

# PLD LISP

Torben Mogensen

March 16, 2021

PLD LISP is a simple LISP variant designed for “Programming Language Design”. While it is similar to other statically-scoped LISP variants (such as Scheme or Clojure), it differs from these by having pattern matching as an integral part of the `lambda` construct, and by supporting both statically and dynamically scoped functions.

This document describes PLD LISP and how to compile and run it.

## 1 S-Expressions

The values used in PLD LISP are *S-expressions*. An S-expression can be one of the four following forms:

**A number.** A number is an integer with the same range as integers in F#. Numbers are read and printed as F# integers. When reading, hexadecimal notation is allowed.

**A symbol.** A symbol is a name constructed from letters (Danish alphabet, both upper and lower case), digits, and the following special characters: `+-*/%=<>?!#,:{};`. Any such sequence that is not recognized as an integer constant is a valid symbol. For example, the sequence `0x3f` is a number, but `0x3g` is a symbol.

**Nil.** Nil (read and written as `()`) is used to represent the empty list, the false boolean value, or whatever else a programmer wants it to represent (since PLD LISP is dynamically typed).

**A pair.** A pair of two S-expressions  $s_1$  and  $s_2$  is read and written as  $(s_1 . s_2)$ , but there are a number of abbreviated forms used when reading or printing pairs. See below.

**A closure.** A closure is a pair of an S-expression and an environment (a binding from symbols to S-expressions). A closure can not be read (only constructed), and it is written as `#closure#`. The only use of a closure is applying it to a list of arguments.

We will, when reading and printing S-expressions, use the usual LISP shorthands:  $(s_1 s_2 \dots s_n)$  means the same as  $(s_1 . (s_2 \dots (s_n . ())))$ . For example, `(car x)` is short for `(car . (x . ()))`. The notation using spaces can be combined with the dot notation, so `(a b . c)` is short for `(a . (b . c))`. Additionally, we use the shorthand `'s` to mean `(quote s)`, which can also be written (or read) as `(quote . (s))` or `(quote . (s . ()))`.

Note that these are only shorthands in the same way that, in F#, `[1; 2; 3]` is a shorthand for `1 :: (2 :: (3 :: []))`.

An S-expression of the form  $(s_1 s_2 \dots s_n)$  represents a list of the elements  $s_1, s_2, \dots, s_n$ . Values that can not be written in this form are not considered proper lists. For example, `(1 . 2)` is not a proper list, but `(1 . (2))` is a proper list because it can be written as `(1 2)`. We consider `()` to be a proper list (the empty list).

When reading an S-expression, it can span several lines, but only whitespace is allowed on a line after an S-expression. So the following is valid input:

```
( 1 2
  (3 ) 4)
```

that represents the list `(1 2 (3) 4)`, but the following is not:

```
( 1 2
  (3 ) 4) 5
```

because the line that ends the S-expression that represents the list (1 2 (3) 4) has non-blank characters after the closing bracket. Comments start with a backslash and span to the end of the line, so

```
( 1 2 \ the first two elements
  \ followed by
  (3 ) 4) \ the last two elements
```

is legal input.

The module `Sexp.fs` defines an `F#` data structure that represents S-expressions and functions for reading and printing S-expressions.

## 2 PLD LISP

The syntax of PLD LISP is represented as S-expressions. This means that some S-expressions can be evaluated to other S-expressions. Not all S-expressions are valid expressions in PLD LISP, though. We go through the valid forms and their evaluation below:

- `()` evaluates to itself
- A number evaluates to itself.
- A symbol  $x$  can be a predefined operator or keyword, in which case it evaluates to itself, or it can be a variable that evaluates to whatever  $x$  is bound to in the current environment, which consists of the global environment extended with the current local environment. Static scoping is used, so the local environment takes precedence over the global environment. If a symbol is neither a predefined operator or keyword nor a variable, evaluating the symbol gives a run-time error. Predefined operators and keywords are

<code>quote</code> , <code>lambda</code> , <code>lambdaD</code> , <code>define</code> , <code>load</code> , and <code>save</code>	are keywords
<code>number?</code> and <code>symbol?</code>	are unary operators (predicates)
<code>cons</code> , <code>apply</code> , <code>/</code> , <code>%</code> , and <code>!=</code>	are binary operators
<code>+</code> , <code>-</code> , <code>*</code> , <code>&lt;</code> , <code>=</code> , <code>&lt;=</code> , <code>&gt;</code> , and <code>&gt;=</code>	are variadic operators

