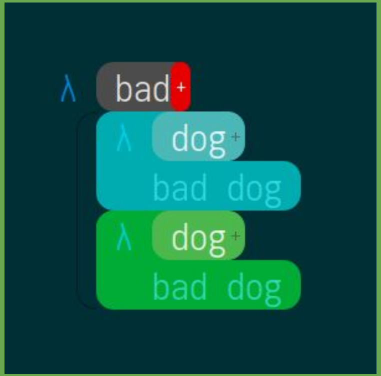


```
λ bad+
  λ dog+
    bad dog
  λ dog+
    bad dog
```

fructure

extended unedited version
do not present!!

andrew blinn

What is Fructure?

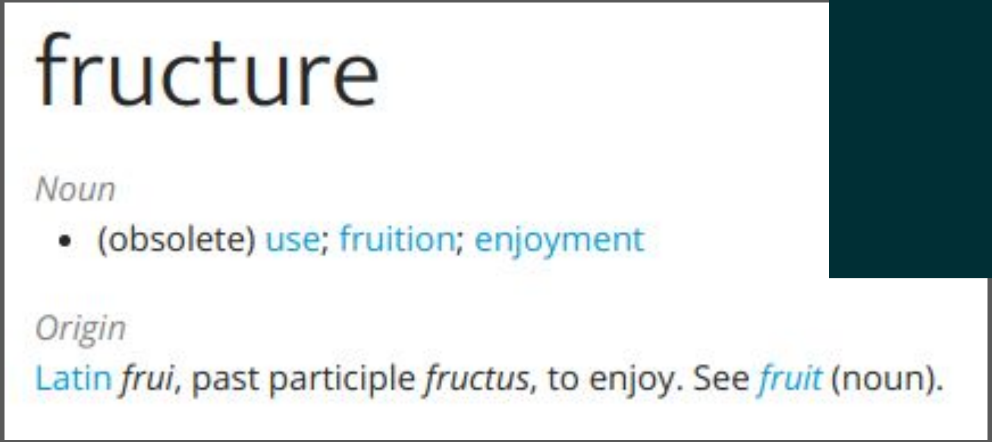
- Aspirationally: An engine for structured interaction
  - General-purpose tool for engaging with structured information in a structured way
  - Oriented toward - but not limited to - structures involved in programming: programs, data structures, run-time state, & user interfaces
- Initial goal, in progress: A prototype editor for racket code
  - Not usable beyond code snippets, but packaged so you can play around a bit
  - Not intended to go head-to-head with your time-honed emacs profile
- Designed from the ground up for easy extensibility to other lispy languages and data representations

Functional Reactive strUCTURE editor?

- (At least, that was the idea. Things have changed since the name)
- **Functional**? Yes! All object actions are declarative transformations of the whole object syntax; editor actions are transformations of the whole editor state; in fact, the distinction between the object syntax and editor state is intentionally blurred
- **Reactive**? Not really! Except in the trivial sense that the whole thing is an overgrown htdp world, driven by a (big-bang)
- **Structure editor**? Hold this thought

Fructure, Actually: Joy of Use

- Accidentally a real word
- More accurate than intended



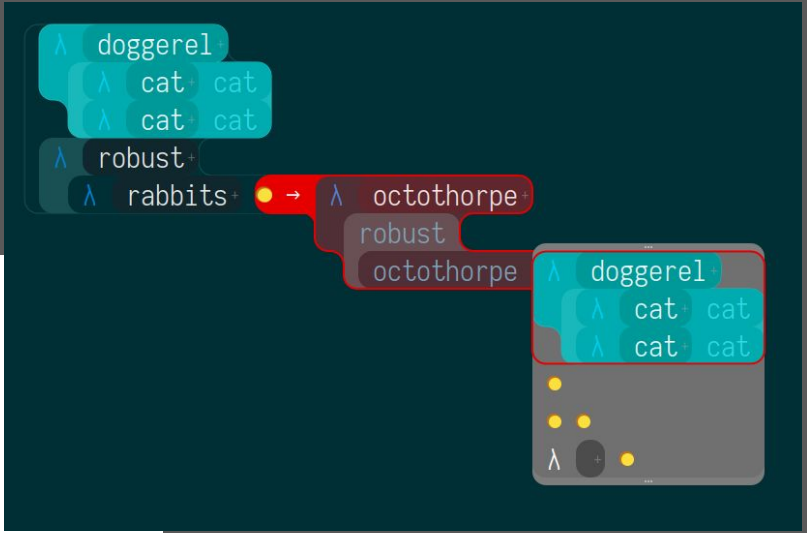
fructure

Noun

- (obsolete) use; fruition; enjoyment

Origin

Latin *frui*, past participle *fructus*, to enjoy. See *fruit* (noun).

