CSE 341
Lecture 18

symbolic data; code as data;
writing a REPL loop; symbolic differentiation

slides created by Marty Stepp
http://www.cs.washington.edu/341/

Symbols and evaluation

- Scheme has a type called `symbol`
  - a symbol is very similar to a one-token immutable string
  - a symbol's *intrinsic value* is simply its name
    - but it can be connected to / associated with other values

- *all* Scheme code is treated as lists of symbols
  - the list `(+ 4 7)` is a list containing 3 symbols: +, 4, and 7

- the Scheme interpreter reads and *evaluates* symbols
  - symbols are keys in *(name, value)* pairs in the environment

Defining and using symbols

```
(quote name)
'name                    ; shorthand
```

- `(define name 'Suzy)`

- a list can contain symbols:
  - `(define mylist (list 'a 'b 42 'c 17 'd))`
- precede the list with `'` to make all its elements symbols:
  - `(define mylist2 '(a b c d))`

Symbol procedures

```
(symbol? expr)                 ; type test
(symbol=? sym1 sym2)           ; eq? also works
(symbol->string sym)
(string->symbol str)
```

- symbols are *interned*; two identical symbols are equal:

```
(define s1 'Hello)
(define s2 'Hello)
(eq? s1 s2)              → #t
```

Symbols vs. strings

- Schemers tend to favor using symbols over strings
  - symbols are atomic, while a string is an array of characters
  - symbols are immutable, while Scheme strings are not
  - Scheme's syntax often makes manipulating symbols easier

- much of the language syntax uses symbols:
  - `(define` ***`symbol`*** ***`expr`***`)`
  - `(let ((`***`symbol`*** ***`expr`***`) ...)` ***`expr`***`)`
  - `(`***`symbol`*** ***`expr`*** `...` ***`expr`***`)    ; procedure call`

    – most parts of the languages evaluate symbols; some don't

Symbol / value mappings

- the Scheme interpreter implements its environment as a table of mappings between *symbols* and *values*

```
(define x 5)
(define (square n) (* n n))
(define f square)
(define gpa 3.98)
```

symbol	**value**
`gpa`	`3.98`
`f`	`<procedure>`
`square`	`<procedure>`
`x`	`5`
system libraries...	...

global environment

- when code runs, it looks up values for each symbol:

```
> (+ x 2)
7
> (square 4)
16
```

How Scheme evaluates

- When it sees a list, Scheme *evaluates* each element, then *applies* the first (procedure) to the rest (params).

```
(+ 4 (* 2 3))

(  +  4  (* 2 3)  )
^  ^  ^  ^
|  |  |  |
|  |  |  +--list: evaluate! → ...6
|  |  +--number: evaluate! → 4
|  +--symbol: evaluate! → <procedure +>
+--list
```

  - What's the difference between *symbol* + and *procedure* +?

Quoted lists

`'(`***`expr expr ... expr`***`)`

- one way of thinking of `'` is that it "turns off the interpreter" for the duration of that list
  - i.e., it creates the list *without evaluating* its elements
  - a list of symbols rather than a list of their assoc. values

- this allows us to store **Scheme code as data**

```
> (define mycode '(+ 2 3))
```

Code as data

- Java and ML don't really have a way to do the following:

```
String code = "System.out.println(2 + 3);"
execute(code);
```

  - what would have to be done for this to work?

- manipulating code is much easier in a dynamic language
  - syntax/type checking are being done at runtime already
  - Scheme is looser about types, what is defined, etc.

Manually evaluating code

`(eval `***`code`***`)`

- tells interpreter to evaluate a symbol or list of symbols
  - Example:

```
> (define code '(+ 2 3))
> code
(+ 2 3)
> (eval code)
5
```

Evaluating symbols

- Symbols can be evaluated as identifiers, but they become references to identifiers if you interpret them:

```
> (define sym 'abc)
> sym
abc
> (eval sym)
reference to undefined identifier: abc
> (define abc 123)
> (eval sym)
123
```

Various uses of quotes

- What's the difference between these? Which are errors?
  - `(2 + 2)`
  - `(2 '+ 2)`
  - `'(2 + 2)`
  - `(list 2 + 2)`
  - `(list 2 '+ 2)`
  - `('list '2 '+ '2)`
  - `(list list 'list "list" '(list))`

References to procedures

- What is the difference between these two?

```
> (define f +)           ; what type is f?
> (define g '+)          ; what type is g?
> (define h '(+ 2 3))    ; what type is h?
```

- What is the result of each expression? Which ones fail?

```
> (f 2 3)
> (eval f)
> (g 2 3)
> (eval g)
> ((eval g) 2 3)
> (eval h)
```

Writing a REPL loop

- **REPL** ("read-eval-print") **loop**: Reads a statement or expression at a time, runs it, and shows the result.
  - examples: The Scheme and ML interpreters

- Exercise: Let's write our own crude Scheme REPL loop as a procedure named `repl` ...
  - *__l__**oop** while not done:*
    - *__r__**ead** command from user.*
    - *__e__**valuate** result of command.*
    - *__p__**rint** result on screen.*

Console I/O procedures

```
(display expr)    ; output expr or list to console
(newline)         ; output a line break (\n)
(read)            ; read token of input as a symbol
```

- note that read returns the *symbol* it read, not a string

```
> (define x (read))
hello how are you
> x
hello
> (symbol? x)
#t
```

