CPSC 311: Syntax **(DRAFT)**
(“lec-syntax”)

Joshua Dunfield
University of British Columbia

September 8, 2016

1 Bird’s-eye view

As discussed in the introduction, syntax describes “which sequences of symbols are reasonable”. Given an input string, the first thing done by interpreters and compilers is to *parse* the string into an *abstract syntax tree* (AST).

Parsing is not the focus of this course, but we need to spend a little time on it. (If you take 411, parsing will probably be covered in somewhat more detail.)

For example, in a Java program, the string `x = y + 11;` would be parsed into something like

```
  Assignment
    /    \
 Var    Plus
  x      /  \
       Var   Num
        y    11
```

The Racket expression `(+ y 11)` could be parsed into something like

```
     Plus
     /   \
   Id     Num
   y      11
```

(This is not how DrRacket actually handles syntax, but it’s close enough for now.)

Parsing filters out strings that don’t make any sense in the language. A Java compiler, for example, will reject `(+ y 11)` with a syntax error.

The area of computer science called *formal languages* tends to focus on filtering; for example, formal languages theorists try to study which classes of automata can *recognize* strings, that is, decide whether or not a string is syntactically well-formed. In that setting, a “language” is just a set of strings. But in *programming* languages, we almost always care about the *meaning* of a particular string, so the main purpose of parsing is to get an abstract syntax tree.

It can be quite tricky to transform a string into an abstract syntax tree. Languages in the Lisp family, including Racket, make the problem easier by making the syntax unusually simple. For example, the precedence rules for Java say that + has higher precedence than =, that is, + “binds tighter”. If + had *lower* precedence than =, then the Java statement `x = y + 11;` would produce the following:

```
         Plus
       /    \
Assignment   Num
   /  \       11
Var   Var
 x     y
```

In Racket, the issue just doesn't arise: the parentheses in `(+ y 11)` are *not* optional. (The rough equivalent to the Java statement, in Racket, would be `(set! x (+ y 11))`.)

2 Phases of parsing

Parsing is usually a two-step process:

1. *Lexical analysis* (also called *lexing* or *tokenizing*) turns a string into a sequence of *tokens*.

   For example, the Java string `x = y + 11;` would become a sequence of 6 tokens:

   id(x), equals, id(y), plus, num(11), semicolon

   This is the *only* thing lexical analysis does. It will not reject the string `x y = + + 11 + ;`—it is syntactically invalid, but it is a sequence of valid Java tokens, which is the only thing this step cares about.

   Note that lexical analysis ignores comments and whitespace (spaces, tabs, and newline characters) between tokens: the Java string

   ```
   x= /* hi */ y+11   ;
   ```

   would also become the above sequence of tokens.

   In the vast majority of languages, whitespace *inside* a token is usually not allowed: the Java string

   ```
   x= /* hi */ y+1 1   ;
   ```

   is a different sequence of tokens, with two num(1) tokens rather than one num(11) token.

   But whitespace inside some tokens, like *string literals*, is allowed: in Java (and C),

   ```
   x= "ab cd ef";
   ```

   is a sequence of 4 tokens, where the 3rd token is string(`ab cd ef`).

2. The second step is called *parsing*. (Confusingly, "parsing" can mean either the lexer and parser together, or just this second step.) This step turns a sequence of tokens into an abstract syntax tree.

Techniques for lexing are well-developed. Parsing is more difficult, but again, techniques have been developed—along with tools such as yacc, bison, and ANTLR that automatically generate parsers from *grammars*.

But we will not need to go into the details of either step. As we build interpreters, we will use DrRacket's built-in lexer/parser to do most of the work. This means that the languages we interpret must have syntax that looks a lot like Racket; this may not be to your taste, but it's less work, and leaves more time for us to discuss the more interesting aspects of programming languages.

Unfortunately, there is a gap between what DrRacket gives us, and what we need. This is the gap between *concrete syntax* and *abstract syntax*. First, we need to explain grammars.

3 Grammars

A grammar specifies which strings are syntactically valid programs. Different people use different notations for grammars, but most notations are based on "Backus Normal Form", or BNF, which was first used to specify the syntax of Algol-60.