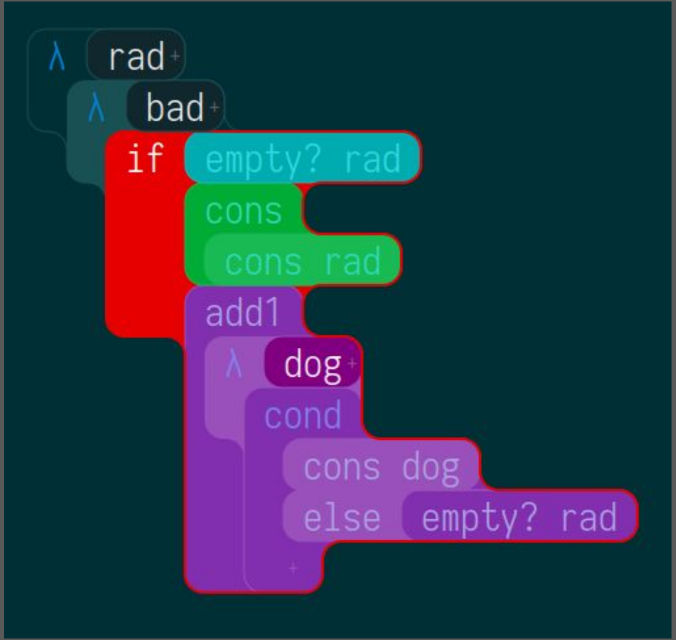


Structured editing

- AKA: structure editing, syntax-directed editing, language-based-editing, projectional editing.
- Related: visual programming, live programming & incremental parsing/evaluation
- Wiki: "editors cognizant of the document's underlying structure"
- True but uninformative, likely all-encompassing. in practice, often: editor operating on AST (or other non-linear representation) instead of text
- (insert: screenshots of other editors)

Structured editing, continued

- Many such editors since at least the 80s; no breakouts successes. (mention Jetbrains MPS)
- Limited structural editing in emacs+cursive with paredit, smartparens, parinfer.
    - (note about s-expressions editing & grammar hierarchy)
- My take: Editors where the ways you can transform data are informed by the data's semantics, not its serialization; 'structure' refers to structuring the editing process first, and the edited objects second



```
λ rad
  λ bad
    if empty? rad
       cons
         cons rad
       add1
         λ dog
           cond
             cons dog
             else empty? rad
```

What’s wrong with current editors?

1. Nothing! Modern options are more diverse & capable than ever before...
    a. Emacs: roll your own text editor
    b. Vim: compositional language for text editing
    c. Code/Atom/Sublime: Good defaults and consistent UI
    d. Your favorite IDE: deep language support

2. A lot of little things...
    a. (hard to generalize, given the above, Most things are possible, somewhere, But some simple things are stuck behind deep paths)

A lot of little things

- i'm clumsy. most editors give me a lot of undesirable states to fall into. it would be nice if i could selectively restrict this space of possibilities
  - Operating on off-by-one-character selections either breaks code (sometimes in invisible ways) or is disallowed, even when intent should be unambiguous
  - in many editors, editing operations are ad-hoc in that they are grounded in the text serialisation of an object rather than the nature of the object as a syntactic + semantic entity
  - navigation, initial entry, rewriting in particular are very text-focused, and refactoring often relegated to an unstructured junk drawer

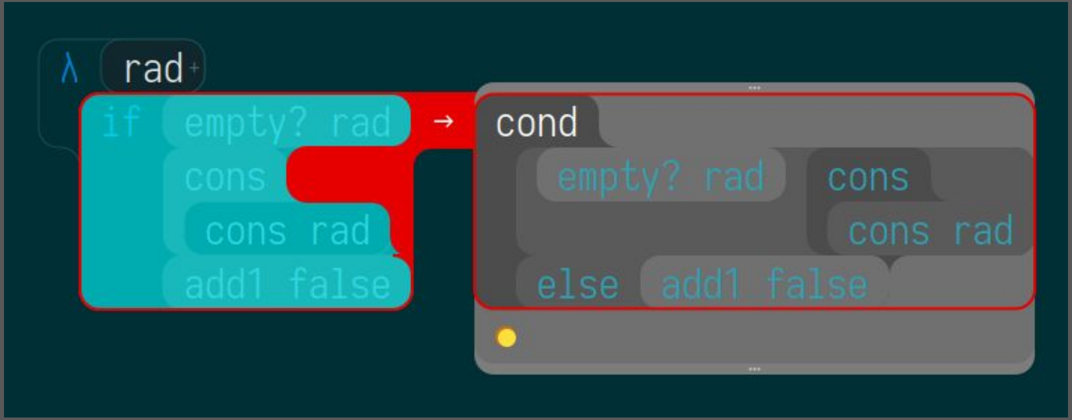
- need for complex non-semantic motor planning for simple transformations
  - a lot of editing is about destructuring and restructuring existing code. in most editors, this involves either moving back and forth or remembering what's in registers/ring
- Editor implicit state fights for limited working memory
- Keyboard shortcuts are often arbitrary; don't use our ability to build and recall spatial maps

… adding up to a lot

- Our object-languages and editor-concepts usually live in different worlds, resulting in:
- A brittleness & lack of solidity - editors should make sure that what you’re editing ‘feels real’, reflective of the nature of thing thing you’re editing, and avoid spilling implementation details at the user

