))      ; Same as  '(a b c . d)

'(a . (b . (c . ())))     ; Same as  '(a b c)
```

- Some pieces of syntax are "self-quoting",
  e.g. `#t` and 7.

Read the Revised^5 Report

- The details of the functions available on strings, numbers, lists, vectors, etc are given in report on the language.
- We will not go over the collection of functions in class: you do that on your own.

`begin` for multiple expressions

- Sometimes one wishes to evaluate multiple expressions where only one expression is allowed.
- The `begin` construct can be used in these circumstances.

```
(if (pair? l)
    (begin
       (set! la (car l))
       (f la) )
    (f 0) )
```

- The form `(begin expr1 expr2 ...)` is a short-hand for `((lambda () expr1 expr2 ...))`
- By making this a formal equivalence we need only specify how `lambda` behaves, and need no special rules for `begin`

Parameters as variables

- As in many other languages, parameters act as local variables within a function.

```
(define myfunction (lambda (n)
   (set! n (* 3 n))
   (+ n 1) ))
```

Local bindings, `let`

- This idea may be used to create as many local variables as needed.

```
BEGIN
    local a := a0, b := b0, c := c0;
    expr1; expr2; expr3;
END
```

  may be written in Scheme as:

```
(  (lambda (a b c)
        expr1 expr2 expr3)  a0 b0 c0  )
```

- The is so common, there is a special short-hand:

```
(let ((a a0) (b b0) (c c0))
   expr1 expr2 expr3 )
```

Nested functions

- Functions (`lambda` expressions) may be nested.

  Instead of

```
(define double (lambda (c) (+ c c)))

(define foo (lambda (a)  (double (+ a 1))))
```

  we could write

```
(define foo (lambda (a)
    ( (lambda (c) (+ c c))  (+ a 1) )))
```

Block Structure

- Scheme is block structured:

  An inner lambda exprssion function may access the parameters/variables of all the enclosing lambda expressions

```
(define foo (lambda (a b)
    ( (lambda (c) (+ c b))  (+ a 1) )))
```

- This is called “lexical scoping”
- Since `let` is defined in terms of ”lambda”,
  we have that `let`s can be nested.
- An inner parameter will make an outer parameter of the same name invisible.

Functions as values

- Recall that the result of evaluating a `lambda` expression is an ordinary value which happens to be a function.
- Function values may be passed as arguments...

```
(define f (lambda (n) (+ n 1)))
(define g (lambda (n) (* 3 n)))

(define compose-call (lambda (f1 f2 a)
   (f1 (f2 a)) ))

(compose-call f g 7)
```

- Function values may also be returned as values...

```
(define compose-fn (lambda (f1 f2)
   (lambda (n) (f1 (f2 n))) ))

(define myfun (compose-fn f g))

(myfun 7)
```

Closures

- The result of `compose-fn` is a new function which captures the values of `f1` and `f2`.
- This way of pairing a function with a set of bindings is a fundamental idea, and the resulting functions are known as “closures”.
- Closures are a direct consequence of the orthogonal combination of
  - lexical block structure
  - functions as first class values.

N-ary functions and `apply`

- The number of arguments a function takes is known as the function's *arity*.
  E.g. The arity of `cons` is 2. We say `cons` is *binary*.
  The arity of `cdr` is 1. We say `cdr` is *unary*.

- Some functions can take any number of arguments.

  These functions are called *n-ary*. `(+ 2 3 4 5 99)`

- Sometimes we have constructed a list of arguments and wish to call an N-ary function with these as arguments. To do this, we use `apply`.

```
(set! mylist '(10 12 30 14 50))
...
(apply + mylist)
```

- NOTE:

```
(+ 1 2 3)   ==   (apply + (list 1 2 3)) !=  (+ (list 1 2 3))
```

Defining N-ary functions

- So far all of our functions definitions have had a fixed arity, i.e. for some n

```
(lambda (v1 v2 ... vn) E1 E2 ... Em)
```

- If all the arguments are handled as one list, then it is possible to deal with an arbitrary number:

```
(lambda vlist E1 E2 ... Em)
```

- Then in the body of the function, the usual list operations may be used to access individual argumetns. E.g.

