Lecture Notes on Context-Free Grammars

15-411: Compiler Design
Frank Pfenning

Lecture 7
September 15, 2009

1 Introduction

Grammars and parsing have a long history in linguistics. Computer science built on the accumulated knowledge when starting to design programming languages and compilers. There are, however, some important differences which can be attributed to two main factors. One is that programming languages are designed, while human languages evolve, so grammars serve as a means of specification (in the case of programming languages), while they serve as a means of description (in the case of human languages). The other is the difference in the use of grammars and parsing. In programming languages the meaning of a program should be unambiguously determined so it will execute in a predictable way. Clearly, this then also applies to the result of parsing a well-formed expression: it should be unique. In natural language we are condemned to live with its inherent ambiguities, as can be seen from famous examples such as "*Time flies like an arrow*".

In this lecture we review an important class of grammars, called *context-free grammars* and the associated problem of parsing. They end up to be too awkward for direct use in a compiler, mostly due the problem of ambiguity, but also due to potential inefficiency of parsing. Alternative presentations of the material in this lecture can be found in the textbook [App98, Chapter 3] and in a seminal paper by Shieber et al. [SSP95]. In the next lecture we will consider more restricted forms of grammars, whose definition, however, is much less natural.

2 Context-Free Grammars

Grammars are designed to describe languages, where in our context a *language* is just a set of strings. Abstractly, we think of strings as a sequence of so-called *terminal symbols*. Inside a compiler, these terminal symbols are most likely *lexical tokens*, produced from a bare character string by lexical analysis.

A *context-free grammar* consists of a set of productions of the form $X \longrightarrow \gamma$, where X is a *non-terminal symbol* and γ is a potentially mixed sequence of terminal and non-terminal symbols. It is also sometimes convenient to distinguish a *start symbol* traditionally named S, for *sentence*. We will use the word *string* to refer to any sequence of terminal and non-terminal symbols. We denote strings by $\alpha, \beta, \gamma, \ldots$. Non-terminals are generally denoted by X, Y, Z and terminals by $\mathtt{a}, \mathtt{b}, \mathtt{c}$

For example, the following grammar generates all strings consisting of matching parentheses.

$$\begin{array}{lll} S & \longrightarrow & \\ S & \longrightarrow & [\, S\,] \\ S & \longrightarrow & S\, S \end{array}$$

The first rule looks somewhat strange, because the right-hand side is the empty string. To make this more readable, we usually write the empty string as ϵ.

A *derivation* of a sentence w from start symbol S is a sequence $S = \alpha_0 \longrightarrow \alpha_1 \longrightarrow \alpha_n = w$, where w consists only of terminal symbols. In each step we choose an occurrence of a non-terminal X in α_i and a production $X \longrightarrow \gamma$ and replace the occurrence of X in α_i by γ.

We usually label the productions in the grammar so that we can refer to them by name. In the example above we might write

$$\begin{array}{llll} [\mathsf{emp}] & S & \longrightarrow & \\ [\mathsf{pars}] & S & \longrightarrow & [\, S\,] \\ [\mathsf{dup}] & S & \longrightarrow & S\, S \end{array}$$

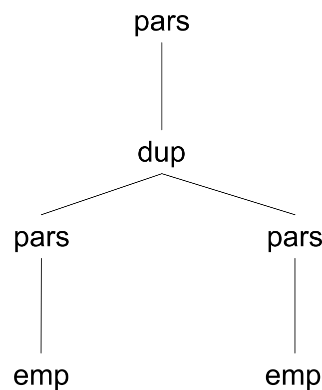
Then the following is a derivation of the string `[[][]]`, where each transi-

tion is labeled with the production that has been applied.

$$\begin{array}{rcll} S & \longrightarrow & [\,S\,] & [\mathsf{pars}] \\ & \longrightarrow & [\,SS\,] & [\mathsf{dup}] \\ & \longrightarrow & [\,[\,S\,]\,S\,] & [\mathsf{pars}] \\ & \longrightarrow & [\,[\,]\,S\,] & [\mathsf{emp}] \\ & \longrightarrow & [\,[\,]\,[\,S\,]\,] & [\mathsf{pars}] \\ & \longrightarrow & [\,[\,]\,[\,]\,] & [\mathsf{emp}] \end{array}$$

We have labeled each derivation step with the corresponding grammar production that was used.

