Cursive

Cursive

First public beta Oct 2013.

v1.0 Dec 2015.

Quickly became the second most used dev environment.

Currently developed as a plugin to IntelliJ.

Will eventually be a standalone IDE as well.

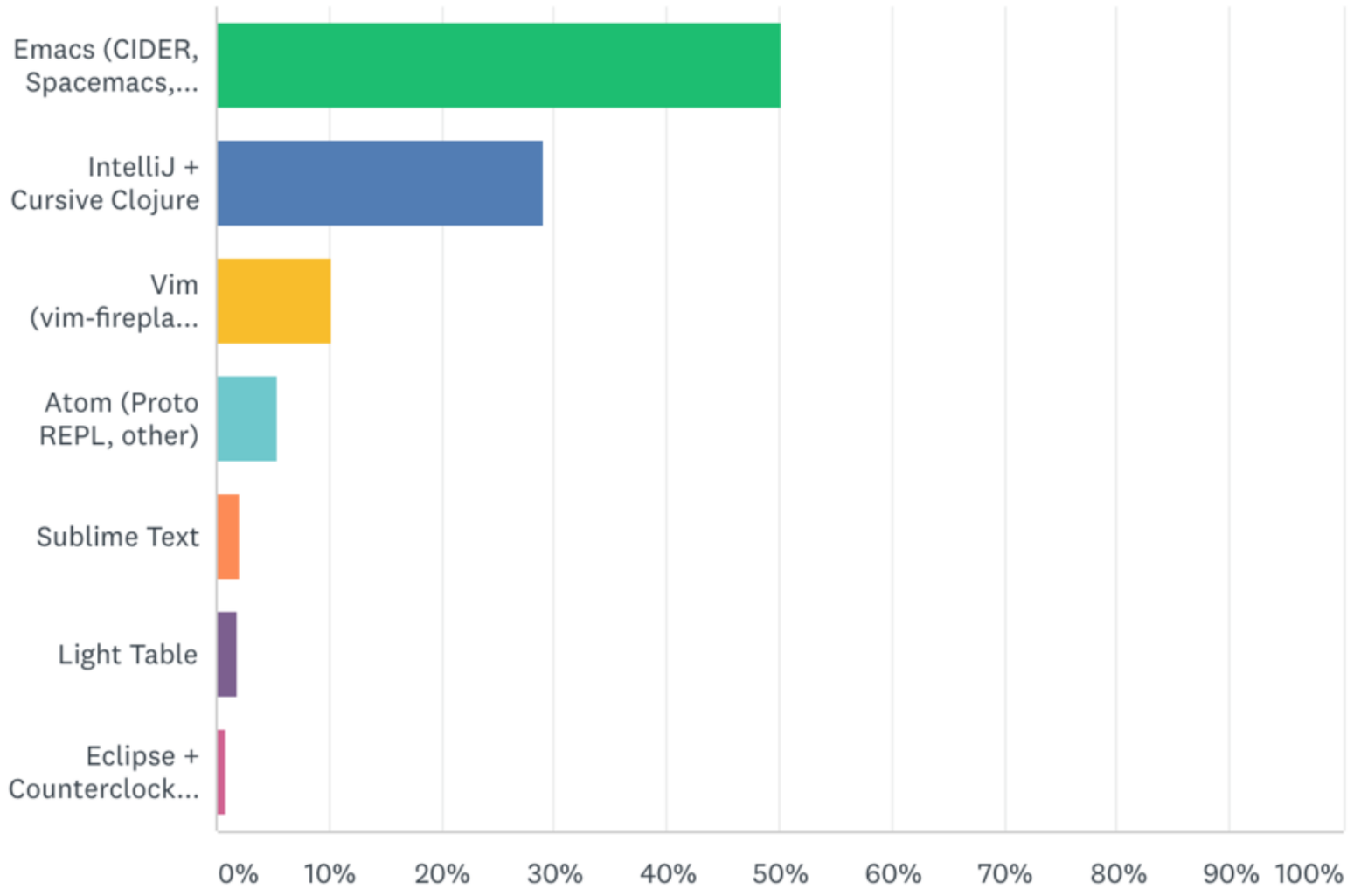
Commercial - $99 USD/$199 USD

Also has a free non-commercial licence for OSS, personal hacking, student work etc.

The Competition

What is your *primary* Clojure, ClojureScript, or ClojureCLR development environment?

Answered: 2,242 Skipped: 83



REPL Based Editors

The others are all *text editors* - they have no syntactic analysis.

