CSCI 4230 Lecture Notes – PLP 11.1-11.3

Functional Programming

Functional programming expresses a program as a mathematical function that calculates its output based on a given input.

- Pure functional programs have no internal state or side effects.
- Functions are first-class objects.
- Recursion is used to implement repetition.
- Anonymous functions are created using **lambda expressions**.
- Polymorphism is widely used.
- List-based data structures are used.
- Functions can return structured values.
- Garbage collection is used to deallocate dynamically allocated memory.

The first functional programming language was Lisp.

- Lisp is interpreted and dynamically typed.
- Programs and data are both represented using list-based structures.
- A Lisp interpreter can be easily written in Lisp.
- The interpreter provides a **read-eval-print** loop.

Scheme

Scheme is a dialect of Lisp developed by Guy Steele and Gerald Sussman in the late 1970's.

The Scheme interpreter evaluates Scheme expressions and displays the resulting value to the user.

- Expressions are written in **Cambridge Polish Notation**:

  ```
  (<operation> <arg1> <arg2> ... <argN>)
  ```

- The interpreter reads the expression, evaluates it, then displays the result back to the user:

  ```
  > (+ 3 4)
  7
  ```

Expressions

Scheme has most of the usual building blocks for expressions:

- Numbers like `4`, `78`, `-13`, and `3.14`.
- Boolean values `#t` for true and `#f` for false.
- Characters, represented using `#\` followed by the literal character, or a longer sequence of symbols for special characters: `#\a`, `#A`, `#\space`
- Strings, represented by characters in double quotes with the usual escape sequences like `\"` and `\\`.
- Identifiers consist of a sequence of characters that adhere to the following rules:
  1. An identifier may not start with any character that can begin a number.
  2. An identifier may contain letters, digits and any of the following characters:

     `! $ % & * + - . / : < = > ? @ ^ _ ~`
  3. `+`, `-`, and `...` are all identifiers

Compound expressions can be built using the Cambridge Polish Notation describe above:

- The interpreter attempts to use the first expression after the `(` as a **procedure** and the remaining expressions as **arguments** to the procedure.
- If the first expression is not defined as a procedure, or the wrong number of arguments is provided, the interpreter will display an error:

```
> (+ 3 4)
7
> (3 + 4)
application: not a procedure;
```

- Note that extra parentheses can cause errors:

```
> (+ 3 4)
7
> ((+ 3 4))
application: not a procedure;
```

Scheme can be prevented from evaluating non-literal values using `quote` or `'`.

- Quoting an identifier turns it into a **symbol**:

```
> (quote a)
a
> 'a
a
> (quote (+ 3 7))
(+ 3 7)
> '(+ 3 7)
(+ 3 7)
```

- Quoting a parenthesized expression turns it into a **list**.

```
> (quote (+ 3 7))
(+ 3 7)
> '(+ 3 7)
(+ 3 7)
```

Types of expressions can be determined using **predicates**:

```
> (number? 34)
#t
> (number? "34")
#f
> (symbol? 'xyz)
#t
> (boolean? #t)
#t
> (list? '(a b c))
#t
```

Conditional expressions can be written using the special form (like a procedure except its arguments are not evaluated) `if`:

```
> (if (< 3 4) 6 7)
6
> (if #f 8 9)
9
> (if (symbol? 'xyz) "hello" "goodbye")
"hello"
```

Procedures can be defined using the special form `lambda`:

```
> (lambda (x) (* x x))
#<procedure>
> ((lambda (x) (* x x)) 6)
36
```

Bindings

Bindings with global environment can be created using `define`:

```
> (define x 5)
> x
5
> (define square (lambda (x) (* x x)))
> (square 5)
25
> (define (cube x) (* x x x))
> (cube 7)
343
```

Note that there are two different notations for defining procedures. The first shows that a procedure definition in Scheme is the binding of a name in the global environment to an anonymous function. The second is just syntactic sugar (a sweeter way of expressing something) for the first notation.

