Functional Programming with Lisp

Klaus Köhler
Munich University of Applied Sciences
FH - München
E-Mail: koehler@hm.edu
Home Page: http://www.cs.hm.edu/~koehler

pgp Key ID: 0xBE42584D, Algorithm: DSA/ELG, Size: 1024/1792
fingerprint: 8D67 19C6 70C7 14D0 E33D 42AA AA36 42F3 BE42 584D
Contents

1. Introduction
2. Language Characteristics
3. Lisp Syntax
4. Predefined Functions
   with interspersed examples
5. User-defined Functions
6. Lisp Exercise: symbolic differentiation
Introduction

- **List Programming:**
  - List is the only data structure: `(LISP)`
  - Functions are lists: `(LAMBDA (L) (CAR (CDR L)) )`
- **Lots of Indecipherable Silly Parentheses**
- **Lots of Irritating Superfluous Parentheses:** see later
- **Lisp Characteristics:**

  *Another Glitch in the Call*
  (Sung to the tune of a recent Pink Floyd song.)

  We don't need no indirection
  We don't need no flow control
  No data typing or declarations
  Did you leave the lists alone?
  - Hey! Hacker! Leave those lists alone!
  **Chorus:**
  - All in all, it's just a pure-LISP function call.
  - All in all, it's just a pure-LISP function call.
Introduction (cont.)

- **Lisp History:**
  - 1960: Definition by J. McCarthy
    implementation of A. Church's $\lambda$-Calculus
  - 1965: LISP 1.5 by J. McCarthy
  - 1970: several dialects: ZetaLISP, Franz Lisp, InterLISP
  - 1975: Scheme by MIT
  - 1981: Common Lisp by DARPA, 1994 ANSI standard
  - 1980: Common Lisp Object-oriented System CLOS (multi-methods, inheritance)
  - 1980: LISP-Machine by Xerox

- "Successors" strongly influenced by LISP:
  - 1970: Logo (turtle graphics) by S. Papert
  - 1970: Meta Language ML (proof system LCF, type inference) by R. Milner
  - 1980: HOPE (type declaration, pattern matching) by R. Burstall and D. MacQueen
  - 1985: Miranda (recursion equations, higher order functions) by St. Andrews
  - 1988: Haskell (static typing, polymorphic data types) by P. Hudak and P. Wadler
  - 1989: Gofer (simplified Haskell for teaching) by M. Jones

[J. McCarthy]
Recursive Functions of Symbolic Expressions and their Computation by Machine
CACM, 3(4), Apr.1960, p. 184-195

[A. Church's]
The Calculi of Lambda-Conversion
Princeton University Press, Annals of Mathematical Studies, No. 6, 1941
Language Characteristics

- **Functional language**
  programming = composition of functions
  no side effects
- **"Implementation" of Lambda-calculus**
  universal programming language
- **Simple syntax**
  data and functions are lists
- **Interpreted language**
  self-modifying programs
- **Statically untyped language**
  no declaration of variables
  variables = function parameters
- **Intended use**: symbolic computation
  applied in AI
- **Great impact on all functional languages**
LISP Syntax

- **Data Types** (pure LISP: bold)
  
  S-Expression ::= Atom | List
  
  Atom ::= Literal | Boolean | Number
  
  e.g. Lisp T, NIL 2, 3.14
  
  List ::= ( { Atom | List } )
  
  e.g. (L I S P), (D(E F)((G H)I)), () = NIL

- **Variables**
  
  = function parameters = symbols (bound to a value through function calls)

- **Function calls** (prefix notation)
  
  = evaluated lists with
  
  – first element = function name
  
  – rest list = argument list
  
  e.g. (last 'L I S P) → P

- **Evaluation** of S-Expressions by interpreter
  
  – atoms evaluate to themselves
  
  – lists evaluated as function calls
  
  – variables evaluate to the value they are bound to
  
  – evaluation prevented by quote '
LISP Functions

- **List functions**  (pure LISP: **bold**)
  - **car**: select first element (head) of a non-empty list
    
    e.g. \((\text{CAR } '(L I S P)) \rightarrow L\)
  
  - **cdr**: select rest list (tail) of a non-empty list
    
    e.g. \((\text{CDR } '(L I S P)) \rightarrow (I S T)\)
  
  - **cons**: insert an element at the beginning of a list
    
    e.g. \((\text{CONS } 'L 'I S P)) \rightarrow (L I S T)\)
  
  - **list**: generate a list from elements
    
    e.g. \((\text{LIST } 'A 'B 'C ) \rightarrow (A B C)\)

