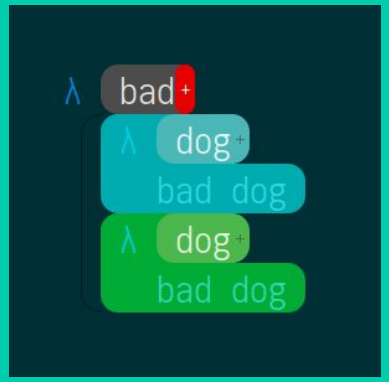


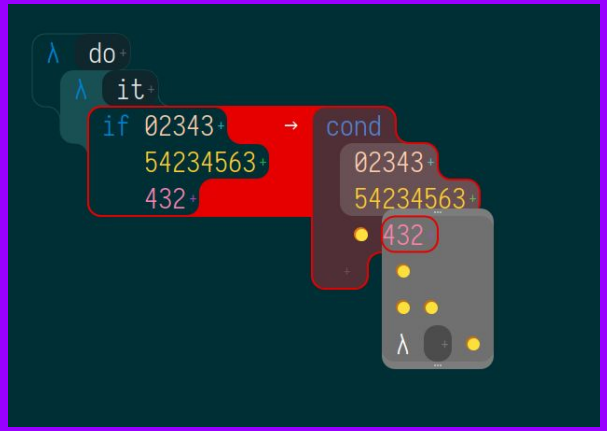
fructure

a structured interaction engine for racket

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 - the structures involved in programming: programs, data structures, run-time state, & user interfaces
- Actually: A prototype structured editor for racket code
  - Edit actions preserve syntactic well-formedness
  - Currently suitable for code snippets in a tiny subset of Racket
  - Not intended to go head-to-head with your time-honed emacs profile
- Designed from the ground up for straightforward extensibility



demo

structure editing show & tell

Why Fructure?

- **Functional Reactive Structure Editor?**
- Fructure (noun, obsolete): use; fruition; enjoyment
  - “From Latin frui, past participle fructus (to enjoy). See fruit (noun)”
- Structure editor?
  - **Wiki**: "editors cognizant of the document’s underlying structure”
  - **Prosaically**: Editor actions preserve syntactic (or, in some cases, semantic) validity

What’s wrong with current editors?

1. Nothing! Modern options are more diverse & capable than ever before…

2. A lot of little things…

A lot of little things...

- Text-based layout encourages needless fiddling about code formatting (locally) and language syntax (globally)
- Operating on non-meaningful selections either breaks code or fails silently
- Complex, non-semantic motor planning needed for destructuring and restructuring existing code
- Implicit editor state fights for limited working memory: registers/ring
- Keyboard shortcuts often arbitrary and don't leverage our ability to build spatial maps
- Refactoring often relegated to an unstructured junk drawer

Adding up to a lot

- Our object-languages and editor-concepts usually live in different worlds, creating an endless source of friction

A way forward

- The need for an editor strictly generalizing text-based input
  - Without mandating a new language/ecosystem for end users
  - Basic usability similarly to a text editor, while offering much more on the side
  - Not jumping directly to visual or live programming, but bridging the gap between text and richer structure

theory

the fructure editing model

Fructure's editing model - I

- Transformation-based editing
  - An editing model, based on edit-time term-rewriting
  - Base unit: meaningful whole-program transformations
- A global list of transformations, filtered by current syntactic context
  - specifying grammar via production rules
  - specifying simple refactorings via rewrite rules

Fructure's editing model - II

- Backend: Editor state is an AST augmented with
  - Static properties (later: dynamic)
  - Syntactic affordances, which serve as scaffolding informing edits
- Frontend: A projectional layout engine for syntax-like things
  - Visually explicit editor state, spatially oriented editor actions

Fructure's editing model - III

- Combined model for object and editor transformations
  - Edits are rewriting on the object language
  - Navigation/projection are rewriting on the encompassing editor metalanguage
- Turning informal editing tactics into editor concepts, via…
- Syntactic affordances: edit-time widgets which
  - Are an attempt at modularizing ui primitives
  - Act as scaffolding for informing object-level transformations
  - Act as guides for how syntax is projected & navigated on-screen

