Reading Inside Out

An algorithm to indent S-expressions by reading them inside out.

2024-05-06 | Alex Vear <alex@vear.uk> | ELS 2024 Lightning talk

```clojure
(defn distance
  "Calculate the Euclidean distance between two points
  (\"p\" and \"q\") of any dimension."
  [p q]
  (math/sqrt
   (reduce +
           (map (fn [p' q'] (math/pow (- p' q') 2))

                p q))))
```

```clojure
(defn distance
  "Calculate the Euclidean distance between two points
  (\"p\" and \"q\") of any dimension."
  [p q]
  (math/sqrt
   (reduce +
           (map (fn [p' q']
                  (math/pow (- p' q') 2))
                p q))))
```

How can we do this efficiently?

Build a special-purpose Clojure reader inside of Vim.

Constraints

- Vim script (the implementation language) is painfully slow.
  - Loop iterations especially so!
  - Vim 9 script and Lua?

- The algorithm is invoked separately on each line to be indented.
  - State cannot be shared between invocations.

- Cannot assume that all code in the file is valid.

```
(read-string "(defn foo [x]\n  (* x\n 42))")
                  ^-->
```

```
(read-string "(defn foo [x]\n  (* x\n 42))")
                          <--^
```

The base algorithm

```clojure
(defn distance
  "Calculate the Euclidean distance between two points
  (\"p\" and \"q\") of any dimension."
  [p q]
  (math/sqrt
   (reduce +
           (map (fn [p'
                    q']
(math/pow (- p' q') 2))
                p q))))
```

```
(defn distance
  "Calculate the Euclidean distance between two points
  (\"p\" and \"q\") of any dimension."
  [p q]
  (math/sqrt
   (reduce +
           (map (fn [p'
                     q']
(math/pow (- p' q') 2))
                p q))))
```

```
 1 (defn distance
 2   "Calculate the Euclidean distance between two points
 3   (\"p\" and \"q\") of any dimension."
 4   [p q]
 5   (math/sqrt
 6    (reduce +
 7            (map (fn [p'
 8                      q']
 9 (math/pow (- p' q') 2))
10                  p q))))
```

Token	Location

Looking for: `[ ] { } ( ) ; "`

```
 1 (defn distance
 2   "Calculate the Euclidean distance between two points
 3   (\"p\" and \"q\") of any dimension."
 4   [p q]
 5   (math/sqrt
 6    (reduce +
 7            (map (fn [p'
 8                      q']
 9 (math/pow (- p' q') 2))
10                  p q))))
```

Looking for: `[ ] { } ( ) ; "`

Token	Location
]	8, 23

```
 1 (defn distance
 2   "Calculate the Euclidean distance between two points
 3   (\"p\" and \"q\") of any dimension."
 4   [p q]
 5   (math/sqrt
 6    (reduce +
 7            (map (fn [p'
 8                      q']
 9 (math/pow (- p' q') 2))
10                  p q))))
```

Token	Location
]	8, 23

Looking for: `[ ] { } ( ) ; "`

```
 1 (defn distance
 2   "Calculate the Euclidean distance between two points
 3   (\"p\" and \"q\") of any dimension."
 4   [p q]
 5   (math/sqrt
 6    (reduce +
 7            (map (fn [p'
 8                      q']
 9 (math/pow (- p' q') 2))
10                  p q))))
```

Looking for: `[ ] { } ( ) ; "`

Token	Location
]	8, 23
[	7, 20
(	7, 16
(	7, 11

```
 1 (defn distance
 2   "Calculate the Euclidean distance between two points
 3   (\"p\" and \"q\") of any dimension."
 4   [p q]
 5   (math/sqrt
 6    (reduce +
 7            (map (fn [p'
 8                      q']
 9 (math/pow (- p' q') 2))
10                   p q))))
```

Looking for: `[ ] { } ( ) ; "`

Token	Location
]	8, 23
[	7, 20
(	7, 16
(	7, 11

```
 1 (defn distance
 2   "Calculate the Euclidean distance between two points
 3   (\"p\" and \"q\") of any dimension."
 4   [p q]
 5   (math/sqrt
 6    (reduce +
 7            (map (fn [p'
 8                      q']
 9 (math/pow (- p' q') 2))
10                   p q))))
```

Looking for: `[ ] { } ( ) ; "`

Token	Location
(	7, 16
(	7, 11

```
 1 (defn distance
 2   "Calculate the Euclidean distance between two points
 3   (\"p\" and \"q\") of any dimension."
 4   [p q]
 5   (math/sqrt
 6    (reduce +
 7            (map (fn [p'
 8                      q']
 9 (math/pow (- p' q') 2))
10                  p q))))
```

