Lecture Notes on Context-Free Grammars

15-411: Compiler Design
Frank Pfenning, Rob Simmons, André Platzer, Jan Hoffmann

Lecture 7
September 18, 2018

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: in the sentence *"Prachi spotted Vijay with binoculars,"* it is possible that Prachi had the binoculars (using them to spot Vijay) or that Vijay had the binoculars (which were with him when he was spotted by Prachi).

In this lecture we review an important class of grammars, called *context-free grammars* and the associated problem of parsing. Some context-free grammars are not suitable for the use in a compiler, mostly due to the problem of ambiguity, but also due to potential inefficiency of parsing. However, the tools we use to *describe* context-free grammars remain incredibly important in practice. We use Backus-Naur form, a way of specifying context-free grammars, to specify our languages in technical communication: you've already seen this in programming assignments for this class. We also use the language of context-free grammars to describe grammars to computers. The input to *parser generators* such as Yacc or Happy is a context free grammar (and possibly some precedence rules), and the output is efficient parsing code written in your language of choice.

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].

2 Grammars

Grammars are a general way to describe languages. You have already seen regular expressions, which also define languages but are less expressive. A *language* is a set of sentences and a sentence is a sequence drawn from a finite set Σ of *terminal symbols*. Grammars also use *non-terminal symbols* that are successively replaced using *productions* until we arrive at a sentence. Sequences that can contain non-terminal and terminal symbols are called *strings*. We denote strings by $\alpha, \beta, \gamma, \ldots$. non-terminals are generally denoted by X, Y, Z and terminals by $\mathtt{a}, \mathtt{b}, \mathtt{c}$. Inside a compiler, 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 grammar is defined by set of productions $\alpha \to \beta$ and a *start symbol* S, a distinguished non-terminal symbol. In the productions, α and β are strings and α contains at least one non-terminal symbol.

For a given grammar G with start symbol S, a derivation in G is a sequence of rewritings

$$S \to \gamma_1 \to \cdots \to \gamma_n = w$$

in which we apply productions from G. For instance, if $\alpha \to \beta$ is a production then γ_{i+1} might be the string that we get by replacing one occurrence of α in γ_i by β. We often simply write $S \to^* w$ instead of $S \to \gamma_1 \to \cdots \to w$.

The language $L(G)$ of G is the set of sentences that we can derive using the productions of G.

Consider for example the following grammar.

$$\begin{array}{lrcl} [1] & S & \longrightarrow & aBSc \\ [2] & S & \longrightarrow & abc \\ [3] & Ba & \longrightarrow & aB \\ [4] & Bb & \longrightarrow & bb \end{array}$$

To refer to the productions, we assign a label $[\ell]$ to each rule. Following a common conversion, lower-case letters are terminal symbols and upper-case denote non-terminal symbols. The following is a derivation of the sentence $a^3b^3c^3$. We annotate

Class	Languages	Automaton	Rules	Word Problem	Example
type-0	recursively enumerable	Turing machine	no restriction	undecidable	Post's corresp. problem
type-1	context sensitive	linear-bounded TM	$\alpha \to \gamma$ $\lvert\alpha\rvert \leq \lvert\gamma\rvert$	PSPACE-complete	$a^n b^n c^n$
type-2	context free	pushdown automaton	$A \to \gamma$	cubic	$a^n b^n$
type-3	regular	NFA / DFA	$A \to a$ or $A \to aB$	linear time	$a^* b^*$

Figure 1: Chomsky Hierarchy

each step in the derivation with the label of the production that we applied.

$$\begin{aligned} S &\longrightarrow_1 aBSc \\ &\longrightarrow_1 aBaBScc \\ &\longrightarrow_3 aaBBScc \\ &\longrightarrow_2 aaBBabccc \\ &\longrightarrow_3 aaBaBbccc \\ &\longrightarrow_3 aaaBBbccc \\ &\longrightarrow_4 aaaBbbccc \\ &\longrightarrow_4 aaabbbccc \end{aligned}$$

Grammars are very expressive. In fact, we can describe all recursively enumerable languages with grammars. As a consequence, it is in general undecidable if $w \in L(G)$ for a string w and a grammar G (the so-called word problem). Of course, we would like our compiler to be able to quickly decide whether a given input program matches the specification given by the grammar. So we will use a class of grammars for which we can decide if $w \in L(G)$ more efficiently.

