Binary Trees

Binary trees are also called S-Expressions in Lisp and Scheme. They are of the form

\[( \text{item} . \text{item} )\]

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:

```
          /
         /  \
        A    B
```

```
          /
         /  \
        1.2  "xyz"
```
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.

\[ \text{cons}(E_1,E_2) = (E_1 \ . \ E_2) \]

`car` returns are left subtree of an S-Expression.

\[ \text{car} \ (E_1 \ . \ E_2) = E_1 \]

`cdr` returns are right subtree of an S-Expression.

\[ \text{cdr} \ (E_1 \ . \ E_2) = E_2 \]
Lists

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

() represents the empty or null list

(A) represents 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 ) ≡

```
  /\   /\   /\   /\   /\
 A  /\ /\ /\ /\ /\ /\ B
    /\ /\ /\ /\ /\ /\ C
      /\ /\ /\ /\ /\ (/)
```

(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
Then use the value returned by the call as the “meaning” of \((A \ B \ C)\).

\texttt{cons}, \texttt{car} and \texttt{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 (⇒ means “evaluates to”)

(cons 1 2) ⇒ (1 . 2)
(cons 1 () ) ⇒ (1)
(car (cons 1 2)) ⇒ 1
(cdr (cons 1 ())) ⇒ ()

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

\[(\text{lambda} \ (\text{args}) \ (\text{body}))\]
evaluates to a function with \(\text{args}\) as its argument list and \(\text{body}\) as the function body.

No quotes are needed for \(\text{args}\) or \(\text{body}\).

Thus

\[(\text{lambda} \ (x) \ (+ \ x \ 1))\] evaluates to the increment function.

Similarly,

\[
((\text{lambda} \ (x) \ (+ \ x \ 1)) \ 10) \Rightarrow 11
\]
We can bind values and functions to global symbols using the \texttt{define} function.

The general form is

\begin{verbatim}
(define id object)
\end{verbatim}

\texttt{id} is not evaluated but \texttt{object} is. \texttt{id} is bound to the value \texttt{object} evaluates to.

For example,

\begin{verbatim}
(define pi 3.1415926535)
(define plus1 (lambda (x) (+ x 1)))
(define pi*2 (* pi 2))
\end{verbatim}

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

\begin{verbatim}
(plus1 12) \Rightarrow 13
\end{verbatim}
Since functions are frequently defined, we may abbreviate

\[(\text{define id}
(\lambda (\text{args}) \text{(body)}))\] as

\[(\text{define (id args) (body))\] Thus

\[(\text{define (plus1 x) (+ x 1))\]