Lecture Notes on Shift-Reduce Parsing

15-411: Compiler Design

Frank Pfenning, Rob Simmons, André Platzer, Jan Hoffmann

Lecture 09
February 13, 2024

1 Introduction

In this lecture we discuss *shift-reduce* parsing, which is the basis of most modern parser generator tools. Shift-reduce parsing is based on the idea of *predictive parsing with lookahead*. To improve upon the inefficiency of CYK parsing, we process strings in some fixed order: generally left-to-right, since most computer languages have been designed to be read and written in that direction. While processing the string in this fixed order, we attempt to correctly predict how to continue parsing the string based on the part of the string we have already seen and a finite amount of the string that we haven't yet considered (the *lookahead*). It is possible to do so successfully in a surprising number of cases, and when possible it allows us to have extremely efficient parsing algorithms.

Alternative presentations of the material in this lecture can be found in the textbook [App98, Chapter 3] and a paper by Shieber et al. [SSP95].

2 Deductive Shift-Reduce Parsing

Recall our original rules for deductive parsing of context-free grammars.

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

Interpreted as the CYK parsing algorithm, we allow these rules to be applied in any order or combination, so long as we only derive facts $w : \gamma$ where w is a substring of the original string w_0 we are trying to parse.

In shift-reduce parsing, we modify the form of the facts we use. Instead of $w : \gamma$, where w is a sequence of terminals and γ is a single terminal or nonterminal, we conclude facts of the form $w : \beta$, where $\beta = \gamma_1 \ldots \gamma_n$ is a (possibly empty) series of terminals and nonterminals, respectively. Then, we restrict both of our previous rules so that they only manipulate the rightmost element side of w or β, adding a new rule start as a base case:

$$\frac{}{\epsilon : \epsilon}\ \mathsf{start} \qquad\qquad \frac{w : \beta}{w\,\texttt{a} : \beta\,\texttt{a}}\ \mathsf{shift} \qquad\qquad \frac{\begin{array}{l}[r]X \longrightarrow \alpha \\ w : \beta\,\alpha\end{array}}{w : \beta\,X}\ \mathsf{reduce}(r)$$

Our previous restriction, that we only consider substrings of the sentence w_0 that we're trying to parse, remains in effect. This means that we know, when we conclude $w : \beta$, that w is a *prefix* of w_0. The requirement that we effectively scan the string from left to right means that we know exactly what any deduction of `[] [[] []]` $: S$ looks like this, where each of the omitted portions consist of zero or more applications of the rule reduce:

$$\begin{array}{c}
\dfrac{}{\epsilon : \epsilon}\ \mathsf{start} \\
\vdots \\
\dfrac{\epsilon : \beta_0}{\texttt{[} : \beta_0\,\texttt{[}}\ \mathsf{shift} \\
\vdots \\
\dfrac{\texttt{[} : \beta_1}{\texttt{[]} : \beta_1\,\texttt{]}}\ \mathsf{shift} \\
\vdots \\
\dfrac{\texttt{[]} : \beta_2}{\texttt{[] [} : \beta_2\,\texttt{[}}\ \mathsf{shift} \\
\vdots \\
\dfrac{\texttt{[] [} : \beta_3}{\texttt{[] [[} : \beta_3\,\texttt{]}}\ \mathsf{shift} \\
\vdots \\
\dfrac{\texttt{[] [[} : \beta_4}{\texttt{[] [[]} : \beta_4\,\texttt{]}}\ \mathsf{shift} \\
\vdots \\
\dfrac{\texttt{[] [[]} : \beta_5}{\texttt{[] [[] [} : \beta_5\,\texttt{[}}\ \mathsf{shift} \\
\vdots \\
\dfrac{\texttt{[] [[] [} : \beta_6}{\texttt{[] [[] []} : \beta_6\,\texttt{]}}\ \mathsf{shift} \\
\vdots \\
\dfrac{\texttt{[] [[] []} : \beta_7}{\texttt{[] [[] []]} : \beta_7\,\texttt{]}}\ \mathsf{shift} \\
\vdots \\
\texttt{[] [[] []]} : S
\end{array}$$

