15–212: Fundamental Structures of Computer Science II

Some Notes on Grammars and Parsing

Robert Harper, Spring 1997
edited by Frank Pfenning, Fall 1997

Draft of November 4, 1997

1 Introduction

These notes are intended as a "rough and ready" guide to grammars and parsing. The theoretical foundations required for a thorough treatment of the subject are developed in the *Formal Languages, Automata, and Computability* course. The construction of parsers for programming languages using more advanced techniques than are discussed here is considered in detail in the *Compiler Construction* course.

Parsing is the determination of the structure of a sentence according to the rules of grammar. In elementary school we learn the parts of speech and learn to analyze sentences into constituent parts such as subject, predicate, direct object, and so forth. Of course it is difficult to say precisely what are the rules of grammar for English (or other human languages), but we nevertheless find this kind of grammatical analysis useful.

In an effort to give substance to the informal idea of grammar and, more importantly, to give a plausible explanation of how people learn languages, Noam Chomsky introduced the notion of a *formal grammar*. Chomsky considered several different forms of grammars with different expressive power. Roughly speaking, a grammar consists of a series of rules for forming sentence fragments that, when used in combination, determine the set of well-formed (grammatical) sentences. We will be concerned here only with one, particularly useful form of grammar, called a *context-free grammar*. The idea of a context-free grammar is that the rules are specified to hold independently of the context in which they are applied. This is clearly limits the expressive power of the formalism, but is nevertheless powerful enough to be useful, especially with computer languages. To illustrate the limitations of the formalism, Chomsky gave the now-famous sentence "Colorless green ideas sleep furiously." This sentence is grammatical according to some (fairly obvious) context-free rules: it has a subject and a predicate, with the subject modified by two adjectives and the predicate by an adverb. It is debatable whether it is "really" grammatical, precisely because we're uncertain exactly what is the boundary between that which is grammatical and that which is meaningful.

We will dodge these questions by avoiding consideration of interpreted languages (those with meaning), and instead focus on the mathematical notion of a *formal language*, which is just a set of strings over a specified alphabet. A formal language has no intended meaning, so we avoid questions like those suggested by Chomsky's example.[1]

[1] He has given many others. For example, the two sentences "Fruit flies like a banana." and "Time flies like an arrow." are superficially very similar, differing only by two noun-noun replacements, yet their "deep structure" is clearly very different!

2 Context-Free Grammars

Let us fix an alphabet Σ of letters. Recall that Σ^* is the set of strings over the alphabet Σ. In the terminology of formal grammars the letters of the alphabet are called *terminal symbols* or just *terminals* for short, and a string of terminals is called a *sentence.* A *context-free grammar* consists of an alphabet Σ, a set V of *non-terminals* or *variables*, together with a set P of *rules* or *productions* of the form

$$A \to \alpha$$

where A is a non-terminal and α is any sequence of terminals and non-terminals (called a *sentential form*). We distinguish a particular non-terminal $S \in V$, the *start symbol* of the grammar. Thus a grammar G is a four-tuple (Σ, V, P, S) consisting of these four items.

The significance of a grammar G is that it determines a language, $L(G)$, defined as follows:

$$L(G) = \{\, w \in \Sigma^* \mid S \Rightarrow^* w \,\}$$

That is, the language of the grammar G is the set of strings w over the alphabet Σ such that w is *derivable* from the start symbol S. The derivability relation between sentential forms is defined as follows. First, we say that $\alpha \Rightarrow \beta$ iff $\alpha = \alpha_1\, A\, \alpha_2$, $\beta = \alpha_1\, \alpha\, \alpha_2$, and $A \to \alpha$ is a rule of the grammar G. In other words, β may be derived from α by "expanding" one non-terminal from α by one rule of the grammar G. The relation $\alpha \Rightarrow^* \beta$ is defined to hold iff β may be derived from α in zero or more derivation steps.