A path forward

- The need for an editor strictly generalizing text
  - Without mandating a new language/ecosystem for end users
  - Basic usability similarly to a text editor, while offering other options
  - Not jumping directly to visual or live programming, but bridging the gap between text and richer structures



```
λ rad
  if empty? rad
     cons
       cons rad
     add1 false
```

→

```
cond
  empty? rad  cons
                cons rad
  else  add1 false
```

fructure’s editing model

- Transformation-based editing
  - An editing model, based on edit-time term-rewriting
  - base unit: meaningful whole-program transformations
  - specifying grammar via production rules
  - specifying simple refactorings via rewrite rules
- Global list of transformations, filtered by current syntactic context
- Entire editor state is an ast augmented with
  - 1. static/dynamic properties (for now, syntactic sorts & scope)
  - 2. syntactic affordances which serve as scaffolding informing edits
- Transforming Object vs Representation
  - Editing as rewriting the object language, navigating/projecting as rewriting the editor metalanguage which extends the object language
- A projectional layout engine for syntax-like things
  - Visually explicit editor state, spatially oriented editor actions

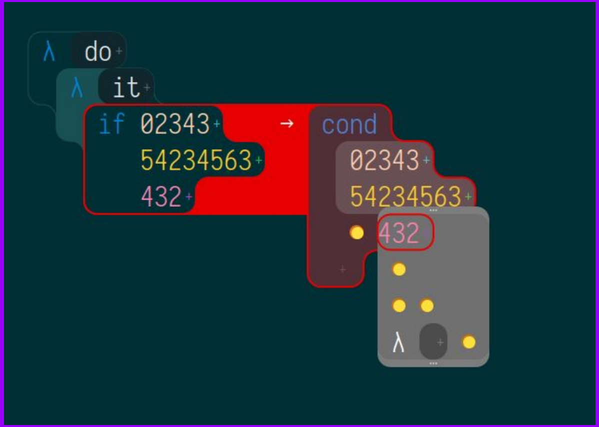
fructure editing model

Yes, this is the same as last slide. Condense!

- Editing as rewriting
- First, transformation rules on the object grammar
  - Expansion / Insertion
  - Contraction / Deletion - semantic implications
  - Refactoring aka transformation with guarantees
- Then, as transformation on a metagrammar: the object grammar enriched with syntactic affordances
- Syntactic affordances are edit-time widgets which
  - 1. Act as scaffolding for negotiating object-level transformations
  - 2. Act as guides for how syntax is projected to the real world
- With the goal of: language/editor conversation, cooperation, coeducation, & codevelopment
- (? exploration/debugging via rewriting-based eval sim)

Syntactic Affordances

- syntactic affordances - a widget/scaffolding metalanguage
  - determining transformations on the representation
  - informing transformations of the object structure
- Attempt at modular ui primitives
- none of these affordances are original - attempt at synthesis
  - (mention? In-line vs wrapping vs annotative affordances)
- Three basic affordances, which I’ll demo
  - Hole, Selector, Capture
- Three planned affordances, if there’s time
  - Top, Fold, Portal



demo

does this thing work?

Demo summary (not a real slide!)

- Video forthcoming (will serve as backup)
- Aiming for a couple minutes
- Outline - build up a small program
  - first, use menu/arrows, with enter after each selection
  - then show type to search (delete and redo?)
    - explain gaps in current parsing approach
    - make ‘a’ mistake (lol) and show undo
  - navigate though the program to show traversal
  - use captures to destructure an if
  - transform it manually to a cond
  - use automatic transform to turn it back

theory

the fructure editing model

Theory? In my editor?

- It’s approximately as likely as you’d think
- Emphasis in fructure not formalism per se, but expressivity which might help enable formalism - see Cyrus Omar’s incredible work on Hazel, where he develops an editor calculus based on actions which preserve well-typedness
- Fructure is more about an attempt at an expressive underlying model; providing a grounding for more esoteric extensions

Term-rewriting in Programming Languages

- Systems of declaratively-specified tree transformations
  - e.g. macro transformations
  - (let ([var rhs] …) body) -> ((λ(var. . .) body) rhs …)
  - (cond [a b] [else c]) -> (if a b c)

- Term-rewriting / Reduction Rules are used everywhere in PL to explain and extend languages at
  - Run-time
  - Compile-time

- But don’t see enough use at language
  - Edit-time!