They introspect a running system over a REPL to get the information they need to provide editor functionality.

Are very accurate - they represent the actual running system *for reified elements* by introspection.

Require application to be running - no editor support otherwise.

Generally don't allow renaming or indexing - only a very loose concept of a project.

Cursive

Works by analysing source code.

Everything is organised by project - allows full-project indexes.

Allows indexing and manipulating elements that are not reified at runtime (e.g. keywords, reader macro forms).

Works mostly transparently across Clojure & ClojureScript.

Why IntelliJ?

Why IntelliJ (for Java)?

True syntactic analysis.

IDE lexes and parses source code, builds AST.

Pervasive refactoring.

Flawless navigation.

Inspections - live static analysis in editor.

Intentions - many tiny code manipulations.

Why IntelliJ (for Java)?

I feel like I'm working directly on the syntax of my program.

The commands provided by the editor reflect almost exactly the manipulations I want to do to my program.

In a perfect world, this removes all distractions from the task you're trying to achieve.

This is always my ultimate aim with Cursive.

We're doing it wrong

"IntelliJ is great and all, but I type much faster in <something else>."

– Lots of people on the internet

We're doing it wrong

We should be manipulating our programs, not text.

Lack of typing is a *feature*, not a bug.

Text manipulation is the wrong level of abstraction.

But what about Paredit?

Necessary, but not sufficient.

Is not really syntax-aware, and provides no help with interesting problems of syntax.

Equivalent to ensuring sentences end with full stops.

clj-refactor is much closer to what we're looking for.

We're doing it wrong

"Never send an IDE to do a programming language's job."

– Lots more people on the internet

We're doing it wrong

More sophisticated or expressive languages are not a solution on their own.

Editor support is largely orthogonal to language features.

Arguably more sophisticated languages could benefit more from this sort of support.

Mostly this sort of support doesn't exist because it's very hard to develop, not because it's not desirable.

“I object to doing things that computers can do.”

– Olin Shivers

Macros in Cursive

The elephant in the room

The promise of Lisps

"Programmable programming languages"

Clojure code is represented using the language's own data structures (homoiconicity).

After code is read, it is compiled - during compilation, macros are expanded.

Can be used for simple additions to language syntax, or can fundamentally change the language.

Macro Examples

```
Connecting to local IDE...
Clojure 1.7.0
```

Macro Examples

```
Connecting to local IDE...
Clojure 1.7.0

(defmacro foreach [[sym coll] & body]
  `(loop [coll# ~coll]
     (when-let [[~sym & xs#] (seq coll#)]
       ~@body
       (recur xs#))))
=> #'user/foreach
```

Macro Examples

```
Connecting to local IDE...
Clojure 1.7.0

(defmacro foreach [[sym coll] & body]
  `(loop [coll# ~coll]
     (when-let [[~sym & xs#] (seq coll#)]
       ~@body
       (recur xs#))))
=> #'user/foreach

(foreach [x [1 2 3]]
  (println x))
1
2
3
=> nil
```

Macro Examples

```
Connecting to local IDE...
Clojure 1.7.0

(defmacro foreach [[sym coll] & body]
  `(loop [coll# ~coll]
     (when-let [[~sym & xs#] (seq coll#)]
       ~@body
       (recur xs#))))
=> #'user/foreach

(foreach [x [1 2 3]]
  (println x))
1
2
3
=> nil
```

Macros in Clojure

Most core functionality is actually macros - defn, let, etc.

Cursive has no way to see inside macros to see which symbols they define.

Symbols defined by macros may not even exist in source code.

Macro Example

```
(defrecord Point [x y])
=> user.Point

(->Point 3 4)
=> #user.Point{:x 3, :y 4}
```

Macro Example

```
(defrecord Point [x y])
=> user.Point

(->Point 3 4)
=> #user.Point{:x 3, :y 4}
```

Why not just expand macros?

Macros in Clojure are totally arbitrary code (CL style).

It might never terminate (and you cannot automatically determine termination — that's the Halting Problem).

It could use a large amount of resources (memory, time).

It could format your HD.

(with thanks to @fbellomi of CrossClj)

Why not just expand macros?

Also very difficult to relate macro expansion back to source.

Trivial cases are (relatively) trivial, but even moderately complex cases get very tricky.

Still difficult to handle cases such as virtual symbols.

Extensibility

Internally based around an extension API, which is used even for built-in forms.

Keyed off fully qualified head symbol for lists.

Extensions are just functions, can be re-used, composed.

API & existing implementations will be open source.