Example 1 *Let G be the following grammar over the alphabet $\Sigma = \{\, a, b \,\}$:*[2]

$$\begin{array}{lcl} S & \to & \epsilon \\ S & \to & a\,S\,a \\ S & \to & b\,S\,b \end{array}$$

It is easy to see that $L(G)$ consists of strings of the form $w\,w^R$, where w^R is the reverse of w. For example,

$$\begin{array}{lcl} S & \Rightarrow & a\,S\,a \\ & \Rightarrow & a\,b\,S\,b\,a \\ & \Rightarrow & a\,b\,a\,S\,a\,b\,a \\ & \Rightarrow & a\,b\,a\,a\,b\,a \end{array}$$

To *prove* that $L(G) = \{\, w\,w^R \mid w \in \Sigma^* \,\}$ for the above grammar G requires that we establish two containments. Suppose that $x \in L(G)$ — that is, $S \Rightarrow^* x$. We are to show that $x = w\,w^R$ for some $w \in \Sigma^*$. We proceed by induction on the length of the derivation sequence, which must be of length at least 1 since x is a string and S is a non-terminal. In the case of a one-step derivation, we must have $x = \epsilon$, which is trivially of the required form. Otherwise the derivation is of length $n + 1$, where $n \geq 1$. The derivation must look like this:

$$\begin{array}{lcl} S & \Rightarrow & a\,S\,a \\ & \Rightarrow^* & x \end{array}$$

[2]The set of non-terminals is effectively specified by the notation conventions in the rules. In this case the only non-terminal is S, which we implicitly take to be the start symbol.

or like this:

$$\begin{array}{rcl} S & \Rightarrow & b\,S\,b \\ & \Rightarrow^* & x \end{array}$$

We consider the first case; the second is handled similarly. Clearly x must have the form $a\,y\,a$, where $S \Rightarrow^* y$ by a derivation of length n. Inductively, y has the form $u\,u^R$ for some u, and hence $x = a\,u\,u^R\,a$. Taking $w = a\,u$ completes the proof.

Now suppose that $x \in \{\, w\,w^R \mid w \in \Sigma^* \,\}$. We are to show that $x \in L(G)$, *i.e.*, that $S \Rightarrow^* x$. We proceed by induction on the length of w. If w has length 0, then $x = w = \epsilon$, and we see that $S \Rightarrow^* \epsilon$. Otherwise $w = a\,u$, and $w^R = u^R\,a$ or $w = b\,u$ and $w^R = u^R\,b$. In the former case we have inductively that $S \Rightarrow^* u\,u^R$, and hence $S \Rightarrow a\,u\,u^R\,a$. The latter case is analogous. This completes the proof.

Exercise 2 *Consider the grammar G with rules*

$$\begin{array}{rcl} S & \Rightarrow & \epsilon \\ S & \Rightarrow & (\,S\,)\,S \end{array}$$

over the alphabet $\Sigma = \{\,(,)\,\}$. Prove that $L(G)$ consists of precisely the strings of well-balanced parentheses. Warning: *the argument is fairly tricky!*

A word about notation. In computer science contexts we often see context-free grammars presented in *BNF (Backus-Naur Form)*. For example, a language of arithmetic expressions might be defined as follows:

$$\begin{array}{rcl} \langle\texttt{expr}\rangle & ::= & \langle\texttt{number}\rangle \mid \langle\texttt{expr}\rangle + \langle\texttt{expr}\rangle \mid \langle\texttt{expr}\rangle - \langle\texttt{expr}\rangle \\ \langle\texttt{number}\rangle & ::= & \langle\texttt{digit}\rangle\langle\texttt{digits}\rangle \\ \langle\texttt{digits}\rangle & ::= & \epsilon \mid \langle\texttt{digit}\rangle\,\langle\texttt{digits}\rangle \\ \langle\texttt{digit}\rangle & ::= & \texttt{0} \mid \ldots \mid \texttt{9} \end{array}$$

where the non-terminals are bracketed and the terminals are not.

3 Parsing

The *parsing problem* for a grammar G is to determine whether or not $w \in L(G)$ for a given string w over the alphabet of the grammar. There is a polynomial (in fact, cubic) time algorithm that solves the parsing problem for an arbitrary grammar, but it is only rarely used in practice. The reason is that in typical situations restricted forms of context-free grammars admitting more efficient (linear time) parsers are sufficient. We will briefly consider two common approaches used in hand-written parsers, called *operator precedence* and *recursive descent* parsing. It should be mentioned that in most cases parsers are not written by hand, but rather are automatically generated from a specification of the grammar. Examples of such systems are Yacc and Bison, available on most Unix platforms, which are based on what are called LR grammars.