The syntax of programming languages is usually give by a context-free grammar. Context-free grammars (and languages) are also called type-2 grammars following the classification in the Chomsky hierarchy [Cho59]. We have already seen type-3 languages in the lecture about lexing. Type-3 languages are regular languages. The Chomsky hierarchy is shown in Figure 1. In the table, you find the grammar class, the alternative name of the corresponding languages, the corresponding automata model that recognizes languages of the class, restrictions on the rules, and an example language. We say a language is of type-n if it can be described by a grammar of type-n. The example languages for type-n are not of type-$(n-1)$. Note that every grammar (language) of type-$n+1$ is of type-n.

The table also describes the complexity of the word problem for the respective class of grammars, that is, given a grammar G and a word $w \in \Sigma^*$, decide if $w \in L(G)$. Regular expressions are not expressive enough for programming languages since they cannot describe "matching parenthesis structures" such as $\{a^n b^n \mid n \geq$

1}. So we have to use at least context-free grammars (type 1). In general, deciding the word problem for type-1 grammars is cubic in the length of the word. However, we will use a specific context-free grammars that can be parsed more efficiently.

3 Context-Free Grammars

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.

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

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

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.

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

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

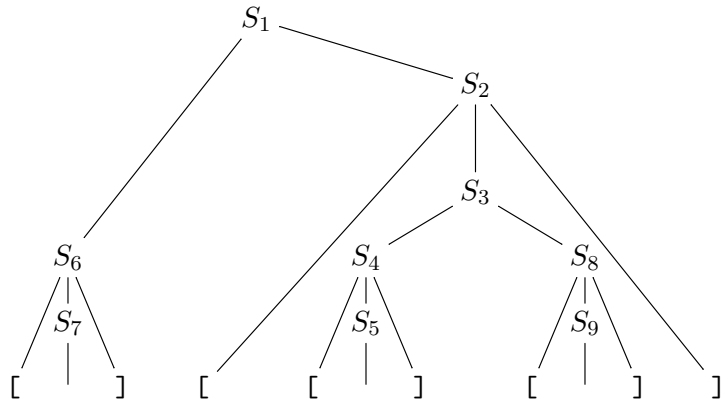
The following is a derivation of the string `[][[][]]`, where each transition is labeled with the production that has been applied.

$$\begin{array}{lllll} \text{Step 1:} & S & \longrightarrow & SS & [\mathsf{dup}] \\ \text{Step 2:} & & \longrightarrow & S\texttt{[}S\texttt{]} & [\mathsf{pars}] \\ \text{Step 3:} & & \longrightarrow & S\texttt{[}SS\texttt{]} & [\mathsf{dup}] \\ \text{Step 4:} & & \longrightarrow & S\texttt{[[}S\texttt{]}S\texttt{]} & [\mathsf{pars}] \\ \text{Step 5:} & & \longrightarrow & S\texttt{[[]}S\texttt{]} & [\mathsf{emp}] \\ \text{Step 6:} & & \longrightarrow & \texttt{[}S\texttt{][[]}S\texttt{]} & [\mathsf{pars}] \\ \text{Step 7:} & & \longrightarrow & \texttt{[][[]}S\texttt{]} & [\mathsf{emp}] \\ \text{Step 8:} & & \longrightarrow & \texttt{[][[][}S\texttt{]]} & [\mathsf{pars}] \\ \text{Step 9:} & & \longrightarrow & \texttt{[][[][]]} & [\mathsf{emp}] \end{array}$$

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. If we always replace the rightmost non-terminal then we call a derivation the *rightmost derivation*. If we always replace the leftmost non-terminal then we have the *leftmost derivation*.

In order to avoid this kind of harmless ambiguity, 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:



The subscripts on the grammar productions just correspond to steps in the step-by-step derivation: S_3 corresponds to step 3, where we used the production dup to rewrite S`[`S`]` to S`[`SS`]`. The nonessential choices in our changes below correspond to the fact that there are a large number of ways of traversing the single parse tree above.