Term-rewriting @ Run-time

- Not an implementation thing, for the most part
    - Exception: mathematica
- But for explanation: Language Semantics via reduction rules
- And for exploration: Dr. Racket Algebraic Stepper
- And both: PLT Redex
- The Revised Report on the Syntactic Theories of Sequential Control and State
    - Felleisen, Hieb
    - Writing an ‘affordance’ representing state of control flow / continuations into the tree

Term-rewriting @ Compile-time

- Very much an implementation thing
- Macro systems
  - Macros by example
  - Dr. Racket Macro Stepper
  - Fortyinfing macros (Culpepper, Felleisen)
    - (Wish I had read this before starting; provides a much more robust model for transforming annotated syntax)
  - Nanopass - multi-pass compilation

Term-rewriting @ edit-time

- Edit-time: relatively new term, used mostly in proof assistants, in reference to tactics; meta-languages for semi-automated proof composition

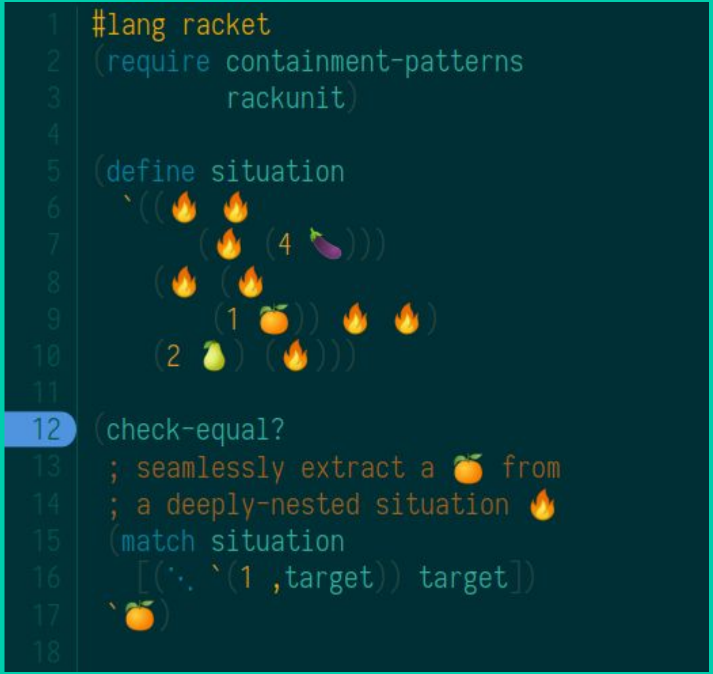
- Fructure, now:
    - Insertion via grammatical production rules
    - Simple refactorings via transformation rules

- Future, future:
    - Complex refactoring with extended pattern combinations + recursion schemes

Aside - Containment Patterns

A racket/match expander,
Implemented for fructure, but
more generally useful.

(more text + diagram forthcoming…)



```racket
#lang racket
(require containment-patterns
         rackunit)

(define situation
  `((🔥 🔥
       (🔥 (4 🍆)))
    (🔥 (🔥
        (1 🍊)) 🔥 🔥)
    (2 🍐) (🔥)))

(check-equal?
 ; seamlessly extract a 🍊 from
 ; a deeply-nested situation 🔥
 (match situation
   [(⋱ `(1 ,target)) target])
 `🍊)
```

Syntactic Affordance #1 - Holes

Holes act as scaffolding to maintain syntax shape and trigger production rules
See: Typed Holes in Agda, Haskell and esp. Hazel (Cyrus Omar)

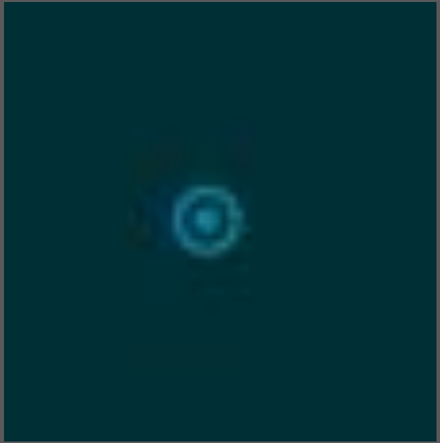


Fig 1: hole, in syntax

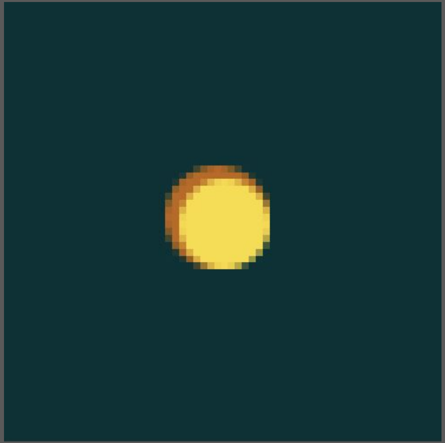
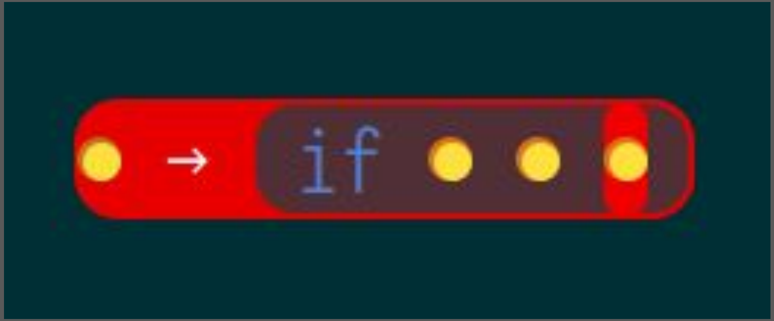


