Lecture Notes on
Top-Down Predictive LL Parsing

15-411: Compiler Design
Frank Pfenning[*]

Lecture 8

1 Introduction

In this lecture we discuss a parsing algorithm that traverses the input string from left to right giving a left-most derivation and makes a decision on which grammar production to use based on the first character/token of the input string. This parsing algorithm is called LL(1). If that were ambiguous, the grammar would have to be rewritten to fall into this class, which is not always possible. Most hand-written parsers are recursive descent parsers, which follow the LL(1) principles.

Alternative presentations of the material in this lecture can be found in a paper by Shieber et al. [SSP95]. The textbook [App98, Chapter 3] covers parsing. For more detailed background on parsing, also see [WM95].

2 LL(1) Parsing

We have seen in the previous section, that the general idea of recursive descent parsing without restrictions forces us to non-deterministically choose between several productions which might be applied and potentially backtrack if parsing gets stuck after a choice, or even loop (if the grammar is left-recursive). Backtracking is not only potentially very inefficient, but it makes it difficult to produce good error messages in case the string is not grammatically well-formed. Say we try three different ways to parse a given input and all fail. How could we say which of these is the source

[*]With edits by André Platzer

of the error? How could we report an error location? This is compounded because nested choices multiply the number of possibilities. We therefore have to look for ways to disambiguate the choices, making parsing more efficient and syntax errors easier to locate.

One way is to *require* of the grammar that at each potential choice point we can look at the next input token and based on that token decide which production to take. This is called *1 token lookahead*, and grammars that satisfy this restriction are called *LL(1)* grammars. Here, the first *L* stands for *L*eft-to-right reading of the input; the second *L* stands for *L*eftmost parse (which a recursive descent parser generates) and *1* stands for *1 token lookahead*. We can also define LL(2), LL(3), etc. But for higher k, LL(k) parsers become fairly inefficient and are used less frequently. The LL parser generators (or SLL parser generators which are a simplification of LL) themselves, however, are much more efficient than the LR parser generators.

Since we are restricting ourselves to parsing by a left-to-right traversal of the input string, we will consider only tails, or suffixes of the input strings, and also of the strings in the grammar, when we restrict our inference rules. Those suffixes capture what still has to be parsed or worked on. For short, we will say γ is a *suffix substring* of the grammar, or w is a suffix substring of the input string w_0. For example, in the grammar

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

the only suffix substrings are ϵ, $\texttt{[}\ S\ \texttt{]}$, $S\ \texttt{]}$, $\texttt{]}$, S, and $S\,S$, but not the prefix $\texttt{[}\ S$.

Firsts & Nulls The main idea behind LL(1)-parsing is that we restrict our attention to grammars where we can use a 1 token lookahead to disambiguate which grammar production to use. That is, by peeking at the next input token, we want to know which grammar production helps us. The first thing we want to know for this is what the tokens are that could be generated from a sequence β of terminals and non-terminals at all. So imagine β describes a shape of how far we have come with parsing and what we expect for the rest of the input. It could be something like $S]$. We begin by defining two kinds of predicates (later we will have occasion to add a third), where β is either a non-terminal or suffix substring of the grammar. The predicate $\mathsf{first}(\beta, \mathtt{a})$ captures if token $\mathtt{a}$ can be the first token occurring in a word that matches the expression β. What we also need to know is if

β could possibly match the empty word ϵ, because then the first token of $\beta\gamma$ could actually come from the first token of γ. This is what we use the predicate $\mathsf{null}(\beta)$ for.

$\mathsf{first}(\beta, \mathtt{a})$	Token $\mathtt{a}$ can be first in any string produced by β
$\mathsf{null}(\beta)$	String β can produce the empty string ϵ

These predicates must be computed entirely statically, by an analysis of the grammar before any concrete string is ever parsed. This is because we want to be able to tell if the parser can do its work properly with 1 token look-ahead regardless of the actual input strings it will later have to parse. We want the parser generator to tell us right away if it will work on all input. We do not want to wait till runtime to tell us that it doesn't know what to do with some particular input.