```
(define my-plus (lambda argle
    (if (null? argle)
        0
        (+ (car argle) (apply my-plus (cdr argle))) )))
```

Reprise: let

- `let` introduces a lexical scope with a number of new variables.
- The form is defined in terms of lambda.

```
(let ((v1 i1)  (v2 i2) ...)
   expr1
   expr2
   ... )
```

```
( (lambda (v1 v2 ...) expr1 expr2 ...)
  i1 i2 ... )
```

- Note the initial values are evaluated in the old, outer scope.

Initializations which depend on each other

- Question: How do we have some of the initializations $\mathtt{i}_j$ use the values of $\mathtt{v}_k$?
- Two cases
  - $\mathtt{i}_j$ depends only on $\mathtt{v}_k$ for $k < j$
  - $\mathtt{i}_j$ can depend on any $\mathtt{v}_k$.
- Note, in the first case we can evaluate in order. The second case allows mutually recursive definitions, e.g. for functions.

`let*` – Initialization in sequence

- $\mathtt{i}_j$ depends only on $\mathtt{v}_k$ for $k < j$
- Very common case
- Use

```
(let* ((v1 i1) (v2 i2) ...)
   expr1
   expr2
   ... )
```

- Equivalent to

```
(let ((v1 i1))
   (let* ((v2 i2) ...)
      expr1
      expr2
      ... )
```

- Example

```
(let* ((r2 (+ (* x x) (* y y)))
       (pi (* 3 (atan 1)))
       (area (* pi r2)))
   (write "A circle at the origin with the point ")
   (write (list x y))
   (write " on the circumference has area ")
   (write area))
```

`letrec` — recursive initializations

- $\mathtt{i}_j$ depends on any $\mathtt{v}_k$.
- Use:

```
(letrec ((v1 i1) (v2 i2) ...)
   expr1
   expr2
   ... )
```

- Each initial value can *refer* to to the $\mathtt{v}_k$, but should not *evaluate* them.
- In practical terms, this means the references to the $\mathtt{v}_k$ should be inside `lambda` expressions.

- Example:

```
(letrec
   ( (f (lambda (n) (if (= n 0) 1 (* n (g (- n 1))))))
     (g (lambda (n) (if (= n 0) 1 (* n (f (- n 1)))))) )
   (f 10) )
```

- Example:

```
(letrec
   ( (wt-nodes (lambda (a)
        (cond
           ((not (pair? a))  1)
           ((eq? '+ (car a)) (apply wt-plus  (cdr a)))
           ((eq? '* (car a)) (apply wt-times (cdr a)))
           (else (error "wt-nodes: Cannot handle " a))) ))
     (wt-plus (lambda args
         (apply + (map wt-nodes args)) ))
     (wt-times (lambda args
         (* 2 (apply + (map wt-nodes args))) )) )

     (wt-nodes '(* (+ a b c) (+ d e f))) )
```

- Equivalence:

```
(letrec ((v1 i1) (v2 i2) ...)
   expr1
   expr2
   ... )


(let ((v1 '())  (v2 '()) ...)
   (set! v1  i1)
   (set! v2  i2)
   ...
   expr1
   expr2
   ... )
```

Loops

- Like C/Java — initialize, test, step
- But the loop is an expression
  with a value.
- Use:

```
(do  ((v1 init1 step1) (v2 init2 step2) ...)
     (end-test  end-expr1 ... end-exprN)
   body-expr1
   body-expr2
   ... )
```

- The value of the loop is `end-exprN`.

- Example:

```
(define factorial (lambda (n)
     (do ((i 1 (+ 1 i)) (prod 1))
         ((> i n) prod)

         (set! prod (* i prod)) ) ))
```