LISP

- Introduction (some words about history...)
- Theoretical Assumptions−Bridging the Gap
- Overview of Common Lisp
- Lisp versus ML
- Conclusion

*

For the lecture "Functional Programming"
held by Dr.Ute Schmid
at the University of Osnabrueck in winter term 2001/02
as an abstract for a brief report about Lisp, by Martin Beckmann

Introduction

****

Lisp is the second oldest programming language that's still in widespread use today (Fortran is oldest). John Mc Carthy (MIT, then Stanford University) is said to be its inventor. Development began in the 1950s at IBM as *FLPL − Fortran List Processing Language.*

In April 1960 Mc Carthy published a paper *"in which he did for programming something like what Euklid did for geometry"* [PG02] *:

"Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I".

The intent of this paper was to provide a basic mathematical and notational framework for the *LISt Processor* implementation developped for the IBM 704 computer by the A.I. Group at MIT. This concrete implementation was described in quite technical terms later in the article.

The motivation for inventing Lisp was a relatively pragmatic one: to realize a system called *The Advice Taker* a language was needed which would be able to treat functions as data (bluntly spoken, because I do not refer to the much stricter mathematical meaning of 'function'), or, as Mc Carthy has put it in the Introduction:

"The main requirement was a programming system for manipulating expressions representing formalized declarative and imperative sentences so that the Advice Taker system could make deductions."

Another important feature of the new language was ***Garbage Collection*** which took the burden from the programmer to assign and free memory cells explicitely. In fact, Garbage Collection was first exhibited by Lisp. [1]

---

1 * ... What Graham means, is building a large, powerful and infinitely extensible language from small pieces. Okay, this is an advantage of FP in general and also of OOP; but keep in mind that Lisp was the first...

Theoretical Assumptions − Bridging the Gap

****

Since Mc Carthy's paper mentioned above seems important to me (and by the way is interesting to read) I will try to give a brief summary of the theoretical issues it's concerned with:

First, he explains some (well−known) mathematical concepts:

- Partial functions

- Propositional expressions and predicates

- Conditional expressions ... *(p1 −> ε1,...,pn −> εn)*
  (he then defines a computational ***noncommutative*** *interpretation of the logical connectives:*
  $p \wedge q = (p{->}q, T{->}F)$
  $p \vee q = (p{->}T, T{->}q)$
  $\neg p = (p{->}F, T{->}T)$
  $p \supset q = (p{->}q, T{->}F))$

- Recursive functions

- Distinction between functions and forms:
  A ***form*** *is something like* y^2+x *...let it be ε. The notation for a* ***function*** *is Church's lambda−notation:*
  If ε is a form in variables x1,...,xn then λ((x1,..,xn), ε) is the corresponding function...For the example above and x=3, y=4 follows: $\lambda((x,y),y^2+x)(3,4) = 19$

2

- A way to express recursive functions in lambda−notation:
  Consider the following example:
  The recursive function n! = (n=0 −> 1, T −> n(n−1)!)*
  should be translated into a lambda−expression.
  !=λ((n),(n=0−>1,T−>n(n−1)!)) would be wrong, because*
  there is no clear reference from '!' inside the lambda clause
  to the expression as a whole.
  Solution: A new notation is introduced. ***label(a,***ε) *where* ε denotes an expression and a refers to the expression as a whole. a may occur in ε. For the example given follows:
  label(!,λ*((n),(n=0−>1,T−>n*(n−1)!)))*

Second, Mc Carthy defines a class of symbolic expressions the so−called S−expressions:

- S−expressions are composed of the special characters
  (...*start of a composed expression*
) ...*end of a composed expression*
  • ...*composition*
  and of "an infinite set of distinguishable atomic symbols"
  e.g.: *A*
  ABA
  APPLE_PIE_NUMBER_3

- ***1. Atomic symbols are S−expressions***
  2. The composition (ε1•ε2***) is an S−expression iff*** ε1 ***and*** ε2 ***are.***

- The list notation (m1,m2,...,mn) is an abbreviation for
  (m1•(m2•(...(mn•NIL)...)))
- *(where NIL is a special atomic constant)*

3

Third, a class of Meta−expression (M−expressions) is defined:

- M−expressions represent functions over S−expressions.

- Brackets, semicolons and lowercase letters are used to distinguish between S−expressions and M−expressions.

