PLD LISP (v. 3.0)

Torben Mogensen

February 3, 2025

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 and by supporting both statically and dynamically scoped functions. Details of the syntax are deliberately different from other LISP variants, but the overall "feel" is the same.

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

1 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 and run PLD LISP both using Mono and .NET.

- With Mono, You can compile the interpreter with the command `source compileMono.sh` and run it with the command `mono lisp.exe`.
- With .NET, you can run PLD LISP by the command `dotnet fsi RunLISP.fsx`.

In both cases, you must have the files from the zip file in your current directory when running the commands.

1.1 The PLD LISP REPL

When the REPL is started, it shows the text

```
PLD LISP version 3.0
>
```

you can type in an expression at the "`> `" prompt. It may span several lines. When a line is completed and a complete expression is found, this is evaluated and the result shown after a "`= `".

If an error is found during parsing or evaluation, a message is displayed after a "`! `", and a backtrace is shown. The backtrace shows the recursion stack by listing (in reverse orer) which functions that were applied to which values to reach the error. Note that functions are shown as lambda-expressions and not as names.

In both cases, the "`> `" prompt is shown again. Close a session by pressing `^D` (control-D) at the prompt (`^Z` if using Windows).

2 Simple expressions

The simplest expression you can type is an integer, which evaluates to itself:

```
> 42
= 42
>
```

Funnction applications are fully parenthesized prefix expressions using spaces between function names and arguments. For example

```
> (* 3 4 (+ 5 6 7))
= 216
>
```

Note that + and * can take more than two arguments. This is true of many functions, including -, where it means the first argument minus all the subsequent arguments. / and % (modulus) only take two arguments, as these operations are not naturally defined on more than this:

```
> (/ 3 4 5)
! Wrong number of arguments to /
>
```

All arithmetic is on integers, and division round towards zero.

PLD LISP does not have Booleans, so comparison returns the first argument if the comparison is true and a special value `()` (pronounced "nil") if false:

```
> (< 3 4)
= 3
> (< 3 2)
= ()
>
```

Conditionals treat `()` as false and every other value a true:

```
> (if (< 2 3) 7 9)
= 7
> (if (> 2 3) 7 9)
= 9
>
```

Most comparison operators work on arbitrary many arguments, including none:

```
> (<)
= ()
> (< 2 3 4)
= 2
> (< 2)
= 2
> (< 2 3 3)
= ()
> (<= 2 3 3)
= 2
>
```

With more than two arguments, the comparison is true if the arguments are ordered by the operator, so `(< 2 3 4)` is true (indicated by the non-nil value 2), but `(< 2 3 3)` is false. Operators `<, >, <=, >=, =` work this way, but `!=` (different from) only accepts exactly two arguments.

Predefined operator names evaluate to themselves:

```
> +
= +
> -
= -
> <
= <
> (+ + <)
= ()
>
```

Arithmetic and comparison operators are only defined on numbers, so if any argument is not a number (e.g., +), they return `()`, as seen in the last expression above. So `()` is not just used for the false value, but also to indicate certain kinds of undefined behaviour and, as we will see later, the empty list.

3 Symbols, variables and functions

Operator names such as +, >, and `if` are *symbols.*

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 hexadecimal number, but `0x3g` is a symbol. Note that underscores are allowed (and ignored) in number constants, so `2_3` is read as the number 23, and `_2` as the number 2.

Predefined symbols such as `+` and `if` evaluate to themselves, but other symbols are used as names of variables and user-defined functions and are initially undefined:

```
> if
= if
> x
! Undefined variable x
>
```

Global variables can be defined using the `define` keyword:

```
 (define x (+ 3 4))
= x
> x
= 7
> (+ x x)
= 14
>
```

Global functuions can be defined using the `fun` keyword:

```
> (fun square (x) (* x x))
= square
> (square 7)
= 49
>
```

Note that the arguments in the definition are enclosed in parentheses. Functions of no or multiple parameters can easily be defined

```
> (fun getx () x)
= getx
> (getx)
= 7
> (fun average (x y)
      (/ (+ x y 1) 2))
= average
> (average 4 7)
= 6
> getx
= λ
> average
= λ
>
```

Note that since `x` is not bound locally in `getx`, the global variable `x` (which we previously defined to be 7) is used. Note, also, that the defintion of `average` spans several lines. The REPL waits until all parentheses are closed before completing the expression.

