CPSC 311:
Definition of Programming Languages
2015 Winter Term 1

Joshua Dunfield

University of British Columbia

2015–09–16: Lecture 4

www.ugrad.cs.ubc.ca/~cs311

Logistics

- **Start on the assignment!**
  We can tell how many people have run `handin`.

Logistics

- **Start on the assignment!**
  We can tell how many people have run `handin`.
- The assignment is due 1 day **after** the drop deadline.
  You won't know your mark—but you will know whether you passed (most of) our test cases.

Logistics

- **Start on the assignment!**
  We can tell how many people have run `handin`.
- The assignment is due 1 day **after** the drop deadline.
  You won't know your mark—but you will know whether you passed (most of) our test cases.
- If you aren't (almost) done with the assignment by the drop deadline, you should probably drop the course.

Logistics

- **Start on the assignment!**
  We can tell how many people have run `handin`.
- The assignment is due 1 day **after** the drop deadline.
  You won't know your mark—but you will know whether you passed (most of) our test cases.
- If you aren't (almost) done with the assignment by the drop deadline, you should probably drop the course.
- Run `handin` early and often!
  (Even if you haven't done a single problem yet!)

Today:

- Getting into dynamic semantics: substitution.

Definition of Programming Languages

A programming language is a
precise, **symbolic** method of describing computations.

Three sides of PLs

- 1. **Syntax** describes **which sequences of symbols are reasonable**.
- 2. **Dynamic semantics** describes **how to run programs**.
- 3. **Static semantics** describes **what programs are**.

Three sides of PLs

- 1. **Syntax** describes **which sequences of symbols are reasonable**.
- 2. **Dynamic semantics** describes **how to run programs**.
- 3. **Static semantics** describes **what programs are**.

1. Syntax

- Syntax is (usually) the easiest to define, understand, and process.

1. Syntax

- Syntax is (usually) the easiest to define, understand, and process.
- Racket makes it even easier than usual!

  (By accident: the inventors of Lisp designed a more complex syntax, but the simple syntax had already spread. For once, simplicity won.)

1. Syntax

- Syntax is (usually) the easiest to define, understand, and process.
- Racket makes it even easier than usual!

  (By accident: the inventors of Lisp designed a more complex syntax, but the simple syntax had already spread. For once, simplicity won.)
- We won't spend much time on syntax.

2. Dynamic semantics

Dynamic semantics is about **how** programs behave:

- Dynamic semantics tells you how to “step” a program.

2. Dynamic semantics

Dynamic semantics is about **how** programs behave:

- Dynamic semantics tells you how to “step” a program.
- Or how to “evaluate” a program.

2. Dynamic semantics

Dynamic semantics is about **how** programs behave:

- Dynamic semantics tells you how to “step” a program.
- Or how to “evaluate” a program.
- These methods work a little differently, but they have the same purpose: they tell you what your interpreter is supposed to do.

Defining dynamic semantics

- "Interpreter semantics":
  "to explain a language, write an interpreter for it."
  ... "When we finally have what we think is the correct representation of a language's meaning..."

Defining dynamic semantics

- "Interpreter semantics":
  "to explain a language, write an interpreter for it."
  …"When we finally have what we think is the correct representation of a language's meaning…"

- If the interpreter you write defines the language, you **cannot** know whether it's correct. (It's trivially correct, because it defines itself. "When the President does it, that means it is not illegal.")

Defining dynamic semantics

- “Interpreter semantics”:
  “to explain a language, write an interpreter for it.”
  . . . “When we finally have what we think is the correct representation of a language’s meaning. . . ”

- If the interpreter you write defines the language, you **cannot** know whether it’s correct. (It’s trivially correct, because it defines itself. “When the President does it, that means it is not illegal.”)

- You can test it on programs, but tests can only show the presence of bugs, not their absence!

Substitution

- Substitution is needed to define “laws of computation”.

Substitution

- Substitution is needed to define “laws of computation”.
  - (Language designers get to make their own laws.)

Substitution

- Substitution is needed to define “laws of computation”.
  - (Language designers get to make their own laws.)
- Interestingly, the textbook doesn’t really follow “interpreter semantics” for substitution: it defines substitution *separately* (“Definition 8”, etc.).

Substitution

- Substitution is needed to define "laws of computation".
  - (Language designers get to make their own laws.)
- Interestingly, the textbook doesn't really follow "interpreter semantics" for substitution:
  it defines substitution *separately* ("Definition 8", etc.).
- It doesn't define it particularly "rigorously"—just with words. But Def. 8 *is* separate from the Racket code for `subst`.