- Example: atom[x] ...
  atom[X] =T
  atom[(X•A)] =F

- Note that the S−expressions are a subset of the M−expressions (just like numbers are a subset of the set of functions over numbers)!

Fourth, a transformation mechanism is defined that translates an M−expression ε into an S−expression ε*:

- The mechanism is the following:
  1. If ε is an S−expression then ε* is (QUOTE, ε).
  2. Lowercase letters in expressions in ε are translated to the corresponding uppercase letters. Thus atom* is ATOM.
  3. An M−expression f[ε1,..., εn] is translated to (f*, ε1*,...,εn*).
  4. {[p1−> ε1;...;pn−> εn]}* is (COND,(p1*, ε1*),...,(pn*, εn*)).
  5. {λ[[x1;...;xn]; ε]}* is (LAMBDA,(x1*,...,xn*), ε*).
  6. {label[a; ε]}* is (LABEL, a*, ε*).

4

Given the opportunity to unify the Symbol− and the Meta−level, expressions over symbols can be treated exactly in the same way as symbols themselves. Functions become data. This bridging, which was the intention for developing Lisp is finally reached and remarkably not only for primitive symbols but also for higher−level symbols and the corresponding Meta−levels.

Fifth, the S−function (=M−expression, which now can be rewritten as S−expression as well, of course) *apply* is defined, which "*plays the theoretical role of a universal Turing machine and the practical role of an interpreter.*"(Mc Carthy)

apply[f;args] itself invokes some other basic Lisp−functions, which are nowadays known as *special forms* (or *special form operators*) such as *eval, appq, cons, list, quote, eq, car, atom,...* . (Some of them will be introduced later.)
apply as a whole is to complex to be shown here. Read Mc Carthy's paper for further information.

5

Overview of Common Lisp
****

"No doubt about it. Common Lisp is a *big* language."
(Guy L. Steele, 1990)

Common Lisp is a family of Lisp dialects, including for example the Lisp standard *ANSI Common Lisp*. There exist several implementations of Common Lisp variants, for example *GNU CLisp* which I used for test purposes. Additionally probably thousands of extensions for Common Lisp can be found, many of which actually define a new language or a new dialect, as will become obvious later in this section.

But this shouldn't lead to the illusion, that "pure" Common Lisp were a compact or fast learnable language. It adds a plethora of new functions, datatypes and special forms (at least macros which appear like special forms) to the relatively small language set presented by Mc Carthy in 1960.

In fact Backus' criticism of the Lisp reality is not unjustified:
"Pure Lisp is often buried in large extensions with many von Neumann features."

On the other hand many of these features allow for fast development processes and for a semantic specificity which should not be regarded as just a question of style.

And: one must not forget that even the most complex constructs (like *loop*) are defined in terms of smaller, already verified ones. Thus the very majority of them is actually written *in* Lisp.

Steele's "Common Lisp − the Language. 2nd edition" *is known to be* the "bible" for Common Lisp.

6

LISP − the LISt Processor:

Following directly from the the theoretical assumptions summarized in the last section most input (in pure Lisp *all* input except atoms) is passed to the interpreter or compiler in the form of lists which are just a syntactic simplification of S−expressions.
S−expressions are either atomic or a composition of two S−expressions.
Lists are either the empty list − *NIL* or a composition of an element and a list − returned by *(cons element list)*. This is in principle the same recursive definition.
In the earliest Lisp the elements of a list were separated by commas. However this changed during Lisp's evolution. Nowadays blanks are used instead of commas.
Since lists are used to represent both − functions and data it is important to know how Lisp evaluates expressions...

Evaluation rules:

Lisp uses *strong evaluation* but *weak typing*.
Symbol naming is not case−sensitive.