We define the relation $\mathsf{first}(\beta, \mathtt{a})$ by the following rules.

$$\frac{}{\mathsf{first}(\mathtt{a}\,\beta, \mathtt{a})}\ F_1$$

This rule *seeds* the first predicate with the knowledge that parse strings starting with a token $\mathtt{a}$ always start with that token $\mathtt{a}$, no matter what β yields. Then is it propagated to other strings appearing in the grammar by the following three rules.

$$\frac{\mathsf{first}(X, \mathtt{a})}{\mathsf{first}(X\,\beta, \mathtt{a})}\ F_2 \qquad \frac{\mathsf{null}(X) \quad \mathsf{first}(\beta, \mathtt{a})}{\mathsf{first}(X\,\beta, \mathtt{a})}\ F_3 \qquad \frac{\begin{array}{c}[r]X \longrightarrow \gamma \\ \mathsf{first}(\gamma, \mathtt{a})\end{array}}{\mathsf{first}(X, \mathtt{a})}\ F_4(r)$$

Rule F_2 says that if X can start with $\mathtt{a}$, then so can $X\beta$. Rule F_3 says that if X can produce ϵ and β can start with $\mathtt{a}$, then $X\beta$ can start with $\mathtt{a}$ as well. Rule $F_4(r)$ captures that X can start with whatever any of the right-hand sides γ of its productions $[r]X \to \gamma$ can start with. Even though ϵ may be technically a suffix substring of every grammar, it can never arise in the first argument of the first predicate, because it is not a proper token, so we do not need any information about it. The auxiliary predicate null is also easily defined.

$$\frac{}{\mathsf{null}(\epsilon)}\ N_1 \qquad \frac{\mathsf{null}(X) \quad \mathsf{null}(\beta)}{\mathsf{null}(X\,\beta)}\ N_2 \qquad \frac{\begin{array}{c}[r]X \longrightarrow \gamma \\ \mathsf{null}(\gamma)\end{array}}{\mathsf{null}(X)}\ N_3$$

N_1 expresses that ϵ can produce ϵ – surprise. That $X\beta$ can produce ϵ if both X and β can (N_2). And that X can produce ϵ if one of its productions has a right-hand side γ hat can (N_3).

We can run these rules to saturation because there are only $O(|G|)$ possible strings in the first argument to both of these predicates, and at most the number of possible terminal symbols in the grammar, $O(|\Sigma|)$, in the second argument. Naive counting the number of prefix firings (see [GM02]) gives a complexity bound of $O(|G| \times |\Xi| \times |\Sigma|)$ where $|\Xi|$ is the number of non-terminals in the grammar. Since usually the number of symbols is a small constant, this is roughly equivalent to $\approx O(|G|)$ and so is reasonably efficient. Moreover, it only happens once, before any parsing takes place.

Constructing LL(1) Parsing Rules Next, we modify the rules for recursive descent parsing from the last lecture to take these restrictions into account. The first two compare rules stay the same.

$$\frac{}{\epsilon : \epsilon}\,L_1 \qquad \frac{w : \gamma}{\mathtt{a}\,w : \mathtt{a}\,\gamma}\,L_2$$

The third generate rule,

$$\frac{\begin{array}{l}[r]X \longrightarrow \beta \\ w : \beta\,\gamma\end{array}}{w : X\,\gamma}\,L_3(r)$$

is split into two, each of which has an additional precondition on when to use it:

$$\frac{\begin{array}{l}[r]X \longrightarrow \beta \\ \mathsf{first}(\beta, \mathtt{a}) \\ \mathtt{a}\,w : \beta\,\gamma\end{array}}{\mathtt{a}\,w : X\,\gamma}\,L_3' \qquad \frac{\begin{array}{l}[r]X \longrightarrow \beta \\ \mathsf{null}(\beta) \\ w : \beta\,\gamma\end{array}}{w : X\,\gamma}\,L_3''?$$

We would like to say that a grammar is LL(1) if the additional preconditions in these last two rules make all choices unambiguous when an arbitrary non-terminal X is matched against a string starting with an arbitrary terminal $\mathtt{a}$. Unfortunately, this does not quite work yet in the presence non-terminals that can rewrite to ϵ, because the second rule above does not even look at the input string. This nondeterministic rule cannot possibly ensure the appropriate parse without looking at the input. To disambiguate, we need to know what tokens could follow X. Only if the first token of w could follow X would we want to use a production for X that would make it vanish.

Follows We thus need to know which tokens could follow X in order to know if it would make sense to skip over it by using one of its productions that produces ϵ. For that, we define one additional predicate, again on suffix strings in the grammar and non-terminals.

$\mathsf{follow}(\beta, \mathtt{a})$ Token $\mathtt{a}$ can follow string β in a valid string

See Figure 1 for an illustration of where first and follow come from in a parse tree. The set of all $\mathtt{a}$ for which $\mathsf{first}(X, \mathtt{a})$ characterizes what tokens X itself can start with. The set of all $\mathtt{a}$ for which $\mathsf{follow}(X, \mathtt{a})$ characterizes what tokens can follow X.

