

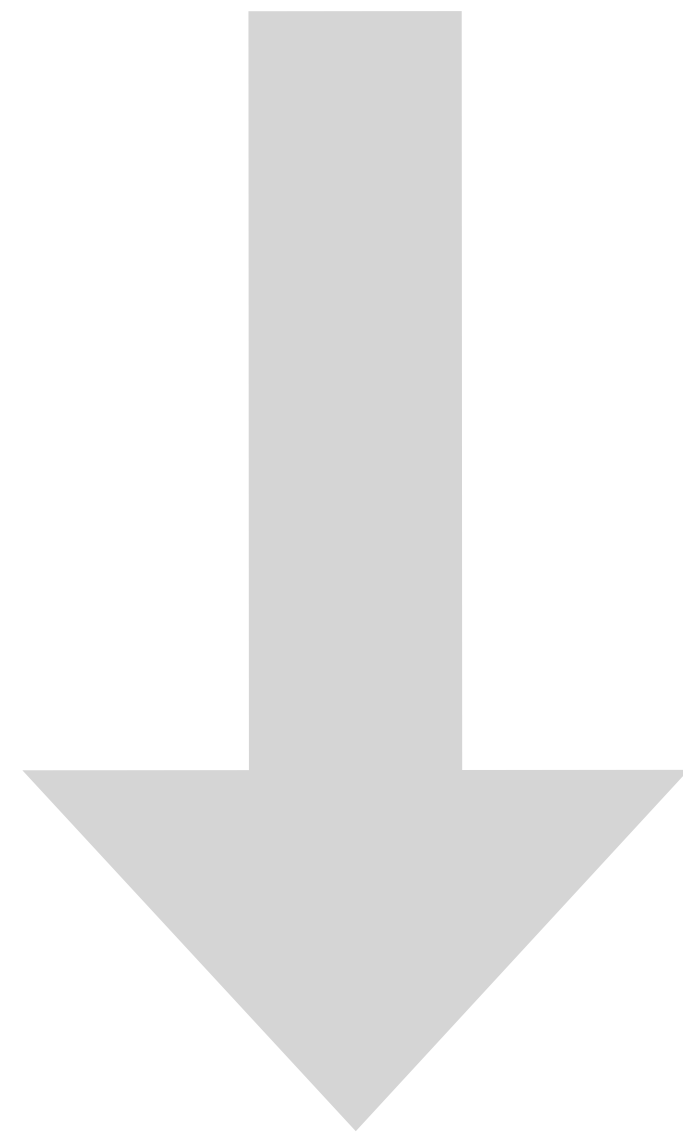
Final Project Guidelines

Dec 2

Building a DSL

Roadmap

Learn a bunch of **modern PL features** in Racket: recursion, function composition, currying, closures, higher-order-functions, contracts, type inference...



Syntax & AST, defining the *implementation* of a PL interpreter in terms of itself...

Extend our PL with **Macros**, syntax transformations, syntactic sugar...

Implement toy DSLs with **syntax and semantics defined by us** (you).