I) Every expression is either an atom or a list.
II) Every atom is either a non−symbol or a symbol
III) Every list is either a function to be evaluated or a special form

>II) A *non−symbol* evaluates to itself. Examples are numbers and strings.

```
[1]> 13.7
13.7
```

A *symbol* names a variable and evaluates to the value most recently assigned to it.

```
[1]> (setf symbol "a symbol")
"a symbol"
[2]> symbol
"a symbol"
```


7

>III) A function call is a list of the form (function−name *arguments*)

```
[1]> (1 2 3)
*** – EVAL: 1 is not a function name
```

...this fails because Lisp tries to treat 1 as a function−name
but...

```
[1]> (+ 1 2 3)
6
```

...succeeds. *

A special form is a list whose first element is a special form operator. It is evaluated according to its own evaluation rules. An important special form is quote, abbreviated by a ’
The argument of quote is not evaluated and simply evaluates to itself. This is important to represent data.

```
[1]> (quote John)
JOHN
[2]> ’John
JOHN
[3]> ’(John)
(JOHN)
[4]> (setf p (second ’("one" "two" "three")))
"two"
```

setf is also a special form − p is not evaluated

```
[5]> (setf (second ’(one two three)) "two"))
"two"
```

the left hand side of an assignment may be any expression that evaluates to a symbol

```
[6]> p          but:      [7]> ’p
"two"                     P
```

A special form for defining named functions:

```
[1]> (defun faculty (n) (cond ((= n 0) 1)
                              (t  (* n (faculty (− n 1))))))
FACULTY
```

cond is also a special form, denoting conditional expressions...

```
[2]> (faculty 4)
24
```


8 *...That’s the reason why all function calls in Lisp have to be in prefix notation.

Some words about Datatypes in Lisp

The first Lisp only dealt with symbols, numbers and lists.
More modern Lisps added new datatypes.
In principle *every* expression has a type.
Remember: All expressions can be treated as data.
Data needs type. (...and type information is data, thus the type of of a type as datum is of type *type*)

There is a type *function*. An object of type *function* can be accessed by *#'function_type_object.*

Arrays include *vectors* and higher−dimensional arrays.
*Vectors (*one dimensional *arrays)* and *lists* were subsumed under the *sequence* type.
A *cons* is a non−NIL *list*.
The empty *list* NIL is the only object of type *null*.
A *string* is a special *vector* of *characters.*
A *vector* can be created by #(e1 e2 ... en).

Numbers can be *integers* or *floats*. *Integers* can be *fixnums* (16 bit integer) or *bignums* (unlimited precision integers).

Symbols can be *functions* and variables.
An *atom* is everything that is not a *cons.*

There are predicates for testing whether an expression is of a certain type.
e.g. *(numberp 3)* or *(typep 3 'number)* evaluates to T.
And there is a function that returns the most specific type of an expression: *type−of.* **But take care: The type hierarchy in Common Lisp is not simply a tree but quite a complex graph.**

It stays to mention that there is no boolean type. The *null*−object *NIL* stands for *false* and every other expression for *true* (but by *convention* the symbol−constant *t* is used to represent truth).

9

Anonymous function objects − the Lambda notation:

Anonymous functions can be generated in Lisp by the help of the *lambda* form.

The general pattern of a lambda expression is

(lambda (parameters...) body...).

The *whole* expression is just a *non−atomic name* for a function.

Thus

```
[1]> ((lambda (x) (* x x)) 4)
16
```

is a valid expression, whereas *(lambda (x) (* x x))* is not (*lambda* is not a function−name). *

1.) Lambda expressions avoid superfluous naming in programs.

2.) more important...
Lambda can be used to create new functions at run−time:

Consider the following example:

```
[1]> (defun adder (c)
"return a function that adds c to its argument"
#'(lambda (x)(+ x c)))
ADDER
[2]> (mapcar (adder 3) '(1 2 3))
(4 5 6)
```

This is what is called a* closure *of course.

*(...*mapcar *is a a higher−order function which maps a function on a list of arguments...)*

10 *...some interpreters treat this expression in a way that evaluates to an inaccessible closure.

Lisp versus ML
****
Both − Lisp and ML can be regarded as functional languages and thus both offer the general FP features (functions as data, modularity, no need to use the Hoare calculus for program verification...)
However, although Lisp is much older and it can be assumed that the experience gained from the fundamental concepts of Lisp and and their application in practice have influenced the design of ML, it would be too easy to say, ML was the superior language.

In Lisp lists are the dominating structure throughout the language, whereas lists in ML are just a (nevertheless important) datatype.

Lisp uses *Strict Evaluation* − in contrast ML in general performs *Strict Evaluation* as well, but in a few subdomains *Lazy Evaluation* is used, which has an advantage in efficiency but can lead to problems and semantic confusion when functions are called that return *unit* and are just written for their side−effects (e.g. print).