- **Relations between list functions**
  
  \[
  \begin{align*}
  (\text{car } (\text{cons } X \text{ List})) &= X \\
  (\text{cdr } (\text{cons } X \text{ List})) &= \text{List} \\
  (\text{cons } (\text{car } \text{List}) \ (\text{cdr } \text{List})) &= \text{List} \\
  (\text{cons } X1 \ (\text{cons } X2 \ldots \ (\text{cons } Xn \text{ nil}) \ldots)) &= (\text{list } X1 \ X2 \ldots \ Xn)
  \end{align*}
  \]

\[
\begin{align*}
\text{CAR} &= \text{Contents of Address part of Register} \\
\text{CDR} &= \text{Contents of Decrement part of Register} \\
\text{CONS} &= \text{CONStruct}
\end{align*}
\]
LISP Functions (cont.)

- **Arithmetic functions**
  1+ or **ADD1** increment by 1
  e.g. (1+ 7) → 8
  1− or **SUB1** decrement by 1
  e.g. (1- 7) → 6
  * or **Times** multiply numbers
  e.g. (* 3 4 5) → 60
  +, −, / analogous
LISP Functions (cont.)

• **Predicate functions** (pure LISP: bold)
  - **ATOM** Is operand an atom?
    - e.g. (ATOM 'A) → T, (ATOM 7) → T, (ATOM '(3)) → NIL
  - **NULL** Is list empty?
    - e.g. (NULL '()) → T, (NULL '(LISP)) → NIL
  - **NUMBERP** Is operand a number?
    - e.g. (NUMBERP 'A) → NIL, (NUMBERP 7) → T
  - **LISTP** Is operand a list?
    - e.g. (LISTP 'A) → NIL, (LISTP '(L I S P)) → T

• **Compare functions** (pure LISP: bold)
  - **EQ** Are operands identical?
    - e.g. (EQ Value 'A) → T, if variable Value is bound to atom A,
      (EQ '(A) '(A)) → NIL
    - (sometimes EQ is not applicable to numbers ⇒ use EQL)
  - **EQUAL** Are operands equal?
    - e.g. (EQUAL '(Value) 'Value) → NIL,
      (EQUAL (CDR '(L I S P)) '(I S P)) → T
LISP Functions (cont.)

- **Logical functions**
  - **AND**
    - conjunction of S-expressions
    - evaluation of parameters from left to right
    - returns NIL, as soon as an argument evaluates to NIL,
      and the value of the last argument otherwise
    - e.g. (AND 'A ' () 7) → NIL, (AND 'A ' (L) 7) → A
  - **OR**
    - disjunction of S-expressions
    - evaluation of parameters from left to right
    - returns the value of the first argument ≠ NIL
    - e.g. (OR ' () 7 'A) → 7
  - **NOT**
    - negation of S-expression
    - argument value ≠ NIL is interpreted as T
    - e.g. (NOT 7) → T
    - (AND (EQUAL ' () 7) (OR (NUMBERP 7) (NOT T))) → NIL
User-defined LISP Functions

• **Named functions**
  Named-function ::= (DEFUN Name Parameter-list Body)
  Name ::= Literal
  Parameter-list ::= List
  Body ::= S-expression
  e.g. (DEFUN SECOND(L) (CAR (CDR L)))
       (SECOND '(L I S P)) → I

• **Anonymous functions**
  used for direct evaluation or as function argument
  Anonymous-function ::= (LAMBDA Parameter-list Body)
  e.g. ((LAMBDA (L) (CAR (CDR L))) '(L I S P)) → I
LISP Functions (cont.)

- Control function
  - COND conditional evaluation of S-expressions
    - nested ?: operator in C or nested if.. else if.. else if...
  - Cond-function ::= (COND Pair-list)
    - Pair-list ::= (Pair)
    - Pair ::= (S-Expression S-Expression)

- Examples
  - (DEFUN LENGTH (L) ; recursive function for length of list
    (COND ((NULL L) 0)
          (T (1+ (LENGTH (CDR L)))))
    )
  - (DEFUN LAST(L) ; recursive function for last element
    (COND ((NULL L) NIL) ; undefined for empty list
          ((NULL (CDR L)) (CAR L)) ; list of 1 element
          (T (LAST (CDR L)))
        )
    )
LISP Functions (cont.)