If we the manually-derived unambiguous grammar for matching parentheses discussed in the previous lecture:

$$\begin{array}{lrcl} [\textsf{emp}] & S & \longrightarrow & \epsilon \\ [\textsf{next}] & S & \longrightarrow & \texttt{[}\,S\,\texttt{]}\,S \end{array}$$

we can create a deductive parse tree, again treating the reduce rule as two rules specialized to the [emp] and [next] grammar productions:

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{\epsilon : \epsilon}\,\textsf{start}
}{\texttt{[} : \texttt{[}}\,\textsf{shift}
}{\texttt{[} : \texttt{[}S}\,\textsf{reduce(emp)}
}{\texttt{[]} : \texttt{[}S\texttt{]}}\,\textsf{shift}
}{\texttt{[] [} : \texttt{[}S\texttt{] [}}\,\textsf{shift}
}{\texttt{[] [[} : \texttt{[}S\texttt{] [[}}\,\textsf{shift}
}{\texttt{[] [[} : \texttt{[}S\texttt{] [[}S}\,\textsf{reduce(emp)}
}{\texttt{[] [[]} : \texttt{[}S\texttt{] [[}S\texttt{]}}\,\textsf{shift}
}{\texttt{[] [[] [} : \texttt{[}S\texttt{] [[}S\texttt{] [}}\,\textsf{shift}
}{\texttt{[] [[] [} : \texttt{[}S\texttt{] [[}S\texttt{] [}S}\,\textsf{reduce(emp)}
}{\texttt{[] [[] []} : \texttt{[}S\texttt{] [[}S\texttt{] [}S\texttt{]}}\,\textsf{shift}
}{\texttt{[] [[] []} : \texttt{[}S\texttt{] [[}S\texttt{] [}S\texttt{]}S}\,\textsf{reduce(emp)}
}{\texttt{[] [[] []} : \texttt{[}S\texttt{] [[}S\texttt{]}S}\,\textsf{reduce(next)}
}{\texttt{[] [[] []} : \texttt{[}S\texttt{] [}S}\,\textsf{reduce(next)}
}{\texttt{[] [[] []]} : \texttt{[}S\texttt{] [}S\texttt{]}}\,\textsf{shift}
}{\texttt{[] [[] []]} : \texttt{[}S\texttt{] [}S\texttt{]}S}\,\textsf{reduce(emp)}
}{\texttt{[] [[] []]} : \texttt{[}S\texttt{]}S}\,\textsf{reduce(next)}
}{\texttt{[] [[] []]} : S}\,\textsf{reduce(next)}
$$

Looking back at our general template, we can identify $\beta_0 = \epsilon$, $\beta_1 = \texttt{[}S$, $\beta_2 = \texttt{[}S\texttt{]}$, $\beta_3 = \texttt{[}S\texttt{] [}$, $\beta_4 = \texttt{[}S\texttt{] [[}S$, $\beta_5 = \texttt{[}S\texttt{] [[}S\texttt{]}$, $\beta_6 = \texttt{[}S\texttt{] [[}S\texttt{] [}S$, and $\beta_7 = \texttt{[}S\texttt{] [}S$. The advantage of shift-reduce parsing comes, in part, from the fact that not only is this deduction unambiguous as a proof tree, it's unambiguous as a series of applications of grammar productions. There is no other way to conclude $\texttt{[] [[] []]} : S$ according to our rules. The modified rules do not change the sentences that we can parse, but they force us to apply grammar productions in a specific order. In fact, we produce the *rightmost derivation*.

3 Predictive parsing

Only allowing rightmost derivations means that at most one sequence of shift and reduce rules for every given parse tree. If a grammar is unambiguous, this means that, as we apply a series of rules to try to derive $w : S$ from $\epsilon : \epsilon$, there is *at most one rule* we can apply that will lead us to success. The goal of predictive parsing is to always be able to pick the *correct* rule.