Extensibility

Local bindings

Global vars

Extracting index information - aliases, refers, imports…

Threading forms

Java class/interface definition

Formatting

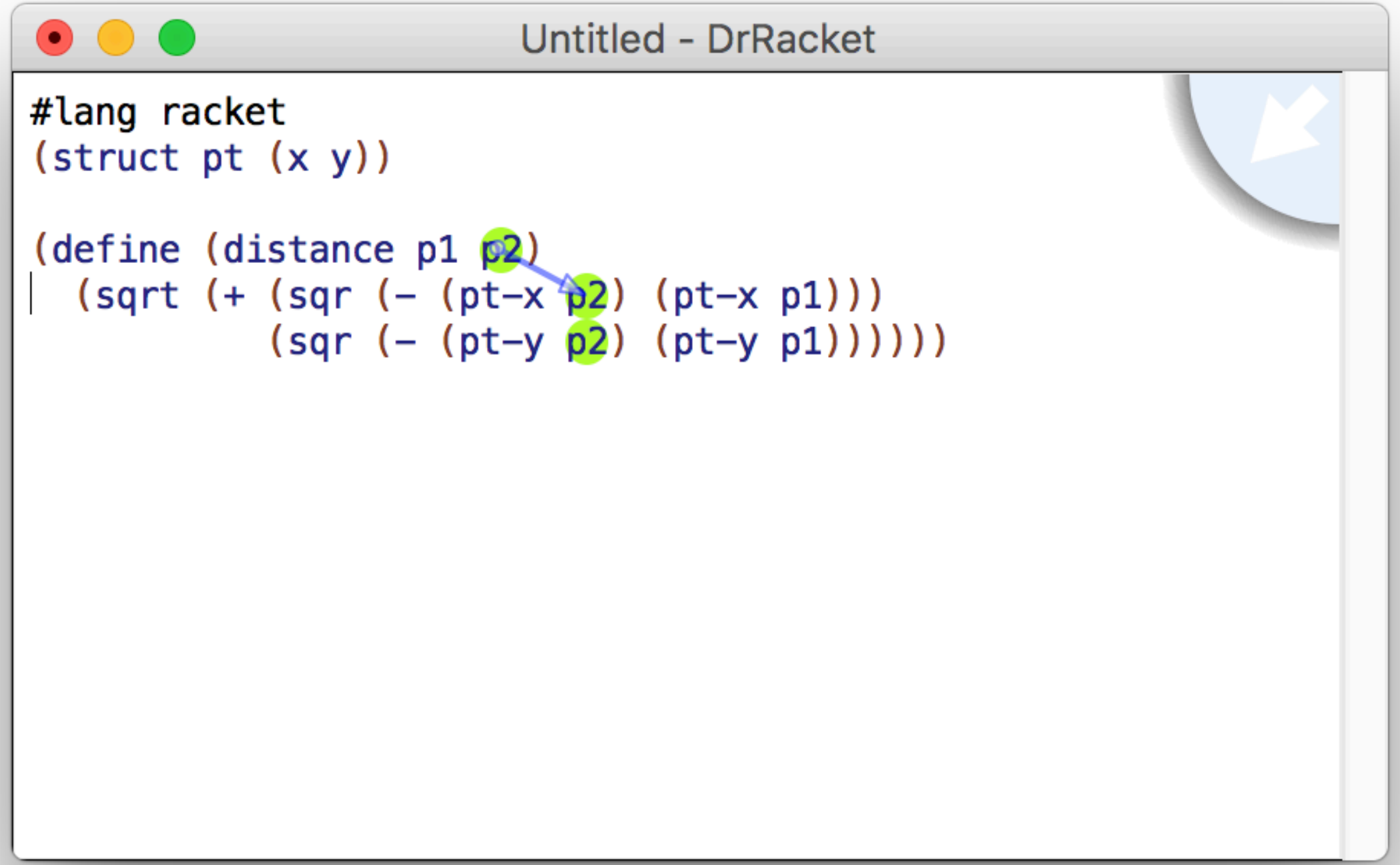
Type propagation information

Implicit grouping of forms (e.g. cond/case)

