More about Scheme

Applying Scheme: Programs as data

Natural representation for Scheme programs as S-expressions

Classic example is Scheme interpreter in Scheme

"metacircular evaluator"

```
-> (read-eval-print '(
       (+ 3 (* 4 5))
       (val abs (lambda (x) (if (< x 0) (- 0 x) x)))
       (abs -77)
       (cdr (quote (a b c)))))
23
77
(b c)
```

Metacircular evaluator, part 1

To begin, simple arithmetic and constants:

```
(define eval (exp)
   (if (number? exp) exp
       (apply-op (car exp)
                 (eval (cadr exp))
                 (eval (caddr exp)))))
(define apply-op (f x y)
        (if (= f '+) (+ x y)
        (if (= f '-) (- x y)
        (if (= f '*) (* x y)
        (if (= f '/) (/ x y) 'error!)))))
-> (eval '(+ 3 (* 4 5)))
```

```
23
```
; applications are always binary

Evaluation with variables

For variables, need environments:

variable/value pairs in association list `rho`

```
(define eval (exp rho)
  (if (number? exp) exp
  (if (symbol? exp) (assoc exp rho)
    (apply-op   (car exp)
        (eval (cadr exp) rho)
        (eval (caddr exp) rho)))))
```

Evaluating quotation, unary ops

```
(define eval (exp rho)
  ...
 (if (= (car exp) 'quote) (cadr exp)
 (if (= (length exp) 2)

(apply-unary-op (car exp) (eval (cadr exp) rho))
   (apply-binary-op (car exp) (eval ...))))))

-> (eval '(cons 3 (cons (+ i j) (quote ( ))))
     (mkassoc 'i 5 (mkassoc 'j 3 '( ))))
(3 8)
```

Evaluation with functions

For functions, pass extra association list that binds each function name to its formal parameters and body

```
-> (eval '(double 4) '( )
         '((double ((a) (+ a a)))))
8
```

Given fun bound to `l`, `(car l)` is formals, `(cadr l)` is body

```
(define eval (exp rho fundefs)
   ...
   (if (userfun? (car exp) fundefs)
       (apply-userfun (assoc (car exp) fundefs)
                      (evallist (cdr exp) rho fundefs)
                      fundefs)
   ... )
```

Applying user-defined functions

Note similarity with code in Impcore's evaluator:

```
(define apply-userfun (fundef args fundefs)
  (eval (cadr fundef) ; body of function
        (mkassoc* (car fundef) args '( ))
; local env
        fundefs))
```

where

```
(define mkassoc* (keys values al) ; like mkEnv
  (if (null?  keys) al
    (mkassoc* (cdr keys) (cdr values)
      (mkassoc (car keys) (car values) al))))
```

Top-level eval

Don't have `read` to get S-expression, so use quoting:

```
(r-e-p-loop* '(
      (+ 3 4)
      (define double (a) (+ a a))
      (double (car (quote (4 5)))))  )
(7 double 8) ; "results list"
```

where

```
(define r-e-p-loop* (inputs fundefs)
   ...
  (if (= (caar inputs) 'define) ; function definition
   (process-def (car inputs) (cdr inputs) fundefs)
   (process-exp (car inputs) (cdr inputs) fundefs))
  ...)
```

Elements of top-level eval

`process-exp` conses onto result list:

```
(define process-exp (e inputs fundefs)
  (cons (eval e '( ) fundefs) ; cons value of e
        (r-e-p-loop* inputs fundefs)))
```

`process-def` adds to `fundefs`:

```
(define process-def (e inputs fundefs)
  (cons (cadr e) ; cons function name to results
        (r-e-p-loop* inputs
            (mkassoc (cadr e) (cddr e) fundefs))))
```

Evaluator is "meta-circular" — can evaluate itself

Summary: μScheme in one slide

Abstract syntax: imperative core, `let`, `lambda`

Values: S-expressions, function closures

Environments: name stands for mutable location holding value

Evaluation rules: `lambda` captures environment

Initial basis: useful higher-order functions

Scheme as it really is

Conditional expressions avoid if-else parenthesis nightmare

```
(cond (e1 e1')     if e1 then e1'
      (e2 e2')     elsif e2 then e2'
        ⋮            ⋮
      (en en'))    elsif en then en'
```