Fig 2: hole, rendered

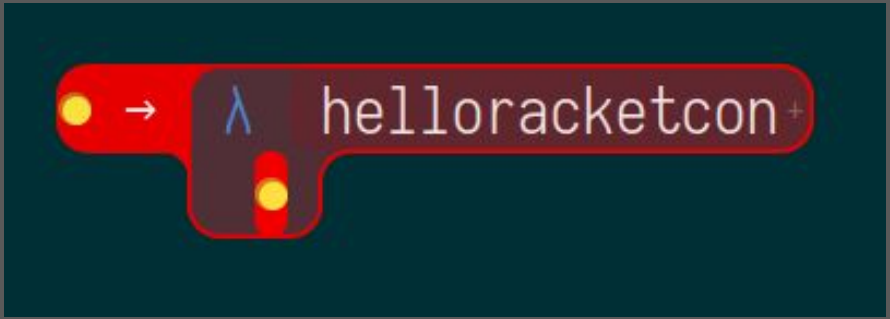
Holes - Inserting an if-expression

```
[⋱
   ([sort expr] xs ... / (▹ ⊙)) □
   ([sort expr] xs ... / (if ([sort expr] / ⊙)
                             ([sort expr] / ⊙)
                             ([sort expr] / (▹ ⊙))))]
```



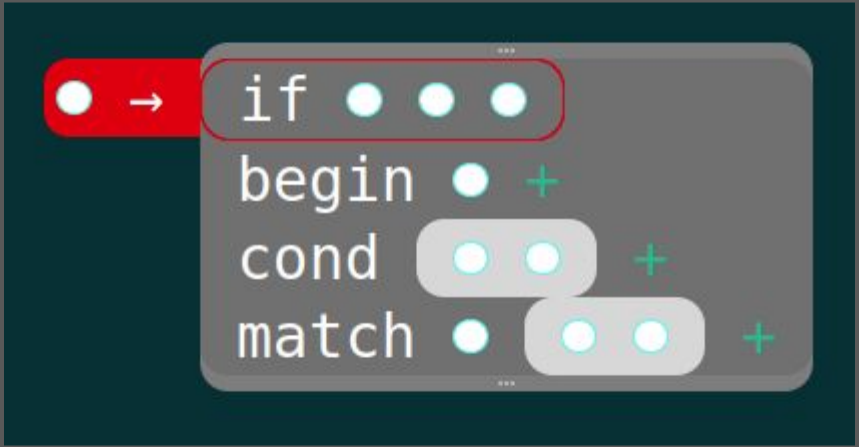
Holes - Inserting a lambda

```
[⋱
   ([sort expr] xs ... / (▷ ⊙)) □
   ([sort expr] xs ... / (λ ([sort params]
                              / (id ([sort char] / ⊙+)))
                            ([sort expr] / (▷ ⊙))))]
```



Holes - putting it together

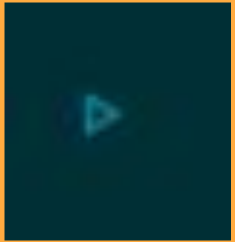
When you transform anything in fructure (here, a hole), you get a menu generated by finding all transformation rules (here, productions) with matching left-hand-sides.

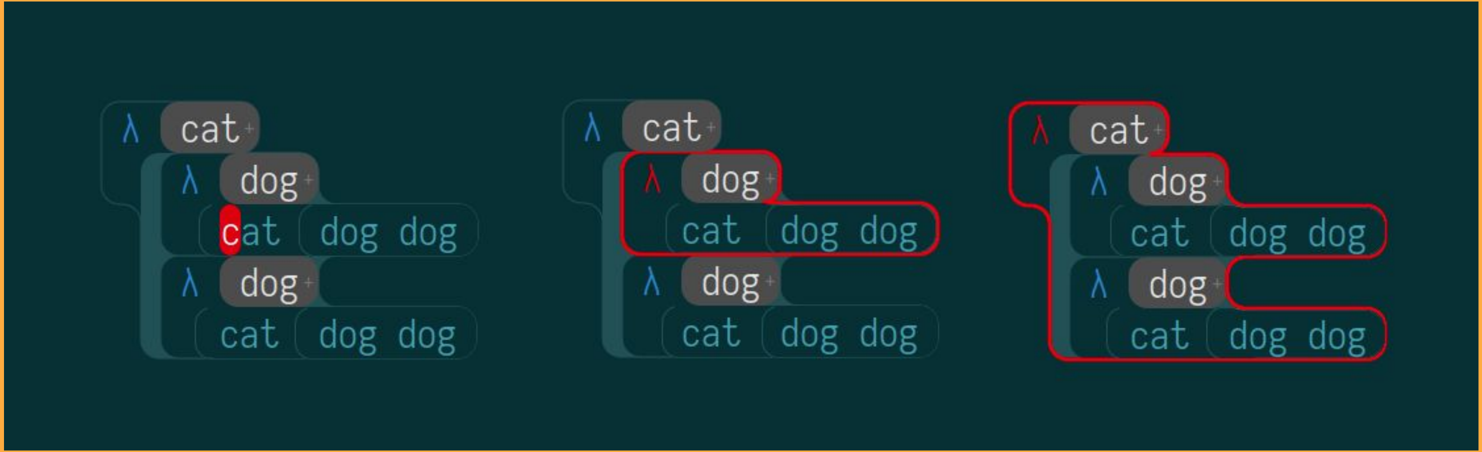


Syntactic Affordance #2 - Selectors

- Right: Selector & handle in code; (handles indicate selectability)

- Below: Selectors as rendered





Why write-in selectors?