Derivations are clearly not unique, because when there is more than one non-terminal we can replace it in any order in the string. In order to avoid this kind of harmless ambiguity in rule order, we like to construct a *parse tree* in which the nodes represents the non-terminals in a string, with the root being S. In the example above we obtain the following tree:



While the tree removes some ambiguity, it turns out the sample grammar is ambiguous in another way. In fact, there are infinitely many parse trees of every string in the language. This can be seen by considering the cycle

$$S \longrightarrow SS \longrightarrow S$$

where the first step is dup and the second is emp, applied either to the first or second occurrence of S.

Whether a grammar is ambiguous in the sense that there are sentences permitting multiple different parse trees is an important question for the use of grammars for the specification of programming languages. We will see an example shortly.

3 Parse Trees are Deduction Trees

We now present a formal definition of when a terminal string w matches a string γ. We write:

$[r]X \longrightarrow \gamma$	production r maps non-terminal X to string γ
$w : \gamma$	terminal string w matches string γ

The second judgment is defined by the following four simple rules. Here we use string concatenation, denoted by juxtaposing to strings. Note that $\gamma\,\epsilon = \epsilon\,\gamma = \gamma$ and that concatenation is associative.

$$\frac{}{\epsilon : \epsilon}\,P_1 \qquad \frac{w_1 : \gamma_1 \quad w_2 : \gamma_2}{w_1\,w_2 : \gamma_1\,\gamma_2}\,P_2 \qquad \frac{}{\texttt{a} : \texttt{a}}\,P_3 \qquad \frac{\begin{array}{c}[r]X \longrightarrow \gamma \\ w : \gamma\end{array}}{w : X}\,P_4(r)$$

We have labeled the fourth rule by the name of the grammar production, while the others remain unlabeled. This allows us to omit the actual grammar rule from the premises since it can be looked up in the grammar by its name. Then the earlier derivation of `[[][]]` becomes the following deduction.

$$\cfrac{\cfrac{\cfrac{}{\texttt{[} : \texttt{[}}\,P_3 \qquad \cfrac{\cfrac{\cfrac{\cfrac{}{\texttt{[} : \texttt{[}}\,P_3 \quad \cfrac{\cfrac{}{\epsilon : \epsilon}\,P_1}{\epsilon : S}\,P_4(\mathsf{emp}) \quad \cfrac{}{\texttt{]} : \texttt{]}}\,P_3}{\cfrac{\texttt{[]} : \texttt{[}\,S\,\texttt{]}}{\texttt{[]} : S}\,P_4(\mathsf{pars})}\,P_2^2 \qquad \begin{array}{c}\vdots \\ \texttt{[]} : S\end{array}}{\texttt{[][]} : S\,S}\,P_2}{\texttt{[][]} : S}\,P_4(\mathsf{dup}) \qquad \cfrac{}{\texttt{]} : \texttt{]}}\,P_3}{\texttt{[[][]]} : \texttt{[}\,S\,\texttt{]}}\,P_2^2}{\texttt{[[][]]} : S}\,P_4(\mathsf{pars})$$

The one omitted subdeduction is identical to its sibling on the left. We observe that the labels have the same structure as the parse tree, except that it is written upside-down. Parse trees are therefore just deduction trees.

4 CYK Parsing

The rules above that formally define when a terminal string matches an arbitrary string can be used to immediately give an algorithm for parsing.

Assume we are given a grammar with start symbol S and a terminal string w_0. Start with a databased of assertions $\epsilon : \epsilon$ and $a : a$ for any terminal symbol occuring in w. Now arbitrarily apply the given rules in the following way: if the premises of the rules can be matched against the database, and the conclusion $w : \gamma$ is such that w is a substring of w_0 and γ is a string occurring in the grammar, then add $w : \gamma$ to the database.

We repeat this process until we reach *saturation*: any further application of any rule leads to conclusion are already in the database. We stop at this point and check if we see $w_0 : S$ in the database. If yes, we succeed; if not we fail.