Eval until one of the *guards* is true, then take corresponding expression. So `(if e1 e2 e3)` is really

```
(cond (e1 e2)
      (#t e3))
```

More real Scheme

Macros

- functions that manipulate S-expressions (at compile time)
- hygienic macros — name clashes impossible
- `let`, `and`, etc., implemented as macros

Even more real Scheme

Mutation

```
(set-car! '(a b c) 'd) => (d b c)
```

- modifies original list
- can create circular lists, sharing
- can avoid allocation (`cons`)

Garbage collection: reclaim and reuse unreachable cons cells

Real Scheme — continuations

Call with current continuation:

```
(call/cc (lambda (k) ... body... ) )
```

Continuation `k` is "what will be done with result of body"

E.g., can call `(k 1)` to return 1 "instantly"

activations in progress are abandoned

If `k` escapes, could return to body even after it finishes!

Like closures, need activation records on the heap

Building block for control flow:

- multithreading
- exception handling

Will revisit continuations later in this course (μProlog)

Real Scheme — tail calls

Imperative style list-reverse (in C):

```
List revimp(List l) {
  List r;
  for (r=NULL; l!=NULL; l = l->cdr)
    r = cons(l->car, r);
  return r;
}
```

Uses constant stack space

Tail calls, continued

Write functionally?

```
List revapp(List l, List r) {
  if (l!=NULL) then
    return revapp(l->cdr, cons(l->car, r));
  else
    return r;
}
List rev(List l) { return revapp(l, NULL); }
```

Uses stack space proportional to length of list `l`

call; call; call; .... call; return; return; return; ... ; return

But, the call is the last thing in the body. Try this:

call; return; call; return; ... call; return

Optimized tail calls

call; return; call; return; ... call; return

Idea: when call is recursive, implement return; call; by assignment and goto

Total result is

call; return

(Also OK even when tail call is *not* recursive)

True Scheme implementations *must* optimize tail calls

“proper tail recursion”

(Actually applies to all tail calls, whether recursive or not)

Why tail calls matter

Recursive function becomes the same as a loop

consumes *constant space*, and also faster!

Function call becomes

“goto with arguments” or

“assignment plus goto”

Tail recursion and factorial

Non-tail-recursive factorial

```
(set fact (lambda (n)
  (if (= n 1) 1
     (* n (fact (- n 1))))))
```

Requires `n` simultaneously active calls to `fact`

Tail-recursive version (with accumulating parameter):

```
(set fact (lambda (n)
  (letrec (
    (f (lambda (i product) ; product of 1..i − 1
          (if (> i n) product
             (f (+ i 1) (* i product)))))
    ) (f 1 1))))
```

At most one `f` activation at any time: compiles into a loop!

Real systems

Common Lisp:

- **Big systems ($>$ 1000 builtins)**

Scheme:

- **Big programming environments (DrScheme)**
- **Tiny embedded interpreters (libscheme)**
- **Everything in between**

Assessment

High-level data structures

- lists powerful for programming
- symbols give instant enumeration
- tables also powerful, can be efficiently implemented

Cheap, easy recursion

- a good fit for recursively defined types
- a natural for symbolic computing

Assessment

Safety and convenience of garbage collection

- **hard to overestimate**
- **historical performance highly variable**
  - **early systems embarrassing**
  - **modern systems outperform hand allocation**

Programs as data a remarkable paradigm

- **dynamic analysis**
- **dynamic construction (e.g., tactics)**

Assessment

LISPers invented first (and best) programming environments

everything interactive and dynamic

Difficult to eliminate errors at compile time

- **no compile-time checking in language**
- **everything represented as (exposed) list**

  hence mind-boggling `caddr` and friends
- **so you *need* a good programming environment**

Assessment

`lambda` is a major win

- **can't do it justice in this short time**
- **but we'll see it some more when we learn ML**
- **has a real implementation cost**
  - **heap-allocated closures**
  - **(copying environments)**

Before leaving the LISP family, some comments about parentheses

- **a major barrier to many people**
- **but as many people find it elegant**
- **last word: enables programs as data**

What LISP really stands for

Land of

Infinite

Stupid

Parentheses