4 Ambiguity

While the parse tree removes some ambiguity, it turns out that the example grammar is ambiguous in other, more important, ways. In fact, there are an infinite number of parse trees of every string in the above language. This can be most easily seen by considering the cycle

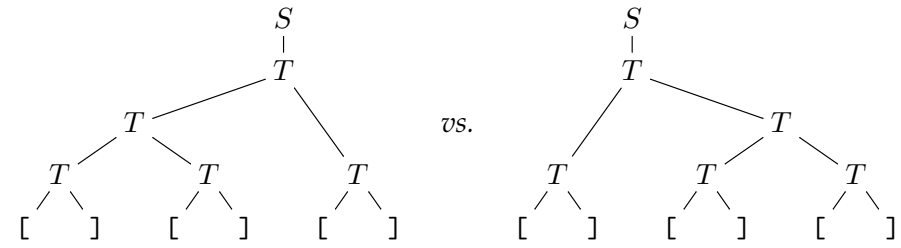
$$S \longrightarrow SS \longrightarrow S$$

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. (Why is this a problem?)

Resolving this ambiguity is not too difficult: we observe that the only reason we need to have a specific production for parsing the empty string is so that $\epsilon \in S$. Otherwise, we only need ϵ in order to parse the middle of the string `[]`, and we can add that string specifically as the new base case for the language of non-empty strings.

[emp]	S	$\longrightarrow$	ϵ
[nonemp]	S	$\longrightarrow$	T
[sing]	T	$\longrightarrow$	`[]`
[pars]	T	$\longrightarrow$	`[`T`]`
[dup]	T	$\longrightarrow$	TT

While this new grammar gets rid of the fact that all strings in the language can be parsed an infinite number of ways, there is still another ambiguity: the string `[] [] []` has two distinct parsings, which are represented by two structurally-distinct parse trees.



Why does this ambiguity matter? There are reasons why it's more efficient to parse unambiguous grammars. More fundamentally, though, when we are writing compilers we are primarily interested in the parse trees, and a parsing algorithm for the ambiguous grammar above might generate one of two data structures:

```
(nonemp (dup (dup sing sing) sing))
```
vs.
```
(nonemp (dup sing (dup sing sing)))
```

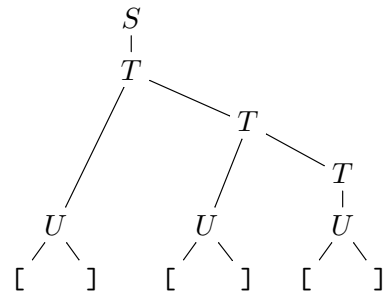
(Note: we won't always generate exactly the tree structure represented by our parse trees, because parser generators allow us to define `nonemp`, `dup`, `pars`, and `sing` either as constructors or functions. When grammar productions are associated with functions, they are sometimes called *semantic actions*.)

It is not immediately obvious which parse tree we ought to prefer, but the differences might be important to the meaning of the program! If we prefer the second parse and want to exclude the first one, we can rewrite the grammar as follows:

[emp]	S	$\longrightarrow$	ϵ
[nonemp]	S	$\longrightarrow$	T
[dup]	T	$\longrightarrow$	$U\,T$
[atom]	T	$\longrightarrow$	U
[sing]	U	$\longrightarrow$	`[]`
[pars]	U	$\longrightarrow$	`[`T`]`

Then there is only one parse remaining, which follows the structure of the original

parse.



If we made the identity function the semantic action associated with the new [atom] production, then we would end up with the same data structure that we associated with the leftmost tree before: `(nonemp (dup (dup sing sing) sing))`.

We've progressive rewritten the grammar in a way that makes it relatively clear that we haven't changed the language. Sometimes, this can be done in a less obvious way.

$$\begin{array}{lllll} [\text{emp}] & S & \longrightarrow & \\ [\text{next}] & S & \longrightarrow & \texttt{[}S\texttt{]}\,S \end{array}$$

The grammar is unambiguous, but does it really generate the same language? It is an interesting exercise to show that this is the case.

Working through another example, let's take an ambiguous grammar for arithmetic:

$$\begin{array}{lllll} [\text{plus}] & E & \longrightarrow & E \texttt{ + } E \\ [\text{minus}] & E & \longrightarrow & E \texttt{ - } E \\ [\text{times}] & E & \longrightarrow & E \texttt{ * } E \\ [\text{number}] & E & \longrightarrow & num \\ [\text{parens}] & E & \longrightarrow & \texttt{(} E \texttt{)} \end{array}$$

There are very strong conventions about this sort of mathematical notation that the grammar does not capture. It is important that we parse `3 - 4 * 5 + 6` as `((3 - (4 * 5)) + 6)` and not `(3 - ((4 * 5) + 6))` or `(3 - (4 * (5 + 6)))`.