Pros

- Directness
- Generalizability, e.g. multiple cursors

Cons

- Need to work harder for efficiency

Structural Navigation, Take One

```
(define select-first-child
  [(▷ (,a ,b ..)) ↦ ((▷ ,a) ,b ..)])


(define select-parent
  [(,a .. (▷ ,b ..) ,c ..) ↦ (▷ (,a .. ,b .. ,c ..))])


(define next-sibling-wrap
  (↓ [(,a .. (▷ ,b) ,c ,d ...) ↦ (,a .. ,b (▷ ,c) ,d ..)
     [(,a ,b ... (▷ ,c)) ↦ ((▷ ,a) ,b .. ,c)]))
```

Trying to extend Take One

Problem: Using naive tree-structured navigation is kind of annoying; how do we escape from deeply-nested forms?

```
(define next-escape
  (↓ [(,a … (▷ ,b) ,c ,d …) ↦
      (,a … ,b (▷ ,c) ,d …)]
     [(,x … (,a … (▷ ,b)) ,y ,z …) ↦
      (,x … (,a … ,b) (▷ ,y) ,z …)]
     [(,s … (,x … (,a … (▷ ,b))) ,r ,t …) ↦
      (,s … (,x … (,a … ,b)) (▷ ,r) ,t …)]))
```

What now??

Structural Navigation, Take Two

Contextual navigation example; using containment patterns to move upwards to the nearest containing handle:

```
[⋱
   (and (/ ⊃ a/
            (⋱ c⋱ (/ b/ (▷ b))))
          (not (/ ⊃ _/
                   (⋱ _⋱ (/ ⊃ _/
                              (⋱ _⋱ (/ _/ (▷ _))))))))
   (/ ⊃ a/
       (▷ (⋱ c⋱ (/ b/ b))))]
```

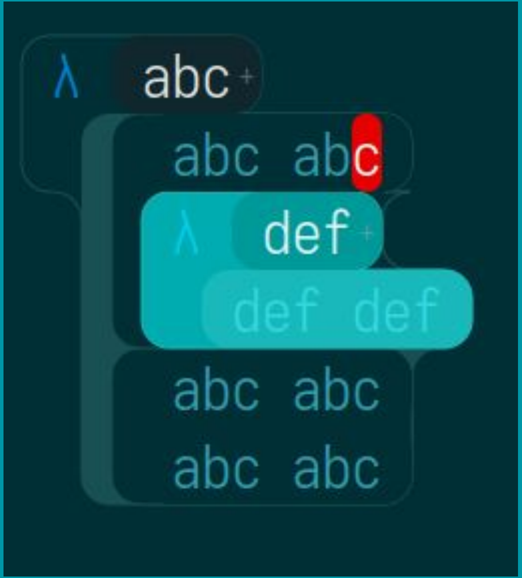
This approach strains the limits of clarity. Does succeed in declaratively expressing full tree traversals

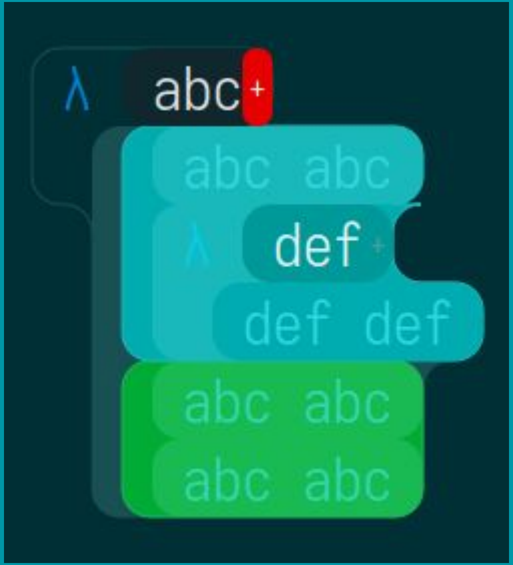
Structural Navigation, Take Three?

