lec1: Course overview; syntax

Joshua Dunfield

January 6, 2019

This first lecture discusses course goals and logistics, gives some flavour of what this course is about, and introduces a few essential technical ideas.

Website: dunfieldlab.ca/865

Email: joshuad@cs.queensu.ca

Me, previously: UBC, MPI-SWS, McGill, CMU

1 Course goals

Let's start with some questions of practical importance:

- How do we know that a given program is correct?
- How do we know that two programs behave the same way?
- How do we know that a program doesn't leak private data?
- How do we know that a program doesn't crash?

To know something, we need to prove it, so we can amend these questions: "How do we *prove* that a given program is correct?"

These questions suggest more general questions.

- How do we design languages to make it easier to prove that programs are correct, or behave the same way, or don't leak private data?
- How do we prove that *all programs* in a particular language don't[1] crash?

To answer these questions, we need to know what programs *mean*—and what programming *languages* mean. That is, we need to know their semantics.

You will learn how to

- read and write formal definitions of what programming languages mean;
- relate those formal definitions to informally defined language features;
- apply formal definitions (e.g. applying inference rules to construct derivations);
- reason about derivations, including proving that languages satisfy specific properties, such as type soundness.

[1] Subject to reasonable assumptions about several other parts of a computer system, including the CPU's speculative execution. . .

You will also learn how to

- read and write formal definitions of logics;
- relate those formal definitions to informal logical features, such as proof by contradiction;
- relate general aspects of logic to aspects of programming (formulas ~ types, proofs ~ programs, truth ~ inhabitation);
- relate specific features of logics to features of programming languages: conjunction ~ pairing (a record or struct with two fields), existential formulas ~ abstract data types/modules (there *exists* an implementation with an interface, but you can't look at the implementations), negation ~ first-class continuations).

As a consequence of the above skills, you will be able to understand substantial parts of current research papers in the field of programming languages.

(You will *not* learn how to write a compiler—that's CISC 858, being taught this term by James Cordy—though much of what you'll learn in this course is relevant.)

2 Logistics

2.1 Prerequisites

There are no specific formal prerequisites. However, some background in *at least some* of the following topics is helpful:

- basic set theory (e.g. set notation, subset, power set)
- mathematical proof techniques, particularly proof by induction
- functional programming languages
- logic programming (e.g. Prolog)

If you have not been exposed to these topics, expect to put extra time into catching up. I am happy to discuss any concerns and advise you as to whether this course is a good fit.

2.2 Assessment

Your course grade will be determined by a combination of assignments, an in-class exam, and a course project.

- 45% Assignments (approximately 6): some or all will be written assignments, some may involve programming
- 20% Exam (in class)
- 35% Project:
  - 5% Proposal

- 20% Report
- 10% Presentation

I do not plan to use onQ. Instead, we will use Piazza—you should "enroll" via the link on the web page. If you decide you don't want to take this course, you can easily "un-enroll" from Piazza. Announcements will be posted *only* to Piazza.

2.3 Lectures

I will use some combination of traditional (but participatory) lecturing and (brief) in-class exercises.

I will try to take a 5- to 10-minute break midway through each lecture. If I forget, remind me!

If you don't understand something, or you aren't sure if you understand it, ask! It is *very* likely that other students want to ask the same question.

I sometimes have trouble understanding spoken English, especially in an unfamiliar accent. You may need to repeat or reword a question for me to understand.

I have trouble remembering names, but I find it much easier to remember a name if I know how it's spelled—even if it's a "difficult" name—so don't be surprised if I ask you to spell your name (maybe repeatedly).

If you miss a lecture, read my posted notes, get other students' notes, and/or come to my office hours. (If you are ill with something that seems contagious, *please* miss the lecture!)

Even if you attend every lecture, **you need to read the lecture notes!** My lecture notes are usually more complete than the lectures.

3 Semantics of Programming Languages

Semantics: "the branch of linguistics and logic concerned with meaning" (Oxford Dictionary of English). More specifically, the *semantics* of a thing is the *meaning* of the thing.

The semantics of programming languages includes the semantics of programs. So, this course is about what programming languages mean, and what programs mean. How can we say what programs mean?