In order the rewrite it to make the parse tree unambiguous we have to analyze how to *rule out* the unintended parse tree. In the expression `3 + 4 * 5` we have to all the parse equivalent to `3 + (4 * 5)` but we have to *rule out* the parse equivalent to `(3 + 4) * 5`. In other words, the left-hand side of a product is *not allowed to be a sum* (unless it is explicitly parenthesized).

Backing up one step, how about `3 + 4 + 5`? We want addition to be *left associative*, so this should parse as `(3 + 4) + 5`. In other words, we have to *rule out* the parse `3 + (4 + 5)`. Instead of

$$E \longrightarrow E \texttt{ + } E$$

we want

$$E \longrightarrow E \texttt{ + } P$$

where P is a new nonterminal that does not allow a sum. Continuing the above thought, P *is* allowed to be a product, so we proceed

$$P \longrightarrow P * A$$

Since multiplication is also left-associative, we have made up a new symbol A which cannot be a product. In fact, in our language A can only be an identifier, a number, or a parenthesized (arbitrary) expression.

[plus]	E	$\longrightarrow$	E + P
[minus]	E	$\longrightarrow$	E - P
[times]	P	$\longrightarrow$	P * A
[number]	A	$\longrightarrow$	num
[parens]	A	$\longrightarrow$	(E)

This is not yet complete, because it is in fact empty: it claims an expression must always be a sum. But it could also just be a product. Similarly, products P may just consist of an atom A. This yields:

[plus]	E	$\longrightarrow$	E + P
[minus]	E	$\longrightarrow$	E - P
[e/p]	E	$\longrightarrow$	P
[times]	P	$\longrightarrow$	P * A
[p/a]	P	$\longrightarrow$	A
[number]	A	$\longrightarrow$	num
[parens]	A	$\longrightarrow$	(E)

You should convince yourself that this grammar is now unambiguous. It is also more complicated! In this case (as in many cases) we can describe the problem unambiguously in terms of *associativity* and *precedence*: we expect multiplication to have higher precedence than addition and subtraction, and all these operators are left associative. Similarly, in our first example we were asking for the [dup] production to behave in a left-associative way, similar to functional programming languages where fxy is parsed the same way as $(fx)y$. In the other direction, multiple assignment, which appears in C but not C0, is right-associative:

```
x = y = z = w = 3;
```
is the same as
```
x = (y = (z = (w = 3)));
```

It is important that programming languages be unambiguous in practice. We can usually rewrite grammars to remove ambiguity, but sometimes we extend the language of context-free grammars to resolve ambiguity. One example is explicitly stating precedence and associativity as a way of resolving ambiguities.

5 Parse Trees are Deduction Trees

We now present a formal definition of when a terminal string w matches a string γ. We will treat string concatenation as associative ($w_1w_2w_3$ is the same string as w_1 concatenated with w_2w_3 and w_1w_2 concatenated with w_3), and write ϵ for the empty string (so w, ϵw, and $w\epsilon$ are the same string).

$[r]X \longrightarrow \gamma_1 \ldots \gamma_n$	production r maps non-terminal X to string $\gamma_1 \ldots \gamma_n$
$w : \gamma$	terminal string w matches string γ

The second judgment is defined by two rules:

$$\frac{}{\mathtt{a} : \mathtt{a}}\ D_1 \qquad \frac{[r]X \longrightarrow \gamma_1 \ldots \gamma_n \quad w_1 : \gamma_1 \quad \ldots \quad w_n : \gamma_n}{w_1 \ldots w_n : X}\ D_2$$

The rule D_1 just says that the only way to match a terminal is with the string consisting of just that terminal. The second rule says that if a series of strings w_i match the individual components γ_i on the right-hand side of the grammar production $X \rightarrow \gamma_1 \ldots \gamma_n$, then the concatenation of those strings $w_1 \ldots w_n$ matches the nonterminal X.