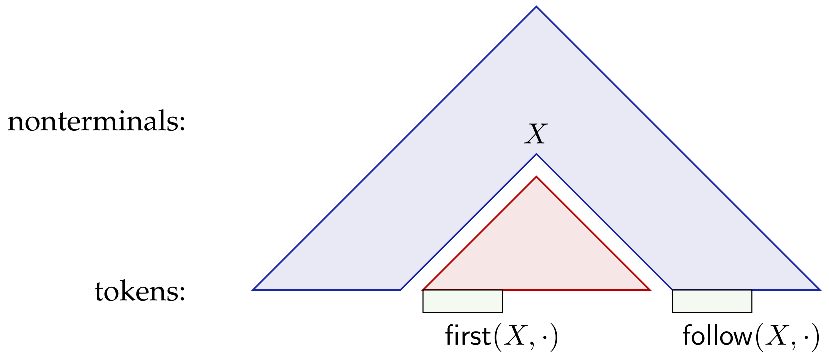

Figure 1: Illustration of first and follow sets of non-terminal X in a parse tree in which X occurs

We seed this follows relation with the rules

$$\frac{X\gamma\ \mathit{suffix} \qquad \mathsf{first}(\gamma, \mathtt{a})}{\mathsf{follow}(X, \mathtt{a})}\ W_1$$

Here, $X\gamma\ \ \mathit{suffix}$ means that the string $X\gamma$ appears as a suffix substring on the right-hand side of some production, because then we would want to know if we should choose the productions for X that ultimately produce ϵ, because the lookahead token comes from γ. Whatever can come first in γ can follow X if $X\gamma$ occurs in the right-hand sides of some production. We then propagate this information applying the following rules from premises to conclusion until saturation is reached.

$$\frac{\mathsf{follow}(\mathtt{b}\,\gamma, \mathtt{a})}{\mathsf{follow}(\gamma, \mathtt{a})}\ W_2 \qquad \frac{\mathsf{follow}(X\,\gamma, \mathtt{a})}{\mathsf{follow}(\gamma, \mathtt{a})}\ W_3 \qquad \frac{\mathsf{follow}(X\,\gamma, \mathtt{a}) \qquad \mathsf{null}(\gamma)}{\mathsf{follow}(X, \mathtt{a})}\ W_4 \qquad \frac{[r]X \longrightarrow \gamma \qquad \mathsf{follow}(X, \mathtt{a})}{\mathsf{follow}(\gamma, \mathtt{a})}\ W_5$$

Rule W_2 says that whatever can follow $\texttt{b}\gamma$ can also follow the last part γ. Similarly, rule W_3 says that whatever can follow $X\gamma$ can also follow the last part γ. Rule W_4 says that, in addition, everything that can follow $X\gamma$ can follow X itself if γ can produce ϵ (otherwise the follows of $X\gamma$ are no follows of X, because there is always at least one token from γ in between). Rule W_5 says that whatever follows a nonterminal X can also follow the right-hand side γ of each production $[r]X \to \gamma$.

The first argument of follow should remain a non-empty suffix or a non-terminal here, because we are not interested in what could follow ϵ.

Final LL(1) Parser Rules Now we can refine the proposed L_3'' rule from above into one which is no longer ambiguous (for LL(1) grammars).

$$\frac{\begin{array}{l}[r]X \longrightarrow \beta \\ \mathsf{first}(\beta, \texttt{a}) \\ \texttt{a}\,w : \beta\,\gamma\end{array}}{\texttt{a}\,w : X\,\gamma}\ L_3' \qquad \frac{\begin{array}{l}[r]X \longrightarrow \beta \\ \mathsf{null}(\beta) \\ \mathsf{follow}(X, \texttt{a}) \\ \texttt{a}\,w : \beta\,\gamma\end{array}}{\texttt{a}\,w : X\,\gamma}\ L_3''$$

We avoid creating an explicit rule to treat the empty input string by appending a special end-of-file symbol $\texttt{\$}$ symbol at the end before starting the parsing process. We repeat the remaining compare rules for completeness.

$$\frac{}{\epsilon : \epsilon}\ L_1 \qquad \frac{w : \gamma}{\texttt{a}\,w : \texttt{a}\,\gamma}\ L_2$$

These rules are interpreted as a parser by proof search, applying them from the conclusion to the premise. We say the grammar is LL(1) if for any goal $w : \gamma$ at most one rule applies.

1. If X cannot derive ϵ, this amounts to checking that there is at most one production $X \longrightarrow \beta$ such that $\mathsf{first}(\beta, \texttt{a})$.