Token	Location
(	7, 16
(	7, 11

Looking for: `[ ] { } ( ) ; "`

```
 1 (defn distance
 2   "Calculate the Euclidean distance between two points
 3   (\"p\" and \"q\") of any dimension."
 4   [p q]
 5   (math/sqrt
 6    (reduce +
 7            (map (fn [p'
 8                     q']
 9                   (math/pow (- p' q') 2))
10                 p q))))
```

Token	Location
(	7, 16
(	7, 11

Looking for: [ ] { } ( ) ; "

```
 1 (defn distance
 2   "Calculate the Euclidean distance between two points
 3   (\"p\" and \"q\") of any dimension."
 4   [p q]
 5   (math/sqrt
 6    (reduce +
 7            (map (fn [p'
 8                     q']
 9                   (math/pow (- p' q') 2))
10                 p q))))
```

Token	Location
(	7, 16
(	7, 11

Looking for: `[ ] { } ( ) ; "`

Tricky syntax

Escaped characters

- While scanning for tokens, if any are preceded by an odd number of escape characters, we ignore those tokens.
  - Backslash is the escape character in Clojure.
  - It was either a Clojure character literal, or an escape code in a string.

Semicolon comments

- Semicolon comments are unlike the rest of the syntax as they are not S-expressions, instead throwing away everything until the next newline.

```
 1 (defn distance
 2   "Calculate the Euclidean distance between two points
 3   (\"p\" and \"q\") of any dimension."
 4   [p q]
 5   (math/sqrt
 6    (reduce +
 7            (map (fn [p'
 8                      ;; ( [
 9                      q']
10 (math/pow (- p' q') 2))
11                   p q))))
```

Looking for: `[ ] { } ( ) ; "`

Token	Location

```
 1 (defn distance
 2   "Calculate the Euclidean distance between two points
 3   (\"p\" and \"q\") of any dimension."
 4   [p q]
 5   (math/sqrt
 6    (reduce +
 7            (map (fn [p'
 8                      ;; ( [
 9                      q']
10 (math/pow (- p' q') 2))
11                   p q))))
```

Looking for: `[ ] { } ( ) ; "`

Token	Location
]	9, 23

```
 1 (defn distance
 2   "Calculate the Euclidean distance between two points
 3   (\"p\" and \"q\") of any dimension."
 4   [p q]
 5   (math/sqrt
 6    (reduce +
 7            (map (fn [p'
 8                      ;; ( [
 9                      q']
10 (math/pow (- p' q') 2))
11                   p q))))
```

Looking for: `[ ] { } ( ) ; "`

Token	Location
]	9, 23
[	8, 26
(	8, 24
;	8, 22
;	8, 21

```
 1 (defn distance
 2   "Calculate the Euclidean distance between two points
 3   (\"p\" and \"q\") of any dimension."
 4   [p q]
 5   (math/sqrt
 6    (reduce +
 7            (map (fn [p'
 8                      ;; ( [
 9                      q']
10 (math/pow (- p' q') 2))
11                   p q))))
```

Looking for: `[ ] { } ( ) ; "`

Token	Location
]	9, 23
[	8, 26
(	8, 24
;	8, 22
;	8, 21

```
 1 (defn distance
 2   "Calculate the Euclidean distance between two points
 3   (\"p\" and \"q\") of any dimension."
 4   [p q]
 5   (math/sqrt
 6    (reduce +
 7            (map (fn [p'
 8                      ;; ( [
 9                      q']
10 (math/pow (- p' q') 2))
11                   p q))))
```

Looking for: `[ ] { } ( ) ; "`

Token	Location
]	9, 23

Multiline strings

- Huge pain to deal with.
  - Made worse due to opening and closing string delimiters being the same.

- If we detect a lone string delimiter on a line, we activate the "maybe in a multiline string" mode and apply special rules to figure it out for sure.
  - Fairly uncommon (except as docstrings) so the performance hit of the additional checks is minimal.

Results

Performance improvements

Code version	200 lines of Clojure	600 lines of EDN
Original	0.96s	4.6s (`let clojure_maxlines = 0`)
New	0.33s	2.6s
New (Vim 9)	0.06s	0.46s

Other benefits

- Much simpler and more configurable with no hacks.
  - >400 LOC → ~300 LOC.

- Not reliant on syntax highlighting.
  - Works with Treesitter.

- Reusable for other Lisps with Clojure-like syntax.

Find out more

- https://github.com/clojure-vim/clojure.vim/pull/31
  - Will be merged soon(ish), just a few more small changes to make.

- Email: alex@vear.uk
- WWW: https://www.alexvear.com