To move forward, we're going to reframe the problem a bit. Instead of talking about facts $w : \beta$ where w is a prefix of w_0, we're going to think about facts of the form $\beta \mid\mid w'$, where w' is the complement of w, the string where $ww' = w_0$. This makes the interpretation of the shift rule more natural, because we are shifting a character from left-hand side of the un-processed w' string onto the right-hand side of the processed w string. This re-interpretation of the rules gives us the following summary of the proof tree above:

```
              || [][[][]] shift  [
             [ || ][[][]]  reduce emp
            [S || ][[][]]  shift  ]
           [S] || [[][]]   shift  [
          [S][ || [][]]    shift  [
         [S][[ || ][]]     reduce emp
        [S][[S || ][]]     shift  ]
       [S][[S] || []]      shift  [
      [S][[S][ || ]]       reduce emp
     [S][[S][S || ]]       shift  ]
    [S][[S][S] || ]        reduce emp
   [S][[S][S]S || ]        reduce next
      [S][[S]S || ]        reduce next
         [S][S || ]        shift
        [S][S] ||          reduce emp
       [S][S]S ||          reduce next
          [S]S ||          reduce next
             S ||
```

If we can successfully predict what the next step should be at every point, then we can implement this proof search with a stack holding the terminals and nonterminals (on the left) and a queue or array index tracking the unprocessed tokens (on the right).

What would we need to know how to always make the right decision for the grammar above? It's *not* enough to just look at the contents of the stack. We can prove this by giving a counterexample: in two different states, both starting with the same stack but with different queues of un-shifted terminals, we can see that the correct action in one case is to reduce by emp and the correct action in the other case is to shift a [.

```
  [[S] || ] reduce emp         [[S] || []] shift [
 [[S]S || ] reduce next       [[S][ || ]]  reduce emp
    [S || ] shift ]     vs   [[S][S || ]]  shift ]
   [S] || reduce emp        [[S][S] || ]   reduce emp
  [S]S || reduce next      [[S][S]S || ]   reduce next
     S ||                     [[S]S || ]   reduce next
                                 [S || ]   shift ]
                                [S] ||     reduce emp
                               [S]S ||     reduce next
                                  S ||
```

Therefore, in addition to inspecting the structure of the stack, we will need to use extra information to decide what to do. In particular, we will allow ourselves to use the *first* unshifted token when we decide which action to take next. This means that our shift-reduce algorithm needs a *lookahead* of 1.

At any step while parsing, we can (potentially) shift a `[`, shift a `]`, reduce with emp, or reduce with next. We will begin constructing a *parse table*, where the columns correspond to the next unshifted token and the rows correspond to patterns that we match against the stack β. The following table allows us to always unambiguously make decisions about what to do next when parsing our grammar of parentheses:

$\beta \setminus$ a	`[`	`]`	`$`
ϵ	shift	error	reduce(emp)
β`[`	shift	reduce(emp)	error
β`]`	shift	reduce(emp)	reduce(emp)
β`[`S	error	shift	error
β`]`S	error	reduce(next)	reduce(next)
ϵS	error	error	accept(S)

4 Generating a Parse Table

A parser generator will take a context free grammar and construct a parse table for use. If the grammar is an unambiguous LR(1) grammar then the process will succeed. Otherwise, it will result in shift-reduce and reduce-reduce conflicts, which we discuss in the following section. For now, assume that we have an unambiguous LR(1) grammar.

To automatically generate the parse table, we first determine all the "spots" (the different situation) in the derivation at which we could be. To this end, we consider the right-hand sides of the productions in the grammar.

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

For the production next, there the following possibilities:

- We just have processed `[` and are now expecting an input string that reduces to S`]`S.
- We have processed `[`S and are now expecting a string that reduces to `]`S.
- We have processed `[`S`]` and are now expecting a string that reduces to S.
- We have processed `[`S`]`S and do not expect anything.

For the production emp, we only have the case in which we produced something (it doesn't matter what) and expect nothing. In other words, this production can always be applied. This information can can be generated mechanically from the productions.