3.1 Operator Precedence Parsing

Operator precedence parsing is designed to deal with infix operators of varying precedences. The motivating example is the language of arithmetic expressions over the operators $+, -, *$, and $/$. According to the standard conventions, the expression $3 + 6 * 9 - 4$ is to be read as $(3 + (6 *$

$9)) - 4$ since multiplication "takes precedence" over addition and subtraction. Left-associativity of addition corresponds to addition yielding precedence to itself (!) and subtraction, and, conversely, subtraction yielding precedence to itself and addition. Unary minus is handled by assigning it highest precedence, so that $4 - -3$ is parsed as $4 - (-3)$.

The grammar of arithmetic expressions that we shall consider is defined as follows:

$$E \;\rightarrow\; n \mid -E \mid E+E \mid E-E \mid E*E \mid E/E$$

where n stands for any number. Notice that, as written, this grammar is *ambiguous* in the sense that a given string may be derived in several different ways. In particular, we may derive the string $3+4*5$ in at least two different ways, corresponding to whether we regard multiplication to take precedence over addition:

$$\begin{array}{lcl} E & \Rightarrow & E+E \\ & \Rightarrow & 3+E \\ & \Rightarrow & 3+E*E \\ & \Rightarrow & 3+4*E \\ & \Rightarrow & 3+4*5 \end{array}$$

$$\begin{array}{lcl} E & \Rightarrow & E*E \\ & \Rightarrow & E+E*E \\ & \Rightarrow & 3+E*E \\ & \Rightarrow & 3+4*E \\ & \Rightarrow & 3+4*5 \end{array}$$

The first derivation corresponds to the reading $(3+(4*5))$, the second to $((3+4)*5)$. Of course the first is the "intended" reading according to the usual rules of precedence. Note that both derivations are *leftmost derivations* in the sense that at each step the leftmost non-terminal is expanded by a rule of the grammar. We have exhibited two distinct leftmost derivations, which is the proper criterion for proving that the grammar is ambiguous. Even unambiguous grammars may admit distinct leftmost and rightmost (defined obviously) derivations of a given sentence.

Given a string w over the alphabet $\{\, n, +, -, *, / \mid n \in \mathbf{N}\,\}$, we would like to determine (a) whether or not it is well-formed according to the grammar of arithmetical expressions, and (b) parse it according to the usual rules of precedence to determine its meaning unambiguously. The parse will be represented by translating the expression from the *concrete syntax* specified by the grammar to the *abstract syntax* specified by the following ML datatype:[3]

```
datatype expr =
    Int of int
  | Plus of expr * expr
  | Minus of expr * expr
  | Times of expr * expr
  | Divide of expr * expr
```

Thus $3+4*5$ would be translated to the ML value `Plus(Int 3, Times (Int 4, Int 5))`, rather than as `Times(Plus(Int 3, Int 4), Int 5)`.

How is this achieved? The basic idea is simple: scan the string from left-to-right, translating as we go. The fundamental problem is that we cannot always decide immediately when to translate.

[3]Since the abstract syntax captures the structure of the parse, the translation into abstract syntax is sometimes called a *parse tree* for the given string. This terminology is rapidly becoming obsolete.

Consider the strings $x = 3 + 4 * 5$ and $y = 3 + 4 + 5$. As we scan from left-to-right, we encounter the sub-expression $3 + 4$, but we cannot determine whether to translate this to `Plus(Int 3, Int 4)` until we see the next operator. In the case of the string x, the right decision is not to translate since the next operator is a multiplication which "steals" the 4, the right-hand argument to the addition being 4*5. On the other hand in the case of the string y the right decision is to translate so that the left-hand argument of the second addition is `Plus(Int 3, Int 4)`.

The solution is to defer decisions as long as possible. We achieve this by maintaining a stack of terminals and non-terminals representing a partially-translated sentential form. As we scan from left to right we have two decisions to make based on each character of the input:[4]

Shift Push the next item onto the stack, deferring translation until further context is determined.

Reduce Replace the top several elements of the stack by their translation according to some rule of the grammar.