When a fnction name such as `getx` or `average` is used as a variable, its value is a *closure*, i.e., a functional value which can not be displayed, so the text λ is shown instead. This is not an actual value, just an indication that the value can not be displayed.

We can define anonymous functions using the \ symbol (chosen, as in Haskell, because it looks a bit like λ):

```
> ((\ (x) (* x x)) 9)
= 81
> (\ (x) (* x x))
= λ
>
```

In fact, the definition `(fun f (x) (* x x))` is just shorthand for `(define f (\ (x) (* x x)))`.

4 Lists, pairs, and quoted values

We noted above that using a symbol that is not bound to a value gives an "undefined variable" error, and that a bound symbol evaluates to the value to which it is bound. But we can also use symbols as values: If we prefix a symbol by a single quote, it evaluates to the symbol itself. In fact, any expression prefixed by a quote evaluates to the expression itself (and not its value):

```
> 'hello
= hello
> '()
= ()
> '7
= 7
> '(+ 2 3)
= (+ 2 3)
> ''(+ 2 3)
= '(+ 2 3)
>
```

Note that a double-quoted expression evaluates to a single-quoted expression. We can also use a notation that looks like a function application:

```
> (quote (+ 2 3))
= (+ 2 3)
> (quote (quote (+ 2 3)))
= '(+ 2 3)
>
```

In fact, `'e` is just a short notation for `(quote e)`, but `quote` is not a real function, as a function always evaluates its arguments.

We saw that `'(+ 2 3)` evaluated to `(+ 2 3)`. This value is actually a list of three elements, `+`, `2`, and `3`. Lists are shown in parentheses with elements separated by whitespace. We can add elements in front of a list using the `::` operator (as in F# and Standard ML):

```
> (:: 1 '(+ 2 3))
= (1 + 2 3)
> (:: 1 (:: 2 (:: 3 ())))
= (1 2 3)
>
```

Note that `()` is the empty list, so the expression `(:: 1 (:: 2 (:: 3 ())))` is equivalent to the F# expression `1 :: 2 :: 3 :: []` that evaluates to the list `[1; 2; 3]`. The difference is that F# uses an infix operator while PLD LISP uses a parenthesized prefix operator, and that the notation for lists is different.

Since PLD LISP is dynamically typed, a list does not have to end with `()`. Some examples:

```
> (:: 2 3)
= (2 . 3)
> (:: 2 (:: 3 4))
= (2 3 . 4)
> '(1 . (2 . (3 . ())))
= (1 2 3)
>
```

The first of these expressions essentially builds a pair of 2 and 3, and the second adds 2 in front of the pair of 3 and 4. In essence, a list in PLD LISP is nested pairs ending with `()`, as the last evaluation above shows. Note the difference between `.` and `::` – the dot is used infix in values to indicate pairs (but it is not an operator), and the double-colon is a prefix operator/function that builds pairs.

A list that ends in `()` (so it can be shown without a dot) is called *a proper list*.

5 Pattern matching

The function definitions we have seen so far specify formal parameters as a list of variables, but functions can use pattern matching. As an example, the factorial function can be defined as shown here:

```
(fun factorial
   (0) 1
   (n) (* n (factorial (- n 1))))
```

The body of the function consists of two *rules*, each consisting of a pattern and an expression. The pattern `(0)` matches an argument list consisting of a single value, which is equal to 0. The pattern `(n)` matches an argument list consisting of any single value v and binds `n` to v when evaluating the expression to the right of the pattern. Note that newlines and indentation do not matter, so we could lay the function out on one line or split it over more lines with no indentation.

We can also use pattern matching on lists and pairs, and that is actually the only way to access the elements of lists or components of pairs – that is, until we define functions that can do so. These functions must, initially, be defined using pattern matching. We can, for example, define functions that takes the left or right components of a pair by:

```
(fun #L ((L . R)) L)

(fun #R ((L . R)) R)
```

Note that, if the argument is not a pair, the pattern does not match, so a run-time error is reported, as shown below

```
> (#L 7)
! No patterns matched arguments (7)
! when applying (\ ((L . R)) L)
! to (7)
>
```

Note that, since `(fun #L ((L . R)) L)` is a shorthand for `(define #L (\ ((L . R)) L))`, the error message is given in terms of this.