Building for edit-time semantics

- My emphasis is not on formal semantics but instead UI to support integration of semantic features into editors in a less ad-hoc way
- For formal foundations for editing, see Cyrus Omar's work on Hazel, where he develops an editor calculus based on actions which preserve well-typedness
- Fructure itself is more about moving towards an expressive underlying model, providing a grounding for more esoteric extensions

Term-rewriting / Reduction Rules

- These terms have specific technical connotations, but here I use them loosely to refer to systems of declaratively-specified syntax transformations

- A canonical example:
  (let ([var rhs] …) body) -> ((λ(var. . .) body) rhs …)

- Term-rewriting / reduction rules are used everywhere in PL to explain, explore, and implement languages

Term-rewriting / Reduction Rules

- At run-time
  - As an evaluation model, e.g. Mathematica
  - For explanation: Language Semantics via reduction rules
  - For exploration: Dr. Racket's algebraic stepper, PLT Redex
  - The Revised Report on the Syntactic Theories of Sequential Control and State
- At compile-time
  - Extending languages: Macros by example, Fortifying macros - syntax/parse
  - Exploration/debugging: Dr. Racket Macro Stepper
  - Simplifying compilation: Nanopass
- At edit-time?

Term-rewriting @ edit-time

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

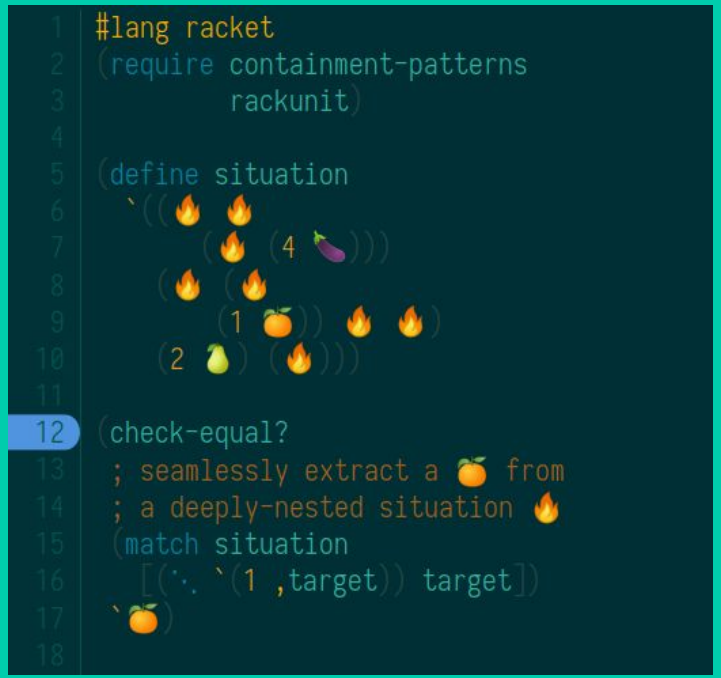
- Fructure, now:
  - Insertion via grammatical production rules
  - Simple refactorings via transformation rules
  - Simple editor navigation & projection via transformations + syntactic affordances

- Fructure, future:
  - Richer refactoring with extended pattern combinators + recursion schemes
  - More syntactic affordances supporting richer navigation & projection

Aside - Containment Patterns

A racket/match expander, for capturing contexts with composable continuations.

Implemented for fructure, but more generally useful.



