Lecture Notes on Context-Free Grammars

15-411: Compiler Design
Frank Pfenning*

Lecture 8
September 18, 2014

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* (Chomsky-2 in the Chomsky hierarchy [Cho59]) and the associated problem of parsing. They end up to be too awkward for direct use in a compiler, mostly due to 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.

*With edit by André Platzer

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 that already groups substrings into tokens of appropriate type and skips over whitespace.

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.

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

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 exactly one occurrence of a non-terminal X in α_i and one production $X \longrightarrow \gamma$ and replace this 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

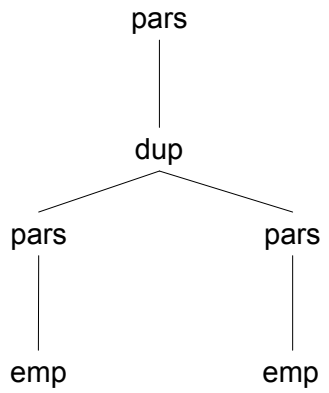
$$\begin{array}{lcll} [\mathsf{emp}] & S & \longrightarrow & \\ [\mathsf{pars}] & S & \longrightarrow & \mathtt{[}S\mathtt{]} \\ [\mathsf{dup}] & S & \longrightarrow & S\,S \end{array}$$

Then the following is a derivation of the string `[[][]]`, where each transition is labeled with the production that has been applied.

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

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, then 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 parse 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 above 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. We can get arbitrarily long parse trees for the same string with this.

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. The basic problem is that it becomes ambiguous in which grammatical function a specific terminal occurs in the source program. This could lead to misinterpretations. 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 the empty string ϵ satisfies $\gamma\,\epsilon = \epsilon\,\gamma = \gamma$ and that concatenation is associative (mathematically speaking, strings form a monoid, which is like a group that does not have inverse elements).

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

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 directly by its name. Then the earlier derivation of `[[] []]` becomes the following deduction.

$$\dfrac{\dfrac{\dfrac{}{\texttt{[} : \texttt{[}}\ P_3 \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\epsilon : \epsilon}\ P_1}{\epsilon : S}\ P_4(\mathsf{emp}) \quad \dfrac{}{\texttt{]} : \texttt{]}}\ P_3}{\texttt{[]} : \texttt{[}\,S\,\texttt{]}}\ P_2^2}{\texttt{[]} : S}\ P_4(\mathsf{pars}) \quad \vdots}{\texttt{[]}\,\texttt{[]} : S\,S}\ P_2 \quad \dfrac{}{\texttt{]} : \texttt{]}}\ P_3}{\texttt{[]}\,\texttt{[]} : S}\ P_4(\mathsf{dup})}{\texttt{[]}\,\texttt{[]}\,\texttt{]} : \texttt{[}\,S\,\texttt{]}}\ P_2^2}{\texttt{[]}\,\texttt{[]}\,\texttt{]} : S}\ P_4(\mathsf{pars})$$

The one omitted subdeduction (marked $\vdots$) 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 occurring in w_0. 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 the input w_0 and γ is a string occurring in the grammar, then add $w : \gamma$ to the database. The side conditions are used to focus the parsing process to the facts that may matter during the parsing (i.e., that talk

about the actual input string w_0 being parsed and that fit to the actual grammatical productions in the grammar).

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 we see $w_0 : S$, we succeed parsing w_0; 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 [McA02, 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 the splitting rule

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

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 $O(n^2 * n) = O(n^3)$.