- A non-empty list is interpreted by evaluating the first element of the list and then applying this (as a keyword, operator or function) to the remaining elements of the list as follows:
  - `(quote s)` evaluates to  $s$ . Note that `'s` is a shorthand for `(quote s)`.
  - `(define x e)` evaluates  $e$  to  $s$ , binds  $x$  to  $s$  in the global environment, and returns  $x$ . If  $x$  is already bound, the new binding overwrites the old.
  - `(cons e1 e2)` evaluates  $e_1$  to  $a$  and  $e_2$  to  $d$  and returns `(a . d)`.
  - `(load f)` loads a sequence of expressions from the file `f.le`, evaluates these, and then returns `()`. The effect is the same as if the expressions are typed in from the REPL (see below), except that `()` is returned.
  - `(save f)` saves the current global environment in the file `f.le`. The saved environment is represented as a list of definitions (using `define`), one per line, so they can be read back in using `load`.
  - `(if e1 e2 ... en)` is a multi-way conditional. The behaviour depends on the number of arguments  $n$  to `if`. If  $n = 0$ , it returns `()`. If  $n = 1$ , it evaluates  $e_1$  and returns its value. If  $n \geq 2$ , it first evaluates  $e_1$  to a value  $v$ . If  $v \neq ()$ , it then evaluates  $e_2$  and returns its value. If  $v = ()$ , it applies `if` to the arguments after  $e_2$ , i.e., does `(if e3 ... en)`. Note that this implies that `(if e1 e2 e3)` works like `if e1 then e2 else e3` and that `(if e1 e2 e3 e4 e5)` works like `if e1 then e2 else if e3 then e4 else e5`.
  - `(lambda p1 e1 ... pn en)` is a statically scoped function. It evaluates to a closure consisting of itself and the current local environment.
  - `(lambdaD p1 e1 ... pn en)` is a dynamically scoped function. It evaluates to itself.

- $(u\ e)$ , where  $u$  is an unary operator, evaluates  $e$  and applies  $u$  to this value. The unary operator **number?** will return  $()$  if applied to anything other than a number, and return the number unchanged if applied to a number. The unary operator **symbol?** will return  $()$  if applied to anything other than a symbol, and return the symbol unchanged if applied to a symbol.
- $(b\ e_1\ e_2)$ , where  $b$  is a binary operator, evaluates  $e_1$  to  $v_1$  and  $e_2$  to  $v_2$  and applies  $b$  to  $v_1$  and  $v_2$ . The binary operator **cons** will return the pair  $(v_1\ .\ v_2)$ . The binary operator **apply** will apply the value  $v_1$  to the list of arguments given in  $v_2$ . For example, `(apply cons '(1 2))` will return `(1 . 2)`. `/` divides  $v_1$  by  $v_2$  using integer division. If  $v_1$  and  $v_2$  are not numbers or if  $v_2 = 0$ ,  $()$  is returned. `%` divides  $v_1$  by  $v_2$  using integer division and returns the remainder (using the semantics of the similar F# operator). If  $v_1$  and  $v_2$  are not numbers or if  $v_2 = 0$ ,  $()$  is returned. `!=` compares the numbers  $v_1$  and  $v_2$ . If they are different,  $v_1$  is returned, but if they are equal or they are not both numbers,  $()$  is returned.
- $(v\ e_1 \cdots e_n)$ , where  $v$  is a variadic operator, evaluates the  $e_i$  to  $v_i$  and applies  $v$  to  $v_1 \cdots v_n$ . The operator `+` adds its arguments. If applied to no arguments, it returns 0. If any arguments are not numbers,  $()$  is returned. The operator `-`, when applied to no arguments, returns 0. If applied to a single number  $n$ , it returns  $-n$ , if applied to  $n_1 n_2 \cdots n_m$ , it returns  $n_1 - n_2 - \cdots - n_m$ . If any argument is not a number, it returns  $()$ . The operator `*` multiplies its arguments. If applied to no arguments, it returns 1. If any arguments are not numbers,  $()$  is returned. The operators `<`, `=`, `<=`, `>`, and `>=` checks if the arguments are all numbers and sorted according to the operator. If they are, the first element is returned, otherwise  $()$  is returned. For example, `(< 7 9 13)` returns 7, while `(> 7 9 13)` returns  $()$ . If applied to no arguments or to arguments that are not numbers, comparison operators return  $()$ . Note that, while `!=` is a comparison operator, it is not variadic, as there is no sensible definition of a list being sorted by `!=`. So `!=` is a binary operator.
- If the first element evaluates to a closure that pairs an S-expression of the form `(lambda p1 e1 ... pn en)` with an environment  $\rho$ , the remaining elements of the list are evaluated to values  $v_1 \cdots v_m$ , and then the patterns  $p_1 \cdots p_n$  are in sequence matched against the list  $(v_1 \cdots v_m)$ . If  $p_i$  matches, this builds an environment  $\rho'$ , which is used to extend the closure environment to form a new local environment in which the corresponding expression  $e_i$  is evaluated. If no pattern matches, a run-time error is reported.  
Note that this is similar to the behaviour when applying a F# expression of the form `function p1 -> e1 | ... | pn -> en` to an argument  $(v_1, \dots, v_m)$ .  
If the first element evaluates to an S-expression of the form `(lambdaD p1 e1 ... pn en)`, pretty much the same thing happens, except that instead of using an environment stored in a closure, it uses the current environment.