- **Evaluation functions**
  
  **QUOTE** or ' prevent evaluation of S-expression
  
  e.g. (QUOTE A) or 'A → A

  **EVAL** evaluate S-expression
  
  e.g. (EVAL '(1+ 7)) → 8

  **APPLY** apply a function to the elements of an argument list
  
  e.g. (APPLY 'TIMES '(3 4 5)) → 60
  
  XLisp: (APPLY (function *) '(3 4 5))
  (APPLY '1+ 7) is undefined
  
  (APPLY 'LAMBDA (L) (CAR(CDR L))) '((1 2 3)) → 2

- **Relation between evaluation functions**
  
  APPLY can be defined as
  
  (DEFUN APPLY (FN ARGS) (EVAL (CONS FN ARGS)))
**LISP Example:** Symbolic Differentiation

- **Grammar for infix expressions** (no operator priority: use brackets!)
  - `Expression ::= DyadicExpr | MonadicExpr | SimpleExpr`
  - `DyadicExpr ::= (SimpleExpr DyadicOperator SimpleExpr)`
  - `MonadicExpr ::= (MonadicOperator SimpleExpr)`
  - `SimpleExpr ::= Symbol | Number | (Expression)`
  - `DyadicOperator ::= + | - | * | / | ^`
  - `MonadicOperator ::= + | - | FunctionId`
  - `FunctionId ::= sin | cos | exp | log`

- **Examples:**
  - `Pi`
  - `x`
  - `13`
  - `(- 13)`
  - `(x)`
  - `(LOG(x))`
  - `(x * (-13))`
  - `(Pi ^ (SIN x))`
  - `(((Pi ^ (SIN x)) / (x + 13)))`
**LISP Example**: Symbolic Differentiation (cont.)

- **Differentiation rules for infix expressions**
  - **DyadicExpr**
    - $(a \pm b)' = a' \pm b'$
    - $(a \times b)' = (a' \times b) + (a \times b')$
    - $(a / b)' = ((a' \times b) - (a \times b')) / (b ^ 2)$
    - $(a ^ b)' = (exp(log(a^b)))' = (exp(b*log(a)))' = ((exp(b*log(a))) * (b*log(a)))' = ((a ^ b) * (b * log(a)'))$
  - **MonadicExpr**
    - $(+ a)' = a'$
    - $(- a)' = (- a')$
    - $(sin a)' = ((cos a) * a')$
    - $(cos a)' = ((-sin a) * a')$
    - $(exp a)' = ((exp a) * a')$
    - $(log a)' = (a' / a)$
  - **SimpleExpr**
    - Symbol' =
    - Number' = 0
    - $(a)' = a'$
LISP Example: Symbolic Differentiation (cont.)

(defun deriv (e) ; derivation with respect to variable 'x
  (cond ((null e) 0) ; empty expr.
         ((equal e 'x) 1)
         ((atom e) 0)
         ((null (cdr e)) (deriv (car e)))
         (t (derexpr (car e) (cadr e) (caddr e)))));

(defun derexpr (arg1      op     arg2 )
  (cond ((equal op '+ ) (deradd arg1 arg2 ))
         ((equal op '- ) (dersub arg1 arg2 ))
         ((equal op '* ) (dermult arg1 arg2))
         ((equal op '/ ) (derdiv arg1 arg2))
         ((equal op '^' ) (derpower arg1 arg2))
         (t (print 'error)) ))
LISP Example: Symbolic Differentiation (cont.)

(defun derfun (fun arg)
  (cond ((equal 'SIN fun) (list (list 'COS arg) '* (deriv arg)))
        ((equal 'COS fun) (list (list '-' (list 'SIN arg)) '*
                                  (deriv arg)))
        ((equal 'EXP fun) (list (list 'EXP (list arg)) '*
                                  (deriv arg)))
        ((equal 'LOG fun) (list (deriv arg) '/ arg))
        ; can be continued for further functions
        (t (print 'illegal_function)))
)
LISP Example: Symbolic Differentiation (cont.)

(defun deradd (arg1 arg2)
  (list (deriv arg1) '+ (deriv arg2)))

(defun dersub (arg1 arg2)
  (list (deriv arg1) '- (deriv arg2)))

(defun derdiv (arg1 arg2)
  (list (list (list (deriv arg1) '* arg2)
              '- (list arg1 '* (deriv arg2)))
        '/ (list arg2 '^ '2)))

(defun dermult (arg1 arg2)
  (list (list (deriv arg1) '* arg2)
        '+ (list arg1 '* (deriv arg2))))

(defun derpower (arg1 arg2)
  ; (a(x)^b(x))' = (exp(b(x)*log(a(x))))' = (a(x)^b(x))*(b(x)*log(a(x))'
  (list (list arg1 '^ arg2)
        '*' (dermult arg2 (list 'LOG(list arg1))))))
LISP Exercise: Symbolic Differentiation

1. Modify the program so that prefix instead of infix expressions are derived.

- **Grammar for prefix expressions** (no operator priority: use brackets!)