3.1 Denotational semantics

In *denotational semantics* (a questionable name, since *denotation* means *meaning*, so denotational semantics is "meaning-based meaning"...), we give a *denotation* to each program. For example, if a program P always returns the number zero, we might write

The hollow square brackets are called *semantic brackets*.

$$[\![\mathsf{P}]\!] = \{0\}$$

to say that "the denotation of P is the set containing only zero". Or, if a program R sometimes returns zero and sometimes returns one, we might write

$$[\![\mathsf{R}]\!] = \{0, 1\}$$

to say that "the denotation of R is the set containing zero and one". Actual denotational semantics are more interesting: for example, we may care about a program's effect on memory and not just what it "returns", in which case the denotation of a program might be a (mathematical) function. Instead of the meaning of P being a set of integers (a subset of the power set of the integers)

$$[\![P]\!] \subseteq \mathbb{Z} \qquad \text{or} \qquad [\![P]\!] : \mathcal{P}(\mathbb{Z})$$

the meaning of P might be a function

$$[\![P]\!] : \mathit{Mem} \to \mathcal{P}(\mathit{Mem} \times \mathbb{Z})$$

where *Mem* is a set of descriptions of memory (such as a sequence of word values). If P never changes memory and always returns zero, then

$$[\![P]\!] \ = \ f$$

where $f(s) = \langle s, 0 \rangle$. (We could say the same thing by writing $[\![P]\!](s) = \langle s, 0 \rangle$.)

3.2 Operational semantics

Denotational semantics, at least in the style hinted at above, is a little like a novel that only has its last page: it tells you where you ended up, but not what happened along the way. Operational semantics gives you the whole story. In operational semantics, the meaning of a program is how it affects the world at each "time step". The question then is: what counts as the world?

In the simplest form of operational semantics, the program *is* the world, so operational semantics is a *program transforming itself*. In this form of operational semantics, the program P could in fact *be* the number zero, which does not do anything. On the other hand, the program R might be a program that flips a coin, so that we have two possible transformations, called *steps*:

$$R \mapsto 0 \qquad \text{"R steps to zero"}$$
$$R \mapsto 1 \qquad \text{"R steps to one"}$$

More generally, in this form of operational semantics, the meaning of a program e is a series of expressions:

$$e \mapsto e_1 \mapsto e_2 \mapsto \cdots \mapsto v$$

where v is a *value*, an expression of a particular kind, perhaps the integers. If e is already a value, as in our program P, then $e = v$ and we have a "sequence" with no steps. In the λ-calculus, the "stepping function" $\mapsto$ is given by something called β-reduction, which models a single function call. Each step then corresponds to a single function call.

This may remind you of Turing machines, where (given a fixed "program", that is, a transition function) we can model a configuration as a tuple $\langle q, t, p \rangle$ where q is the current "state", t describes the symbols written on the tape, and p is the position of the head on the tape. (Many alternatives exist, such as $\langle q, t_L, t_R \rangle$ where t_L describes the tape to the left of the head, and t_R describes the tape underneath and to the right of the head.) When a Turing machine is in a halting state, the program is finished; when the current expression is a value, the program is finished.

In fact, the λ-calculus corresponds to a universal Turing machine, and just as the number of transitions a Turing machine takes to is a model of running time (not a particularly realistic model, since Turing machines spend a lot of time "commuting" from one part of the tape to another), the number of steps needed to turn an expression into a value gives us a model of running time. In that model, the program P takes no time at all, and the program R takes a little time but not very much.

3.3 Dynamic semantics

Both denotational and operational semantics are concerned with the *dynamic semantics* of programs: what programs *do*. A programming languages researcher might say "for the dynamic semantics, I prefer operational approaches" or "for the dynamics, I prefer the operational style."

Different approaches can be used for the same language. Each style has advantages and disadvantages, some of which will become evident later in the course. The kind of reasoning I need to do in my research usually seems easier to do in the operational style. (As usual in research, this assessment of which style is easier should be treated with skepticism. A researcher's methods and techniques are self-perpetuating: the more you work with a technique T, the more effective it seems to be—because you're better at using it. You also become more adept at noticing opportunities to apply technique T.)

3.4 Static semantics

In addition to "the dynamics", we have "the statics". *Static* semantics are about *what programs are*. The most successful approach to static semantics is based on *type systems*.

4 Syntax

Of the three parts of a language definition—syntax, dynamic semantics, and static semantics—syntax is often the easiest to define, understand, and process. (Languages descended from Lisp, like Racket, make it even easier than usual. This was an 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.

4.1 Grammars

A *grammar* is a set of *nonterminals* (usually called *meta-variables* in this course), each with a list of their possible syntactic forms.

You may have seen grammars in formal languages and automata theory. Our treatment of them is, intentionally, concrete and example-driven. (Later in the course, there will be plenty of opportunities to be excessively abstract!)

Each nonterminal *generates a language*. A language, in this sense, is a set of strings: all the strings that match one of the listed syntactic forms.