Note that, if applied to a list, `#L` returns the first element of the list, and `#R` returns the tail of the list:

```
> (#L '(1 2 3))
= 1
> (#R '(1 2 3))
= (2 3)
>
```

We can define and use a function that returns the second element of a list by

```
> (fun #2 ((_1 _2 . rest)) _2)
= #2
> (#2 '(1 2 3))
= 2
>
```

Note that underscores are allowed in variable names. A single underscore is *not* a wildcard pattern, but just a "normal" variable.

If a variable occurs several times in a pattern, it will only match values if all the occurrences of the same variable match the same value. So, a general equality function can be defined by

```
(fun equal
 (x x) 'T
 (x y) ())
```

If the two arguments are equal, the function returns the symbol `T` (which is considered a true Boolean value), and otherwise it returns `()` (which is considered a false Boolean value). Note that, while `+` only works on numbers, `equal` works on arbitrary values.

In the examples above, functions are defined on a fixed number of parameters, but you can define functions that can take different number of arguments. For example, we can define and use a function that builds a list of its arguments by

```
> (fun list x x)
= list
> (list 1 2 3)
= (1 2 3)
>
```

The pattern `x` (not in parentheses!) matches the complete argument list.

We can also use this to define a function that can take either one or two parameters by using patterns with different length. This allows us to define a reverse function using an accumulating parameter in a single function definition:

```
(fun reverse
  (as)          (reverse as ())
  (()       bs) bs
  ((a . as) bs) (reverse as (:: a bs)))
```

The first rule matches a list of one argument, and calls `reverse` with two arguments, the second of which is an empty list, which will be used as an accumulating parameter in the other rules. The second rule matches the case where the first argument is an empty list, and returns the second argument (the accumulating parameter). The third rule matches the case where the first argument is a non-empty list, and calls `reverse` with the rest of the list as firt parameter and adding the first element to the accumulating parameter.

This facility should not be over-used, as it can make code quite unreadable.

6 Miscellaneous

6.1 Comments

When a semicolon (`;`) is encountered, the rest of the line is a comment.

6.2 The `apply` function

The binary function `apply` takes a function and a list of arguments and applies the function to these arguments. For example,

```
> (apply + '(1 2 3))
= 6
>
```

This is useful when you compute a list and want to apply a variadic function to it. For example, we can define a function that checks if a list is ordered by a comparison operator given a argument:

```
> (fun orderedBy (op list) (apply op list))
= orderedBy
> (orderedBy < '(1 2 4))
= 1
>
```

We can also use it to define a variadic equality function:

```
> (fun Equal
  (x) 'T
  (x x . rest) (apply Equal (:: x rest))
  otherwise ())
= Equal
> (Equal 'x 'x 'x)
= T
> (Equal  'x 'x 'x 'xx)
= ()
>
```

6.3 Let-expressions

PLD LISP supports let-expressions similar to those found in F#, ML, or Haskell. The syntax is

`(let` p e_1 e_2`)`

It evaluates e_1 and matches the pattern p to this. If successful, it evaluates e_2 in the current envionment extended with the bindings made by matching p to the value of e_1. Otherwise, an error is reported. Examples:

```
> (let x (+ 3 4) (* x x))
= 49
> (let (h . t) '(1 2 3 4) t)
= (2 3 4)
> (let (h . t) 42 t)
! The pattern (h . t) did not match the value 42
>
```

6.4 Symbol and number tests

The predefined unary operators `symbol?` and `number?` test if their arguments are symbols or numbers, respectively:

```
> (symbol? 'x)
= x
> (number? 7)
= 7
> (symbol? 7)
= ()
> (number? 'x)
= ()
>
```

If the test succeeds, they return their arguments, otherwise they return `()`.

6.5 Variadic conditional

The `if` expression actually works with any number of arguments, which is mostly used as a shorter form to write nested conditionals. See Section 7 for details.

6.6 Dynamically scoped functions

A λ-expression can be defined using `lambda` instead of `\`, as for example in

```
(define g (lambda (x) (+ x y))
```

The difference is that this defines a dynamically scoped function. This means that the variable `y` in the body does not need to be defined when the function is defined (as would be the case with static scoping), but it must be defined when the function is used.

6.7 Loading and saving code

`(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.

The file can contain additional `load` commands, and these will cause new files to be loaded recursively. There is no check against infinite recursion (a file containing a load of itself).