  - Expression ::= DyadicExpr | MonadicExpr | SimpleExpr
  - DyadicExpr ::= ( DyadicOperator SimpleExpr SimpleExpr )
  - MonadicExpr ::= ( MonadicOperator SimpleExpr )
  - SimpleExpr ::= Symbol | Number | ( Expression )
  - DyadicOperator ::= + | - | * | / | ^
  - MonadicOperator ::= + | - | FunctionId
  - FunctionId ::= sin | cos | exp | log

  - Only the order of parameters for DyadicExpr have to be changed:
    
```lisp
(defun derexpr (op arg1 arg2)
  (list op arg1 arg2))
```

  - And the generated dyadic expressions, e.g.
    
```lisp
(defun deradd (arg1 arg2)
  (list '+ (deriv arg1) (deriv arg2) ))
```
LISP Exercise: Symbolic Differentiation (cont.)

2. Modify the program so that partial derivatives are determined.

(defun deriv (e X)
; derivation with respect to variable X
(deriv e X)

(defun derexpr (arg1 op arg2 X)
...)

There is a simpler solution:

(defun partderiv (e X)
; pure LISP version
(partderiv e)

(defun partderiv (e dvar)
; COMMON LISP version
(setq X dvar)
(deriv e)

It uses binding of variable X to the differentiation variable. X is free with respect to deriv.)
LISP Exercise: Symbolic Differentiation (cont.)

3. Simplification of the derived expressions: avoid derivation of constants.

Simplification rules for infix expressions

DyadicExpr:
- \((a \pm b)' = a'\) if \(b\) is constant
- \(= b'\) if \(a\) is constant
- \((a \cdot b)' = (a' \cdot b)\) if \(b\) is constant
- \(= (a \cdot b')\) if \(a\) is constant
- \((a / b)' = (a' / b)\) if \(b\) is constant
- \(= - (a' / b) \cdot (b' / (b^2))\) if \(a\) is constant
- \((a ^ b)' = ((b \cdot (a ^ (b - 1))) \cdot a')\) if \(b\) is constant

MonadicExpr
- left to the interested reader

SimpleExpr
- \((a)' = 0\) if \(a\) is constant

Modify the program by using a function \texttt{constexprp}, e.g.

\[((\text{constexprp } e) \ 1)\] instead of \[((\text{atom } e) \ 1)\]

First correct the erroneous function

\begin{verbatim}
(defun constexprp (e)
  (cond ((equal e 'x) nil)
        ((constexprp (car e)) (constexprp (cdr e))
         (t nil)))
\end{verbatim}
LISP Exercise: Symbolic Differentiation (cont.)

Lisp Lab Exercises
We use the freeware Lisp interpreter
• Emacs-Editor run in Lisp-Mode
• gcl = Gnu Common Lisp: interpreter without GUI
• XLisp, available for many platforms (no longer maintained)
  It has object-oriented extensions. In its recent versions it migrates
  towards CommonLisp.
• WXLisp, a Windows-GUI for Xlisp with integrated editor and help-
  windows.

Invoke the Lisp interpreter from the subdirectory that contains your source
  programs or make it your current directory.

Load your source through
  ?- (load "datei") or ?- (load 'datei)

Leave the Lisp interpreter with
  ?- (bye) or Ctrl/q or ?- (exit) .

Exercises
Develop the following Lisp functions and send your programs via email to
  your lecturer.

1. Write a function that determines the dot-product of two vectors \( \mathbf{a} = (a_1, a_2, \ldots, a_n) \) und \( \mathbf{b} = (b_1, b_2, \ldots, b_n) \) represented by lists.