The shadow falls

- In fact, it's not rigorous at all!
  Look at Definitions 3 through 6...

The shadow falls

- In fact, it's not rigorous at all!
  Look at Definitions 3 through 6…

- How do you know that the scope of `x` in

```
{with {x e1} e2}
```

  is `e2`?

The shadow falls

- How do you know that the scope of `x` in

```
{with {x e1} e2}
```

  is `e2`?

The shadow falls

- How do you know that the scope of `x` in

```
{with {x e1} e2}
```

  is `e2`?

- "It's obvious, isn't it?" (professor from grad school)

The shadow falls

- How do you know that the scope of `x` in

```
{with {x e1} e2}
```

  is `e2`?

- "It's obvious, isn't it?" (professor from grad school)

- Maybe it's obvious **here**.
  But compare how scope works in Racket's `let`...
  (Racket Guide: 4.6 Local Binding)

The shadow falls

- How do you know that the scope of `x` in

  ```
  {with {x e1} e2}
  ```

  is `e2`?

- "It's obvious, isn't it?" (professor from grad school)

- Maybe it's obvious **here**.
  But compare how scope works in Racket's `let`...
  (Racket Guide: 4.6 Local Binding)

- (DrRacket's mouse-over arrows are **great**, but have the same limitations as writing test cases.)

The shadow falls

- How do you know that the scope of `x` in

```
{with {x e1} e2}
```

  is `e2`?

- "It's obvious, isn't it?" (professor from grad school)

- Maybe it's obvious **here**.
  But compare how scope works in Racket's `let`...
  (Racket Guide: 4.6 Local Binding)

- (DrRacket's mouse-over arrows are **great**, but have the same limitations as writing test cases.)

- "because of the examples on page 15"

The missing definition

- **Definition 3a**
  In `{with {x e1} e2}`,
  `x` is a binding instance and its scope is `e2`.

The missing definition

- **Definition 3a**
  In `{with {x e1} e2}`,
  `x` is a binding instance and its scope is `e2`.

- This is better, but imagine a misunderstanding or disagreement about what all these words mean.

The missing definition

- **Definition 3a**
  In `{with {x e1} e2}`,
  `x` is a binding instance and its scope is `e2`.
- This is better, but imagine a misunderstanding or disagreement about what all these words mean.
- (“language lawyering”)

Defining substitution: a more precise way

- (Aside: I'm "the precise one")

Defining substitution: a more precise way

- (Aside: I'm "the precise one")
- Substitution is a **mathematical function**

$$subst(e, x, v)$$

Defining substitution: a more precise way

- (Aside: I'm "the precise one")
- Substitution is a **mathematical function**

$$subst(e, x, v)$$

- "Intuitively", it should substitute v for x in e.
  More precisely, it should replace all **free instances** of x with v, throughout e.

Defining substitution: a more precise way

- Substitution is a **mathematical function** $subst(e, x, v)$ that replaces free instances of x in e
- Let's define substitution over the WAE language.
  I'll use the concrete syntax (EBNF) near the top of page 16.

Defining substitution: a more precise way

- Substitution is a **mathematical function** $subst(e, x, v)$ that replaces free instances of x in e

- Let's define substitution over the WAE language.
I'll use the concrete syntax (EBNF) near the top of page 16.

$$subst(n, x, v) = n \qquad \text{where } n \text{ is a number}$$

$$subst(x, x, v) = v$$
$$subst(y, x, v) = y \qquad \text{if } x \neq y$$

Defining substitution: a more precise way

- Substitution is a **mathematical function**
  $subst(e, x, v)$ that replaces free instances of x in e

- Let's define substitution over the WAE language.
  I'll use the concrete syntax (EBNF) near the top of page 16.

$$subst(n, x, v) = n \quad \text{where } n \text{ is a number}$$

$$subst(x, x, v) = v$$

$$subst(y, x, v) = y \quad \text{if } x \neq y$$

$$subst(\{+\ eL\ eR\}, x, v) = \{+\ subst(eL, x, v)\ \ subst(eR, x, v)\}$$

$$subst(\{-\ eL\ eR\}, x, v) = \{-\ subst(eL, x, v)\ \ subst(eR, x, v)\}$$

Defining substitution: a more precise way

- Substitution is a **mathematical function** $\textit{subst}(e, x, v)$ that replaces free instances of x in e