The algorithm we have just presented is an abstract form of the Cocke-Younger-Kasami (CYK) parsing algorithm invented in the 1960s. It originally 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(emp) 3
5	`[]`	:	`[` S `]`	P_2^2 1, 4, 2
6	`[]`	:	S	P_4(pars) 5
7	`[][]`	:	$S\, S$	P_2 6, 6
8	`[][]`	:	S	P_4(dup) 7
9	`[[][]]`	:	`[` S `]`	P_2^2 1, 8, 2
10	`[[][]]`	:	S	P_4(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 (except for a particular input) or give good error messages when parsing fails. What we would like 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 exclusively from left-to-right instead of being symmetric. This means we do not use general concatenation of strings that are split arbitrarily. Instead, we just consider the left-most terminal or left-most non-terminal. We just prepend a single non-terminal to the beginning of a string. This left non-terminal is then the only part where we allow expansion by a production. 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.

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

Rule R_2 *compares* the first terminal `a` of the actual input string $\mathtt{a}w$ with the first terminal `a` of the currently parsed expression $\mathtt{a}\gamma$. For grammar production $[r]X \rightarrow \beta$, rule $R_3(r)$ *generates* or *expands* the righthand side β for the left-most non-terminal X in the currently parsed expression $X\gamma$. Rule $R_3(r)$ uses the grammar production forward to produce the result β. Of course, ultimately, the parse derivation will only be successful if the compare rule R_2 can also match the ultimately generated terminals in the input and the generated parse expression.

At this point the rules are entirely linear (each rule has zero or one premises, note that we count the static grammar productions $[r]X \to \beta$ as part of the rule $R_3(r)$ here) 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 from the desired conclusions to the premisses? After all, we know what we are trying to get at. 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. Using the compare rules R_1 and R_2 from conclusions to premisses just successively simplifies the strings by consuming the first token (or ϵ). But the expansion rule R_3 presents a problem, since we may not be able to determine which production we should 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? Guessing which expansion β of X in R_3 will enable us to parse w as $\beta\gamma$ could be difficult. Yet, we can turn the question around: for which grammars can we always decide which grammar expansion r to use for $R_3(r)$?