This process must always terminate, since there are only a fixed number of substrings of the grammar, and only a fixed number of substrings of the query string w_0. In fact, only $O(n^2)$ terms can ever be derived if the grammar is fixed and $n = |w|$. Using a meta-complexity result by Ganzinger and McAllester [GM02] we can obtain the complexity of this algorithm as the maximum of the size of the saturated database (which is $O(n^2)$) and the number of so-called *prefix firings* of the rule. We count this by bounding the number of ways the premises of each rule can be instantiated, when working from left to right. The crucial rule is

$$\frac{w_1 : \gamma_1 \quad w_2 : \gamma_2}{w_1\, w_2 : \gamma_1\, \gamma_2}\ P_2$$

There are $O(n^2)$ substrings, so there are $O(n^2)$ ways to match the first premise against the database. Since $w_1\, w_2$ is also constrained to be a substring of w_0, there are only $O(n)$ ways to instantiate the second premise, since the left end of w_2 in the input string is determined, but not its right end. This yields a complexity of $O(n^3)$.

The algorithm we have just presented is an abstract form of the Cocke-Younger-Kasami (CYK) parsing algorithm. It assumes the grammar is in a normal form, and represents substring by their indices in the input rather than directly as strings. However, its general running time is still $O(n^3)$.

As an example, we apply this algorithm using an n-ary concatenation rule as a short-hand. We try to parse `[[][]]` with our grammar of matching parentheses. We start with three facts that derive from rules P_1 and P_3. When working forward it is important to keep in mind that we only infer

facts $w : \gamma$ where w is a substring of $w_0 =$ `[[][]]` and γ is a substring of the grammar.

1	`[`	:	`[`	
2	`]`	:	`]`	
3	ϵ	:	ϵ	
4	ϵ	:	S	$P_4(\mathsf{emp})\ 3$
5	`[]`	:	`[` S `]`	$P_2^2\ 1, 4, 2$
6	`[]`	:	S	$P_4(\mathsf{pars})\ 5$
7	`[][]`	:	$S\,S$	$P_2\ 6, 6$
8	`[][]`	:	S	$P_4(\mathsf{dup})\ 7$
9	`[[][]]`	:	`[` S `]`	$P_2^2\ 1, 8, 4$
10	`[[][]]`	:	S	$P_4(\mathsf{pars})\ 9$

A few more redundant facts might have been generated, such as `[]` $: S\,S$, but otherwise parsing is remarkably focused in this case. From the justifications in the right-hand column is it easy to generate the same parse tree we saw earlier.

5 Recursive Descent Parsing

For use in a programming language parser, the cubic complexity of the CYK algorithm is unfortunately unacceptable. It is also not so easy to discover potential ambiguities in a grammar or give good error messages when parsing fails. What we would like it an algorithm that scans the input left-to-right (because that's usually how we design our languages!) and works in one pass through the input.

Unfortunately, some languages that have context-free grammars cannot be specified in the form of a grammar satisfying the above specification. So now we turn the problem around: considering the kind of parsing algorithms we would like to have, can we define classes of grammars that can be parsed with this kind of algorithm? The other property we would like is that we can look at a grammar and decide if it is ambiguous in the sense that there are some strings admitting more than one parse tree. Such grammars should be rejected as specifications for programming languages. Fortunately, the goal of efficient parsing and the goal of detecting ambiguity in a grammar work hand-in-hand: generally speaking, unambiguous grammars are easier to parse.

We now rewrite our rules for parsing to work from left-to-right instead of being symmetric. This means we do not use general concatenation, but just prepend a single non-terminal to the beginning of a string. We also

have to change the nature of the rule for non-terminals so it can handle a non-terminal at the left end of the string.

$$\frac{}{\epsilon : \epsilon}\ R_1 \qquad \frac{w : \gamma}{\mathtt{a}\, w : \mathtt{a}\, \gamma}\ R_2 \qquad \frac{\begin{array}{c}[r] X \longrightarrow \beta \\ w : \beta\, \gamma\end{array}}{w : X\, \gamma}\ R_3(r)$$

At this point the rules are entirely linear (each rule has zero or one premises) and decompose the string left-to-right (we only proceed by stripping away a terminal symbol a).

Rather than blindly using these rules from the premises to the conclusions (which wouldn't be analyzing the string from left to right), couldn't we use them the other way around? Recall that we are starting with a given goal, namely to derive $w_0 : S$, if possible, or explicitly fail otherwise. Now could we use the rules in a goal-directed way? The first two rules certainly do not present a problem, but the third one presents a problem since we may not be able to determine which production to use if there are multiple productions for a given non-terminal X.