Syntactic Affordance #1 - Holes

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

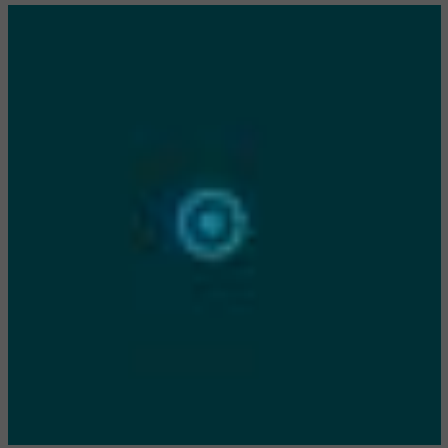


Fig 1: hole, in syntax

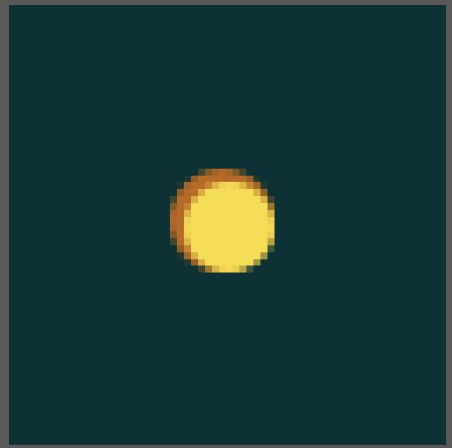


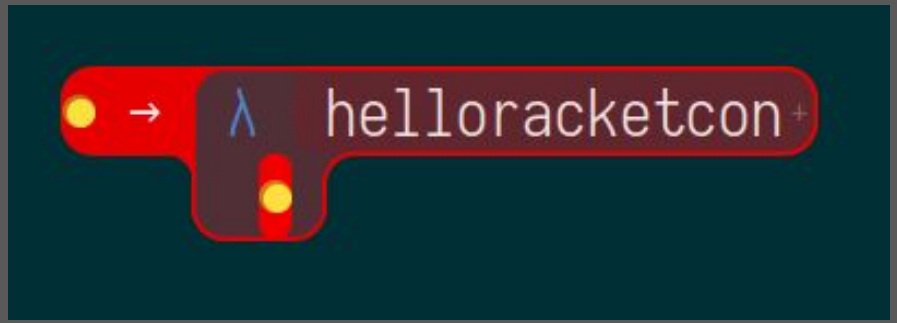
Fig 2: hole, rendered

Holes - Inserting an if-expression

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

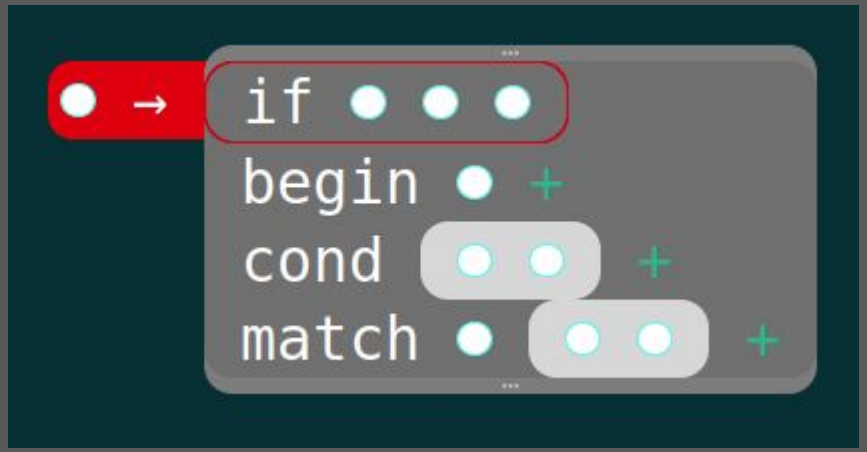
Holes - Inserting a lambda

```
[⋱
   ([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.



How do I fructurize my #lang?

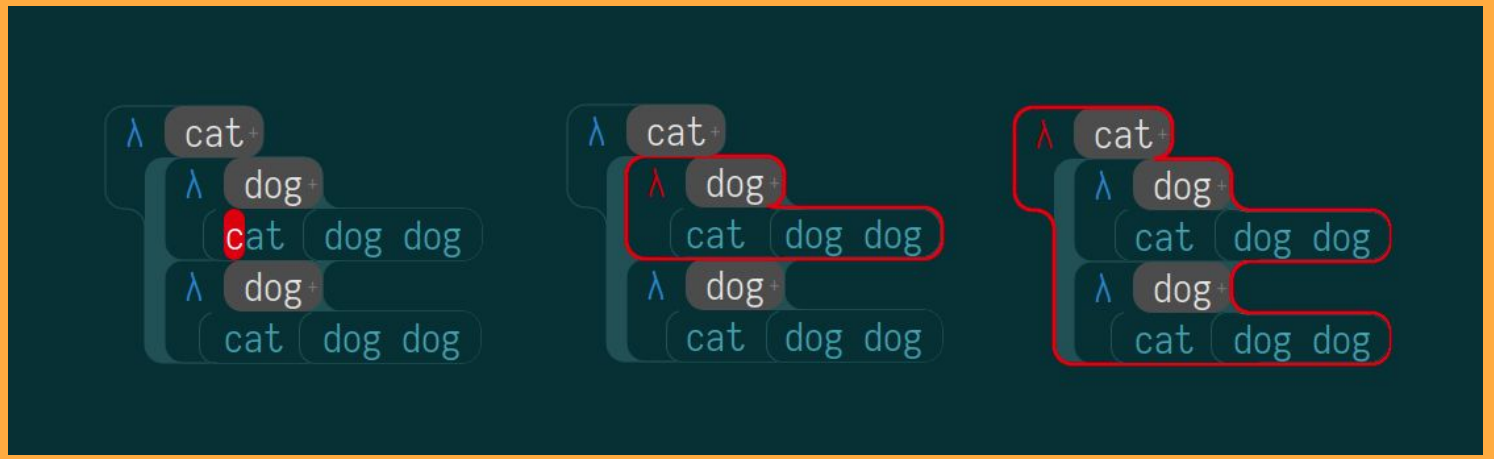
Currently, you'd write production rules for each form, as well as provide rules for propagating / modifying scope.

You'd also define layout for each form, or simply reference an existing form's layout

In principle, though, many production rules, scoping rules, and even simple refactorings should be derivable automatically from (some classes of) macros

Syntactic Affordance #2 - Selectors

- Right: Selector & handle in code; (handles indicate selectability)
- Below: Selectors as rendered



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 …)]))
```