- Let's define substitution over the WAE language. I'll use the concrete syntax (EBNF) near the top of page 16.

$$\textit{subst}(n, x, v) = n \quad \text{where } n \text{ is a number}$$

$$\textit{subst}(x, x, v) = v$$

$$\textit{subst}(y, x, v) = y \quad \text{if } x \neq y$$

$$\textit{subst}(\{+\ eL\ eR\}, x, v) = \{+\ \textit{subst}(eL, x, v)\ \ \textit{subst}(eR, x, v)\}$$

$$\textit{subst}(\{-\ eL\ eR\}, x, v) = \{-\ \textit{subst}(eL, x, v)\ \ \textit{subst}(eR, x, v)\}$$

$$\textit{subst}(\{\texttt{with}\ \{x\ e\}\ eB\}, x, v) = \{\texttt{with}\ \{x\ \textit{subst}(e, x, v)\}\ \ eB\}$$

Defining substitution: a more precise way

- Substitution is a **mathematical function** $\mathit{subst}(e, x, v)$ that replaces free instances of x in e
- Let's define substitution over the WAE language. I'll use the concrete syntax (EBNF) near the top of page 16.

$$\mathit{subst}(n, x, v) = n \qquad \text{where } n \text{ is a number}$$

$$\mathit{subst}(x, x, v) = v$$
$$\mathit{subst}(y, x, v) = y \qquad \text{if } x \neq y$$

$$\mathit{subst}(\{+\ eL\ eR\}, x, v) = \{+\ \mathit{subst}(eL, x, v)\ \ \mathit{subst}(eR, x, v)\}$$
$$\mathit{subst}(\{-\ eL\ eR\}, x, v) = \{-\ \mathit{subst}(eL, x, v)\ \ \mathit{subst}(eR, x, v)\}$$

$$\mathit{subst}(\{\texttt{with}\ \{x\ e\}\ eB\}, x, v) = \{\texttt{with}\ \{x\ \mathit{subst}(e, x, v)\}\ \ eB\}$$

Defining substitution: a more precise way

- Substitution is a **mathematical function** $\mathit{subst}(e, x, v)$ that replaces free instances of x in e
- Let's define substitution over the WAE language. I'll use the concrete syntax (EBNF) near the top of page 16.

$$\mathit{subst}(n, x, v) = n \qquad \text{where } n \text{ is a number}$$

$$\mathit{subst}(x, x, v) = v$$
$$\mathit{subst}(y, x, v) = y \qquad \text{if } x \neq y$$

$$\mathit{subst}(\{+\ eL\ eR\}, x, v) = \{+\ \mathit{subst}(eL, x, v)\ \ \mathit{subst}(eR, x, v)\}$$
$$\mathit{subst}(\{-\ eL\ eR\}, x, v) = \{-\ \mathit{subst}(eL, x, v)\ \ \mathit{subst}(eR, x, v)\}$$

$$\mathit{subst}(\{\texttt{with}\ \{x\ e\}\ eB\}, x, v) = \{\texttt{with}\ \{x\ \mathit{subst}(e, x, v)\}\ \ eB\}$$
$$\mathit{subst}(\{\texttt{with}\ \{y\ e\}\ eB\}, x, v) = \{\texttt{with}\ \{y\ \mathit{subst}(e, x, v)\}\ \mathit{subst}(eB, x, v)\} \qquad \text{if } x \neq y$$

Defining substitution: a more precise way

- Substitution is a **mathematical function** $\mathit{subst}(e, x, v)$ that replaces free instances of x in e
- Let's define substitution over the WAE language. I'll use the concrete syntax (EBNF) near the top of page 16.

$$\mathit{subst}(n, x, v) = n \qquad \text{where } n \text{ is a number}$$

$$\mathit{subst}(x, x, v) = v$$

$$\mathit{subst}(y, x, v) = y \qquad \text{if } x \neq y$$

$$\mathit{subst}(\{+\ eL\ eR\}, x, v) = \{+\ \mathit{subst}(eL, x, v)\ \ \mathit{subst}(eR, x, v)\}$$

$$\mathit{subst}(\{-\ eL\ eR\}, x, v) = \{-\ \mathit{subst}(eL, x, v)\ \ \mathit{subst}(eR, x, v)\}$$

$$\mathit{subst}(\{\mathtt{with}\ \{x\ e\}\ eB\}, x, v) = \{\mathtt{with}\ \{x\ \mathit{subst}(e, x, v)\}\ \ eB\}$$

$$\mathit{subst}(\{\mathtt{with}\ \{y\ e\}\ eB\}, x, v) = \{\mathtt{with}\ \{y\ \mathit{subst}(e, x, v)\}\ \mathit{subst}(eB, x, v)\} \qquad \text{if } x \neq y$$

More precise or more clear?