From this information, we can generate the set of potentially "good" stack states. These are the stacks from which we can still hope to reduce them to the start symbol S. We always include ϵ and S. If these states are not good then the language of the grammar is empty. Additionally, we add all other possible states that we can still reduce with some production:

$$\beta\,\texttt{[} \quad \beta\,\texttt{[}S \quad \beta\,\texttt{[}S\texttt{]} \quad \beta\,\texttt{[}S\texttt{]}\,S$$

This gives us the columns of the table. The lines are just the terminal symbols in our alphabet and the end-of-file marker. Next, we have to determine what to put into the cells of the table. To this end, we first compute the prefix set for every non-terminal symbol. In our example, the only non-terminal symbol is S. We can see right away that if $S \to^* w$ then $w = \texttt{[}w'$ for some string w'. That's why the prefix set for S is $\{\texttt{[}\}$. To compute the prefix set, we usually have to do more iterations. For example if we have a rule like $X \to Y Z$ then the prefix set of X includes the prefix set of Y, and even the prefix set of Z if $Y \to^* \epsilon$.

Now we can fill the parse table. However, we add another column *next?* that tells us what the remainder of the input should reduce to in a successful parse. We consider the next input token. If it is in the prefix set of the first symbol in next the we do a shift. Otherwise we go for the *reduce*. However, if this would lead to a stack pattern that does not appear in the table we put *error* in the cell. For example, in the first line `[` is in the prefix set of S so we chose shift. The other two tokens are not in the prefix set of S and reduce(emp) would lead to the stack in the last row. So we chose reduce(emp) there.

$\beta \setminus$ `a`	$next?$	`[`	`]`	`$`
ϵ	S	shift	reduce(emp)	reduce(emp)
β`[`	S`]`S	shift	reduce(emp)	reduce(emp)
β`[`S	`]`S	error	shift	error
β`[`S`]`	S	shift	reduce(emp)	reduce(emp)
β`[`S`]`S	ϵ	error	reduce(next)	reduce(next)
ϵS	ϵ	error	error	accept(S)

The second row is similar. However, in the third row, `[` is not in the prefix set of `]` and reduce(emp) would lead to the stack state β `[`SS, which does not appear in the table. So we chose error.

The resulting parse table is correct but can be further optimized to arrive the the one that we constructed by hand. In particular, we sometimes continue to reduce even though it is already clear that we will reach an error state. However, since the time is at most linear, this is not a terrible efficiency problem.

5 Parsing Ambiguous Grammars

We can use shift-reduce parsing to parse ambiguous grammars, but our parse tables will no longer be able to give us unambiguous guidance about what the next step to take is. We'll consider a subset of our ambiguous grammar of arithmetic again:

$$\begin{array}{llll}
[\mathsf{plus}] & E & \longrightarrow & E + E \\
[\mathsf{times}] & E & \longrightarrow & E * E \\
[\mathsf{number}] & E & \longrightarrow & num \\
[\mathsf{parens}] & E & \longrightarrow & (\ E\ )
\end{array}$$

If we begin parsing $num + num + num$, then our first several steps are unambiguous:

```
          || num + num + num   shift num
      num || + num + num       reduce number
        E || + num + num       shift +
      E + || num + num         shift num
  E + num || + num             reduce number
    E + E || + num             ???
          ...
```

At this point, we have a real decision. We can either reduce by the plus rule or shift the next + token. Either way, we will be able to complete the derivation:

```
    E + E || + num                E + E || + num
        E || + num            E + E + || num
      E + || num     vs   E + E + num ||
  E + num ||                E + E + E ||
    E + E ||                    E + E ||
        E ||                        E ||
```

these two examples are a counterexample proving that we cannot use shift-reduce parsing to unambiguously parse this grammar. (That's not surprising: it's an ambiguous grammar!)

Cases where we can make different decisions and still successfully parse the string are called *conflicts*. This is a *shift/reduce* conflict, because our parse table has a single entry that could contain either the rule to shift + or the rule to reduce with the plus production.

Resolving the shift/reduce conflict in favor of shifting causes addition to be right-associative, so we should instead resolve the conflict, in this case, by reducing, because we treat addition as left-associative. Rather than rewriting the grammar to avoid these ambiguities, we can supplement our context free grammar with precedence and associativity information, and the parser generator can use this supplementary information to avoid some conflicts.