ML and Lisp behave differently in what concerns typing. ML expects *strong typing* and for flexibility allows for *type polymorphism*. Nevertheless when a polymorphic type (*'a,'b,*...) is assumed by the type inferencer, this happens at *compile−time* and the polymorphic type can be regarded as a *dummy type*. Lisp on the other hand uses *weak typing* or *dynamic typing*, what means a type ambiguity stays *really* unresolved until *run−time*.

Following directly from the fact that data *and* function calls are represented by lists Lisp necessarily allows for *lists with mixed element types* (contrasting ML). This can cause problems concerning semantics but also enables the programmer to represent arbitrary sets in a more straightforward way.
There are two points where ML definitely *is* superior to Lisp:
1) The powerful *functor concept* is totally unknown to Lisp. What is called a *structure* in Lisp is called a *record* in ML and in most other programming languages.

11

2) ML offers *pattern−matching* but there is *none* in Lisp, at least not by default. However, you can define pattern matching manually.
Here is a simple *pattern matcher* realized as a predicate function and implemented by Peter Norvig: *reference page 17

```
(defun pat-match (pattern input)
    "Does pattern match input?
     Any variable can match anything."
    (if (variable-p pattern)
        t
        (if (or (atom pattern) (atom input))
            (eql pattern input)
            (and   (pat-match (first pattern)
                              (first input))
                   (pat-match (rest pattern)
                              (rest input))))))
```

On the other hand there are some useful Lisp features not contained in ML:

1) Macros:

It is impossible to the programmer to define new *real special forms.* But you can add new constructs that *behave* like special forms: so called *macros*.
This is something you should only choose to do if there is a significant advantage in readability or semantic specificity because when using own macros you are somehow leaving the Lisp terrain by modifying syntax.
Macros *expand* into special forms or other macros.

Four steps for the development of macros [PN]:
- Decide if the macro is really necessary
- Write down the syntax of the macro
- Figure out what the macro should expand into
- Use defmacro to implement the syntax/expansion correspondence

12

Example:

a new macro for a while loop...[PN]

*The desired syntax is *(while test body...).*

*The macro should expand into *(loop*
(unless test (return nil))
body...).

*The macro is implemented in the following way:

```
(defmacro while (test &rest body)
  "Repeat body while test is true."
  (list*   'loop
           (list 'unless test '(return nil))
           body))
```

Now the macro can be used:

```
>(setf i 7)
7
>(while (< i 10)
    (print (* i i))
    (setf i (+ i 1)))
49
64
81
NIL
```

...
"...Besides, when someone asks, 'What did you get done today?' it sounds more impressive to say 'I defined a new language and wrote a compiler for it' than just to say 'I hacked up a couple of macros.'"
(Peter Norvig)

13

Functions with optional parameters:

Lisp allows for functions with *optional parameters* which may be or may not be given in a function call. This is from the mathematical perspective quite problematic but it can be useful.

Consider the following example where a *linear recursion* is transformed to a *tail−recursion*:

```
(defun length (list)
  "Compute the length of a list"
  (if  (null list)
    0
    (+ 1 (length (rest list)))))
```

can be made tail−recursive without the need to write an auxiliary function:

```
(defun length  (list &optional (len−so−far 0))
  (if  (null list)
       len−so−far
       (length (rest list) (+ 1 len−so−far))))
```

The optional parameter len−so−far here is set to 0 by default.
If no default value is specified Lisp assumes NIL.

Functions with multiple return values:

The following is *very* problematic in mathematical respect: Lisp offers the opportunity to write functions that return ***really*** *multiple results*
(not just *tuples* or *lists*)!

An example is the built−in function *round* which returns in an ordinary context the nearest *integer* to a given *float*. But in a special context, for example when it is called from inside the form *multiple−value−bind* it returns *two* results − the rounded *integer* ***and*** the *remainder*.

This is a *dubious* feature in my eyes, so I don’t describe it any further.*[14]

14 *...However, I added an example on page 17.

Conclusion

****

In the paper *"**Lisp − Notes on its Past and Future**"* first published in 1980 and revised in 1999, Mc Carthy himself takes stock of the development of Lisp and lists some possible reasons why Lisp and even more Lisp's functional approach could survive and spread out over the programming community especially over those people working in the field of A.I.