For example, we can define a nonterminal B that has exactly two syntactic forms, 0 and 1. Read this definition as "B can have two forms: 0 and 1"; equivalently, "a B is either 0 or 1".

$$\begin{array}{rcl} B & ::= & 0 \\ & & \mid 1 \end{array}$$

This is a "BNF grammar" (identifiable by the "::="); it is equivalent to the more traditional notation with two *productions*:

$$\begin{array}{rcl} B & \rightarrow & 0 \\ B & \rightarrow & 1 \end{array}$$

In a BNF grammar, productions are often called *alternatives*, and can be written on separate lines (as above) or squashed together:

$$\mathsf{B} \ ::= \ 0 \mid 1$$

Such a small grammar is not so interesting; it seems to be only a clumsier way of writing the set $\{0, 1\}$. The power of grammars comes from using nonterminals within alternatives. For example, the following grammar describes *sequences* of one or more bits:

$$\begin{array}{rcl} \mathsf{B} & ::= & 0 \mid 1 \\ \mathsf{S} & ::= & \mathsf{B} \\ & & \mid \mathsf{B}\ \mathsf{S} \end{array} \qquad\qquad \begin{array}{rcl} \mathsf{B} & \rightarrow & 0 \\ \mathsf{B} & \rightarrow & 1 \\ \mathsf{S} & \rightarrow & \mathsf{B} \\ \mathsf{S} & \rightarrow & \mathsf{B}\ \mathsf{S} \end{array}$$

(We put a space between B and S to avoid confusion with a potential nonterminal called BS. You may not have seen nonterminals that are more than one letter long, but they occasionally pop up in programming languages.)

We can use this grammar to *produce* the string 0 1 1, as follows:

$$\begin{array}{rcl} \mathsf{S} & \rightarrow & \mathsf{B}\ \mathsf{S} \\ & \rightarrow & 0\ \mathsf{S} \\ & \rightarrow & 0\ \mathsf{B}\ \mathsf{S} \\ & \rightarrow & 0\ 1\ \mathsf{S} \\ & \rightarrow & 0\ 1\ \mathsf{B} \\ & \rightarrow & 0\ 1\ 1 \end{array}$$

Exercise 1. Design a grammar for sequences of bits *without leading zeroes*: for example, 0 and 1 are included, but not 00 and 01.

Now we can add to our grammar above, to define *arithmetic expressions* E over bitstrings:

$$\begin{array}{rcl} \mathsf{E} & ::= & \mathsf{S} \\ & & \mid \mathsf{E} + \mathsf{E} \\ & & \mid \mathsf{E} - \mathsf{E} \end{array}$$

[Do some examples. Note that in a production, the first E doesn't need to be the same as the second E.]

This grammar is *ambiguous*: the nonterminal S can produce the same string of symbols in *more than one way*.

We can resolve this ambiguity in several different ways: for example, we could change the alternatives E + E and E - E to S + E and S - E. Now we have only one way of producing 0 + 1 + 1. While we may not care whether 0 + 1 + 1 is interpreted as $(0 + 1) + 1$ or $0 + (1 + 1)$, it is possible that in the *semantics* of this tiny language, $+$ will not be associative and commutative. In any case, we probably do care whether 0 - 1 - 1 is interpreted as $(0 - 1) - 1$ or as $0 - (1 - 1)$...

Continuing in this vein leads to some headaches: multiplication usually has higher precedence than addition, so we would want to make sure our grammar interprets $0 * 1 + 1$ as $(0 * 1) + 1$—presumably 2 (binary 10)—and not as $0 * (1 + 1)$—presumably 0. We could do that by making a new nonterminal F. However, since the focus of this course is on semantics and not syntax, I plan to take the easier route. If we use prefix notation (like "+ 1 0") instead of infix notation, and require parentheses around all arithmetic operations, we avoid ambiguities. (Lisp fans will argue that the resulting notation is actually superior.)

$$
\begin{aligned}
\mathsf{E} \;::=\;& \mathsf{S} \\
\mid\;& \mathsf{(+\ E\ E)} \\
\mid\;& \mathsf{(-\ E\ E)}
\end{aligned}
$$

[Discuss parse trees.]

Exercise 2. Similar to how we "produced" the string 0 1 1, use the above grammar for E and S to produce the string

```
(+ 10 1)
```

Producing `10` and `1` is tedious, so we'll make a distinction between "smaller" parts of syntax, like S, and "larger" parts of syntax, like E. (Many real programming languages make this distinction, too.) The small parts will be called *tokens* (sometimes *terminal symbols*). Instead of thinking in terms of the individual characters in "`10`", we'll consider `10` to be a single token.

Exercise 3. Produce the string

```
(+ 10 1)
```

without breaking down individual tokens.

Exercise 4. Draw the parse tree for

```
(+ (- (+ 10 1) 11) 0)
```

without breaking down individual tokens. (I don't think there's a standard convention for how to draw the broken-down parse trees anyway!)