2. Otherwise there is a *first/first conflict* between the two ambiguous productions $X \longrightarrow \beta$ and $X \longrightarrow \gamma$ that share a $\mathsf{first}(\beta, \texttt{a})$ and $\mathsf{first}(\gamma, \texttt{a})$.

For nullable non-terminals there are more complicated extra conditions, because it also depends on the follow sets. The conflicts can still easily be read off from the rules. First/first conflicts stay the same.
In addition,

3. There is a *first/follow conflict* if we cannot always decide between L_3' and L_3''. This happens if there is a token $\mathtt{a}$ with $\mathsf{first}(\beta, \mathtt{a})$ for a production $X \longrightarrow \beta$ that would trigger L_3', but that token also satisfies $\mathsf{follow}(X, \mathtt{a})$ with a nonterminal X that is nullable and trigger L_3''.

Example We now use a very simple grammar to illustrate these rules. We have transformed it in the way indicated above, by assuming a special token $\mathtt{\$}$ to indicate the end of the input string.

$$\begin{array}{llll}
[\mathsf{start}] & S & \longrightarrow & S'\ \mathtt{\$} \\
[\mathsf{emp}] & S' & \longrightarrow & \epsilon \\
[\mathsf{pars}] & S' & \longrightarrow & \mathtt{[}\ S'\ \mathtt{]}
\end{array}$$

This grammar generates all strings starting with an arbitrary number of opening parentheses followed by the same number of closing parentheses and an end-of-string marker, i.e., the language $\mathtt{[}^n\ \mathtt{]}^n \mathtt{\$}$.

We have:

$\mathsf{null}(\epsilon)$	N_1
$\mathsf{null}(S')$	N_3
$\mathsf{first}(\mathtt{[}\ S'\ \mathtt{]}, \mathtt{[})$	F_1
$\mathsf{first}(\mathtt{]}, \mathtt{]})$	F_1
$\mathsf{first}(S'\ \mathtt{]}, \mathtt{]})$	F_3
$\mathsf{first}(S', \mathtt{[})$	F_4 [pars]
$\mathsf{first}(S'\ \mathtt{]}, \mathtt{[})$	F_2
$\mathsf{first}(\mathtt{\$}, \mathtt{\$})$	F_1
$\mathsf{first}(S'\ \mathtt{\$}, \mathtt{\$})$	F_3
$\mathsf{first}(S'\ \mathtt{\$}, \mathtt{[})$	F_2
$\mathsf{first}(S, \mathtt{\$})$	F_4 [start]
$\mathsf{first}(S, \mathtt{[})$	F_4 [start]
$\mathsf{follow}(S', \mathtt{\$})$	W_1
$\mathsf{follow}(S', \mathtt{]})$	W_1
$\mathsf{follow}(\mathtt{[}\ S'\ \mathtt{]}, \mathtt{\$})$	W_5
$\mathsf{follow}(\mathtt{[}\ S'\ \mathtt{]}, \mathtt{]})$	W_5
$\mathsf{follow}(S'\ \mathtt{]}, \mathtt{\$})$	W_3
$\mathsf{follow}(S'\ \mathtt{]}, \mathtt{]})$	W_3
$\mathsf{follow}(\mathtt{]}, \mathtt{\$})$	W_4
$\mathsf{follow}(\mathtt{]}, \mathtt{]})$	W_4

3 Parser Generation

Parser generation is now a very simple process. Once we have computed the null, first, and follow predicates by saturation from a given grammar, we specialize the inference rules $L'_3(r)$ and $L''_3(r)$ by matching the first two and three premises against grammar productions and saturated null, first, and follow database. This is essentially partial evaluation on inference rules. In this case, this leads to the following specialized rules (repeating once again the two initial rules).

$$\frac{}{\epsilon : \epsilon}\ L_1 \qquad \frac{w : \gamma}{\mathtt{a}\, w : \mathtt{a}\, \gamma}\ L_2$$

$$\frac{\mathtt{[}\ w : S'\ \mathtt{\$}\ \gamma}{\mathtt{[}\ w : S\ \gamma}\ L'_3(\mathsf{start}) \qquad \frac{\mathtt{\$}\ w : S'\ \mathtt{\$}\ \gamma}{\mathtt{\$}\ w : S\ \gamma}\ L'_3(\mathsf{start})$$

$$\frac{\mathtt{[}\ w :\ \mathtt{[}\ S'\ \mathtt{]}\ \gamma}{\mathtt{[}\ w : S'\ \gamma}\ L'_3(\mathsf{pars}) \qquad \frac{\mathtt{]}\ w : \gamma}{\mathtt{]}\ w : S'\ \gamma}\ L''_3(\mathsf{emp}) \qquad \frac{\mathtt{\$}\ w : \gamma}{\mathtt{\$}\ w : S'\ \gamma}\ L''_3(\mathsf{emp})$$