The difficulty then lies in the third rule: how can we decide which production to use? We can turn the question around: for which grammars can we always decide which rule to use in the third case?

We return to an example to explore this question. We use a simple grammar for an expression language similar one to the one used in Lab 1. We use *id* and *num* to stand for identifier and number tokens produced by the lexer.

[assign]	S	$\longrightarrow$	id = E ; S
[return]	S	$\longrightarrow$	return E
[plus]	E	$\longrightarrow$	E + E
[times]	E	$\longrightarrow$	E * E
[ident]	E	$\longrightarrow$	id
[number]	E	$\longrightarrow$	num
[parens]	E	$\longrightarrow$	(E)

As an example string, consider `x = 3; return x;`. After lexing, `x` and `3` are replace by tokens $id($"x"$)$ and $num(3)$, which we write just as id and num, for short.

If we always guess right, we would construct the following deduction *from the bottom to the top*. That is, we start with the last line, either determine or guess which rule to apply to get the previous line, etc. until we reach $\epsilon : \epsilon$ or get stuck.

$$
\begin{array}{rcll}
\epsilon & : & \epsilon & \\
\texttt{;} & : & \texttt{;} & \\
id\ \texttt{;} & : & id\ \texttt{;} & \\
id\ \texttt{;} & : & E\ \texttt{;} & [\mathsf{ident}] \\
\texttt{return}\ id\ \texttt{;} & : & \texttt{return}\ E\ \texttt{;} & \\
\texttt{return}\ id\ \texttt{;} & : & S & [\mathsf{return}] \\
\texttt{;}\ \texttt{return}\ id\ \texttt{;} & : & \texttt{;}\ S & \\
num\ \texttt{;}\ \texttt{return}\ id\ \texttt{;} & : & num\ \texttt{;}\ S & \\
num\ \texttt{;}\ \texttt{return}\ id\ \texttt{;} & : & E\ \texttt{;}\ S & [\mathsf{number}] \\
=num\ \texttt{;}\ \texttt{return}\ id\ \texttt{;} & : & =E\ \texttt{;}\ S & \\
id=num\ \texttt{;}\ \texttt{return}\ id\ \texttt{;} & : & id=E\ \texttt{;}\ S & \\
id=num\ \texttt{;}\ \texttt{return}\ id\ \texttt{;} & : & S & [\mathsf{assign}]
\end{array}
$$

This parser (assuming all the guesses are made correctly) evidently traverses the input string from left to right. It also produces a *leftmost* derivation, which we can read off from this deduction by reading the right-hand side from the bottom to top.

We have labeled the inference that potentially involved a choice with the chosen name of the chosen grammar production. If we restrict ourselves to look at the first character in the input string on the left, which ones could we have predicted correctly?

In the last line (the first guess we have to make) we are trying to parse an S and the first input token is id. There is only one production that would allow this, namely [assign]. So there is no guess necessary.

In the fourth-to-last line (our second potential choice point), the first token is num and we are trying to parse an E. It is tempting, but wrong, to say that this must be the production [number]. For example, the string $num + id$ also starts with token num, but we must use production [plus].

In fact, no input token can disambiguate expression productions for us. The problem is that the rules [plus] and [times] are *left-recursive*, that is, the right-hand side of the production starts with the non-terminal on the left-hand side. We can never decide by a token look-ahead which rule to choose, because any token which can start an expression E could arise via the [plus] and [times] productions.

In the next lecture we develop some techniques for analyzing the grammar to determine if we can parse by searching for a deduction without backtracking, if we are permitted some lookahead to make the right choice. This will also be the key for *parser generation*, the process of compiling a grammar specification to a specialized efficient parser.

References

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, England, 1998.

[GM02] Harald Ganzinger and David A. McAllester. Logical algorithms. In P.Stuckey, editor, *Proceedings of the 18th International Conference on Logic Programming*, pages 209–223, Copenhagen, Denmark, July 2002. Springer-Verlag LNCS 2401.

[SSP95] Stuart M. Shieber, Yves Schabes, and Fernando C. N. Pereira. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24(1–2):3–36, 1995.