# 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$_1$ ... x$_n$) is represented by

(cons x$_1$ (... (cons x$_n$ 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` |

## Lists and functions

Since Lisp has no static type system, lists can be represented using only functions and a unique symbol none:

```
nil          =   (lambda (k) (k 'none 'none))
(cons x y)   =   (lambda (k) (k x y))
(car l)      =   (l (lambda (x y) x))
(cdr l)      =   (l (lambda (x y) y))
(null? l)    =   (l (lambda (x y) (= x 'none)))
```

The idea is to represent a cons cell as a function that takes another function k as a parameter.

The function k is used to deconstruct the list.

The function cons simply applies k to its arguments.

car and cdr then simply apply the function cons to the appropriate deconstructor functions.

## Lists and functions (cont)

This construction shows that data can, in principle, be constructed using only pure functions.

But in practice, a `cons` cell is represented by a pair of pointers.

## An Example

Here is the definition and a use case of the `map` function in Scheme:

```
(define (map f xs)
  (if (null? xs)
      '()
      (cons (f (car xs)) (map f (cdr xs)))))
(map (lambda (x) (* x x)) '(1 2 3))
```

What does this expression evaluate to?

## Interpreters

Lisp is so simple that it is an ideal workhorse for studying the rules according to which a program is evaluated.

In particular, it is rather easy to write an interpreter for Lisp.

This is what we are about to do.

More precisely, we are going to implement Scheme--, a restricted version of Scheme, using Scala as the implementation language.

We have defined new languages several times before in this course. For example, we defined

- a language for arithmetic expressions,
- a language for digital circuits, and
- a constraint language.

These languages were implemented as libraries of Scala functions.

## Interpreters (cont)

In the case of Scheme--, there are two novelties:

- We will implement a complete programming language that is capable of expressing any algorithm (Turing complete).
- This language will have an external syntax that resembles Lisp, not Scala. This syntax will be translated into an internal Scala data structure through a parser.

The implementation consists of three steps.

1. Define an internal representation for Scheme-- programs.
2. Define a translator from strings to the internal representation.
3. Define an interpreter capable of evaluating the internal representation.

## Internal representation for Scheme--

Our internal representation for Scheme-- strictly follows the data structures used in Scheme. It also reuses the corresponding Scala constructs whenever possible.

We define a type `Data`, representing data in Scheme, as an alias of the type `scala.Any`.

```
type Data = Any
```

- Numbers and strings are represented as values of the Scala types `Int` and `String`,
- lists are represented by Scala lists,
- symbols are represented as instances of the Scala class `Symbol` (see below).

## Scala Symbols

- Symbols in Scala are values that represent identifiers.
- There is a simplified syntax for such symbols: if `name` is an identifier, then

  ```
  'name
  ```

  is a shorthand for `Symbol(”name”)`.

The standard class `Symbol` is defined as follows in the package `scala`:

```
case class Symbol(name: String) {
    override def toString() = ”'” + name
}
```

## Example: a Scheme-- program defining and using the factorial

```
(def factorial
     (lambda (n)
       (if (= n 0)
           1
           (* n (factorial (- n 1)))))
  (factorial 5))
```

Its internal representation can be constructed as follows:

```
List('def, 'factorial,
     List('lambda, List('n),
          List('if, List('=, 'n, 0),
               1,
               List('*, 'n, List('factorial, List('-, 'n, 1)))))),
     List('factorial, 5))
```

## From text to internal representation

We are now going to write a parser that constructs an internal representation for a given string.

This construction consists of two steps:

1. from a string to a sequence of words (called lexemes or tokens)
2. from a sequence of words to a `Data` tree.

Here, a token can be

- an opening parenthesis “`(`” or a closing parenthesis “`)`”,
- a sequence of characters that does not contain any white space characters or parentheses.

Tokens that are not parentheses must be separated by white space, that is, spaces, new line characters or tabs.

## The tokenizer

We represent the sequence of words that make up a Lisp program using an iterator class, defined as follows:

```
class LispTokenizer(s: String) extends Iterator[String] {
    private var i = 0
    private def isDelimiter(ch: Char) = ch <= ' ' || ch == '(' || ch == ')'
    def hasNext: Boolean = {
      while (i < s.length() && s.charAt(i) <= ' ') { i = i + 1 }
      i < s.length()
    }
    ...
```

*Remarks:*

- The iterator holds a private variable `i` that denotes the index of the next character to be read.

## The tokenizer (cont)

- The function `hasNext` skips over all the white space preceding the next token (if it exists). It uses the method `charAt` from the Java class `String` to access the characters of a string.

