Week 11: Lisp

We now discuss the principles of another functional language: Lisp.

Lisp was the first functional language to be implemented; it dates back to 1959/60.

The name is an acronym for List processor.

At the time, Lisp was designed to deal with data structures for symbolic computation, such as lists or trees. (The languages that existed at the time only dealt with arrays).

Lisp has been used to implement many large applications. Examples:

- Macsyma, the first computer algebra program
- AutoCAD, a popular CAD program

Lisp Variants

In its 40+ years of existence, Lisp has evolved, and many dialects exist today.

The three most widespread ones are:

- Common Lisp (commercial, big),
- Scheme (academic, clean),
- Elisp (emacs extension language)

We treat here only a subset of Scheme.

The subset we present is purely functional – there are no variables or assignments. (Real Lisp has these and several other things as well).

Peculiarities of Lisp

Compared to Scala, there are four main points where Lisp differs and which make it interesting to study:

- No syntax, programs are simply nested sequences of words.
- No (static) type system.
- Only one form of compound data: the cons cell (from which lists are built).
- Programs are lists as well, so they can be constructed, transformed and evaluated by other program.

As an example of a Lisp program, consider `faculty`:

```
((define (faculty n)
   (if (= n 0)
       1
       (* n (faculty (- n 1)))))
 (faculty 10))
```


Observations:

- A compound Lisp expression is a sequence of sub-expressions in parentheses. Such an expression is also called a *combination*.
- Every subexpression is either a single word (an *atom*) or a compound expression.
- Subexpressions are separated by space characters.
- The first subexpression in a compound expression denotes an *operator* (usually a function).
- The other expressions denote the *operands*.

Special Forms

Some combinations look like function applications but aren't. In particular:

`(define name expr)`	defines `name` as an alias of the result of `expr` in the program that follows. analogous to Scala's ***def*** *name* = *expr*.
`(lambda (params) expr)`	the unnamed function that takes parameters `params` and returns `expr`, analogous to Scala's (*params* ⇒ *expr*).
`(if cond expr1 expr2)`	returns result of `expr1` if `cond` evaluates to true, or the result of `expr2` otherwise.

Combinations like these are called *special forms*.

Function Applications

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

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

This is essentially all. It is hard to imagine a useful programming language with fewer rules.

In fact, Lisp is so simple because it was meant as an intermediate language for computers, not humans.

It was originally planned to add a fancier Algol-like syntax on top of it, which would be translated to the intermediate form by a preprocessor.

However, humans got used to Lisp syntax rather quickly (at least some of them), and started to appreciate its advantages.

So the fancier syntax was never introduced.

Lisp Data

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

- Numbers are either floating point or integers. Integers have arbitrary size (no overflow!).
- Strings are as in Java.
- Symbols are simple sequences of characters without enclosing apostrophes. Examples:

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

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

There is no separate type of booleans. Instead, false is represented as the number *0*, whereas any value different from *0* is interpreted as true.

Lists in Lisp

Lists are written like combinations, e.g.

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

Note that lists are heterogeneous; they can have elements of different types.

Note also that we may not evaluate a list like that, since its first element is not a function.

To prevent lists from being evaluated, we use the special form `quote`.

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

The argument of `quote` is returned as the result without being evaluated itself. `quote` can be abbreviated with the special character '.

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

Lists Internals

As in Scala, Lisp's list notation is just syntactic sugar.

Internally, lists are formed from:

- The empty list, written `nil`.
- Pairs of head `x` and tail `y`, written `(cons x y)`.

The list (or combination) `(`x_1 `...` x_n`)` is represented as

`(cons` x_1 `( ... (cons` x_n `nil) ... ))`

Lists are accessed using the three operations

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

The names `car` and `cdr` point back to the IBM 704, the first computer on which Lisp was implemented.

On this computer, `CAR` meant "contents of address register" and `CDR` meant "contents of decrement register".

These two registers stored the two halves of a `cons` cell in the IBM 704 implementation.

In comparison to Scala,

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


Lists and Functions

Because Lisp has no static type system, we can represent lists using just functions, and a single symbol `none`. Here's how:

```
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 that a cons-cell can be represented as a function which takes another function `k` as parameter.

The `k` function is intended to decompose the list.

The `cons` function simply applies `k` to its arguments.

Then, `car` and `cdr` simply apply the `cons` function to the appropriate decomposition functions.

This construction shows that in principle all data can be constructed from pure functions.

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

An example

Here is the definition and usage of `map` in Lisp:

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

What is the result of evaluating this expression?

Interpeters

Lisp is so simple that it is an ideal vehicle for studying program evaluation rules.

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

This is what we are going to do now.

More precisely, we will implement Scheme−−, a small but complete subset of Scheme, using Scala as the implementation language.

Defining new languages is something we did often in this course. Examples were:

- A language for arithmetic expressions,
- A language for digital circuits,
- A language for constraints.

All these languages were implemented as libraries of Scala functions.

Two things are new now:

- We will implement a full-blown programming language, which itself can compute any algorithm (Turing complete).
- The language will have an external syntax which looks like Lisp, not Scala.

  That syntax will be mapped to an internal data structure in Scala by a parser program.

The implementation will proceed in three stages.

1. Define internal representation of Scheme−− programs.
2. Define a translator from text strings to internal representations.
3. Define an interpreter program that evaluates internal representations.

Internal Representation of Scheme−−

Our internal representations of Scheme−− closely follow the data structures used in Scheme. They also re-use as much as possible the equivalent constructs in Scala.

We define a type *Data* to represents Scheme data as an alias of *scala.Any*.

