Lisp & Scheme

Lisp (*Lis*t *P*rocessing Language) is one of the oldest programming languages still in wide use.

It was developed in the late 50s and early 60s by John McCarthy.

Its innovations include:

- Support of symbolic computations.
- A *functional programming style* without emphasis on assignments and side-effects.
- A naturally recursive programming style.
- Dynamic (run-time) type checking.

- Dynamic data structures (lists, binary trees) that grow without limit.
- Automatic garbage collection to manage memory.
- Functions are treated as “first class” values; they may be passed as arguments, returned as result values, stored in data structures, and created during execution.
- A formal semantics (written in Lisp) that defines the meaning of all valid programs.
- An Integrated Programming Environment to create, edit and test Lisp programs.

Scheme

Scheme is a recent dialect of Lisp.

It uses lexical (static) scoping.

It supports true first-class functions.

It provides program-level access to control flow via *continuation* functions.

Atomic (Primitive) Data Types

Symbols:

Essentially the same form as identifiers. Similar to enumeration values in C and C++.

Very flexible in structure; essentially any sequence of printable characters is allowed; anything that starts a valid number (except + or -) *may not* start a symbol.

Valid symbols include:

```
abc  hello-world   +  <=!
```

Integers:

Any sequence of digits, optionally prefixed with a + or -. Usually unlimited in length.

Reals:

A floating point number in a decimal format (**123.456**) or in exponential format (**1.23e45**). A leading sign and a signed exponent are allowed (**-12.3**, **10.0e-20**).

Rationals:

Rational numbers of the form integer/integer (e.g., **1/3** or **9/7**) with an optional leading sign (**-1/2**, **+7/8**).

Complex:

Complex numbers of the form num+num i or num-num i, where num is an integer or real number. Example include **1+3i**, **-1.5-2.5i**, **0+1i**).

String:

A sequence of characters delimited by double quotes. Double quotes and backslashes must be escaped using a backslash. For example

`"Hello World"  "\"Wow!\""`

Character:

A single character prefixed by **`#\`**. For example, **`#\a`**, **`#\0`**, **`#\\`**, **`#\#`**. Two special characters are **`#\space`** and **`#\newline`**.

Boolean:

True is represented as **`#t`** and false is represented as **`#f`**.

Binary Trees

Binary trees are also called *S-Expressions* in Lisp and Scheme.

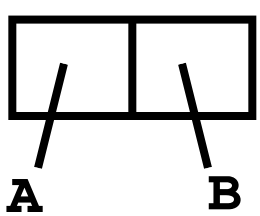
They are of the form

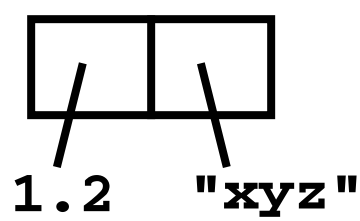
(item . item)

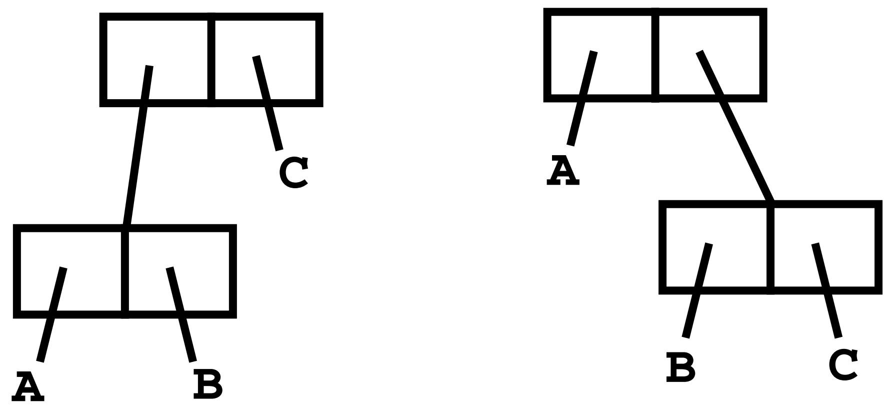
where item is any atomic value or any S-Expression. For example:

```
( A . B )
(1.2  .  "xyz" )
( (A . B ) . C )
( A . (B . C ) )
```

S-Expressions are linearizations of binary trees:







S-Expressions are built and accessed using the predefined functions *cons*, *car* and *cdr*.

cons builds a new S-Expression from two S-Expressions that represent the left and right children.

cons(E1,E2) = (E1 . E2)

car returns are left subtree of an S-Expression.

car (E1 . E2) = E1

cdr returns are right subtree of an S-Expression.

cdr (E1 . E2) = E2

Lists

In Lisp and Scheme lists are a special, widely-used form of S-Expressions.