REPL solution

```
(define (repl)
  (display "expression? ")
  (let ((exp (read)))                 ; read
      (display exp)
      (display " --> ")
      (display (eval exp))            ; eval / print
      (newline)
      (repl)))                        ; loop
```

The begin expression

(begin ***expr1 expr2*** ... ***exprN***)

- evaluates the expressions in order, ignoring the result of all but the last; result of ***exprN*** is the overall result
- useful for printing data and then returning a result

```
> (define x 3)
> (begin (display "x=") (display x)
  (newline)
           (* x x))
```

Differentiation (SICP 2.3.2)

- Suppose we're computing *derivatives* of math functions.
  - e.g. if $f(x) = ax^2 + bx + c$ (for constants a,b,c),
    $df/dx = 2ax + b$
  - suppose functions can consist of:
    – constants
    – variables (e.g. *x*)
    – addition with +
    – multiplication with *
  - we use the rules at right:

$$\frac{dc}{dx} = 0$$

$$\frac{dx}{dx} = 1$$

$$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx}$$

$$\frac{d(uv)}{dx} = u\frac{dv}{dx} + v\frac{du}{dx}$$

A grammar for our functions

<*func*>	::= NUMBER | VARIABLE | <*list*>
<*list*>	::= "(" <*term*> ")"
<*term*>	::= ("*" | "+") <*func*> <*func*>

- Grammar is specified in Extended Backus-Naur Format ("**EBNF**").
  - A **grammar** defines the syntax rules of a language.
  - The grammar maps from <*non-terminals*> to TERMINALs.

Derivative exercise

- Define a procedure `deriv` that takes a mathematical function (represented as a list of symbols) and a variable (a symbol) and differentiates the function.

```
; d/dx (x + 3) -> 1
deriv('(+ x 3) 'x) → 1

; d/dx (5x) -> 5
deriv('(* x 5) 'x) → x

; d/dz (z² + 5z) -> 2z + 5
deriv('(+ (* z z) (* 5 z)) 'z) → (+ (* 2 z) 5)

; d/dx (ax² + bx + c) -> 2ax + b
deriv('(+ (+ (* a (* x x)) (* b x) c)), 'x) →
       (+ (* (2 (* a x)) b))
```

Pseudo-code

- Use the EBNF grammar to guide the creation of the code.

  Pseudo-code:

  - function deriv(func, variable):
    - is func a **number**? if so, ...
    - is func a **variable**? if so, ...
    - is func a **list**?
      - starting with **+** ? if so, ...
      - starting with ***** ? if so, ...
    - ...

Checking types

(***type***? ***expr***)

- tests whether the expression/var is of the given type
  - `(integer? 42)` → `#t`
  - `(rational? 3/4)` → `#t`
  - `(real? 42.4)` → `#t`
  - `(number? 42)` → `#t`
  - `(procedure? +)` → `#t`
  - `(string? "hi")` → `#t`
  - `(symbol? 'a)` → `#t`
  - `(list? '(1 2 3))` → `#t`
  - `(pair? (42 . 17))` → `#t`

Helper procedures

- We suggest writing the following helper code:
  - `(sum? `***`func`***`)` - returns `#t` if `func` represents a sum in our grammar, such as `'(+ (* 2 3) 4)`
  - `(product? `***`func`***`)` - returns `#t` if `func` represents a product in our grammar, such as `'(* 3 (+ 2 5))`
  - `(make-sum `***`func1 func2`***`)` - takes the two operands of a + sum and returns their sum expression
    – `(make-sum 4 '(+ 2 3))` returns `(+ 4 (+ 2 3))`
  - `(make-product `***`func1 func2`***`)` - takes the two operands of a + product, returns the product expression
    – `(make-product '(+ 2 3) 4)` returns `(* (+ 2 3) 4)`

Improved derivative exercise

- Make the `deriv` function **simplify** various patterns:
  - $a+0 \rightarrow a$
  - $a*1 \rightarrow a$
  - $var+var \rightarrow 2*var$
  - $k*0 \rightarrow 0$
  - $k*1 \rightarrow k$

- Make the function produce an **error message** when given an invalid function that doesn't match the grammar.

The error procedure

(error *[symbol] [string]*)

- raises an exception with the given error string/symbol

```
> (error "kaboom!")
kaboom!
> (error 'abc "oh noez!")
abc: oh noez!
```

Quasi-quotes

```
(quasiquote expr expr ... expr)
`(expr expr expr)
```

- quasi-quotes are used to stop evaluation of *most* of a list
  - useful to mostly not evaluate a given expression, so that you don't have to individually quote lots of the pieces

```
> `(1 2 3)
(1 2 3)
> `(* 2 (+ 1 3))
(* 2 (+ 1 3))
```

Unquoting

```
(unquote expr)
,expr
(unquote-splicing list)
,@expr
```

- within quasi-quotes, `,` and `,@` cause a particular sub-expression or list to be evaluated (the rest isn't evaled)

```
> `(1 2 ,(+ 3 4) 5 ,@(list 6 7 8))
(1 2 7 5 (6 7 8))
```

Quasi-quotes versus quotes

- **quotes** are useful when you want to stop evaluation of an entire list, or stop evaluation of just one / a few items:

```
> '(1 2 3 4 5 6)                      ; good
> (list 1 2 3 4 (+ 2 3) 6)            ; good
> (list 'a 'b 'c 'd (+ 2 3) 'e 'f)    ; bad!
```

- **quasi-quotes** are useful when you want to stop evaluation of *most* of the items in a list, except for a few

```
> `(a b c d ,(+ 2 3) e f)             ; good
```