Local bindings can be created using `let`:

```
> (let ((a 3)
        (b 4)
        (square (lambda (x) (* x x)))
        (plus +))
    (sqrt (plus (square a) (square b))))
5
```

Recursive local bindings can be created using `letrec`:

```
> (letrec ((fact
            (lambda (n)
              (if (= n 1) 1
                  (* n (fact (- n 1)))))))
    (fact 5))
120
```

Pairs and Lists

A **pair** is a data structure that groups together two values.

- A pair can be created using the pair constructor `cons`:

  ```
  > (cons 'a 'b)
  (a . b)
  ```

- The values in a pair can be extracted by using the pair accessor procedures `car` and `cdr`:

  ```
  > (car '(a . b))
  a
  > (cdr '(a . b))
  b
  ```

A **list** is either the empty list or a pair where the second element is a list.

- The empty list is written as `'()`.
- Lists can be constructed using `cons` or written out explicitly using parenthesized expressions:

  ```
  > (cons 'a '())
  (a)
  > (cons 'a (cons 'b '()))
  (a b)
  > '(a b c)
  (a b c)
  ```

- List items can be accessed using `car` and `cdr`:

  ```
  > (car '(a b c))
  a
  > (cdr '(a b c))
  (b c)
  > (car (cdr '(a b c)))
  b
  > (cadr '(a b c)) ;syntactic sugar for previous expression
  b
  ```

Values can be tested for equality using the following predicates:

- `=` can be used to compare numerical values.
- `eq?` determines if two values refer to the same object.
- `eqv?` determines if two values are semantically equivalent. (Depending on the implementation, this may give the same results as `eq?`, but is more consistent and therefore preferred.)
- `equal?` determines if two values have the same structure, using `eqv?` to compare non-structured values.

```
> (= (+ 1 1) 2)
#t
> (define list1 '(1 2 3))
> (define list2 '(1 2 3))
> (eqv? list1 list2)
#f
> (define list3 list1)
> (eqv? list1 list3)
#t
> (equal? list1 list2)
#t
```

List membership is determined using the following predicates:

- `memq` uses `eq?` to compare values.
- `memv` uses `eqv?` to compare values.
- `member` uses `equal?` to compare values.
- All three return `#f` if the value is not found in the list, and the longest suffix of the list beginning with the value if it is found.

```
> (memq 'z '(x y z w))
(z w)
> (memv '(z) '(x y (z) w))
#f
> (member '(z) '(x y (z) w))
((z) w)
```

Control Flow and Assignment

We have already seen how the special form `if` can be used to create conditional expressions.

Conditional expressions can be nested, or the special form `cond` can be used to test multiple conditions:

```
> (cond
    ((< 3 2) 1)
    ((< 4 3) 2)
    (else 3))
3
```

Pure functional languages do not have side-effects, but most functional programming languages have some imperative features.

Variables can be modified using `set!`

```
> (define x 23)
> x
23
> (define add-n-to-x
    (lambda (n)
      (set! x (+ x n))))
> (add-n-to-x 5)
> x
28
```

Lists can be modified using `set-car!`, and `set-cdr!`:

```
> (define my-list '(a b c))
> my-list
(a b c)
> (set-car! my-list 'd)
> my-list
(d b c)
> (set-cdr! my-list '())
> my-list
(d)
```

Sequencing can be performed using the special form `begin`:

```
(begin
  (display "hi ")
  (display "mom"))
```

Iteration can be performed using the special form `do`:

```
(define iter-fib
  (lambda (n)
    ; print the first n + 1 Fibonacci numbers
    (do ((i 0 (+ i 1))    ; initially 0, incremented in each iteration
         (a 0 b)          ; initially 0, set to b in each iteration
         (b 1 (+ a b)))   ; initially 1, set to sum of a and b
      ((= i n) b)         ; termination test and final value
      (display b)         ; body of loop
      (display " "))))    ; body of loop
```

Note the use of the `display` procedure for displaying output to the console.

The special form `for-each` is used to iterate through lists:

```
> (for-each
   (lambda (a b) (display (* a b)) (newline))
   '(2 4 6)
   '(3 5 7))
6
20
42
```

The `newline` procedure displays a newline to the console.