Scheme

Features

- Lisp dialect
- Functional programming language
- Lists are the main data structures in Scheme
- Dynamically typed variables
- Statically scoped language
- First class functions
- Anonymous functions (lambda)
- Block structured
- Case insensitive
- Heavy use of parentheses
- Prefix notation
- Heap based storage with garbage collection

Data types

- Numbers
  - Integer
  - Floating point
- Strings
- Symbols
- Characters
- Boolean
- Pairs and lists

Quote Expressions

- Quote or ‘ distinguishes between program code and data.

(quote hello) or ‘hello are equivalent

- In the above example the symbol hello must be quoted in order to prevent Scheme from treating hello as a variable

(quote (1 2 3 4)) or ‘(1 2 3 4) are equivalent and represent a list of 4 elements

- In the above example quote ‘ indicates that it should be treated as a list and not a function call.
  - ‘(+ 2 3) will return (+ 2 3) and not evaluate the procedure

Lists

- Single most important built in data type in Scheme is the list.
- Lists are unbounded, possibly heterogeneous collections of data.
- Important functions that operate on lists:

  length -- length of a list
  equal? -- test if two lists are equal (recursively)
  car -- first element of a list
  cdr -- rest of a list
  cons -- make a new list cell (a.k.a. cons cell)
  list -- make a list
  append – returns the concatenation of two lists.
  null? – returns #t if the argument is a null list

Lists

- car: Returns the head of the list without changing the list
  - (car '(2 6 1 4 5))

  Returns 2
- cdr: Returns everything except the head of the list.
  - (cdr '(2 6 1 4 5))

  Returns (6 1 4 5)
- cons: Creates a new list.
  - (cons 1 '(2 6 1 4 5))

  Returns (1 2 6 1 4 5)

Predicates for lists

- null? -- is the list empty?
- pair? -- is this thing a nonempty list?
- equal?

Variables

- Scheme has both local and global variables.
- No type declarations for variables.

Define :

- Can be used to define functions or variables.
  - (define <name> <expression>)
  - (define (<fn-name> <param1> ... <paramN>)
- Body is not evaluated. Just a binding is created.
- Define is primarily used to bind global variables.

Local variables

- let to declare and bind local, temporary variables.
- Syntax:

```
(let ((name1 value1)
      (name2 value2)
      ...
      (nameN valueN))
      expression1
      expression2
      ...
      expressionQ)
```

- Problem with let: while the bindings are being created, expressions cannot refer to bindings that have been made previously.
- The following wont work:

```
(let ((x 3) (y (+ x 1)))
     (+ x y))
```

Local variables

- To get around the problem of let, scheme provides let*

  (let* ((x 3) (y (+ x 1))) (+ x y))
- The let* values and bindings are computed sequentially, this means that later definitions may be dependent on the earlier ones.
- But in “let” values are computed and bindings are done in parallel, this means that the definitions are independent.

Conditional statements

If condition:

- (if condition expr1 expr 2)
  - Expr1 is executed when the condition is true and expr2 is executed when condition is false.

Cond expression:

```
(cond (test1 expr1)
        (test2 expr2)
        ....
        (else exprn))
```

- Else is optional.
- Finds a test that evaluates to true, then the corresponding expr is evaluated and value is returned. The remaining tests are not evaluated, and all the other expr's are not evaluated.
- If none of the tests evaluate to true then we evaluate exprn (the "else" part) and return its value if you have the else in your cond.

Predicates

- (boolean? arg)
- (number? arg)
- (pair? arg)
- (symbol? arg)
- (procedure? arg)
- (null? arg)
- (zero? arg)
- (odd? arg)
- (even? arg)

Lambda Expressions

- LAMBDA is used to create functions without giving them names (Anonymous functions)
  - (lambda (x) (* x x))
  - Anonymous function taking x as parameter and returns the square of it.

- Structure:

```
(lambda (param1 param2 ... paramk)      ; list of formals
        expr)                           ;body
```

letrec

- Nested “recursive” functions can be defined using letrec.
- Let and let* cannot be used to define nested recursive functions.
- Define statement is used to create global variables and not local variables, so define wont work either.

```
(define (f x)
        (letrec ((fac (lambda (y) (if (= y 0) 1 (* y (fac (- y 1)))))))
                (fac x)))
```

Comments

- ; followed by anything is comment