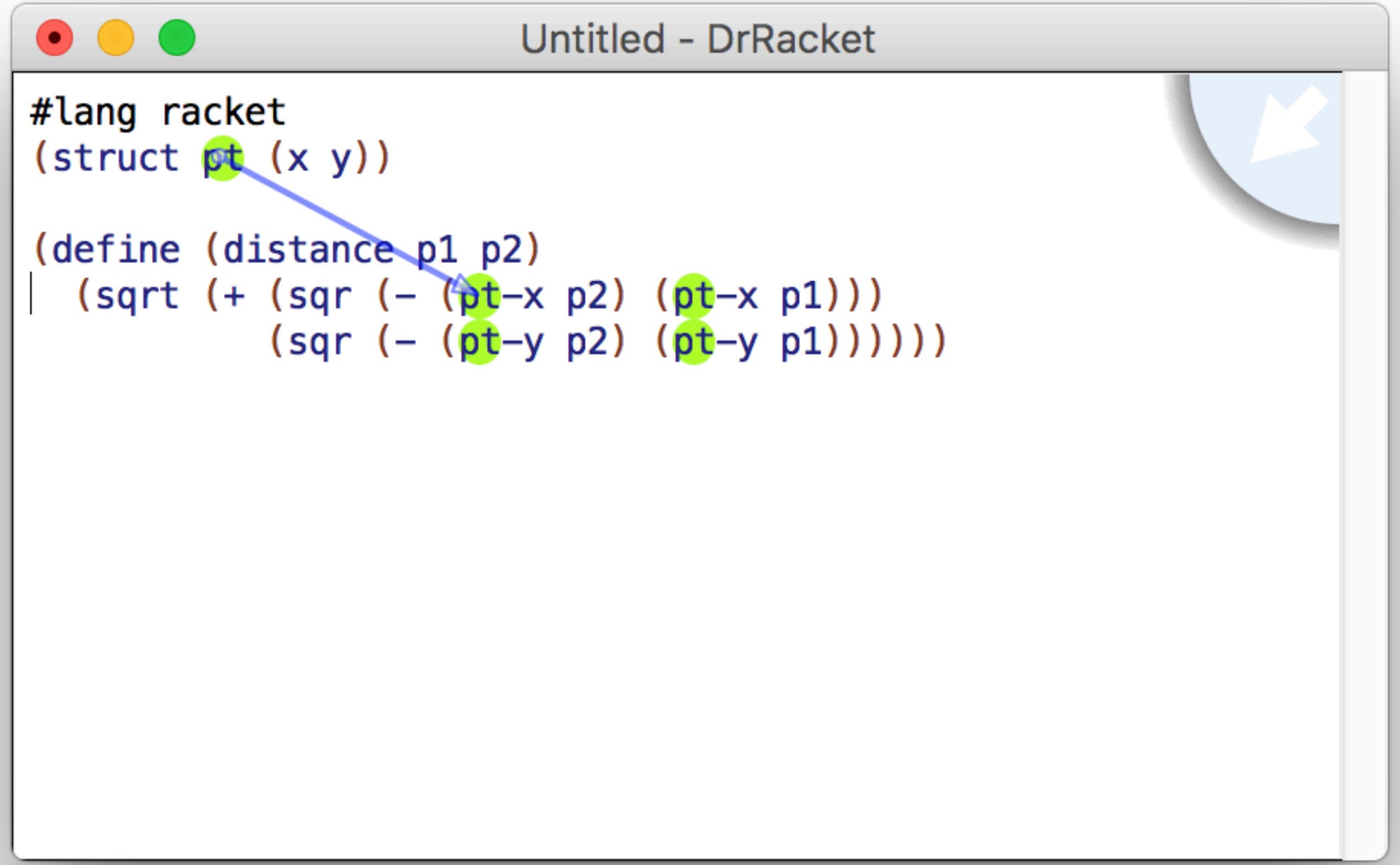
TBD: completion, interaction with refactorings & more

What would Racket do?