```
type Data = Any;
```

- Numbers and strings are represented as in Scala as elements of type *int* and *String*.
- Lists are represented as Scala lists.
- Symbols are represented as instances Scala's *Symbol* class (see below).

Scala Symbols

- Symbols in Scala are values that represent identifiers.
- There is a simplified syntax for such symbol literals: If *name* is a lexical identifier, then

  'name is a shorthand for *Symbol(name)*.

  The standard class *Symbol* is defined in package *scala* as follows.

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


Example: The `faculty` function is written in Scheme−− as:

```
(define faculty (lambda (n)
  (if (= n 0)
     1
     (* n (faculty (- n 1))))))
```

Its internal representation is given by:

```
List('define, 'faculty, List('lambda, List('n),
  List('if, List('=, 'n, 0),
    1,
    List('*, 'n, List('faculty, List('-, 'n, 1)))))))
```


From Strings to Internal Representations

We now develop a parser program that maps strings into internal representation.

The mapping proceeds in two stages.

- From a string to a sequence of words (or: tokens)
- From a sequence of words to a *Data* tree.

Here, a token is one of the following.

- A left or right parenthesis "(", ")".
- A sequence of characters not containing whitespace or parentheses.
- Tokens which are not parentheses need to be separated by whitespace, i.e. blanks, new lines, or tabulator characters.

Tokenizing

We will represent the sequence of words in a Lisp program as an iterator.

The iterator is defined by the following class:

```
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()
  }
  ...
```

Notes:

- The iterator keeps a private variable *i*, denoting the index of the next character to be read.

- The *hasNext* function skips any whitespace in front of the next token (if one exists). It uses the *charAt* method in Java's *String* class to access characters in the string.

```
...
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("premature end of string")
}
```

- The *next* function returns the next token. It uses the *substring* method in class *String*.

Parsing and Tree Construction

Given the extremely simple syntax of Lisp, it is possible to parse it without any advanced parsing techniques.

Here is how it is done.

```
def string2lisp(s: String): Data = {
  val it = new LispTokenizer(s);
  def parseExpr(token: String): Data = {
    if (token == "(") parseList
    else if (token == ")") error("unbalanced parentheses")
    else if (Character.isDigit(token.charAt(0))) Integer.parseInt(token)
    else if (token.charAt(0) == '\"' && token.charAt(token.length()−1)=='\
      token.substring(1,token.length() − 1)
    else Symbol(token)
  }
```



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

Notes:

- The *string2list* function converts a string into a *Data* expression tree.
- It first defines a *LispTokenizer* iterator *it*.
- This iterator is used by the two functions *parseExpr* and *parseList*, which call each other recursively (recursive descent parser).
- *parseExpr* parses a single expression.
- *parseList* parses a list of expressions in parentheses.

Now, if we write in the Scala interpreter:

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

we get (without the indentation):

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

Exercise: Write a function *lisp2string*(*x*: *Data*) which prints Lisp expressions in Lisp form. I.e.

lisp2string(*string2lisp*("(lambda (x) (+ (* x x) 1))"))

should yield

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


Special Forms

Our Lisp interpreter will only be able to implement a single expression.

That expression can have the following special forms:

1.

```
(val x expr rest)
```

Evaluates `expr`, binds the results to `x`, and then evaluates `rest`. This is analogous to Scala's ***val*** *x* = *expr; rest*.

2.

```
(def x expr rest)
```

Binds `x` to `expr`, and then evaluates `rest`. `expr` is evaluated each time `x` is used. This is analogous to Scala's ***def*** *x* = *expr; rest*.

Real Lisps have instead a range of binders called `define` for the toplevel and `let`, `let*` and `letrec` inside combinations.

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

3.

```
(lambda (p_1 ... p_n) expr)
```

Defines an anonymous function with parameters $\mathtt{p}_1, \ldots, \mathtt{p}_n$ and body `expr`.

4.

```
(quote expr)
```

returns `expr` without evaluating it.

5.

```
(if cond then-expr else-expr)
```

The usual if-then-else construct.

Removing Syntactic Sugar

Other forms can be mapped to these by transforming the internal representation of Lisp data.

One can write a function *normalize* that eliminates other special forms in trees.

For instance, Lisp supports the special form

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

for short-circuited and and or. The *normalize* procedure would map these to if-then-else expressions as follows.

$$
\begin{aligned}
&\textbf{def}\ normalize(expr\colon Data)\colon Data = expr\ match\ \{\\
&\quad \textbf{case}\ \text{'and} :: x :: y :: Nil \Rightarrow\\
&\qquad normalize(\text{'}\textbf{if} :: x :: y :: 0 :: Nil)\\
&\quad \textbf{case}\ \text{'or} :: x :: y :: Nil \Rightarrow\\
&\qquad normalize(\text{'}\textbf{if} :: x :: 1 :: y :: Nil)\\
&\quad ...\ //\ further\ simplifications\ go\ here\\
&\}
\end{aligned}
$$

Derived Forms

Our normalization function supports 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 quite an unusual language.

- It does away with elaborate syntax and static type systems.
- It reduces all compound data to lists.
- It identifies programs and data.

These points have advantages as well as disadvantages.

- No syntax:
  \+ easy to learn, − hard to read.
- No static type system.
  \+ flexible, − easy to make mistakes.
- Only lists as compound data:
  \+ flexible, − poor data abstraction.
- Programs are data:
  \+ powerful, − hard to keep safe.