CPSC 311: Definition of Programming Languages: Recap; strings ("14-review")

DRAFT

Joshua Dunfield
University of British Columbia

October 19, 2015

Logistics

- The class that meets after us in this room will have a midterm on Wednesday.
  I will try to finish around 9:40, so we can clear the room for them.
  That's **Wednesday**, not today. Try to remind me on Wednesday.
- There was a wonderful bug in a3.rkt, which I fixed; see Piazza / check your email. I suggest fixing your copy of a3.rkt, but if you exploit the bug to do Problem 2, you could get full marks. You'll probably learn more if you don't do that, though.
- Sometimes, you need to stop the bus. (Possibly-overblown bus metaphor goes here.)

  (Staffer on *The West Wing*: That's really going to cramp our style!
  Chief of staff: Your style could use a little cramping.)
- Midterm update:
  - The "cutoff" for what could be on the midterm is essentially *right now*. However, "right now" includes assignment 3.
  - Several midterm questions have been written, and I expect to post a sample midterm Friday (hopefully earlier).
  - (Digression: Puzzles are bad, and should feel bad. Maybe an election *is* a time to discuss serious issues, but an exam is no time for flashes of inspiration.)

1 Review

Defining programming languages:

- Defining syntax: BNF
- Defining semantics: rules

1.1 BNFs

Here's a BNF:

Characters	⟨ch⟩	::=	`a` | `b` | `c` | ···
Strings	⟨S⟩	::=	`"`⟨ch⟩`..."` ⟨ch⟩... means zero or more repetitions of ⟨ch⟩
Cats	⟨C⟩	::=	⟨S⟩
			| `{+` ⟨C⟩ ⟨C⟩`}`

Read "::=" as "can have the form". Other readings you may come across are "expands to" or "rewrites to", which are reasonable in some contexts, but I feel they're misleading in the context of programming languages. We are given an input string (a program) and want to parse it; if there is "rewriting" happening (and I'm not sure there is), it should be going from right to left: the parser sees `b` and realizes it is a character ⟨ch⟩.

Symbols on the **left** of the "::=", like ⟨ch⟩, ⟨S⟩, ⟨C⟩, are called *nonterminals*. On the right hand side, alternatives separated by "|" are called *productions*.

In a BNF, when a nonterminal appears twice, it can (and usually does) represent a different string. For example,

`{+ "ab" "cd"}`

is a ⟨C⟩ because `"ab"` and `"cd"` are each ⟨C⟩s (because they are each an ⟨S⟩ (because ...)).

By itself, a BNF only tells you what input strings (programs) are syntactically valid. You might be able to *guess* that I want to define a simple language of string concatenation, where you can "run" `{+ "ab" "cd"}` and get `"abcd"`, but the BNF doesn't say that.

Remark. Unlike "formal semantics" (rules), "formal syntax" (BNF) is used for most "real" programming languages, so it's important to understand it. You also have to be prepared for variations in notation (which is why I try to be careful to always tell you what "..." means).

1.2 Abstract syntax

Lisp was supposed to have a "real" syntax, which was never finished. But this piece of vaporware led to something useful: abstract syntax.

Unlike most "real" syntaxes, abstract syntax is not ambiguous; it doesn't need to resolve the ambiguity of `a + b * c`, because everything is in brackets/parentheses/braces.

The **define-type** feature of Racket/PLAI is ideally suited to defining abstract syntax: everything is in parentheses, because everything in Racket is in parentheses.

```
(define-type Cat
   [c/string (s string?)]
   [c/concat (left Cat?) (right Cat?)]
)
```

In the concrete syntax BNF, I could just write ⟨S⟩ by itself as a production of ⟨C⟩, In abstract syntax, each alternative has to begin with a variant name (like `c/string`).

Here, I have written `c/concat` instead of `+`, but this is still only syntax. Using the name `c/concat` strongly suggests that this is meant to be string concatenation, but so far, that's only a name.

1.3 Rules

An *inference rule*, or *rule* for short, looks like

$$\frac{\text{premise}_1 \quad \ldots \quad \text{premise}_m}{\text{conclusion}}\ \text{rule name}$$

There might be no premises. We've seen that with rules like

$$\frac{}{(\mathsf{num}\ n) \Downarrow (\mathsf{num}\ n)}\ \text{Eval-num}$$

The conclusion is always some *judgment*, like $e \Downarrow v$ or $\Gamma \vdash e : A$. The conclusion and premises usually have *meta-variables*, which we can replace with instances.

To apply a rule, you replace all its meta-variables. Eval-num has one meta-variable, n. If I instantiate it with 4, I get a *derivation* of $(\mathsf{num}\ 4) \Downarrow (\mathsf{num}\ 4)$.

$$\frac{}{(\mathsf{num}\ 4) \Downarrow (\mathsf{num}\ 4)}\ \text{Eval-num}$$

In a rule, unlike a BNF, repeated occurrences of the same meta-variable refer to the same thing. So you can't replace the first n with 4, and the second n with 5. (This is a confusing difference, but it's now too standard to change.)

s ::= Racket string

Cat c ::= (c/string s)
       | (c/concat c c)

v ::= ■ (c/string s)

$c \Downarrow v$ Cat c evaluates to v

$$\rightarrow \frac{}{(c/\text{string } s) \Downarrow (c/\text{string } s)} \qquad \frac{}{v \Downarrow v}$$

$$\rightarrow \frac{c_1 \Downarrow (c/\text{string } s_1) \quad c_2 \Downarrow (c/\text{string } s_2) \quad s = s_1 s_2 \; [s_1 \neq s_2]}{(c/\text{concat } c_1\ c_2) \Downarrow (c/\text{string } \underset{s_1 s_2}{s})}$$

(concatenation)

~~c c~~

$$\frac{c_1 \Downarrow c \quad c = (c/\text{string } s_1)}{(c/\text{concat } c_1\ c_2) \Downarrow (c/\text{string } \quad)}$$

$$\frac{\dfrac{(c/\text{string "a"}) \Downarrow (c/\text{string "a"}) \quad (c/\text{string "b"}) \Downarrow (c/\text{string "b"})}{(c/\text{concat } (c/\text{string "a"})\ (c/\text{string "b"})) \Downarrow (c/\text{string "ab"})} \quad (c/\text{string "d"}) \Downarrow (c/\text{string "d"})}{(c/\text{concat } \underbrace{(c/\text{concat } (c/\text{string "a"})\ (c/\text{string "b"}))}_{c_1}\ \underbrace{(c/\text{string "d"})}_{c_2}) \Downarrow (c/\text{string "abd"})}$$