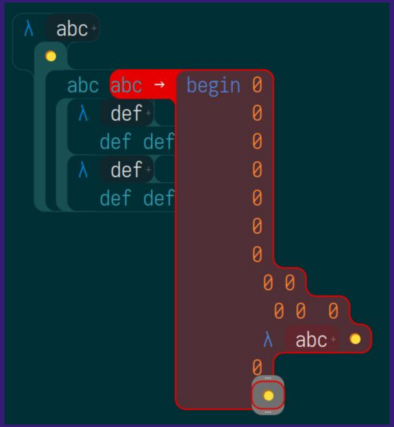
- Efficiency - Zippers
- Clarity - Breadcrumbs
- True spatial navigation - arrow key movement relying on actual spatial positioning data written into the tree
    - Follow principle of least surprise, but can interfere with linear traversal

Syntactic Affordance #3 - Captures

Paint-on destructuring as copy/paste alternative







extentions

plans & thoughts
for the future

Directions for extension

- Addressing user-desired syntactically invalid states
  - Sometimes the easiest way to get to a good state is through bad ones
  - Proposal: New modes complementing Walk (top-down insertion as demoed)
    - Grow mode - true bottom-up editing
    - Hatched editing - escape hatches in the grammar hierarchy
  - Hatched editing
- Leveraging Racket Syntax & GUI Tools
- Further mixing editor and language semantics
  - Dogfooding writing transformations
  - Code quasifolding
  - DAGs for naming, documentation, & temporary transclusion
- Towards deeper compositional editors

Stronger fundamentals

- Backend
  - Attribute grammar system for user-extensible semantic annotation
  - Using Racket syntax objects
    - Leveraging Syntax/parse?
- Frontend
  - Generalized 2d nested layout system
    - Constraint-based
    - Lessons from CSS
  - Using Dr. Racket GUI constructs?
  - My original attempt used nested editor snips, but I found this approach too difficult

Melt quasi-mode - Variadic editing

- Rearranging syntax & multi-selection
- Ctrl-arrows for cycling selected syntax through the sites where it can fit
- Optional: Forcing movement to non-compatibles sites triggers hatching (see hatched editing slide)

Quasi-captures & Uncaptures

- Say you copy a (recursive) function to use as a template for another… But you forget to rename the recursive call! A confusing and potentially hard-to-find bug.
- Now say that when you capture the function, you can ‘punch out’ certain subtrees (like the recursive call) as ‘uncaptures’
- When pasted, these become holes!
- (This Quasi/Un pattern, following quasi-quotation, seems widely usable for syntactic affordances)

Three more syntactic affordances

- Fold/quasifold
  - In current editors, folding offers a (non-semantic) mechanism of information-hiding. In a tree structure, folding can be semantic. Folds can be named, creating an abstraction. With quasifolds, subtress inside a fold can be marked as unfolded. When named, this provides an interface to directly create functions/macros by ‘painting over’ concrete code.
- Top
  - In current editors, the top-level of a view is the top-level of the code. The Top affordance reifies the outer extent of the view; use the selector to drag the top downwards to focus on a particular subtree, or push up to get an overview (combine with fold for abbv)
- Portals
  - Transclude portions of code, for cross-reference or abstraction (see Breckel & Tichy, LIVE2016). In particular, I want to use this functionality for re/naming; variable references are transclusions of the identifier at the definition site.

Meta-refactor: Making History Malleable

- Elaborating Transformation-based editing
- Part of the advantage of transform modes is it nests multiple edits together, at least somewhat semantically
- I want the ability to navigate the whole edit timeline, a similar interface as for syntax; from the above, this is a shallow tree
- This edit tree could be further nested and labelled to emphasize related edits
- (Commutative) edits could even be permutted for clarity, say if you made an unrelated change in the middle of a refactor
- Interface to create editor macros from history selections

Grow mode - Bottom-up editing

- Replace hole with stage; an sexpr with a variadic hole. The initial hole red-transforms to whatever (valid syntax, no scope/type checking) & another hole appears
- Create as many expressions as you want within the stage
- Either
  - Select a segment of expressions
  - Move to the front of the stage
- and you get a menu of wrapping options which take
  - That segment as a prefix of
  - A prefix of the stage
- as children.
- Stage outline colored based on scope/type checking of first element against stage context
- On execution, the first element of the stage fills the original hole and the rest is saved for later.

Hatched editing

- Escape hatches in the grammar hierarchy
- Menu search string is retained as a string
- There are (up to) three new menu entries
  - 1 Plain string literal / comment
  - 2 Quoted s-expression literal
  - 3 Syntactically valid but badly scoped/type/etc
  - 4 (Existing options: editor-semantically valid)
- 1-2 are hosted hatching; embedded in object semantics, so no new affordance is necessarily needed
- 1-3 can be transformed according to their nature
- Manual and automatic options ui options to parse/read (valid) forms of 1-3 to higher grammars, possibly incremental (removing a hatch while creating (syntactically) lower hatches meeting only (semantically) lower grammars)

Hyper-hatched compositional editing

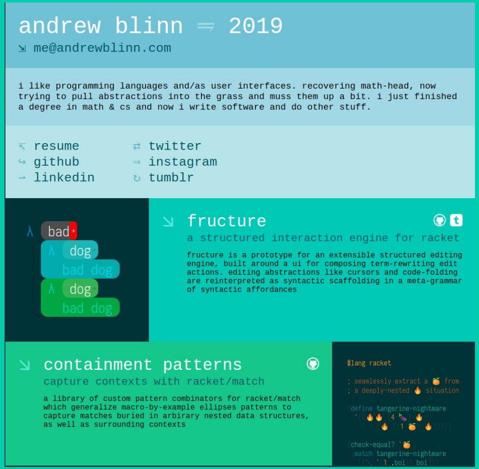
- Compositional Editors: glueing editors together
- From the top
  - Escaping code to the directory level
  - Navigating/Modifying files & folders with same interface as code
- From a bottom
  - Editing a complex literal in a literal-focused editor
- In the middle
  - See: magit in Emacs
  - Sometimes the best explanation/documentation for why code is the way it is is to showcase alternatives
  - Integrating CVS to show variations / past versions inline