Recall that these rules are applied from the bottom-up, starting with the goal $w_0\ \mathtt{\$}\ :\ S$, where w_0 is the input string. It is easy to observe by pattern matching that each of these rules are mutually exclusive: if one of the rules applies, none of the other rules apply. Moreover, each rule except for L_1 (which accepts) has exactly one premise, so the input string is traversed linearly from left-to-right, without backtracking. When none of the rules applies, then the input string is not in the language defined by the grammar. This proves that our simple language $\mathtt{[}^n\ \mathtt{]}^n$ is LL(1).

Besides efficiency, an effect of this LL approach to parser generation is that it supports good error messages in the case of failure. For example, if we see the parsing goal $\mathtt{(}\ w\ :\ \mathtt{)}\ \gamma$ we can report: *Found ′*(*′ while expecting ′*)*′* along with a report of the error location. Similarly for other cases that match none of the conclusions of the rules.

4 Grammar Transformations

This predictive parsing or LL(1) parsing works quite well. But now suppose we have a grammar

$$\begin{array}{rcl} X & \longrightarrow & Yb \\ X & \longrightarrow & Yc \end{array}$$

for a nonterminal Y that may produce token streams of unbounded length. This grammar is left-recursive, which is a problem, because we cannot know which production to use in predictive parsing without unbounded token lookahead. Yet we can *left-factorize* the grammar in order to capture the common prefix of the two productions and only distinguish their suffix in a new common nonterminal X'. This gives

$$\begin{array}{rcl} X & \longrightarrow & Y X' \\ X' & \longrightarrow & b \\ X' & \longrightarrow & c \end{array}$$

This left-factorization is also useful to remove a First/First conflict from grammar

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

turning it into a form where common prefixes have been factored out

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

More generally, consider the grammar

$$\begin{array}{rcl} X & \longrightarrow & \alpha\gamma \\ X & \longrightarrow & \alpha\delta \end{array}$$

where we have difficulties deciding between the two productions based on token lookahead, because both start with α. We can left-factorize it into the following grammar where common prefixes have been factored out and the distinction between γ and δ happens at a new nonterminal X'

$$\begin{array}{rcl} X & \longrightarrow & \alpha X' \\ X' & \longrightarrow & \gamma \\ X' & \longrightarrow & \delta \end{array}$$

The same phenomenon can also happen indirectly, though, even if the grammar rules do not literally start with the same expression Y. Suppose that we have a left-recursive grammar (snippet)

$$\begin{array}{rcl} E & \longrightarrow & E + T \\ E & \longrightarrow & T \\ T & \longrightarrow & ..... \end{array}$$

We cannot choose which grammar rule to use because both E productions can start with terms (T), which may be arbitrarily long. We can change its left recursion into a right recursion by pulling commonalities up. Then we start with what the productions of the nonterminal have in common and postpone the distinction to a new subsequent nonterminal E':

$$\begin{array}{rcl} E & \longrightarrow & TE' \\ E' & \longrightarrow & +TE' \\ E' & \longrightarrow & \epsilon \end{array}$$

This process eliminates left recursion using right recursion. These changes of the grammar have also changed the structure of the parse trees, which needs to be taken into account, e.g., through postprocessing transformations.

Quiz

1. Is there a language that CYK can parse but LL cannot parse? Give example or explain why none exists.
2. Is there a language that LL can parse but CYK cannot parse? Give example or explain why none exists.
3. Is there a language that recursive descent can parse but LL cannot parse? Give example or explain why none exists.
4. Is there a language that LL can parse but recursive descent cannot parse? Give example or explain why none exists.
5. Why do we limit the token lookahead to 1? Does it make a big difference if we would instead limit it to 5?
6. Should we allow token lookahead ∞?
7. Suppose you only have an LL parser generator. Can you still write a compiler for any programming language?
8. Which features of programming languages are difficult or inconvenient for LL parsers?
9. Can $follow(X, \cdot)$ be computed from the grammar productions for X alone? Can $first(X, \cdot)$ be computed from only X productions?

10. Why should parser combinator libraries worry about whether a grammar is LL? Isn't that irrelevant?
11. What is very likely wrong with a compiler when you see a grammar production
$$S ::= E" = "" > "E"; "$$
12. Your combinator parser is beautiful, but for some input files it just produces a stack overflow without giving you any other information. What is wrong?

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. *J. Log. Program.*, 24(1&2):3–36, 1995.

[WM95] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. Addison-Wesley, 1995.