Cells in PicoLisp

Idea: @tankF33der
Review: @Regenaxer
Revision: 24
June 2020

CC0
Fundamental overview

CELL
The PicoLisp reference says:

1. **A cell** is a pair of machine words, which traditionally are called CAR and CDR in the Lisp terminology.
2. These words can represent either a numeric value (scalar) or the address of another cell (pointer).
3. All higher level data structures are built out of cells.
The PicoLisp reference says:

1. A cell is a pair of machine words, which traditionally are called CAR and CDR in the Lisp terminology.
2. These words can represent either a numeric value (scalar) or the address of another cell (pointer).
3. All higher level data structures are built out of cells.

```c
// src/pico.h
typedef struct cell {
    struct cell *car;
    struct cell *cdr;
} cell, *any;
```

Yes, two identical types
The PicoLisp reference says:

1. A cell is a pair of machine words, which traditionally are called CAR and CDR in the Lisp terminology.
2. These words can represent either a numeric value (scalar) or the address of another cell (pointer).
3. All higher level data structures are built out of cells.

```c
// src/pico.h
typedef struct cell {
    struct cell *car;
    struct cell *cdr;
} cell, *any;
```

Yes, two identical types

**Yes, can store identical values**
The PicoLisp reference says:

1. A cell is a pair of machine words, which traditionally are called CAR and CDR in the Lisp terminology.

2. These words can represent either a numeric value (scalar) or the address of another cell (pointer).

3. All higher level data structures are built out of cells.

```c
// src/pico.h
typedef struct cell {
    struct cell *car;
    struct cell *cdr;
} cell, *any;
```

Yes, two identical types

Yes, can store identical values

Yes, cells are everywhere
The PicoLisp reference says:

1. A cell is a pair of machine words, which traditionally are called CAR and CDR in the Lisp terminology.
2. These words can represent either a numeric value (scalar) or the address of another cell (pointer).
3. All higher level data structures are built out of cells.

```c
// src/pico.h
typedef struct cell {
    struct cell *car;
    struct cell *cdr;
} cell, *any;

Yes, two identical types
Yes, can store identical values
Yes, cells are everywhere

// src/pico.h
typedef struct heap {
    cell cells[CELLS];
    struct heap *next;
} heap;
```
The PicoLisp reference says:

1. A cell is a pair of machine words, which traditionally are called CAR and CDR in the Lisp terminology.
2. These words can represent either a numeric value (scalar) or the address of another cell (pointer).
3. All higher level data structures are built out of cells.

The code snippet from `src/pico.h` shows the structure of a cell:

```c
typedef struct cell {
    struct cell *car;
    struct cell *cdr;
} cell, *any;
```

Cells in heap under full control by GC:

```
// src/pico.h
typedef struct cell {
    struct cell *car;
    struct cell *cdr;
} cell, *any;
```

- Yes, two identical types
- Yes, can store identical values
- Yes, cells are everywhere
Fundamental overview

LIST
A list is not part of data type hierarchy.
The PicoLisp reference provides recursive definition:

A list is a sequence of one or more cells (cons pairs), holding numbers, symbols, or cons pairs.
The PicoLisp reference provides recursive definition:

A list is a sequence of one or more cells (cons pairs), holding numbers, symbols, or cons pairs.
The PicoLisp reference provides recursive definition:

A list is a sequence of one or more cells (cons pairs), holding numbers, symbols, or cons pairs.

List is like wagon train
The PicoLisp reference provides recursive definition:

A list is a **sequence** of one or more cells (cons pairs), holding numbers, symbols, or cons pairs.

Remember!

This is a **list** if CDR of last cell points to NIL

If atom in CDR then this is a dotted pair
Construct and view
$\text{pil} +$

: (cons 1 2)
→ (1 . 2)

: (cons 1 2 3)
→ (1 2 . 3)

: (list 1 2 3)
→ (1 2 3)

:
CONStract a cell or sequence of cells are straightforward.
Construct a cell or sequence of cells are straightforward.

Function \texttt{view} will help understand cell structure:

\begin{verbatim}
: (cons 1 2)  -> (1 . 2)
: (cons 1 2 3)  -> (1 2 . 3)
: (list 1 2 3)  -> (1 2 3)
:

(view @)
\end{verbatim}
Construct a cell or sequence of cells are straightforward.

Function **view** will help understand cell structure:

```
: (cons 1 2)
→ (1 . 2)
: (view @)
+-- 1
| 2
→ 2

: (cons 1 2 3)
→ (1 2 . 3)
: (view @)
+-- 1
| +-- 2
|     3
→ 3

: (list 1 2 3)
→ (1 2 3)
: (view @)
+-- 1
| +-- 2
|  +-- 3
→ NIL
```
Construct a cell or sequence of cells are straightforward.

Function \texttt{view} will help understand cell structure:

\begin{verbatim}
: (cons 1 2)
  → (1 . 2)
: (cons 1 2 3)
  → (1 2 . 3)
: (list 1 2 3)
  → (1 2 3)
:

: (cons 1 2)
  → (1 . 2)
: (view @)
  +-- 1
    |    Legend:
    + is CELL
    | is CDR
    2
  → 2
: (cons 1 2 3)
  → (1 2 . 3)
: (view @)
  +-- 1
    |    | is CAR
    2
  → 2
  +-- 3
    3
  → 3
: (list 1 2 3)
  → (1 2 3)
: (view @)
  +-- 1
    |    After practice you will manipulate and view structures in mind.
    +-- 2
    |    Nothing special, right?
    +-- 3
    → NIL
\end{verbatim}
Modify CAR
The PicoLisp reference for function set says:

```
(set 'var 'any ..) -> any
Stores new values any in the var arguments.
See also setq, eval, swap, cons and def.
: (set 'L '(a b c) (cdr L) 999)
   -> 999
: L
   -> (a 999 c)
```

Variable: Either a symbol or a cons pair
The PicoLisp reference for function set says:

\[(\text{set } \text{'var 'any ..}) \rightarrow \text{any}\]

Stores new values any in the var arguments.
See also setq, val, swap, con and def.