Related Work

- cyrus omar - hazel
- sean mcdirmid - apx
- In racket: Leif Andersen - videolang.
    - Also racketcrew in general
- run-time datavis: weston cb, clojure REBL
- education languages
    - Scratch
    - Greenfoot
- lambdu, isomorf, code portals, prune, unison editor, darklang
- ‘Future of programming’ community
    - steve krouse podcast
    - jonathan edwards - subtext, slideshow

special thanks to
Gary Baumgartner

Questions?



andrew blinn ⥱ 2019

⇲ me@andrewblinn.com

i like programming languages and/as user interfaces. recovering math-head, now trying to pull abstractions into the grass and muss them up a bit. i just finished a degree in math & cs and now i write software and do other stuff.

⇱ resume	⇄ twitter
↪ github	⇒ instagram
→ linkedin	↻ tumblr

```
λ bad
  λ dog
    bad dog
  λ dog
    bad dog
```

↘ fructure

a structured interaction engine for racket

fructure is a prototype for an extensible structured editing engine, built around a ui for composing term-rewriting edit actions. editing abstractions like cursors and code-folding are reinterpreted as syntactic scaffolding in a meta-grammar of syntactic affordances

↘ containment patterns

capture contexts with racket/match

a library of custom pattern combinators for racket/match which generalize macro-by-example ellipses patterns to capture matches buried in arbirary nested data structures, as well as surrounding contexts

```
#lang racket

; seamlessly extract a 🍊 from
; a deeply-nested 🔥 situation

(define tangerine-nightmare
  `(((🔥 🔥 (4 🍆) 🔥
        (🔥 ((1 🍊) 🔥))))))

(check-equal? `🍊
 (match tangerine-nightmare
   [(⋱ `(1 ,boi)) boi]))
```

http://andrewblinn.com

Slide graveyard & staging area

ANYTHING PAST HERE ISN’T A REAL SLIDE

They are liars, don’t believe anything they say

UI misc

- rounded rectangle / racket/image graphics challenges

Some sentence fragments I like:

- Compiler as conversation model
- Collaborative Editing
- Explorability + Rich tutorial systems

Dr Racket is awesome & Why Racket?

- Image literals
- Xml boxes
- Plot
- Video lang embedded

Sidebar: text is amazing

- if you want to appreciate text, try to make an post-text editor
- in many ways, i've been converging back to text editing
- simplicity of 1D serilization
- 'free' 2D grid navigation
- existing algorithmic & tool support - efficient data structures, CVS

What is editing?

- loop: find -> change -> compare
- changing the object, and changing the representation
- minimal editor - displays only the code
- why does an editor show anything else? - editor meta
- scaffolding
- editors guide - finding code based existing code or partial information
- editors emphasize - cursor, selection, syntax highlighting
- editors restrict - view:scrolling, zoom, code folding
- non-view?
- editors juxtapose - windowing/sidebaring/modals - comparing pieces of code, or code and associated static/dynamic data
-

Editing Sexprs, Take One

```
(define delete
  [(,a .. (▷ ,b ..) ,c ..) ↦ (▷ (,a .. ,c ..))])


(define wrap-sublist
  [(▷ (,a ..)) ↦ (▷ ((,a ..)))])


(define new-sibling-right
  [(,a .. (▷ (,b ..)) ,c ..) ↦ (,a .. ,b .. (▷ (😊)) ,c ..)])
```

Paredit-style Transformations

```
(define pop/splice
  [(,a .. (▷ (,b ..)) ,c ..) ↦ (▷ (,a .. ,b .. ,c ..))])

(define slurp-right
  [(,a .. (▷ (,b ..)) ,c ,d ..) ↦
   (,a .. (▷ (,b .. ,c)) ,d ..)])

(define barf-right
  [(,a .. (▷ (,b .. ,c)) ,d ..) ↦
   (,a .. (▷ (,b ..)) ,c ,d ..)])
```

The problem of context

We have a transformation we want to apply, but where do we want to apply it?

Cursors are a solution, but what about when our transformations are describe cursors themselves?

What if we want to refer to the context around some syntax as a first-class object?

ui considerations

- Insertion versus selection
    - The menu
    - Spatial selection
    - Typing as search
    - (? audio+ as search)
- Walking the space of programs; inverting autocomplete
- ‘Normal’ text entry possible
- Destructuring and restructuring for transformation
    - Painting patterns onto the syntax
    - As copy/paste

Something about grammar hierarchies

- Editing text
    - Aside: what is a word? (no common definition)
- Editing s-expressions
- Editing languages
- (maybe escape hatch?)