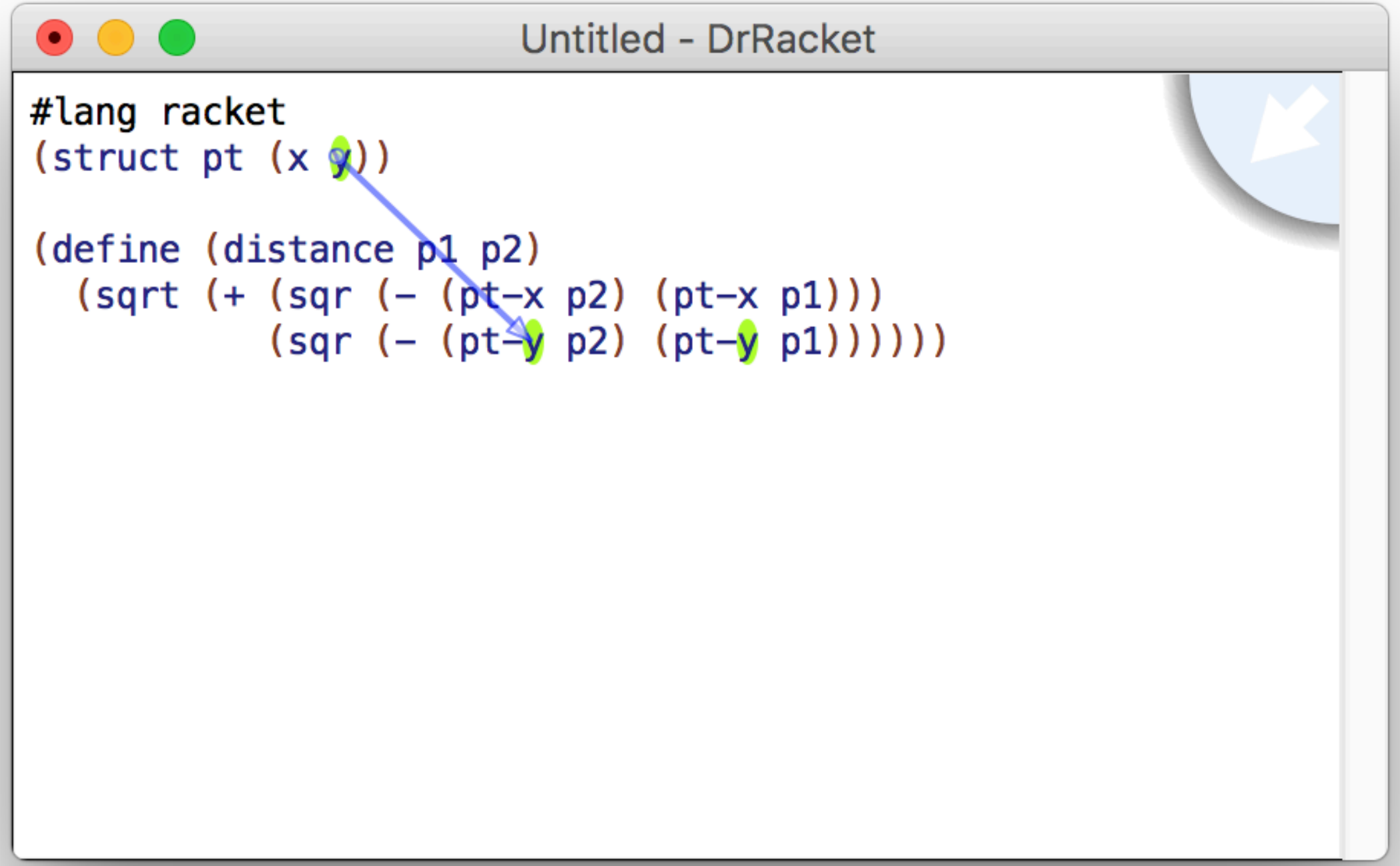
WWRD?


Untitled - DrRacket
#lang racket
(struct pt (x y))
(define (distance p1 p2)
(sqrt (+ (sqr (- (pt-x p2) (pt-x p1)))
(sqr (- (pt-y p2) (pt-y p1))))))