: (set 'L '(a b c) (cdr L) 999)
→ 999
: L
→ (a 999 c)

In case of cell it modify CAR:

$ pil +
: (set 'L (cons 1 2))
→ (1 . 2)
: (set L 3)
→ 3
: L
→ (3 . 2)
: (set L (cons 1 2))
→ (1 . 2)
: L
→ ((1 . 2) . 2)
Modify CDR
The PicoLisp reference for function con says:

\[(\text{con } \text{lst} \text{ any}) \rightarrow \text{any}\]

Connects any to the first cell of lst, by (destructively) storing any in the CDR of lst.
See also set and conc.

: (setq C (1 . a))
→ (1 . a)
: (con C '(b c d))
→ (b c d)
: C
→ (1 b c d)
The PicoLisp reference for function \texttt{con} says:

\begin{verbatim}
(con 'lst 'any) \rightarrow any
\end{verbatim}

Connects \texttt{any} to the first cell of \texttt{lst}, by (destructively) storing \texttt{any} in the CDR of \texttt{lst}.

See also \texttt{set} and \texttt{conc}.

: (setq C (1 . a))
\rightarrow (1 . a)

: (con C '(b c d))
\rightarrow (b c d)

: C
\rightarrow (1 b c d)

\textbf{Remember:}

\begin{enumerate}
  \item modify CDR of dotted pair is just modification
  \item modify CDR of list is \textbf{DESTRUCTIVENESS} of sequence
\end{enumerate}

: (setq 'L (cons 1 2))
\rightarrow (1 . 2)

: (con L 22)
\rightarrow 22

: L
\rightarrow (1 . 22)
The PicoLisp reference for function `con` says:

```
(con 'lst 'any) -> any
```

Connects any to the first cell of lst, by (destructively) storing any in the CDR of lst.
See also set and conc.

```
: (setq C (1 . a))
  → (1 . a)
: (con C '(b c d))
  → (b c d)
: C
  → (1 b c d)
```

**Remember:**

- o) modify CDR of dotted pair is just modification
- o) modify CDR of list is **DESTRUCTIVENESS** of sequence

```
: (setq 'L (cons 1 2))
  → (1 . 2)
: (con L 22)
  → 22
: L
  → (1 . 22)
```

```
: (setq 'L (list 1 2 3))
  → (1 2 3)
: (view @)
  +-- 1
    |   |
    |   +-- 2
    |         |
    |         +-- 3
  +-- NIL
: (con L 22)
  → 22
: (view L)
  +-- 1
    | 22
  → 22
```
Remember:

1) modify CDR of dotted pair is just modification
2) modify CDR of list is **DESTRUCTIVENESS** of sequence

The PicoLisp reference for function `con` says:

`con 'lst 'any) → any`

Connects any to the first cell of lst, by (destructively) storing any in the CDR of lst.

See also `set` and `conc`.

: (setq C (1 . a))
→ (1 . a)
: (con C '(b c d))
→ (b c d)
: C
→ (1 b c d)

**Remember:**

1) modify CDR of dotted pair is just modification
2) modify CDR of list is **DESTRUCTIVENESS** of sequence

: (setq 'L (cons 1 2))
→ (1 . 2)
: (con L 22)
→ 22
: L
→ (1 . 22)

: (set 'L (list 1 2 3))
→ (1 2 3)
: (view @)
+-- 1
|   2
|   +-- 3
|   |   → NIL
|   +-- (con L 22)
|       → 22
|       : (view L)
+-- 1
|   22
→ 22

Access path to two wagons is lost and they will be GC eventually.
The PicoLisp reference for function **con** says:

(con 'lst 'any) → any

Connects any to the first cell of lst, by (destructively) storing any in the CDR of lst.
See also **set** and **conc**.

: (setq C (1 . a))
→ (1 . a)
: (con C '(b c d))
→ (b c d)
: C
→ (1 b c d)

**Remember:**

0) modify CDR of dotted pair is just modification
0) modify CDR of list is **DESTRUCTIVENESS** of sequence

: (setq L (cons 1 2))
→ (1 . 2)
: (con L 22)
→ 22
: L
→ (1 . 22)

: (setq L (list 1 2 3))
→ (1 2 3)
: (view @)
+- 1
| +-- 2
|   +-- 3
   → NIL
: (con L 22)
→ 22
: (view L)
+- 1
| 22
→ 22

Any destructive functions behaves the same way.
No dark corners anymore.
Now you have everything to understand listing of destructive function `chain`:

```
$ pil +
: (make (link 1 2) (view (made)) (chain 3) (view (made)))
  +-- 1
    |    
  +-- 2
    |    
  +-- 1
    |    
  +-- 2
    |    
3 --> (1 2 . 3)
: (make (link 1 2) (view (made)) (chain (cons 3)) (view (made)))
  +-- 1
    |    
  +-- 2
    |    
  +-- 1
    |    
  +-- 2
    |    
  +-- 3
    |    
3 --> (1 2 3)
```
Happy coding!