Scheme

CS 152

Dr. Saptarshi Sengupta

Naya Singhania

1. Basic Arithmetic

+, -, *, and / represent addition, subtraction, multiplication, and division respectively.

1.1. Example(s)

```
( + 1 2 )
```

2. Other Arithmetic Operation

- quotient
- remainder
- modulo
- sqrt
- exp
- log
- trigonometry
  - sin
  - cos
  - tan
  - asin
  - acos
  - atan

3. Lists

3.1. Cons Cells

- Memory spaces which stores two addresses.
  - **car** The part storaing the address to 1
  - **cdr** The part storing the address to 2
- Made by function `cons`.

3.1.1. Lists

- Lists are beaded cons cells with the cr part of the last cons cell being `'()`
- `'()` is called the empty List

4. atoms

- Data structures that do not use cons cells
- Numbers, characters, strings, vectors, and `'()` are atom
- `'()` is an atom and a list

5. quotient

- A special form named `quote` is used to prevent tokens from evaluation
- symbol'

5.1. Special forms

6. Functions car and cdr

- If the value of `car` is a beaded cons cell, it returns the address of the first element of the list.

7. Function List

8. Defining Functions

8.1. Hello World

```
; Hello world as a variable
(define vhello "Hello World")
```

```
(cd "C:\\doc\\scheme")
(load "hello.scm")
```

8.2. With parameters

```
; farg.scm
(define hello
  (lambda (name)
    (string-append "Hello " name)))
```

```
; main.scm
(load "farg.scm")
(hello "World")
```

8.2.1. Another form

```
(define (hello name)
  (string-append "Hello " name "!"))
```

9. Branching

9.1. The `if` expression

```
(if predicate then_value else_value)
```

10. `and` and `or`

10.1. `and`

`(and 1 2 3)` returns 3

- Return `#f` if any argument is `#f`

10.2. `or`

`(or 1 2 3)` returns 1

- Returns value of first argument which is not `#f`

11. `cond` expression

```
(cond
  (predicate_1 clauses_1)
  (predicate_2 clauses_2)
  ...
  (predicate_n clauses_n)
  (else clauses_else))
```

12. Functions that make predicates

12.1. `eq?`

- Compares addresses of two objects and returns `#t` if they are the same

12.2. `eqv?`

12.3. `equal?`

13. Functions that check data type

14. Local Variables

14.1. `let` expression

```
(let binds body)
```

- You can use `let*` as syntactic sugar that doesn't need nesting for `let` to bind variables with nesting

14.1.1. Example(s)

```
(let ((i 1) (j 2))
  (+ i j))
```

15. Repetition

15.1. Recursion

```
(define (fact n)
  (if (= n 1)
      1
      (* n (fact (- n 1)))))
```

15.2. Tail Recursion

```
(define (fact-tail n)
  (fact-rec n n))

(define (fact-rect n p)
  (if (= n 1)
    p
    (let (( m ( - n 1)))
      (fact-rec m (* p m)))))
```

15.3. Named `let`

- Available to express a loop