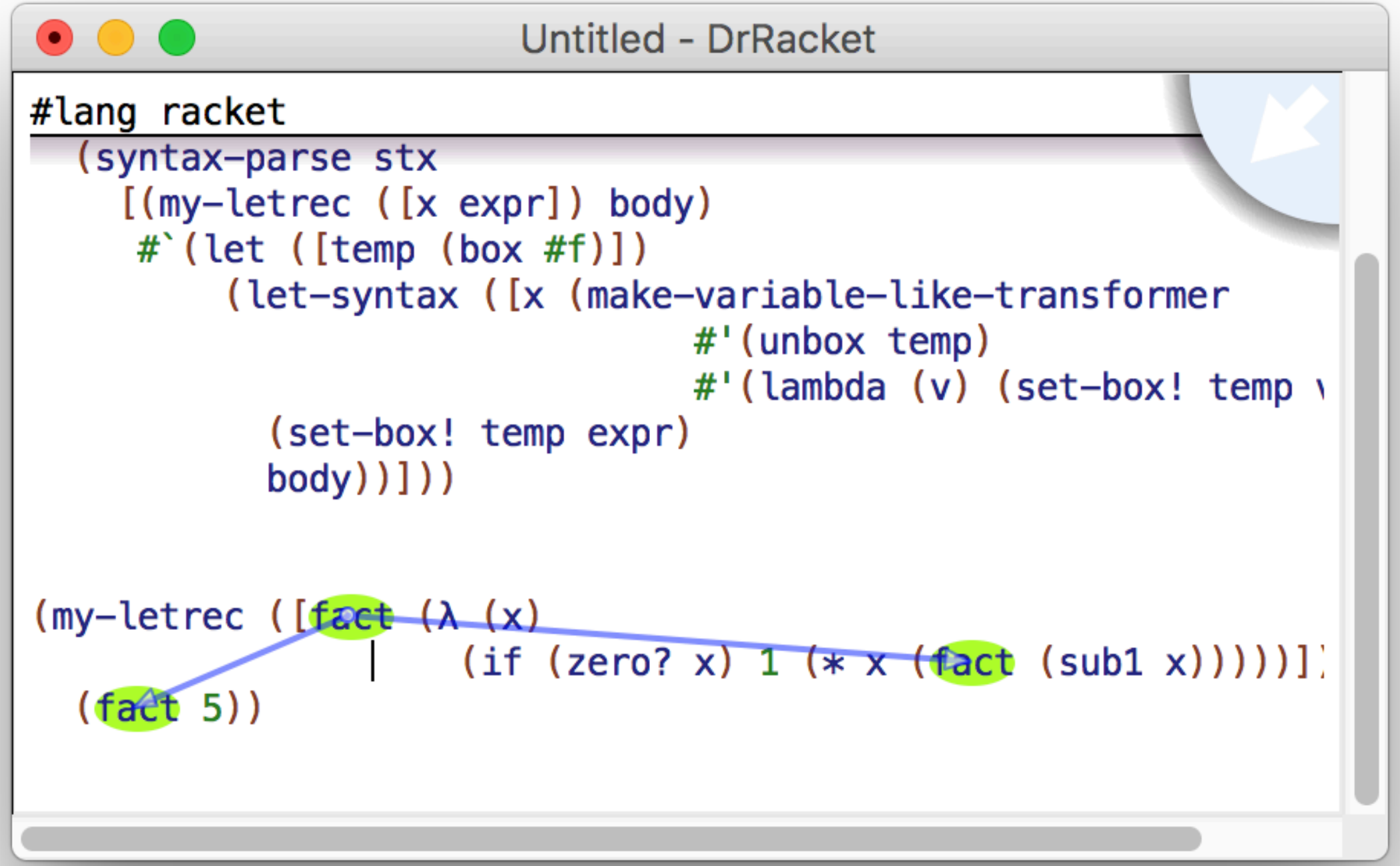
WWRD?


Untitled - DrRacket
#lang racket
(struct pt (x y))
(define (distance p1 p2)
(sqrt (+ (sqr (- (pt-x p2) (pt-x p1)))
(sqr (- (pt-y p2) (pt-y p1))))))

WWRD?


Untitled - DrRacket
#lang racket
(struct pt (x y))
(define (distance p1 p2)
(sqrt (+ (sqr (- (pt-x p2) (pt-x p1)))
(sqr (- (pt-y p2) (pt-y p1))))))

WWRD?


Untitled - DrRacket
#lang racket
(syntax-parse stx
[(my-letrec ([x expr]) body)
#`(let ([temp (box #f)])
(let-syntax ([x (make-variable-like-transformer
#'(unbox temp)
#'(lambda (v) (set-box! temp v
(set-box! temp expr)
body))]))
(my-letrec ([fact (λ (x)
(if (zero? x) 1 (* x (fact (sub1 x)))))]
(fact 5))

WWRD?

Many macros just work out of the box.

Macros can communicate with IDE via syntax properties.

Macros can provide information for tooltips, and hints for disappeared/composite forms.

But…

3k line macro-heavy file takes ~8 seconds to macroexpand.

Memory use is also an issue - full history is tracked through expansion process.

Cursive would have to macro expand in many more situations than DrRacket.

Cursive cannot macro expand during indexing - index data must depend on contents of file being indexed alone.

Racket's macro expander is *much* more sophisticated than Clojure's.

DrRacket is ubiquitous, Cursive is not - I can't assume macro authors will add Cursive support in Clojure.

tl;dr

We should be working on our programs, not on text.

Our tools need to be language aware, and provide language-specific manipulations.

Being hard to implement doesn't mean it's not desirable.

Languages should be designed with tooling in mind.

Thanks!

@CursiveIDE