The file `listfunctions.le` contains a number of simple functions for working on lists, including many of the examples above. It is common to let the first line of a program be `(load listfunctions)`, so these cane be used by the program.

`(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`. It is not formatted for easy human readability, though.

7 PLD LISP reference

PLD LISP has a global environment where values are bound or rebound using `define` or `fun`. It also has a statically scoped local environments containing bindings to variables bound in patterns. The local environment takes precedence over the global environment. An exception to the static scope rules is functions defined with `lambda`, which are dynamically scoped.

We go through the valid forms and their evaluation below:

- `()` evaluates to itself.

- A number evaluates to itself.

- 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 hexadecimal number, but `0x3g` is a symbol.

  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 environmentIf a symbol is neither a predefined operator, nor a keyword, nor a variable, evaluating the symbol gives a run-time error.

  Predefined operators and keywords are

`quote`, `\`, `lambda`, `define`, `fun`, `if`, `let`, `load`, and `save`	are keywords
`number?` and `symbol?`	are predefined unary operators (predicates)
`::`, `apply`, `/`, `%`, and `!=`	are binary operators
`+`, `-`, `*`, `<`, `=`, `<=`, `>`, and `>=`	are variadic operators, meaning they can be applied to any number of arguments

- How an expression of the form `(`e_1 e_2 ... e_n`)` is interpreted depends on the (unevaluated) form of e_1 as follows:

  - `(quote` s`)` evaluates to s. Note that `'`s is a shorthand for `(quote` s`)`, so it too evaluates to 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.
  - `(fun` f *rules*`)` is a shorthand for `(define` f `(\` *rules*`))`.
  - `(::` e_1 e_2`)` evaluates e_1 to a and e_2 to d and returns the value `(`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` e_1 e_2 $\cdots$ e_n`)` 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` e_3 $\cdots$ e_n`)`. Note that this implies that `(if` e_1 e_2 e_3`)` works like **`if`** e_1 **`then`** e_2 **`else`** e_3 in other languages, that `(if` e_1 e_2`)` works like **`if`** e_1 **`then`** e_2 **`else`** `()`, and that `(if` e_1 e_2 e_3 e_4 e_5`)` works like **`if`** e_1 **`then`** e_2 **`else if`** e_3 **`then`** e_4 **`else`** e_5.
  - `(\` p_1 e_1 $\cdots$ p_n e_n`)` is a statically scoped function. It evaluates to a closure consisting of itself and the current local environment. A closure is printed as λ because environments can not be printed.
  - `(lambda` p_1 e_1 $\cdots$ p_n e_n`)` is a dynamically scoped function. It evaluates to itself.

– `(let` p e_1 e_2`)` is evaluated by evaluating e_1 in the current local environment ρ to a value v. The pattern p is matched against v. If p matches v, this builds an environment ρ', which is used to extend ρ. e_2 is evaluated in the extended environment, and the result of this evaluation is returned. If p does not match v, an error is reported.

– `(`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 `::` 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 :: '(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 each 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 e_1 is none of the above, it is evaluated to a value v_1. If v_1 a closure that pairs an S-expression of the form `(lambda` p_1 e_1 $\cdots$ p_n e_n`)` with an environment ρ, the remaining elements $v_2, \ldots, v_m$ of the list are evaluated to values $v_2 \cdots v_m$, and then the patterns $p_1 \cdots p_n$ are in sequence matched against the list `(`$v_2 \cdots v_m$`)`. If p_i matches, this builds an environment ρ', 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` p_1 `->` e_1 `| ... |` p_n `->` e_n`)` to an argument `(`v_1`, ... ,`v_m`)`. In PLD LISP, we just write \ instead of `function` and omit the arrows and bars.

  If the first element evaluates to an S-expression of the form `(lambdaD` p_1 e_1 $\cdots$ p_n e_n`)`, pretty much the same thing happens, except that instead of using an environment stored in a closure, it uses the current environment.

7.1 Pattern matching

Pattern matching tries to match a pattern to a value. If the match is successful, it yields an environment that binds the variables in the pattern to values. 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 ρ_1, p_2 matches s_2 yielding the environment ρ_2, and if a variable x is bound in both ρ_1 and ρ_2, x must have the same value in both environments, otherwise the matching fails. The matching returns the combined environment $\rho_1 \cup \rho_2$.
- In all other cases, the pattern does not match the value.