Consider again the string x given above. We begin by shifting 3, +, and 4 onto the stack since no decision can be made. We then encounter the $*$, and must decide whether to shift or reduce. The decision is made based on precedence. Since $*$ takes precedence over $+$, we shift $*$ onto the stack, then shift 4 as well. At this point we encounter the end of the expression, which causes us to pop the stack by successive reductions. Working backwards, we reduce $[3, +, 4, *, 5]$ to $[3, +, 4 * 5]$, then reduce again to $[3 + (4 * 5)]$, and yield this expression as result (translated into a value of type `expr`).

For the sake of contrast, consider the expression y given above. We begin as before, shifting 3, +, and 4 onto the stack. Since + yields precedence to itself, this time we reduce the stack to $[3 + 4]$ before shifting $*$ and 5 to obtain the stack $[3 + 4, *, 5]$, which is then reduced to $[(3 + 4) * 5]$, completing the parse.

The operator precedence method can be generalized to admit more operators with varying precedences by following the same pattern of reasoning. We may also consider explicit parenthesization by treating each parenthesized expression as a situation unto itself — the expression $(3 + 4) * 5$ is unambiguous as written, and should not be re-parsed as $3 + (4 * 5)$! The "trick" is to save the stack temporarily while processing the parenthesized expression, then restoring when that sub-parse is complete.

A complete operator-precedence parser for a language of arithmetic expressions is given in the course directory. Please study it carefully! A number of instructive programming techniques are introduced there. Note in particular how the stack elements are represented using higher-order functions to capture pending translation decisions.

3.2 Recursive Descent Parsing

Another common method for writing parsers by hand is called *recursive descent* because it proceeds by a straightforward inductive analysis of the grammar. The difficulty is that the method doesn't always work: only some grammars are amenable to treatment in this way. However, in practice, most languages can be described by a suitable grammar, and hence the method is widely used.

Here is the general idea. We attempt to construct a *leftmost* derivation of the candidate string by a *left-to-right* scan using *one* symbol of look-ahead.[5] By "one symbol of lookahead" we mean

[4]We ignore error checking for the time being.

[5]Hence the name "LL(1)" for grammars that are amenable to such treatment. LR(1) grammars are those that admit parsing by construction of a *rightmost* derivation based on a left-to-right scan of the input using at most one symbol of lookahead.

that we decide which rule to apply at each step based solely on the next character of input; no decisions are deferred as they were in the case of operator-precedence parsing. It should be obvious that this method doesn't always work. In particular, the grammar for arithmetic expressions given above is not suitable for this kind of approach. Given the expression $3 + 4 * 5$, and starting with the non-terminal E, we cannot determine based solely on seeing 3 whether to expand E to $E + E$ or $E * E$. As we observed earlier, there are two distinct leftmost parses for this expression in the arithmetic expression grammar.

You might reasonably suppose that we're out of luck. But in fact we can define *another* grammar for the *same* language that admits a recursive descent parse. We present a suitable grammar in stages.

First, we *layer* the grammar to capture the rules of precedence:

$$\begin{array}{lcl} E & \rightarrow & T \mid T + E \mid T - E \\ T & \rightarrow & F \mid F * T \mid F/T \\ F & \rightarrow & n \end{array}$$

The non-terminals E, T, and F stand for "expression", "term", and "factor", respectively. The layering of the grammar resolves the precedences. In particular, observe that the string $3 + 4 * 5$ has precisely one leftmost derivation according to this grammar:

$$\begin{array}{lcl} E & \Rightarrow & T + E \\ & \Rightarrow & F + E \\ & \Rightarrow & 3 + E \\ & \Rightarrow & 3 + E * E \\ & \Rightarrow & 3 + 4 * E \\ & \Rightarrow & 3 + 4 * 5 \end{array}$$

Notice that this derivation corresponds to the intended reading since the outermost structure is an addition whose right-hand operand is a multiplication.

