Lisp

Lisp

In this session we will get to know another functional language: *Lisp*

Lisp is the oldest functional language, its development goes back to 1959-60 (John McCarthy).

The name is an acronym for *Lis*t *p*rocessor.

At the time, Lisp was created for manipulating data structures for symbolic computation, such as trees and lists.

Other high-level languages of the time manipulated only arrays (Fortran, Algol 60), or only records (COBOL).

Applications of Lisp

Lisp has been applied to implement many substantial applications.

Examples:

- Macsyma, the first computer algebra system
- Emacs, the well-known text editor
- Auto-CAD, one of the first computer-aided design programs
- The ITA flight information system

Lisp Variants

Over its 50+ years of existence, Lisp has evolved a lot, and today there are many dialects.

The best known dialects are:

- Common Lisp (several commercial implementations, many language features): Allegro CL, CLISP, GCL, SBCL, ...
- Scheme (clean, well-suited for teaching): Scheme 48, Chicken Scheme, Gambit Scheme, ...
- Racket (complete and comfortable language system derived from Scheme)
- Clojure (Lisp on JVM, emphasis on functional aspects and concurrency)
- Elisp (the extension language of the Emacs editor).

Here, we will cover only a minimalistic and purely functional variant of Scheme.

What's Special About Lisp

Compared to Scala there are four principal areas where classical Lisp is different, and which are interesting to study:

- No elaborate Syntax. Programs are simply nested lists of words.
- No static type system.
- Focus on a single data structure: The *cons-cell*, from which lists and other data are constructed.
- Programs are lists themselves, which makes it easy for programs to construct, transform, and evaluate other programs.

Example

Here’s an example program in Scheme:

```
(define (factorial n) (if (zero? n)
                          1
                          (* n (factorial (- n 1)))))
(factorial 10)
```

Observations

A non-primitive Lisp expression is a sequence of sub-expressions between parentheses `(...)`.

- Such an expression is called a *combination*

Sub-expressions are simple words (called *atoms*) or combinations.

Sub-expressions are separated by spaces.

The first sub-expression of a combination represents an *operator*, which is in general a function.

The other expressions in a combination are the *operands*.

Special Forms

Some combinations look like a function application but are really something different.

For instance, in Scheme:

`(define name expr)`	defines `name` as alias of the result of evaluating `expr` (analogous to `def name = expr` in Scala).
`(lambda (params) expr)`	the anonymous function that takes parameters `params` and returns `expr` (analogous to `params => expr` in Scala).
`(if cond expr1 expr2)`	the result of evaluating `expr1` if `cond` evaluates to true, otherwise the result of evaluating `expr2`.

Combinations like these are called *special forms* in Scheme.

Function Application

A combination (`op` $\texttt{od}_1 \ldots \texttt{od}_n$) that is not a special form is treated as a function application.

It is evaluated by applying the result of evaluating `op` to the results of evaluating the operands $\texttt{od}_1 \ldots \texttt{od}_n$.

That's it! It is hard to imagine a programming language with fewer rules.

The reason Lisp is so simple is that it was originally designed as an intermediate language for computers rather than human beings. The plan was to add a more human-friendly syntax similar to that of Algol, which would then be translated in to the intermediate form by a preprocessor.

However, programmers got used to the Lisp syntax rather quickly (at least some of them did), and started to appreciate its advantages. So much so, that the human-friendly syntax was never added.

Lisp Data

Data types in Lisp are numbers, strings, symbols and lists.

- Numbers are either floating point numbers or integers. In many Lisp dialects, integers have unbounded capacity (no overflows!)
- Strings are similar to those in Java
- Symbols are simple unquoted sequences of characters.
  Examples:

  ```
  x    head    +    null?    is-empty?    set!
  ```

  Symbols are evaluated by looking up the value of a definition in an environment.

In Lisp, any value can be used as a conditional. The convention is that only certain specific values (e.g. the empty list `nil`) represent `false`, and all others represent `true`.

Lists in Lisp

Lists are written as combinations, e.g.

```
(1 2 3)
(1.0 "hello" (1 2 3))
```

Note that lists are heterogeneous, i.e. their elements can be of different types. Also, lists such as those above can not be evaluated since their first element is not a function.

To prevent the evaluation of a list, the special from `quote` is used:

```
(quote (1 2 3))
```

The argument of `quote` is returned without being evaluated. The character ' can be used as a shorthand for `quote`:

```
'(1 2 3)
```

Internal representation of lists

As in Scala, the list notation is just syntactic sugar. Internally, lists are constructed from

- the empty list, denoted by `nil`,
- pairs consisting of a head `x` and tail `y`, denoted by `(cons x y)`.

The list (or combination) `(`$x_1 \dots x_n$`)` is represented by

    (cons x₁ (... (cons xₙ nil) ...))

Lists are accessed using the following three operations:

- `(null? x)` returns true if `x` is empty,
- `(car x)` returns the head of the list `x`,
- `(cdr x)` returns the tail of the list `x`.

car and cdr

The names `car` and `cdr` date back to the original implementation of Lisp on the IBM 704 computer.

On the IBM 704, a single machine word was used to store a `cons` cell, and `car` and `cdr` were functions to extract the *C*ontents of the "*A*ddress part" and "*D*ecrement part" of a given machine *R*egister.

Counterparts in Scala:

`cons`	~	`::`
`nil`	~	`Nil`
`null?`	~	`isEmpty`
`car`	~	`head`
`cdr`	~	`tail`