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;
```
$id(\texttt{"x"}) = num(3)$; `return` $id(\texttt{"x"})$

After lexing, `x` and `3` are replaced by tokens $id(\texttt{"x"})$ and $num(3)$ as indicated in the second line. We write just write those tokens 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 $\epsilon : \epsilon$ (successful parse) or get stuck (syntax error, or wrong guess).

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

This parser (assuming all the guesses are made correctly) evidently traverses the input string from left to right. It also produces a *left-most* derivation (always expand the left-most nonterminal first), 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 only at the first token in the input string on the left, which ones could we have predicted correctly? Which grammar production choices could we predict by looking ahead at the first input token?

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 we do not have to guess but just choose deterministically based on the first token id.

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 to say that this must be the production [number]. But this is wrong! For example, the string num + id also starts with token num, but we must use production [plus] to parse it correctly. This is bad news, because we cannot decide which production rule to use based on the first token.

In fact, no input token can disambiguate expression productions for us here. 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. But this non-terminal could produce a lot of different strings. We can never decide by a finite token look-ahead which rule to choose, because any token which can start an expression E could arise via the [plus] and [times] productions. We cannot decide if we will need the [plus] or [times] production just based on the first token before we have fully understood what the first E is. Yet E could have unbounded length.

The only thing we can do at our current state of knowledge is to parse the

input with a recursive descent parser, guess our choices, and backtrack to different choices whenever things don't work out.

In the next lecture we develop some techniques for analyzing the grammar to determine if we can parse its language 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.

6 Rewriting Grammars

As we have seen in the previous examples, grammars may be ambiguous, and also grammars may force us to backtrack during recursive descent parsing. Such backtracking could lead to very inefficient parsing algorithms and should be avoided if that is possible. One way we can sometimes avoid it (and we'll see more about thi in the next lecture) is to rewrite the grammar, while still accepting the same language. For example, we can rewrite the language of matching parentheses to

$$\begin{array}{lll} [\mathsf{emp}] & S & \longrightarrow \\ [\mathsf{next}] & S & \longrightarrow \quad \texttt{[}S\texttt{]}S \end{array}$$

which combines the dup and pars rule into one. Does this really generate the same language? It is an interesting exercise to show that this is the case. Now let's perform a recursive descent parse using this grammar.

$$\uparrow \quad \texttt{[[][]]} \quad : \quad S$$

At this point, there are two possibilities: we could use the next production or the emp production. But emp fails immediately

$$\begin{array}{|lll} \uparrow & \text{impossible} & \\ & \texttt{[[][]]} \quad : \quad \epsilon & \\ & \texttt{[[][]]} \quad : \quad S & \mathsf{emp} \end{array}$$

because there is no rule to could be applied to conclude `[[][]]` $: \epsilon$. So we apply the next rule instead, followed by R_2 skipping past matching left brackets.

$$\begin{array}{|lll} \uparrow & \texttt{[][]} \quad : \quad S\texttt{]}S & \\ & \texttt{[[][]]} \quad : \quad \texttt{[}S\texttt{]}S & \\ & \texttt{[[][]]} \quad : \quad S & \mathsf{next} \end{array}$$

Again, eps will fail immediately and we have to use next.

$$\begin{array}{|lll} \uparrow & \texttt{][]]} \quad : \quad S\texttt{]}S\texttt{]}S & \\ & \texttt{[][]]} \quad : \quad \texttt{[}S\texttt{]}S\texttt{]}S & \\ & \texttt{[][]]} \quad : \quad S\texttt{]}S & \mathsf{next} \\ & \texttt{[[][]]} \quad : \quad \texttt{[}S\texttt{]}S & \\ & \texttt{[[][]]} \quad : \quad S & \mathsf{next} \end{array}$$

Now, next will fail immediately, because] would not match [!

	impossible		
][]	:	$[S]S]S]S$	
][]	:	$S]S]S$	next
[][]	:	$[S]S]S$	
[][]	:	$S]S$	next
[[][]]	:	$[S]S$	
[[][]]	:	S	next

Instead we apply emp, followed by stripping of the matching right brackets.

[]]	:	$S]S$	
][]]	:	$]S]S$	
][]]	:	$S]S]S$	emp
[][]]	:	$[S]S]S$	
[][]]	:	$S]S$	next
[[][]]	:	$[S]S$	
[[][]]	:	S	next

Performing more steps mechanically now, since we always know which rule to choose, we complete the parse as follows:

ϵ	:	ϵ	
ϵ	:	S	emp
]	:	$]S$	
]	:	$S]S$	emp
]]	:	$]S]S$	
]]	:	$S]S]S$	emp
[]]	:	$[S]S]S$	
[]]	:	$S]S$	next
][]]	:	$]S]S$	
][]]	:	$S]S]S$	emp
[][]]	:	$[S]S]S$	
[][]]	:	$S]S$	next
[[][]]	:	$[S]S$	
[[][]]	:	S	next

We have seen that at each point the rule was determined, and by only looking at the next token in the word on the left we could tell with grammar production to use. A grammar with this property is called LL(1), because we parse *Left-to-right*, generating a *Leftmost parsing derivation*, using *1 token lookahead*. Unfortunately this very nice class of grammars it too restrictive for many programming languages, we we will investigate a more general class in the next lecture on predictive parsing.

Questions

1. What is the benefit of using a lexer before a parser?
2. Why do compilers have a parsing phase? Why not just work without it?
3. Is there a difference between a parse tree and an abstract syntax tree? Should there be a difference?
4. What aspects of a programming language does a parser not know about? Should it know about it?
5. For which programming languages and for which programs is recursive descent parsing slow?
6. What are all the benefits of reading the input from left to right? Are there downsides?
7. Is there a language that CYK can parse but recursive descent cannot parse?
8. What are *all* the difficulties with rule P_2? What are all the difficulties with rule $P_4(r)$?

References

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

[Cho59] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, 1959.

[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.

[McA02] David A. McAllester. On the complexity analysis of static analyses. *Journal of the ACM*, 49(4):512–537, 2002.

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