MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1997

Lecture Notes – September 25, 1997

Lists are a series of `cons` cells whose `cdr`s point to the next cell, and whose last element has a `cdr` pointing to the empty list. The value of `nil` is usually the empty list (in some implementations of Scheme the empty list and `#f` may be the same thing). If `nil` is not defined in a particular implementation of Scheme then use:

```
(define nil (list))
```

`list` always `cons`es exactly as many cells as there are arguments to `list`.

```
(list 2 3 4 5) <==> (cons 2 (cons 3 (cons 4 (cons 5 nil))))
```

Lists point out as a pair of parentheses with all the `car`s, one for each cons cell printed between them:

```
(list 2 3 4 5) ==> (2 3 4 5)
(list 2 (list 3 4) 5) ==> (2 (3 4) 5)
```

The `car` of a list is the first element, the `car` of the `cdr`, or `cadr`, is the second element. The `cddr` is all but the first two elements. In the second example above the `3` can be accessed with `caadr`.

List Operations

```
(define (length lst)
  (if (null? lst)
      0
    (+ 1 (length (cdr lst)))))

(define (increment-list lst)
  (if (null? lst)
      nil
    (cons (+ 1 (car lst))
          (increment-list (cdr lst)))))

(define (map proc lst)
  (if (null? lst)
      nil
    (cons (proc (car lst))
          (map proc (cdr lst)))))

(define (append u v)
  (if (null? u)
      v
    (cons (car u) (append (cdr u) v))))
```

Quotation and Symbols

Another special form is `quote`. When it appears as the first element of a combination then the second element is returned directly as the value of that combination:

```
(quote 2) ==> 2
(quote (2 3 4 5)) ==> (2 3 4 5)
(quote (list 2 3 4 5) ==> (list 2 3 4 5)
```

Besides numbers another primitive data type is symbols, like foo, list, *, etc. Symbols have *names* and they also can have *values*.

```
(define foo 3)
```

The symbol foo is named foo, but has value 3.

```
(define bar (quote baz))
(define baz 5)
(list (quote bar) (quote baz) bar baz) ==> (bar baz baz 5)
```

A shorthand for quote is ', which when preceeding an expression <exp>, as in '<exp> is read as (quote <exp>).

```
(list '+ 2 3) ==> (+ 2 3)
(cons '* '(4 5 6)) ==> (* 4 5 6)
```

The predicate = tests whether two numbers are the same thing. The predicate eq? tests whether two symbols are the same thing. If one argument is a symbol it returns #t if the other argument is the same symbol, and #f otherwise. If neither argument is a symbol then its returned value could be anything (this is not completely true—we will make more cases explicit later).

```
(eq? 'a 'a) ==> #t
(eq? 'a 'b) ==> #f
(eq? 3 'b) ==> #f
(eq? 'b (cadr '(a b c))) ==> #t
```

Algebraic expressions

Earlier, we saw a way to make numerical approximations to a derivative

```
(define (numerical-derivative f)
  (define epsilon 0.0001)
  (lambda (x) (/ (- (f (+ x epsilon))
                    (f x))
                 epsilon)))
```

Now we want to find "exact" or symbolic derivatives of algebraic expressions. Assume that we have the following selectors, constructors and predicates:

(constant? <e>)	Is <e> a constant?
(variable? <e>)	Is <e> a variable?
(same-variable? <v1> <v2>)	Are <v1> and <v2> the same variable?
(sum? <e>)	Is <e> a sum?
(product? <e>)	Is <e> a product?
(addend <e>)	Addend of the sum <e>.
(augend <e>)	Augend of the sum <e>.
(multiplier <e>)	Multiplier of the product <e>.
(multiplicand <e>)	Multiplicand of the product <e>.
(make-constant <c>)	Construct the constant <c>.
(make-sum <a1> <a2>)	Construct the sum of <a1> and <a2>.
(make-product <m1> <m2>)	Construct the product of <m1> and <m2>.

We can easily translate the mathematical notation for derivatives into code:

```
(define (deriv exp var)
  (cond ((constant? exp) (make-constant 0))
        ((variable? exp)
         (if (same-variable exp var)
             (make-constant 1)
             (make-constant 0)))
        ((sum? exp)
         (make-sum (deriv (addend exp) var)
                   (deriv (augend exp) var)))
        ((product? exp)
         (make-sum
           (make-product (multiplier exp)
                         (deriv (multiplicand exp) var))
           (make-product (deriv (multiplier exp) var)
                         (multiplicand exp))))))
```

We need some selectors and constructors for the data abstraction.

```
(define (constant? x) (number? x))

(define (variable? x) (symbol? x))

(define (same-variable? v1 v2)
  (and (variable? v1) (variable? v2) (eq? v1 v2)))

(define (make-constant x) x)

(define (make-sum a1 a2)
  (list '+ a1 a2))
(define (make-product m1 m2)
  (list '* m1 m2))

(define (sum? x)
  (if (pair? x) (eq? (car x) '+) #f))

(define (addend s) (cadr s))

(define (augend s) (caddr s))

(define (product? x)
  (if (pair? x) (eq? (car x) '*) #f))

(define (multiplier p) (cadr p))

(define (multiplicand p) (caddr p))
```

We can add the following extensions to the simple system.

Simplification:

```
(define (make-sum a1 a2)
  (cond ((and (constant? a1) (constant? a2))
         (make-constant (+  a1 a2)))
        ((constant? a1)
         (if (= a1 0)
             a2
             (list '+ a1 a2)))
        ((constant? a2)
         (if (= a2 0)
             a1
             (list '+ a1 a2)))
        (else (list '+ a1 a2))))
(define (make-product m1 m2)
  (cond ((and (constant? m1) (constant? m2))
         (make-constant (*  m1 m2)))
        ((constant? m1)
         (cond ((= m1 0) (make-constant 0))
               ((= m1 1) m2)
               (else (list '* m1 m2))))
        ((constant? m2)
         (cond ((= m2 0) (make-constant 0))
               ((= m2 1) m1)
               (else (list '* m1 m2))))
        (else (list '* m1 m2))))
```

We can expand the system to handle arbitrary numbers of arguments:

```
(define (augend s)
  (if (null? (cdddr s))
      (caddr s)
      (append '(+) (cddr s))))

(define (multiplicand p)
  (if (null? (cdddr p))
      (caddr p)
      (append '(*) (cddr p))))
```

We might also want to change the constructors:

```
(define (make-sum a1 . a2) (cons '+ (cons a1 a2)))

(define (make-product a1 . a2) (cons '* (cons a1 a2)))
```