15-312 Foundations of Programming Languages
Recitation 1: Concrete Syntax

Daniel Spoonhower
spoons+@cs

August 27, 2003

1 Defining a Grammar

We can imagine a spectrum of different representations of a concept or an idea, from the most concrete (e.g. written on a blackboard) to the more ethereal (e.g. your understanding). We'll begin with a description of one of the more concrete: the *syntax* of our language. We will be looking for two characteristics of this representation: its success in conveying the more abstract understanding that underlies it (i.e. readability) and to what extent it specifies a program without ambiguity.

EXAMPLE The number three is an abstract concept. The following are five different concrete representations of the number three.

$$\cdots \qquad \text{III} \qquad 3 \qquad 1+1+1 \qquad 011$$

(Which representation is best?)

We will use a *context-free grammar* to define the syntax of our language. Using a grammar, we will combine strings of symbols, letters, words, tokens, or atoms to form sentences, phrases, statements, and programs. In addition, we will see that syntax lets us make simple syntactic distinctions, for example as between digits (i.e. 0, 1, 2, . . . , 9) and numbers (e.g. 5119). Finally, we will see an example of an ambiguous grammar and how we might disambiguate it.

1.1 Context-Free Grammars (CFGs)

A *context-free grammar* is composed of the following three components:

- An *alphabet* Σ composed of a finite set of *terminals* (or *symbols*)
- A set of *non-terminals* $\mathcal{N}$ that form syntactic categories
- A set of *production rules* $\mathcal{P}$ of the form

  $$A \to \alpha$$

  where α is a string of terminals and non-terminals

(Why is this form *context-free*? What would the productions of a context-*sensitive* grammar look like?)

Consider the following grammar:

$$\begin{array}{llll} d \to 0 & d \to 1 & d \to 2 & d \to 3 \\ d \to 4 & d \to 5 & d \to 6 & d \to 7 \\ d \to 8 & d \to 9 & n \to d & n \to d\ n \end{array}$$

(What are the terminals? The non-terminals?)

Strings are *derived* from the grammar by the *application* of production rules. For each non-terminal A that occurs in the current string β, and given a rule $A \to \alpha$, we substitute the right-hand side of the rule α for each instance of A in β.

Example Using the rule $n \to d\ n$, we can show the following.

$$d\ \underline{n} \Rightarrow d\ \underline{d\ n}$$

How do we show that the numeral "3" can be derived beginning with the non-terminal n? What about "5119"?

$$\begin{array}{l} n \Rightarrow d \Rightarrow 3 \\ n \Rightarrow d\ n \Rightarrow 5\ n \Rightarrow 5\ d\ n \Rightarrow 5\ 1\ n \Rightarrow 5\ 1\ d\ n \Rightarrow \ldots \end{array}$$

Given a grammar, the *language* of a non-terminal A is the set of all strings derivable from applications of the production rules, starting with A. We will call this language $L(A)$.

We have just shown that "3" $\in L(n)$ and "5119" $\in L(n)$. What is another name for $L(n)$? What about $L(d)$? How does it compare to $L(n)$?

1.2 Backus-Naur Form (BNF)

Rather than write the same non-terminals over and over again, we can abbreviate the grammar using Backus-Naur Form:

$$\begin{array}{lcl} d & ::= & 0 \mid 1 \mid 2 \mid \ldots \mid 9 \\ n & ::= & d \mid d\ n \end{array}$$

(Many instances of BNF use ankle-brackets around non-terminals (e.g. $\langle n \rangle ::= \langle d \rangle\ \langle n \rangle$) but I'll avoid doing so unless there is an overlap in the way we write terminals and non-terminals.)

2 Ambiguity

Let's add sums and products to our language using another non-terminal:

$$\begin{array}{lcl} d & ::= & 0 \mid 1 \mid \ldots \mid 9 \\ n & ::= & d \mid d\ n \\ e & ::= & n \mid e + e \mid e * e \end{array}$$

Now take the expression "$3 + 4 * 5$". Here is one derivation of this expression.

$$e \Rightarrow e + e \Rightarrow n + e \Rightarrow d + e \Rightarrow 3 + e \Rightarrow 3 + e * e \Rightarrow \dots$$

But what about this alternative? (How is this significant?)

$$e \Rightarrow e * e \Rightarrow e * n \Rightarrow e * d \Rightarrow e * 5 \Rightarrow \dots$$

The conventional rules of arithmetic would tell us that $*$ has higher *precedence* than $+$, that we should read the expression as "$3 + (4 * 5)$".

How can we make the grammar *unambiguous*? Conceptually, we'd like to separate those expressions that we multiply together (i.e. *factors*) from those we add (i.e. *terms*).

$$\begin{array}{rcl} & \vdots & \\ e & ::= & t \mid t + e \\ t & ::= & f \mid f * t \\ f & ::= & n \mid (e) \end{array}$$

What does a derivation of "$3 + 4 * 5$" look like given our new rules?

$$e \Rightarrow t + e \Rightarrow f + e \Rightarrow n + e \Rightarrow d + e \Rightarrow 3 + e \Rightarrow 3 + t \Rightarrow 3 * f + t \Rightarrow \dots$$

3 Parsing

The process of finding a derivation for a given string is called *parsing*; this is exactly what we have been doing by hand, above. Obviously, this is a process we would like to automate.

How would we write a program to parse our arithmetic expressions starting with the non-terminal e? Consider the two productions for e: both start with the non-terminal t, so we start by parsing a term. Afterward, if we've reached the end of the input string, we use the first production (and then we've finished parsing), otherwise we expect to see "+" followed by another expression: we consume the "+" and start again. We can use a similar algorithm in parsing terms t. Finally, for factors f, if the next symbol is a digit, we use the first production. Otherwise, we expect an "(" followed by an expression and a ")".

Note that for our arithmetic grammar, whenever we need to make a decision as to which production to use, the next terminal of the input *always* tells us which choice to make. This sort of algorithm is called a *recursive-descent* parser. In fact, the prose in the preceding paragraph could easily be translated to form a working recursive-descent parser.

Not all context-free grammars are this simple; in general, finding the derivation for a string given a CFG and a starting non-terminal can be far more difficult. While most programming languages have grammars that can be parsed in a reasonable (linear) amount of time, there are few where you would actually want to write the parser. In many cases, programmers who write compilers use *parser generators* turn a BNF grammar into a working program.

In this course, we won't need a parser generator— our grammar will remain fairly simple— and you won't need to worry much about parsing: we will do most of that work for you.

4 Further reading

- Chapter 3 of Harper's draft covers much of the same material.
- Andrew Appel's "Modern Compiler Implementation in {ML, Java, C}" covers a variety of different types of grammars and their respective parsers.
- "Compilers: Principles, Techniques and Tools" by Aho, Sethi, and Ullman has similar coverage of these topics.