() represents the empty or null list

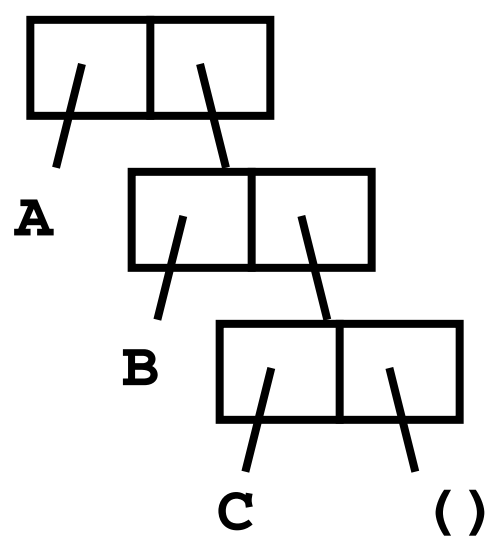
(A) is the list containing **A**.
By definition, **(A) ≡ (A . ())**

(A B) represents the list containing **A** and **B**. By definition,
(A B) ≡ (A . (B . ()))

In general, **(A B C ... Z) ≡**

(A . (B . (C . ... (Z . ()) ...)))

(A B C)≡



Function Calls

In List and Scheme, function calls are represented as lists.

(A B C) means:

Evaluate **A** (to a function)

Evaluate **B** and **C** (as parameters)

Call **A** with **B** and **C** as its parameters

Use the value returned by the call as the “meaning” of **(A B C)**.

cons, **car** and **cdr** are predefined symbols bound to built-in functions that build and access lists and S-Expressions.

Literals (of type integer, real, rational, complex, string, character and boolean) evaluate to themselves.

For example ($\Rightarrow$ means "evaluates to")

`(cons 1 2)` $\Rightarrow$ `(1 . 2)`

`(cons 1 () )` $\Rightarrow$ `(1)`

`(car (cons 1 2))` $\Rightarrow$ `1`

`(cdr (cons 1 ()))` $\Rightarrow$ `()`

But,

`(car (1 2))` fails during execution!

Why?

The expression **`(1 2)`** looks like a call, but **`1`** isn't a function! We need some way to "quote" symbols and lists we *don't* want evaluated.

`(quote arg)`

is a special function that returns its argument *unevaluated*.

Thus **`(quote (1 2))`** doesn't try to evaluate the list **`(1 2)`**; it just returns it.

Since quotation is so often used, it may be abbreviated using a single quote. That is

`(quote arg)` ≡ **`'arg`**

Thus

```
(car '(a b c)) ⇒ a

(cdr '( (A) (B) (C))) ⇒
   ( (B) (C) )

(cons 'a '1) ⇒ (a . 1)
```

But,

`('cdr '(A B))` fails!

Why?

User-defined Functions

The list

```
(lambda (args) (body))
```

evaluates to a function with `(args)` as its argument list and `(body)` as the function body.

No quotes are needed for `(args)` or `(body)`.

Thus

`(lambda (x) (+ x 1))` evaluates to the increment function.

Similarly,

```
((lambda (x) (+ x 1))  10) ⇒
  11
```

We can bind values and functions to global symbols using the **`define`** function.

The general form is

```
(define id object)
```

`id` is not evaluated but **`object`** is. **`id`** is bound to the value object evaluates to.

For example,

```
(define pi  3.1415926535)

(define plus1
   (lambda (x) (+ x 1)) )
(define pi*2  (* pi 2))
```

Once a symbol is defined, it evaluates to the value it is bound to:

`(plus1 12)` $\Rightarrow$ **`13`**

Since functions are frequently defined, we may abbreviate

```
(define id
  (lambda (args) (body)) )
```

as

```
(define (id args) (body) )
```

Thus

```
(define (plus1 x) (+ x 1))
```

Conditional Expressions in Scheme

A *predicate* is a function that returns a boolean value. By convention, in Scheme, predicate names end with "?"

For example,

```
number?  symbol?  equal?
null?    list?
```

In conditionals, **#f** is false, and everything else, including **#t**, is true.

The **if** expression is

(if pred E1 E2)

First **pred** is evaluated. Depending on its value (**#f** or not), either **E1** or **E2** is evaluated (but not both) and returned as the value of the **if** expression.

For example,

```
(if  (=  1  (+  0 1))
     'Yes
     'No
)


(define
  (fact n)
  (if  (= n  0)
       1
       (*  n  (fact  (- n 1)))
  )
)
```

Generalized Conditional

This is similar to a switch or case:

```
(cond
    (p1  e1)
    (p2  e2)
     ...
    (else  en)
)
```

Each of the predicates (**p1**, **p2**, ...) is evaluated until one is true (≠ **#f**). Then the corresponding expression (**e1**, **e2**, ...) is evaluated and returned as the value of the **cond**. **else** acts like a predicate that is always true.

Example:

```
(cond
    ((= a 1)  2)
    ((= a 2)  3)
    (else     4)
)
```