How patterns are matched to values is described below.

Pattern matching uses the following rules:

- The pattern  $()$  matches the value  $()$  and yields the empty environment.
- A pattern that is a number constant matches values equal to this number.
- A pattern that is a symbol  $x$  (not including keywords and predefined operators) matches any value  $s$  and yields an environment that binds  $x$  to  $s$ . Keywords are not allowed as patterns, but can occur in constant patterns (see below). Predefined operators can occur in patterns, but only match themselves.
- A pattern of the form `'v` matches values equal to  $v$ . Note that  $v$  can be any S-expression, including keywords and predefined operators.
- A pattern  $(p_1\ .\ p_2)$ , where  $p_1$  and  $p_2$  are patterns, matches a value  $(s_1\ .\ s_2)$  if  $p_1$  matches  $s_1$  yielding the environment  $\rho_1$ ,  $p_2$  matches  $s_2$  yielding the environment  $\rho_2$ , and if a variable is bound in both  $\rho_1$  and  $\rho_2$ , it has the same value in both environments. The matching returns the combined environment  $\rho_1 \cup \rho_2$ .
- In all other cases, the pattern does not match the value.

The program `RunLISP.fsx` contains an interpreter for PLD LISP. It uses the `Sexp.fs` module.

### 3 Examples

We can define a function `car` that takes the head of a list by the expression

```
(define car (lambda ((a . d)) a))
```

Note that the pattern `((a . d))` specifies that `car` takes a single argument, which must be a pair of two values that are bound to the variables `a` and `d`. The expression `a` that follows this pattern just returns the first component of the pair. There are no other pattern/expression pairs in the `lambda`-expression, so if `car` is applied to something that is not a pair, a run-time error will be reported.

In LISP, it is common to define a function `cadr` such that `(cadr x) ≡ (car (cdr x))`, and similar for `caar`, `cdar`, and `cddr`, and in many cases also for longer sequences of `as` and `ds`. We can define `cadr` directly by the definition `(define cadr (lambda ((a ad . dd)) ad))`.

We can define `append` by

```
(define append (lambda ((          bs) bs
                        ((a . as) bs) (cons a (append as bs))))
```

Note that this uses two rules, one for the empty list and one for the non-empty list. Recursion is possible because `append` is bound in the global environment.

We can define a function `equal?` that tests whether two values are equal by the definition

```
(define equal? (lambda (x x) 'T (x y) ()))
```

If the two arguments are equal, the first pattern matches, so the symbol `T` is returned. Otherwise, `()` is returned. Note that `equal?` defines identity on all S-expressions, where the predefined operator `=` only defines equality on numbers.

We can make a function that tests whether a list of numbers is sorted by a given variadic comparison operator by

```
(define sortedBy (lambda (operator ()) 'T
                    (operator list) (apply operator list)))
```

The first case is needed because we consider empty lists to be sorted. We can call `sortedBy` by, e.g.,

```
(sortedBy < '(1 2 3 4))
```

which will return 1 or by

```
(sortedBy > '(1 2 3 4))
```

which will return `()`.

We can implement list reversal by the function definition

```
(define reverse
  (lambda (as)
    (lambda (bs)
      (reverse as (cons a bs)))))
```

This exploits that a function does not have a fixed number of parameters, so it implements both the usual `reverse` function and a function that takes two arguments and appends the reverse of the first argument to the second argument. The latter is typically used as a helper function when implementing `reverse`. A sequence of calls to `reverse` can be

```
(reverse (a b c))
~> (reverse (a b c) ()) using first rule
~> (reverse (b c) (a))  using third rule
~> (reverse (c) (b a))  using third rule
~> (reverse () (c b a)) using third rule
~> (c b a)              using second rule
```

Note that the arguments are shown as values – to make the initial call from the REPL, you should write `(reverse '(a b c))`.

## 4 Compiling and Running PLD LISP

The file `LISP.zip` contains an interpreter written in F# for this LISP variant. The interpreter implements a read-eval-print loop (REPL) and starts with an empty global environment. You can compile the interpreter with the command `source compile.sh` and run it with the command `mono lisp.exe`.

Type in an S-expression at the `>` prompt. It may span several lines. When (after an "Enter" is typed) a complete S-expression is found it is evaluated and the result shown after a `=`. If an error is found during parsing or evaluation, a message is displayed after a `!`. In both cases, the `>` prompt is shown again. Close a session by pressing `^D` (control-D) at the prompt.

The file `listfunctions.le` contains definitions of some simple list functions, including `car`, `cadr`, `equal?`, `length`, `append`, and `reverse`.

An example session using this LISP REPL is shown below

```
$ mono lisp.exe
PLD LISP version 2.1
> (load listfunctions)
> = car
> = cdr
> = caar
> = cadr
> = cdar
> = cddr
> = list
> = length
> = append
> = reverse
> = equal
> = ()
> (append '(a b c) '(d e f))
= (a b c d e f)
>
```