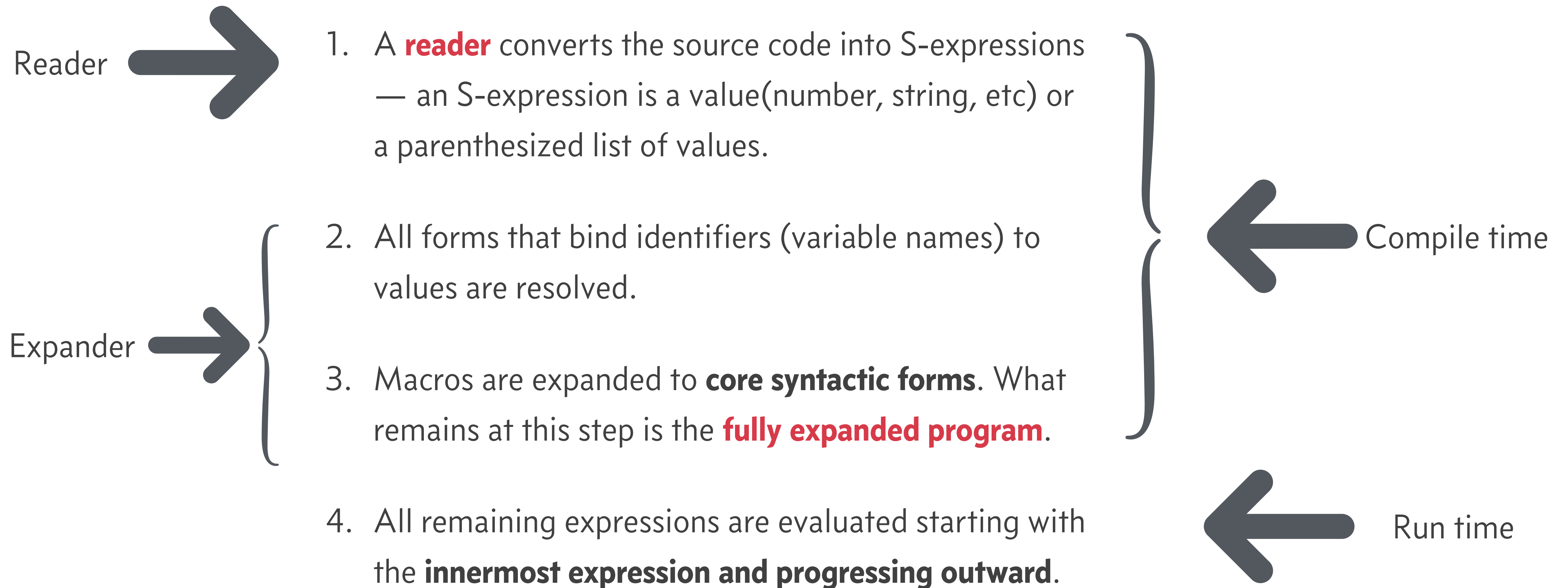
How PLs implemented using interpreters work

$$\text{interpreter} :: \text{Program}_L \rightarrow \text{Value}$$

The default implementation of Racket is as **an interpreter written in Scheme, C++ and Racket itself**.

Crucially an interpreter depends on having a **previous binary of the implementation language** that actually interprets programs to generate values.

Evaluation model of Racket






The recipe for creating a DSL in Racket:

Write your own reader and expander that **converts source code in your DSL to a fully expanded program in Racket.**

Final Project options

Your final submission should be one of these

- Beautiful Racket has a tutorial building the BASIC programming language in Racket from scratch. Implement that tutorial, **but using only racket/base and importing only necessary functions beyond that**. I will also ask you to implement some more features beyond the (first) tutorial.
- Implement a DSL to **solve another Advent of Code problem like wires**.
- Design your own DSL to solve any problem, or class of problems that you like!

```
sample.rkt ▾ (define ...) ▸  Check Syntax  
```

```
1 #lang basic-demo-2
2 30 rem print 'ignored'
3 35
4 50 print "never gets here"
5 40 end
6 60 print 'three' : print 1.0 + 3
7 70 goto 11. + 18.5 + .5 rem ignored
8 10 print "o" ; "n" ; "e"
9 20 print : goto 60.0 : end
```

For the final project, decide if you want to work in groups or individually. Your final submission will just be the Racket source files. Your "finals" will just be a 10 minute meeting with me where I ask some questions about your DSL, how you added features, etc. You will be graded on:

◆ **Completeness (50%)**

Did you implement all the features I asked/does it work on any test program I give it?

◆ **Style/Readability (25%)**

Is the code easy to read and understand? Have you added comments where necessary? Are functions/modules organized and written properly? Most of your time as a programmer is spent reading code not writing it, so write code that is readable and easily understandable.

◆ **Tests (25%)**

You need to write **3 example programs** showcasing what your DSL can do, its limits, and any new features you added to it.