## The tokenizer (cont)

```
  ...
  def next: String =
    if (hasNext) {
      val start = i
      var ch = s.charAt(i); i = i + 1
      if (ch == '(') "("
      else if (ch == ')') ")"
      else {
        while (i < s.length() && !isDelimiter(s.charAt(i))) { i = i + 1 }
        s.substring(start, i)
      }
    } else error("more input expected")
}
```

- The function `next` returns the next token. It uses the method `substring` of the class `String`.

## Parsing and construction of the tree

Given the simplicity of the Lisp syntax, it is possible to parse it without the use of advanced parsing techniques.

This is how it works:

```
def string2lisp(s: String): Data = {
  val it = new LispTokenizer(s)
  def parseExpr(token: String): Data = {
    if (token == ”(”) parseList
    else if (token == ”)”) error(”unmatched parenthesis”)
    else if (token.charAt(0).isDigit) token.toInt
    else if (token.charAt(0) == '\”'
             && token.charAt(token.length()-1)=='\”')
      token.substring(1,token.length - 1)
    else Symbol(token)
  }
```

## Parsing and construction of the tree (cont)

```
  def parseList: List[Data] = {
    val token = it.next
    if (token == ")") Nil else parseExpr(token) :: parseList
  }
  parseExpr(it.next)
}
```

*Remarks:*

- The function `string2lisp` converts a string in a tree expressions of type `Data`.
- It starts by defining a `LispTokenizer` iterator called `it`.
- This iterator is used by the two mutually recursive functions `parseExpr` and `parseList` (recursive descent parsing).
- `parseExpr` parses simple expressions.
- `parseList` parses a list of expressions enclosed by parentheses.

## Parsing and construction of the tree (cont)

Now, if we write the following in the Scala REPL:

```
string2lisp("(lambda (x) (+ (* x x) 1))")
```

we obtain (without the indentation)

```
    List('lambda, List('x),
      List('+,
        List('*,'x,'x),
        1))
```

## Exercise

Write a function `lisp2string(x: Data)` that prints Lisp expression in Lisp format. For example

```
lisp2string(string2lisp(”(lambda (x) (+ (* x x) 1))”))
```

should return

```
(lambda (x) (+ (* x x) 1))
```

## Special forms

Our Scheme-- interpreter will be capable of evaluating a single expression only.

This expression can have one of the following special forms:

- `(val x expr rest)`
  evaluates `expr`, binds the result to `x` and then evaluates `rest`. Analogous to `val x = expr; rest` in Scala.
- `(def x expr rest)`
  binds `expr` to `x` and then evaluates `rest`. `expr` is evaluated every time `x` is used. Analogous to `def x = expr; rest` in Scala.

The real Scheme contains a variety of binders called `define`, for the top level, and `let`, `let*` and `letrec` used inside combinations.

Scheme's `define` and `letrec` correspond roughly to `def`, while Scheme's `let` corresponds roughly to `val`, but their syntax is more complicated.

## Special forms (cont)

- `(lambda (`$\texttt{p}_1 \ldots \texttt{p}_n$`) expr)`
  defines an anonymous function with parameters $\texttt{p}_1 \ldots \texttt{p}_n$ and body `expr`
- `(quote expr)`
  returns `expr` without evaluating it.
- `(if cond then-expr else-expr)`
  is the usual conditional.

## Syntactic sugar

Some other forms can be converted into the above through transformations on the internal representation.

We can write a function `normalize` that eliminates these other special forms from the tree.

For example, Lisp supports the special forms

```
(and x y)
(or x y)
```

for the short-circuited logic relations `and` and `or`. The `normalize` function can convert them to `if` expressions as follows:

## Syntactic sugar (cont)

```
def normalize(expr: Data): Data = expr match {
  case 'and :: x :: y :: Nil =>
    normalize('if :: x :: y :: 0 :: Nil)

  case 'or :: x :: y :: Nil =>
    normalize('if :: x :: 1 :: y :: Nil)

  // other simplifications...
}
```

## Derived forms

Our normalization function accepts the following derived forms:

```
(and x y)                    => (if x y 0)
(or x y)                     => (if x 1 y)

(def (name args_1 ... args_n) => (def name
   body                            (lambda (args_1 ... args_n) body)
 expr)                            expr)

(cond (test_1 expr_1) ...     => (if test_1 expr_1
      (test_n expr_n)              (...
      (else expr'))                  (if test_n expr_n expr')...))
```

## Summary

Lisp is a rather unusual language.

- It is deprived of an elaborate syntax and a static type system.
- It represents all data structures using lists.
- It treats programs and data alike.

These points have advantages and drawbacks.

- No syntax:
  $+$ easy to learn, $-$ hard to read.
- No static type system:
  $+$ flexible, $-$ error-prone.
- The only data structures are lists:
  $+$ flexible, $-$ poor data abstraction.
- Programs are data:
  $+$ powerful, $-$ hard to guarantee safety.