Lisp & Scheme

Lisp (List Processing 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.

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.

Reading Assignment

- Roosta: Sections 13.1, 13.2
- The Scheme Language Definition (linked from class web page)
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. Examples 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 \ 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:

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.
- `car` returns the left subtree of an S-Expression.
- `cdr` returns the right subtree of an S-Expression.

\[
\begin{align*}
\text{cons}(E_1, E_2) &= (E_1 . E_2) \\
car(E_1 . E_2) &= E_1 \\
cdr(E_1 . E_2) &= E_2
\end{align*}
\]

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`.
- `(A B)` represents the list containing `A` and `B`.

In general, `(A B C ... Z)` is defined as:

\[
\begin{align*}
(A B C ) &\equiv (A . (B . (C . ... (Z . () ) ... )))
\end{align*}
\]

Function Calls

In Lisp 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)`.

- `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 (⇒ means “evaluates to”)

\[
\begin{align*}
\text{cons}(1, 2) &\Rightarrow (1 . 2) \\
\text{car}(\text{cons}(1, 2)) &\Rightarrow 1 \\
\text{cdr}(\text{cons}(1, 2)) &\Rightarrow ()
\end{align*}
\]

But,

\[
\text{car}(1 2)
\]

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

```lisp
(quote arg) \equiv 'arg
```

Thus

```lisp
(car '(a b c)) \Rightarrow a
(cdr '( (A) (B) (C))) \Rightarrow
  ( (B) (C) )
(cons 'a '1) \Rightarrow (a . 1)
```

But,

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

Why?

User-defined Functions

The list

```lisp
(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

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

Similarly,

```lisp
((lambda (x) (+ x 1)) 10) \Rightarrow 11
```

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

The general form is

```lisp
(define id object)
```

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

For example,

```lisp
(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:

```lisp
(plus1 12) \Rightarrow 13
```

Since functions are frequently defined, we may abbreviate

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

as

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

Thus

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