With supplementary information about precedence and associativity, we can construct an unambiguous parsing table. As before, we assume that a special end-of-file token $ has been added to the end of the input string. When the parsing goal has the form $\alpha, \beta \mid \mathtt{a}\, w$ where β is a prefix substring of the grammar, we look up β in the left-most column and a in the top row to find the action to take. The non-terminal $\epsilon\, E$ in the last line is a special case in that E must be the only thing on the stack. In that case we can accept if the next token is $ because we know that $ can only be the last token of the input string.

$\beta \setminus \mathtt{a}$	+	*	num	(	)	$
E + E	reduce(plus) (+ left assoc.)	shift (+ $<$ *)	error	error	reduce(plus)	reduce(plus)
E * E	reduce(times) (+ $<$ *)	reduce(times) (* left assoc.)	error	error	reduce(times)	reduce(times)
num	reduce(number)	reduce(number)	error	error	reduce(number)	reduce(number)
(E)	reduce(parens)	reduce(parens)	error	error	reduce(parens)	reduce(parens)
E +	error	error	shift	shift	error	error
E *	error	error	shift	shift	error	error
(E	shift	shift	error	error	shift	error
(	error	error	shift	shift	error	error
ϵ	error	error	shift	shift	error	error
$\epsilon\, E$	shift	shift	error	error	error	accept(E)

We can see that the bare grammar has four shift/reduce conflicts, while all other actions (including errors) are uniquely determined. These conflicts arise when E + E or E * E is on the stack and either + or * is the first character in the remaining input string. It is called a shift/reduce conflict, because either a shift action or a reduce action could lead to a valid parse. Here, we have decided to resolve the conflicts by giving a precedence to the operators and declaring both of them to be left-associative.

It is also possible to have reduce/reduce conflicts, if more than one reduction could be applied in a given situation, but it does not happen in this grammar.

Parser generators will generally issue an error or warning when they detect a shift/reduce or reduce/reduce conflict. For many parser generators, the default behavior of a shift/reduce conflict is to shift, and for a reduce/reduce conflict to apply the textually first production in the grammar. Particularly the latter is rarely what is desired, so we strongly recommend rewriting the grammar to eliminate any conflicts.

One interesting special case is the situation in a language where the else-clause of a conditional is optional. For example, one might write (among other productions)

$$\begin{array}{lcl} E & \longrightarrow & \mathsf{if}\ E\ \mathsf{then}\ E \\ E & \longrightarrow & \mathsf{if}\ E\ \mathsf{then}\ E\ \mathsf{else}\ E \end{array}$$

Now a statement

```
if b then if c then x else y
```

is ambiguous because it would be read as

```
if b then (if c then x) else y
```

or

```
if b then (if c then x else y)
```

In a shift/reduce parser, typically the default action for a shift/reduce conflict is to shift to extend the current parse as much as possible. This means that the above grammar in a tool such as ML-Yacc will parse the ambiguous statement into the second form, that is, the `else` is matched with the most recent unmatched `if`. This is consistent with language such as C (or C0, the language used in this course), so we can tolerate the above shift/reduce conflict, if you wish, instead of rewriting the grammar to make it unambiguous.

We can also think about how to rewrite the grammar so it is unambiguous. What we have to do is rule out the parse

```
if b then (if c then x) else y
```

In other words, the *then* clause of a conditional should be balanced in terms of if-then-else and not have something that is just an if-then without an *else* clause.

$$\begin{array}{lcl} E & \longrightarrow & \mathsf{if}\ E\ \mathsf{then}\ E \\ E & \longrightarrow & \mathsf{if}\ E\ \mathsf{then}\ E'\ \mathsf{else}\ E \\ \\ E' & \longrightarrow & \mathsf{if}\ E\ \mathsf{then}\ E'\ \mathsf{else}\ E \\ E' & \longrightarrow & \ldots \end{array}$$

We would also have to repeat all the other clauses for E, or refactor the grammar so the other productions of E can be shared with E'.