Our notation looks like this:

$$
\begin{array}{rcl}
\langle \text{digit} \rangle & ::= & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
\langle \text{integer} \rangle & ::= & \langle \text{digit} \rangle \\
 & & \mid \langle \text{digit} \rangle \langle \text{integer} \rangle
\end{array}
$$

In this grammar, ⟨digit⟩ and ⟨integer⟩ are *nonterminal symbols*. A nonterminal on the left, like ⟨digit⟩, expands to one of the alternatives on the right. The alternatives are separated by vertical bars |. So this grammar says that a digit can have the form 0, 1, 2, . . . , and that an integer is either a single digit (⟨digit⟩), *or* ("|") a digit followed by an integer (⟨digit⟩⟨integer⟩).

Alternatives are usually written on separate lines, but for something like ⟨digit⟩ it's better to put all the alternatives on a single line.

Nonterminal symbols are usually called nonterminals, and alternatives are sometimes called *productions*.

Exercise 1. Extend the above grammar with a nonterminal ⟨nlz⟩ that represents integers *without* leading zeroes, so that ⟨nlz⟩ can have the form 13 or 130 but not 013, 00130, etc. However, your grammar *should* allow ⟨nlz⟩ to have the form 0.

Hint: First, add a nonterminal that represents a single digit that is *not* zero.

The above grammar is not terribly interesting. In fact, it is a "regular" grammar, and we could have used something simpler than BNF, like regular expressions. But we can extend this grammar to something more interesting, like "arithmetic expressions" ⟨ae⟩ in prefix notation:

$$
\begin{array}{rcl}
\langle \text{digit} \rangle & ::= & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
\langle \text{integer} \rangle & ::= & \langle \text{digit} \rangle \\
 & & \mid \langle \text{digit} \rangle \langle \text{integer} \rangle \\
\langle \text{ae} \rangle & ::= & \langle \text{integer} \rangle \\
 & & \mid \texttt{\{+}\ \langle \text{ae} \rangle\ \langle \text{ae} \rangle \texttt{\}} \\
 & & \mid \texttt{\{-}\ \langle \text{ae} \rangle\ \langle \text{ae} \rangle \texttt{\}}
\end{array}
$$

Recalling data definitions from CPSC 110, an ⟨integer⟩ is shaped like a list: one thing, a digit, is followed by the thing being defined (an integer). In contrast, an ⟨ae⟩ is shaped like a tree, with integers as leaves, and two kinds of branches: + and -.

Remark. Mathematically speaking, BNF grammars are a particular kind of inductive definition: the nonterminal ⟨integer⟩ is defined by a base case (the alternative "⟨digit⟩") and an inductive case—the alternative "⟨digit⟩⟨integer⟩", which mentions the nonterminal ⟨integer⟩. Induction is a form of recursion: the definition of ⟨integer⟩ uses the definition of ⟨integer⟩. The nonterminal ⟨ae⟩ is also defined recursively, with two inductive (recursive) cases instead of one.

3.1 Grammars and abstract syntax

Following the pattern from the Java and Racket examples at the beginning of this chapter, the abstract syntax for the arithmetic expression

```
{+ 3 11}
```

could look something like

```
      Plus
      /   \
Integer   Integer
   3        11
```

I say "could", because other variations are possible. We might call the leaf nodes `Num` rather than `Integer`, or call `+`'s node `Add` rather than `Plus`. We'll encounter some more interesting variations later.

The fact that such variations are possible means there has to be some "distance" between an input string like `{+ 3 11}` and its abstract syntax. For example, the grammar we wrote didn't tell us what name to give nodes for `+`. However, there is a kind of syntax tree that *is* uniquely determined by the grammar: a concrete syntax tree.

3.2 Grammars and concrete syntax

Here is the same grammar again:

$$
\begin{aligned}
\langle\text{digit}\rangle &::= \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \\
\langle\text{integer}\rangle &::= \langle\text{digit}\rangle \\
&\quad\mid \langle\text{digit}\rangle\langle\text{integer}\rangle \\
\langle\text{ae}\rangle &::= \langle\text{integer}\rangle \\
&\quad\mid \texttt{\{+}\ \langle\text{ae}\rangle\ \langle\text{ae}\rangle\texttt{\}} \\
&\quad\mid \texttt{\{-}\ \langle\text{ae}\rangle\ \langle\text{ae}\rangle\texttt{\}}
\end{aligned}
$$

Instead of thinking about parsing a string and getting a tree, let's try to produce the string `{+ 3 11}` *from* the grammar. We can do this by replacing the left-hand sides (nonterminal symbols like ⟨ae⟩) with right-hand sides (alternatives like `{+` ⟨ae⟩ ⟨ae⟩`}`).

