Scheme - Untyped Functional Language

Next 4-5 lectures. Sethi Chapter 10.

Overview:
- History and Introduction
- Primitive and compound expressions
- Definitions
- Lists and their manipulation
- Recursive reasoning
- Higher-order programming
- Memory management in Scheme

History

1958: Lisp (McCarthy et al) - a language for symbolic processing:
- First language with recursion
- First language with first-class functions (almost)
- First language with automatic storage management (garbage collection)
- First language with formal language definition (Lisp interpreter in Lisp)
- pioneering influence in design of integrated programming environments
- but had dynamic scoping

1960: Algol-60 (Naur et al) - general purpose ALGOrithmic Language:
- introduced BNF as a way to specify syntax
- introduced block structure and static scoping
- provided recursion (McCarthy)

History (Cont'd)

1970's: Scheme developed for undergrad curriculum at MIT by Guy Steele and Gerald Sussman.
- "Best features" of Algol-60 and Lisp - first-class functions with static scoping

1984: Guy Steel heads committee on Common Lisp. Static scoping.

Introduction to Scheme

- Statement - evaluated for side-effect:
  `x := x+1`
- Expression - evaluated to obtain a value:
  arithmetic expressions: `x+3`
  `x := x+1` - expression within a statement

Scheme is an *expressional* language: definitions and expressions but not statements.

Expressions - prefix notation:

```
(+ 5 7)
(* 2 3)
(+ 2 3 4 5)
(* 2 (+ 3 4))
(* 2 3 (+ 2 (* 3 4)))
```

- "Lots of Silly Parentheses"
- uniform syntax - can manipulate programs as data

Primive Expressions

"Primitive" - expressing values of atomic datatypes.
- numbers: `1, 2, 3, ..., 1.5, ...`
- booleans: `#t, #f`
- characters: `#\a, #\b, ..., #\1, #\2, ..., #\newline, #\space`
- strings: `"abc"`, `"123ha&%"`
- variables: `a, b, c, d`
- symbols: `'abc, 'apply, 'equal?, 'char->string, '+...`

Symbols and Strings

- Symbols start with quote `'` and then an identifier
- Strings are identifiers in double-quotes `'' ''`
  `"abc"` expresses (string) value "abc"
  `'abc` expresses (symbol) value 'abc
- operations `symbol->string` and `string->symbol` map between symbols and strings:
  `(symbol->string 'abc)` $\Rightarrow$ `"abc"`
  `(string->symbol "abc")` $\Rightarrow$ `abc`

Symbols and Variables

- A variable *denotes* a value (like name for location in which data is stored)
  `xarg` evaluates to value of `xarg`, e.g., 17
- A symbol *is* a value which cannot be evaluated any further.
  `'xarg` evaluates to the value `xarg`
- Strings also evaluate to themselves:
  `"xarg"` evaluates to the value `"xarg"`

Example:
- `Dallas is a popular city`
- `"Dallas" has 2 syllabi`

- Can assign symbol to variable:
  `(define a 'a)` - `a` evalues to symbol `'a`

Big distinction between name and value

Compound expressions

- *List expression*
  `'()`
  `'(a)`
  `'(a b c)`
  `'(a (b c))`
- $'(E_1\ E_2\ ...\ E_k)$ expresses list $(E_1\ E_2\ ...\ E_k)$

- $(E_1\ E_2\ ...\ E_k)$ means $E_1(E_2, ..., E_k)$
- "evaluate parameters before evaluating function"
- so, quote means "do not evaluate what follows"

More on Quote

- Used to treat expressions as data
- How to quote?

```
(quote (a b c)) ⇒ (a b c)
'(a b c) ⇒ (a b c)
```

Examples:

```
pi ⇒ 3.14159
(quote pi) ⇒ pi
(* 2 3) ⇒ 5
('* 2 3) ⇒ ERROR: Bad procedure *
```


Some Built-in Procedures

Predicates are expressions that evaluate to true or false. In Scheme, end in "?".

```
(number? 9)                    ⇒  #t
(zero? 9)                      ⇒  #f
(zero? 0)                      ⇒  #t
(+4 5)                         ⇒
(symbol? "abc")                ⇒
(number? 123)                  ⇒

(string->symbol "comp-sci")    ⇒
(symbol->string 'comp-sci")    ⇒
(symbol->string "comp-sci")    ⇒

(null? '(a b c))               ⇒  #f
(null? '())                    ⇒  #t
(list 'a (+4 5)
   (symbol->string 'xarg))     ⇒
```


food for slide eater

food for slide eater