6 Adapting Grammars for Shift-Reduce Parsing

The set of languages that we can parse with shift-reduce parsers that have lookahead 1 is called LR(1). But even though a language may be describable with an LR(1) grammar, it's not necessarily the case that every grammar for an LR(1) language can be parsed with a shift-reduce parser. To understand the difference, we'll look at two different grammars for a language Z that can be described by a regular expression: $\mathtt{b}^* \cdot (\mathtt{c} + \mathtt{d})$.

$$
\begin{array}{llcl@{\qquad\qquad}llcl}
[\mathrm{xz}] & S & \longrightarrow & X\mathtt{c} & [\mathrm{cz}] & S & \longrightarrow & C\mathtt{c} \\
[\mathrm{yz}] & S & \longrightarrow & Y\mathtt{d} & [\mathrm{dz}] & S & \longrightarrow & D\mathtt{d} \\
[\mathrm{x0}] & X & \longrightarrow & \epsilon & [\mathrm{c0}] & C & \longrightarrow & \epsilon \\
[\mathrm{x1}] & X & \longrightarrow & \mathtt{b}X & [\mathrm{c1}] & C & \longrightarrow & C\mathtt{b} \\
[\mathrm{y0}] & Y & \longrightarrow & \epsilon & [\mathrm{d0}] & D & \longrightarrow & \epsilon \\
[\mathrm{y1}] & Y & \longrightarrow & \mathtt{b}Y & [\mathrm{d1}] & D & \longrightarrow & D\mathtt{b}
\end{array}
$$

Both grammars are completely unambiguous on their own, but only the first can be correctly parsed by a shift-reduce parser that has lookahead of 1.

```
     || bbbc  shift b
   b || bbc   shift b
  bb || bc    shift b
 bbb || c     reduce x0
bbbX || c     reduce x1
 bbX || c     reduce x1
  bX || c     reduce x1
   X || c     shift c
  Xc ||       reduce xz
   Z ||
```

If we want to parse the same string with the second grammar, we must immediately reduce, because we will never be able to handle the b on the stack unless it is preceded by a C or a D.

```
     || bbbc  reduce c0         || bbd  reduce d0
   C || bbbc  shift b         D || bbd  shift b
  Cb || bbc   reduce c1      Db || bd   reduce d1
   C || bbc   shift b         D || bd   shift b
  Cb || bc    reduce c1      Db || d    reduce d1
   C || bc    shift b         D || d    shift d
  Cb || c     reduce c1      Dd ||      reduce dz
   C || c     shift c         Z ||
  Cc ||       reduce cz
   Z ||
```

Both examples have the same initial stack ϵ, and the same first token `b`, but one of them can only be parsed by reducing by `c0` and the other can only be parsed by reducing by `d0`. This proves that there is a reduce/reduce conflict for a LR(1) parser trying to parse this grammar. Even though the grammar is unambiguous, to parse it correctly, we'd need *arbitrary lookahead* – we'd need to look over an arbitrary number of `b` tokens to find whether they were followed by a `c` or a `d`.

Despite the fact that a parser-table-based algorithm cannot unambiguously parse this grammar, the grammar is not unusuable. Using the deduction rules from the beginning, we could consider possible parses as we worked through the string. This idea is the basis of some GLR (Generalized LR) parser generators, but the cost of increased generality is that there are fewer guarantees about efficiency.

Questions

1. What happens if we remove the ϵ from the last entry in the LR parser table? Aren't ϵ's irrelevant and can always be removed?
2. What makes `x*y;` difficult to parse in C and C0? Discuss some possible solutions, once you have identified a problem?
3. Give a very simple example of a grammar with a shift/reduce conflict.
4. Give an example of a grammar with a shift/reduce conflict that occurs in programming language parsing and is not easily resolved using associativity or precedence of arithmetic operators.
5. Give a very simple example of a grammar with a reduce/reduce conflict.
6. Give an example of a grammar with a reduce/reduce conflict that occurs in programming language parsing and is not easily resolved.
7. In the reduce rule, we have used a number of symbols on the top of the stack and the lookahead to decide what to do. But isn't a stack something where we can only read one symbol off of the top? Does it make a difference in expressive power if we allow decisions to depend on 1 or on 10 symbols on the top of the stack? Does it make a difference in expressive power if we allow 1 or arbitrarily many symbols from the top of the stack for the decision?
8. What's wrong with this grammar that was meant to define a program P as a sequence of statements S by $P \to S \mid P; P$

References

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

[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.