A small excerpt of what he names:

Every kind of data can be represented by a single general type (the list −−− the S−expression)

- *Composition of functions as a tool for forming more complex functions*
- *The recursive use of conditional expressions as a sufficient device for building computable functions*
- *The use of Lambda expressions*
- *"The representation of Lisp programs as Lisp data that can be manipulated [...] This has prevented the separation between system programmers and application programmers."*
- *"The conditional expression interpretation of boolean connectives"*
- *"The Lisp function eval that serves both as a formal definition of the language and as an interpreter"*
- *Garbage Collection*
- *Interactive programming environment*

Many of these features have been adopted by other functional and object oriented languages. Whether the more modern functional languages are superior to Lisp is a question I cannot judge about. But doubtless the invention of Lisp was a landmark in the history of information processing.

Apart from its ability to treat symbolic and super−symbolic worlds equally, the great thing about Lisp is that you can put an arbitrary number of abstraction levels one over the other and write functions on any of them. This is a fine quality for natural language processing,

(Read [PN] ch. 2 for an example of a simple generative English grammar...)

and thus in the last consequence for 'automatic' programming. At least in this approach of A.I. I suppose that Lisp is hardly to beat.

15

Annotations to pat−match, page 12

****

```
(defun variable-p (x)
  "Is x a variable (a symbol beginning with '?')?"
  (and (symbolp x) (equal (elt (symbol-name x) 0) #\?)))
```

```
>(variable−p j)
*** − EVAL: variable J has no value
> (variable−p ?j)
*** − EVAL: variable ?J has no value
> (symbolp u)
*** − EVAL: variable U has no value
> (symbolp 'u)
T
> (variable−p 'j)
NIL
> (variable−p '?j)
T
> (pat−match (cons 1 '?x) '(1 2 3))
T
> (pat−match (cons 3 '?x) '(1 2 3))
NIL
> (pat−match (cons 3 '?x) '(1 2 3))
NIL
> (pat−match (cons 1 '?x) (cons 1 2))
T
> (pat−match (cons 1 3) (cons 1 3))
T
```

Start tracing for* pat−match *and* variable−p*:*

```
> (trace pat−match variable−p)
;; Tracing function PAT−MATCH.
;; Tracing function VARIABLE−P.
(PAT−MATCH VARIABLE−P)
```

```
> (pat−match (cons 1 '?x) (cons 1 '(2 3 4)))
1. Trace: (PAT−MATCH '(1 . ?X) '(1 2 3 4))
2. Trace: (VARIABLE−P '(1 . ?X))
2. Trace: VARIABLE−P ==> NIL
2. Trace: (VARIABLE−P '1)
2. Trace: VARIABLE−P ==> NIL
2. Trace: (VARIABLE−P '?X)
2. Trace: VARIABLE−P ==> T
1. Trace: PAT−MATCH ==> T
T
```

16

An example for functions returning multiple values

****

```
(defun show-multiple-value-round (x)
"print rounded and remainder of x to terminal..."

  (if (null (multiple-value-bind (rounded remainder)
  ;if multiple-value-bind returns NIL...

       (round x)
       ;multiple-value-bind refers to (round x)

      (format t "~f is ~r plus ~f"
      ;print to terminal this pattern

     x rounded (/
     ;...applied to x, rounded and...
                  (round
                  (* 10000 remainder))
             10000))))
             ;...the simplified remainder.
  'bye))
  ;...return 'bye.
```

```
> (show-multiple-value-round 5.7)
5.7 is six plus -0.3
BYE
```

To write own multiple value functions use the built in function *values* :

```
> (values 'one 'two 'three)
ONE ;
TWO ;
THREE
```


17

Literature

****

[MC60] *John Mc Carthy, 1960. "Recursive Functions of Symbolic Expressions and Their Computation by Machine"*

[MC80] *John Mc Carthy, 1980,1999. "Lisp−Notes on its Past and Future−1980" (revised in 1999)*

[GLS] *Guy L. Steele Jr, 1990. "Common Lisp the Language 2nd edition"*

[PN] *Peter Norvig, 1992. "Paradigms of Artificial Intelligence Programming. Case Studies in Common Lisp"*

[PG] *Paul Graham, 2002. "The Roots of Lisp − a Draft"*

[LCP] *Larry C. Paulson, 1996. "ML for the Working Programmer 2nd edition"*