2. Write a function myLast that finds the last element in a list.
  Display the value of myLast (Lambda-expression).
  In CommonLisp use (function myLast).
  (function myLast (lambda (list) (first (reverse list)))).
  For example:
  ?- (myLast '(1 2 3))
  (3)
  ?- (myLast '())
  0
Optimization: "local variables"

- Avoid multiple calculations ⇒ use helper variable

\[(\text{defun } \text{ff}(n) \ (h \ (g \ n) \ (k \ (g \ n))) \ ) \quad \text{; call } g \text{ twice}\]

\[(\text{defun } f(qn) \ (h \ qn \ (k \ qn)) \ ) \quad \text{; call } g \text{ once}\]

- Recipe
  - find enclosing block
  - define a new function \(f\) with
    - parameter \(\equiv\) multiply evaluated expression and further variables/expressions of block
    - body \(\equiv\) block with replacements
  - replace block by call of function \(f\) with
    - argument \(\equiv\) multiply evaluated expression and further variables/expressions of block
Optimization: Tail-Recursion

- No loops ⇒ use recursion
  \[
  \text{(defun fak(n); n >= 0}
  \text{(cond (equal n 0) 1)}
  \text{(t (* n (fak (- n 1)))) ) )}
  \]

- Tail-Recursion ⇒ immediate return
  \[
  \text{(defun fak(n) (faka n 1) )}
  \text{; n! using an accumulator}
  \text{(defun faka(n acc); n >= 0}
  \text{(cond ((equal n 0) acc)
  \text{(t (faka (- n 1) (* n acc))))))}
  \]

- Compiler: Tail-Recursion ⇒ convert to loop
  - replace accumulator by local variable
  - avoid stack
LISP Exercise: Optimisation

1. A simple implementation of Hoare's Quicksort algorithm is like this:

   (defun quicksort(L) ; sorts a list L of ordered atoms e.g. integers
       (cond ((null L)       nil) ;L is empty
           ((null (cdr L)) L  ) ;L contains only 1 element
           (t (append (quicksort(car (partition (car L) (cdr L))))
                 (quicksort(cadr (partition (car L) (cdr L))))))) )

   (defun partition(el L)  ; returns a list (L1 L2) of 2 lists L1 and L2,
       (cond ((null (cdr L)) ;L contains only 1 element
               (cond ((< el (car L)) (list (list el) L))
                     (t (list L (list el)))))
               (t (cond ((< el (car L)) (list (partition el (cdr L)))
                        (cons(car L) (cadr (partition el (cdr L)))))
                     (t (list (cons(car L) (car (partition el (cdr L)))))))

   a) Why is it inefficient? Improve it without using SETQ.
   b) Modify the program so that the elements of list L are lists whose first entries
      are integers, e.g. L = '((1 John McCarthy) (7 Haskell Brooks Curry)...).
      Hint: Replace < by a function LT that compares lists.
LISP Exercise: Optimisation

- a) and b): Multiple partition calls with identical arguments cause inefficiency.

```lisp
(defun quicksort(L) ; L is list of elements ordered by 'LT'
  (cond ((null L) nil)
        ((null (cdr L)) L)
        (t (sort (partition (car L) (cdr L))) )
  )

(sort could be replaced with a lambda expression)

(defun sort(LL) ; LL is a pair of lists such that all
  (append (quicksort(car LL)) (quicksort(cadr LL)))

(defun partition(el L) ; L /= nil
  (cond ((null (cdr L)) ; L contains only 1 element
    (cond ((LT el (car L)) (list (list el) L))
          (t (list L (list el)))
    )
  (t ((lambda (el1 el2 LL) ; LL is a pair of lists such that all
      (cond ((LT el1 el2) (list (car LL) (cons el2 (cadr LL))))
            (t (list (cons el2 (car LL)) (cadr LL)))) )
      el (car L)
      (partition el (cdr L))
  )

(defun LT(el1 el2) ; el1 LT el2  iff  (car el1) < (car el2)
  (< (car el1) (car el2))
```
LISP Exercise: Optimisation (cont.)

2. Make the following function tail-recursive.

```lisp
(defun fib (n) ; Fibonacci-numbers without accumulator
  (cond ((equal n 0) 0) ; fib(0) = 0
        ((equal n 1) 1) ; fib(1) = 1
          (t (+ (fib (- n 1)) (fib (- n 2)))))) ; fib(n) = fib(n-1)+fib(n-2)
```

• (defun fiba (n acc1 acc2)
  (cond ((equal n 0) acc1) ; fib(0) = 0
          ((equal n 1) acc2) ; fib(1) = 1
            (t (fiba (- n 1) acc2 (+ acc1 acc2))))) ; fib(n) = fib(n-1)+fib(n-2)
LISP Exercise: Optimisation (cont.)

3. A function is called strict if it is undefined (NIL) whenever at least one of its arguments is undefined. A non-strict function is sometimes called shortcut. It can be defined although not all of its arguments can be evaluated.

C-example: if (d != 0 && n/d > 0) ...

a) Which non-strict Lisp-functions do you know?

b) What is the benefit of non-strict functions?

c) Why can non-strict functions improve performance?