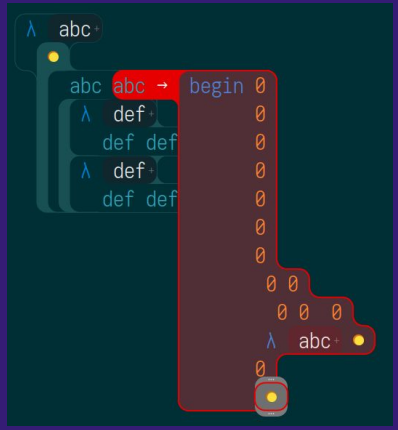
What now??

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, but succeeds in declaratively expressing tree traversals (not shown)



extentions

plans & thoughts
for the future

Problems & Future Directions

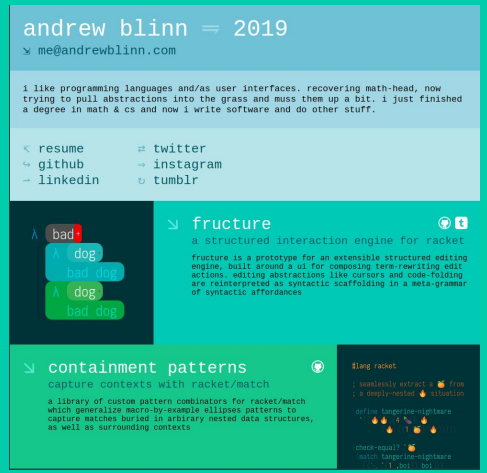
- What if… syntax errors are actually good??
  - Sometimes the easiest way to get to a good state is through bad ones
  - Proposal: New modes complementing Walk (top-down insertion as demoed)
    - Grow mode - bottom-up editing
    - Hatched editing - escape hatches in the grammar hierarchy
    - Composite actions enforcing eventual correctness
- Leveraging the Racket #language ecosystem
  - Automatically populating (some) transforms via macro analysis
  - Converting backend to syntax objects; (some) transformations via syntax parse
- Further mixing editor and language semantics
  - Breaking the dogfooding barrier
  - Multi-holded contexts for direct abstraction
  - History, meta-refactoring & exploratory programming
- Towards deeply compositional editors

Related Work

- Cyrus Omar
  - hazel/hazelnut
  - Toward Semantic Foundations for Program Editors (Omar et al)
- Leif Andersen's embedded videolang editor
- Sean McDirmid's APX live programming environment
- Run-time Datavis: Clojure REBL, Weston CB
- Education languages
  - Scratch
  - Greenfoot
- lambdu, isomorf, code portals, prune, unison editor, darklang
- 'Future of programming' community
  - steve krouse's podcast
  - jonathan edwards - subtext

special thanks to
Gary Baumgartner

Questions?



http://andrewblinn.com