When applied to our original example grammar (with the rules *emp*, *pars*, and *dup*), we can think of D_2 as being a *template* for three rules:

$$\frac{}{\epsilon : S}\ D_2(\mathsf{emp}) \qquad \frac{w_1 : \mathtt{[} \quad w_2 : S \quad w_3 : \mathtt{]}}{w_1w_2w_3 : S}\ D_2(\mathsf{pars}) \qquad \frac{w_1 : S \quad w_2 : S}{w_1w_2 : S}\ D_2(\mathsf{dup})$$

If we inspect the second rule above, we can further observe that the first premise of D_2, $(w_1 : \mathtt{[})$, will always be satisfied by the D_1 rule and can't possibly be satisfied by the D_2 rule: `[` is a terminal, and X only matches nonterminals. This observation allows us to simplify $D_2(\mathsf{pars})$ even further:

$$\frac{}{\epsilon : S}\ D_2(\mathsf{emp}) \qquad \frac{w_2 : S}{\mathtt{[}w_2\mathtt{]} : S}\ D_2(\mathsf{pars}) \qquad \frac{w_1 : S \quad w_2 : S}{w_1w_2 : S}\ D_2(\mathsf{dup})$$

Using this specialized version of the rules, our initial parsing example is now an upside-down version of the parse tree we used as an example in Section 3.

$$\cfrac{\cfrac{\cfrac{}{\epsilon : S}\ D_2(\mathsf{emp}) \quad \cfrac{\cfrac{}{\epsilon : S}\ D_2(\mathsf{emp}) \quad \cfrac{}{\epsilon : S}\ D_2(\mathsf{emp})}{\mathtt{[]} : S}\ D_2(\mathsf{pars}) \quad \cfrac{}{\epsilon : S}\ D_2(\mathsf{emp})}{\mathtt{[]} : S}\ D_2(\mathsf{pars}) \qquad \cfrac{\cfrac{\cfrac{}{\epsilon : S}\ D_2(\mathsf{emp}) \quad \cfrac{}{\epsilon : S}\ D_2(\mathsf{emp})}{\mathtt{[]} : S}\ D_2(\mathsf{pars}) \quad \cfrac{}{\epsilon : S}\ D_2(\mathsf{emp})}{\mathtt{[]} : S}\ D_2(\mathsf{dup})}{\mathtt{[]} : S}\ D_2(\mathsf{pars}) \qquad \cfrac{\cfrac{\cfrac{}{\epsilon : S}\ D_2(\mathsf{emp}) \quad \cfrac{}{\epsilon : S}\ D_2(\mathsf{emp})}{\mathtt{[]} : S}\ D_2(\mathsf{pars}) \quad \cfrac{\cfrac{\cfrac{}{\epsilon : S}\ D_2(\mathsf{emp}) \quad \cfrac{}{\epsilon : S}\ D_2(\mathsf{emp})}{\mathtt{[]} : S}\ D_2(\mathsf{pars}) \quad \cfrac{}{\epsilon : S}\ D_2(\mathsf{emp})}{\mathtt{[]} : S}\ D_2(\mathsf{dup})}{\mathtt{[]} : S}\ D_2(\mathsf{pars})}{\mathtt{[]} : S}\ D_2(\mathsf{dup})$$

$$\mathtt{[]}\,\mathtt{[]}\,\mathtt{[]}\,\mathtt{[]} : S$$

6 CYK Parsing

The rules above already give us an algorithm for parsing! Assume we are given a grammar with start symbol S and a terminal string w_0. Start with a database 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). One way of managing this constraint would be specifying strings by their starting and ending locations in the original string. This makes it readily apparent that the string $w_0[i, j)$ and $w_0[j, k)$ can be concatenated to the string $w_0[i, k)$.

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 $n = |w|$. If our grammar productions only have one or two nonterminals on the right-hand side of the arrow, then the algorithm we have just presented is an abstract form of the Cocke-Younger-Kasami (CYK) parsing algorithm invented in the 1960s. (It is always possible to automatically rewrite a grammar so that it can be used for CYK.)

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 case is when a grammar production has two nonterminals to the right of the arrow:

$$\frac{[r]X \to \gamma_1\gamma_2 \quad w_1 : \gamma_1 \quad w_2 : \gamma_2}{w_1\, w_2 : X}\ D_2$$

There are $O(n^2)$ substrings, so there are $O(n^2)$ ways to match the middle premise $w_1 : \gamma_1$ against the database of facts. Since $w_1\, w_2$ is also constrained to be a substring of w_0, there are only $O(n)$ ways to instantiate the final premise, since the left end of w_2 in the input string is determined, but not its right end. This yields a complexity of $O(n^2 * n) = O(n^3)$, which is also the complexity of traditional presentations of CYK.

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?

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.