- Is a bunch of math I wrote yesterday more precise than the textbook's Racket code?

More precise or more clear?

- Is a bunch of math I wrote yesterday more precise than the textbook's Racket code?
- Maybe not, but it's "shallower":
  it doesn't depend on how Racket is defined.
  To explain what the Racket code does, you need to define Racket scope, which means…

More precise or more clear?

- Is a bunch of math I wrote yesterday more precise than the textbook's Racket code?
- Maybe not, but it's "shallower":
  it doesn't depend on how Racket is defined.
  To explain what the Racket code does, you need to define Racket scope, which means. . .

raincrystal on flickr; CC BY-SA

Specification vs. implementation

- The mathematical definition of *subst* doesn't implement substitution: you can't run it.

Specification vs. implementation

- The mathematical definition of *subst* doesn't implement substitution: you can't run it.
- Instead, it **specifies** how an implementation should behave.

Specification vs. implementation

- The mathematical definition of *subst* doesn't implement substitution: you can't run it.
- Instead, it **specifies** how an implementation should behave.
- This lets us distinguish
  "bug in implementation" vs. "bug in specification".

Specification vs. implementation

- The mathematical definition of *subst* doesn’t implement substitution: you can’t run it.
- Instead, it **specifies** how an implementation should behave.
- This lets us distinguish
  “bug in implementation” vs. “bug in specification”.
- This depends on **the specification being more readable than the implementation**.

Specification vs. implementation

- The mathematical definition of *subst* doesn't implement substitution: you can't run it.
- Instead, it **specifies** how an implementation should behave.
- This lets us distinguish
  "bug in implementation" vs. "bug in specification".
- This depends on **the specification being more readable than the implementation**.

Caveat: When specifications are bad, they are **even less useful** than badly written code. You can maybe run bad code. You can't do **anything** with a bad specification.

When language specifications go bad

c) **WHETHER QUALITY1 TAX resides in QUALITY2 TAX**{a,b,48d} :
where (QUALITY1) is (label) or (QUALITY1) is (DYADIC) or (QUALITY1) is (MODE field), WHETHER (QUALITY1) is (QUALITY2) ;
where (QUALITY1) is (MOID1 TALLETY) and (QUALITY2) is (MOID2 TALLETY), WHETHER MOID1 equivalent MOID2{73a}.

{A nest, except the primal one (which is just '**new**'), is some '**NEST LAYER**' (i.e., some '**NEST new PROPSETY**'). A '**PROP**' is identified by first looking for it in that '**LAYER**' (rule a). If the '**PROP**' is some '**label TAX**' or '**DYADIC TAX**', then a simple match of the '**PROP**'s is a sufficient test (rule c). If the '**PROP**' is some '**MOID TALLETY TAX**', then the mode equivalencing mechanism must be invoked (rule c). If it is not found in the '**LAYER**', then the search continues with the '**NEST**' (without that '**LAYER**'), provided that it is independent of all '**PROP**'s in that '**LAYER**'; otherwise the search is abandoned (rule a). Note that rules b and c do double duty in that they are also used to check the validity of **applied-field-selectors** (4.8.1.d).}

When language specifications are good

- Written in a common* language (not Racket, not PL/I, not Algol): math/logic.
- Papers on type systems from 2015 look a lot like those from 1995.

* to programming language researchers

When language specifications are good

- Written in a common* language (not Racket, not PL/I, not Algol): math/logic.
- Papers on type systems from 2015 look a lot like those from 1995.
- In 311, you'll learn how to **read** and **implement** some of these (less bad) specifications, but we'll skip most of the mathematical foundations—for that stuff, take CPSC 509 from Ron Garcia!

* to programming language researchers

Defining substitution: a more precise way

How would we turn our definition of *subst* into Racket code?

Defining substitution: a more precise way

How would we turn our definition of *subst* into Racket code?

- Fortunately, Racket is a functional language.
  Even better, we have `type-case`!

For next time. . .

- **No assigned reading.**
- But. . . join the club:

For next time. . .

- **No assigned reading.**
- But. . . join the club:

Run `handin` today!