Exercise 3 *Prove that the two grammars determine the same language. Obviously the language of the "layered" grammar is contained in the language of the original grammar (why?). Conversely, given any derivation in the original grammar, we can re-interpret it as a derivation in the layered grammar by choosing non-terminals appropriately. Make this argument precise.*

Next, we "left factor" the grammar by extracting common prefixes of right-hand sides of rules for a given non-terminal. This results in the following grammar:

$$\begin{array}{rcl} E & \rightarrow & T\,E' \\ E' & \rightarrow & +E \mid -E \mid \epsilon \\ T & \rightarrow & F\,T' \\ T' & \rightarrow & *T \mid /T \mid \epsilon \\ F & \rightarrow & n \end{array}$$

At this point the grammar is suitable for recursive-descent parsing. Here is a derivation of the string $3 + 4 * 5$ once again, using the revised grammar. Notice that each step is either forced (no choice) or is determined by the next symbol of input:

$$\begin{array}{rcll}
E & \Rightarrow & T\,E' & \bullet 3+4*5 \\
 & \Rightarrow & F\,T'\,E' & \bullet 3+4*5 \\
 & \Rightarrow & 3\,T'\,E' & 3\bullet +4*5 \\
 & \Rightarrow & 3\,E' & 3\bullet +4*5 \\
 & \Rightarrow & 3+E & 3+\bullet 4*5 \\
 & \Rightarrow & 3+T\,E' & 3+\bullet 4*5 \\
 & \Rightarrow & 3+F\,T'\,E' & 3+\bullet 4*5 \\
 & \Rightarrow & 3+4\,T'\,E' & 3+4\bullet *5 \\
 & \Rightarrow & 3+4*T\,E' & 3+4*\bullet 5 \\
 & \Rightarrow & 3+4*F\,T'\,E' & 3+4*\bullet 5 \\
 & \Rightarrow & 3+4*5\,T'\,E' & 3+4*5\bullet \\
 & \Rightarrow & 3+4*5\,T' & 3+4*5\bullet \\
 & \Rightarrow & 3+4*5 & 3+4*5\bullet
\end{array}$$

Study this derivation carefully, and be sure you see how each step is determined.

Exercise 4 *Extend the language to include unary minus and parenthesized expressions. Give a layered and left-factored grammar for the extended language. Exhibit a derivation of* $3*-(4+5)$.

There are a few "gotchas" to watch out for when defining grammars for recursive descent parsing. One is that "left recursive" rules must be avoided. Consider the following grammar for sequences of digits:

$$N \;\rightarrow\; \epsilon \mid N\,d$$

where d is a single digit. If we attempt to use this grammar to construct a leftmost derivation for a non-empty string of digits, we go into an infinite loop endlessly expanding the leftmost N, and never making any progress through the input! The following right-recursive grammar accepts the same language, but avoids the problem:

$$N \;\rightarrow\; \epsilon \mid d\,N$$

Now we absorb digits one-by-one until we reach the end of the sequence.

Another trouble spot is that several rules may share a common prefix. For example, consider the following grammar of statements in an imperative programming language:

$$\begin{array}{rcl}
S & \rightarrow & \texttt{if}\,E\,\texttt{then}\,S \\
S & \rightarrow & \texttt{if}\,E\,\texttt{then}\,S\,\texttt{else}\,S
\end{array}$$

where E derives some unspecified set of expressions. The intention, of course, is to express the idea that the `else` clause of a conditional statement is optional. Yet consider how we would write a recursive-descent parser for this grammar. Upon seeing `if`, we don't know whether to apply the first or the second rule — that is, do we expect a matching `else` or not? What to do? First, by left-factoring we arrive at the following grammar:

$$\begin{array}{rcl}
S & \rightarrow & \texttt{if}\,E\,\texttt{then}\,S\,S' \\
S' & \rightarrow & \epsilon \\
S' & \rightarrow & \texttt{else}\,S
\end{array}$$

This pushes the problem to the last possible moment: having seen `if` e `then` s, we must now decide whether to expect an `else` or not. But consider the sentence

`if` e `then` `if` e' `then` s `else` s'

To which `if` does the `else` belong? That is, do we read this statement as:

`if` e `then` (`if` e' `then` s `else` s')

or as

`if` e `then` (`if` e' `then` s) `else` s'?

The grammar itself does not settle this question. The usual approach is to adopt a fixed convention (the first is the typical interpretation), and to enforce it by *ad hoc* means — if there is an `else`, then it "belongs to" the most recent `if` statement.

Designing grammars for recursive-descent parsing is usually fairly intuitive. If you're interested, a completely rigorous account of LL(1) grammars can be found in any textbook on formal language theory or any compiler construction textbook.

A complete recursive-descent parser for the language of arithmetic expressions appears in the course directory.