Style/readability (25 points)

Common to both project options

- **Separate reader.rkt, expander.rkt, main.rkt** and any helper files/functions.
- Separate test files (3 each).
- I should be able to install your DSL as a package — have a main.rkt file and ensure that its possible to install your DSL as a package for me to test and use.
- Comments for any custom functions/code that you write that needs disambiguation.

BASIC option

1. Follow the **first** basic tutorial on Beautiful Racket and replicate it. You will be **building on top of that**.

Completeness

Features you must have for all 50 points

- Implement conditionals (if ... then ... else ...) and boolean expressions, variables, and user input (30 points)
- Implement the language in **racket/base** instead of **br/quicklang** and **br**. (20 points)

<https://beautifulracket.com/appendix/from-br-to-racket-base.html>

```
#lang racket/base
(require (for-syntax racket/base racket/syntax))
(require ...
```

- You can get partial credit if you find it hard to implement some features without br/quicklang. Import only the br libraries you need in that (br/macros, br/syntax, etc). **Its fine to use brag for tokenizer/parser/lexer. The more source code files in your package that start with #lang racket/base, the higher your score.**

Test programs

All 3 for 25 points

- test.rkt (5 points): One of the base test cases in the Beautiful Racket tutorial that functions as expected. Or anything you want
- prime.rkt (10 points): Program that calculates **the nth (user input) fibonacci number**. You should be able to input a positive number and it will print the nth fibonacci number to screen. **Use if-then-else and goto**
- fibonacci.rkt (10 points): Program that calculates if **user inputted number is prime or not**. It should simply print "prime" if the number is prime, else it should print "not prime" **Use if-then-else and goto**

Name your language **basic-<your initials>**.
I should be able to install it as a package and load
it using **#lang basic-<your initials>**

Advent of Code option

You'll be solving Advent of Code 2023, Day 8
problem Part 1: Haunted Wasteland.

<https://adventofcode.com/2023/day/8>

There are 3 test cases: 2 on the webpage for the problem, and a big one that you can download if you sign up. It should give **the right answer for all 3 files.**

Completeness plus tests

Simpler evaluation for this one

- However you choose to solve the algorithm, if you get the right answer for all your 3 test cases (and my custom test case): **50 points**
- Implement in racket/base and using only minimal imports: **10 points**
- Solve the problem in a purely **functional style** (no mutation, no state) **15 points**

Refer to the wires and the funstacker tutorial for inspiration on how to solve this problem functionally (and normally too). You can get partial credit for doing some of the stuff towards solving each of the above.

Name your language **haunt-<your initials>**.
I should be able to install it as a package and load
it using **#lang haunt-<your initials>**

Tradeoffs between 2 ideas

You can choose either one, both have pros and cons

basic	haunt
Most of the code you need to write for reader, parser, tokenizer, lexer and expander is from the tutorials.	You need to start from scratch, but you can refer to the wires/funstacker tutorial for guidance.
Multiple files, lots of lines of code	You can definitely solve the whole thing with <50 lines in reader+expander.
You'll need to make changes in parser/tokenizer/lexer, but you can use brag for heavy lifting	There is no hierarchical structure in input, so you don't need to worry about parsing — but you need to handle input reading yourself.
More involved test case writing and migrating to racket/base will be harder.	Test cases can just be copy pasted, migrating to racket/base easier.

Tips

- First write everything using `#lang br` and `#lang br/quicklang` — you can migrate to racket/base bit by bit after you've gotten all your test files working/evaluated correctly.
- For the advent of code option, **be careful how you read in the input and format it in the reader phase**. You can use the code from the wires tutorial to start, but the input file has `=`, parentheses and commas — these have meaning in Racket and will be
- For the AoC option, **start with a solution that uses mutation and state**, then move to a functional solution.
- Getting 75 pts on either option is easy, so work on getting the first 75 points quickly.

Submit your whole package
(reader+expander+helper files+tests) as a zip file
on Canvas by **midnight December 15**

I will ask you all to schedule a 10 minute zoom/in-person meeting with me during finals week (Dec 16 onwards). This is informal since I need to have something during finals week.