1. Start with the nonterminal ⟨ae⟩.

2. We are trying to produce a string that looks like `{+ ...}`, so we use the second alternative (production), `{+` ⟨ae⟩ ⟨ae⟩`}`. Now we have the string of symbols

   `{+` ⟨ae⟩ ⟨ae⟩`}`

3. We are trying to produce `{+ 3 11}`, so we replace the first occurrence of ⟨ae⟩ with the first alternative of ⟨ae⟩, which is ⟨integer⟩. Now we have the string of symbols

   `{+` ⟨integer⟩ ⟨ae⟩`}`

4. We want to produce `{+ 3 ...}`. The number `3` has one digit, so we replace ⟨integer⟩ with its first alternative, which is ⟨digit⟩. That gives us

   {+ ⟨digit⟩ ⟨ae⟩}

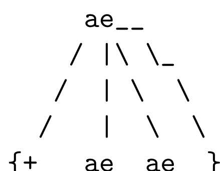
5. We specifically want `3`, so we replace ⟨digit⟩ with the alternative `3`:

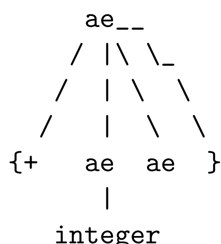
   {+ 3 ⟨ae⟩}

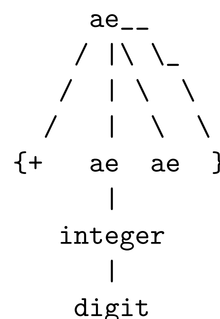
6. (Condensing some steps now.) We want the second number in the string to be `11`, so we replace ⟨ae⟩ with ⟨integer⟩, then ⟨integer⟩ with ⟨digit⟩⟨integer⟩, then ⟨digit⟩ with `1`, then ⟨integer⟩ with `digit`, and finally `digit` with `1`:

   {+ 3 11}

Here's where the "tree" part of "concrete syntax tree" comes in: instead of only writing out the string, we can build a tree as we go. Instead of replacing the nonterminal with the symbols on a right-hand side, we add those symbols to the nonterminal, as its children:







3.3 Tokens and nonterminals

In the grammar for ⟨ae⟩ above, I specified everything using the grammar—even what an integer looks like. Often, language specifications will define smaller pieces of syntax separately from bigger pieces of syntax. In that case, we would define an integer separately from an arithmetic expression, and each integer would be a single token; the main grammar would treat each token as an atom. It would look the same as the previous grammar for ⟨ae⟩, with the first definitions removed:

```
⟨ae⟩ ::= ⟨integer⟩
       | {+ ⟨ae⟩ ⟨ae⟩}
       | {- ⟨ae⟩ ⟨ae⟩}
```

This style of grammar is compatible with reusing DrRacket's lexical analyzer; we'll let it handle the small pieces of syntax, like numbers. We will also use the part of DrRacket's parser that handles parentheses.

Our parser will only have to convert an *S-expression* into an abstract syntax tree. That raises new questions:

- What is an S-expression?
- How do we represent an abstract syntax tree?

4 S-expressions

An interesting feature of Racket (inherited from Scheme, which inherited it from Lisp) is that Racket code is also a data structure that can be directly manipulated in Racket. That data structure is called an S-expression; an S-expression is a tree that represents a Racket expression. Saying that an S-expression "represents" a Racket expression is a little questionable; it's probably more accurate to say that an S-expression *is* a Racket expression.

When we type something like

```
(+ 2 2)
```

into DrRacket, two things happen.

The first thing is that the sequence of characters

"(", "+", " ", "2", " ", "2", ")"

is parsed into an S-expression that looks like this:

```
    cons
   /    \
  +      cons
        /    \
       2      cons
             /    \
            2      empty
```

S-expressions with this shape—a right-leaning tree of "cons cells", ending with `empty`—are called lists.

The second thing that happens is that the S-expression is *evaluated*. DrRacket evaluates this particular S-expression to `4`.

But you can stop DrRacket from doing the second thing: you can *quote* the expression `(+ 2 2)`, by writing an apostrophe `'` before it:

```
> '(+ 2 2)
'(+ 2 2)
```

DrRacket is still doing the first step of converting the sequence of characters into an S-expression. You can't really tell that it's doing this, because it just converts the S-expression back into a string of characters, but it is happening.

```
> '(+ 2 2)
'(+ 2 2)
```

Racket has a built-in function called `eval` that does the evaluation step, even if you've quoted an expression:

```
> (eval '(+ 2 2))
4
```

(Aside: what happens if you type the same thing *without* the quote?)