LL(1) Parsing

Peter Aldous

I. Introduction

Our textbook gives only a cursory explanation of parsing. While there are texts that treat parsing in detail, there is no need to purchase an additional textbook to use a small fraction of its material (especially when the information we use is commonly available). Since we have no readily available reading material on the topic of LL(1) parsing, I have written this short document in the hopes that it would be helpful to students who need to learn parsing early in their computer science education (e.g., students who take a discrete math course that treats parsing in their third semester).

In this document, I will present mathematical definitions and give prose descriptions of those definitions. If you discover a difference between the definitions and the descriptions *trust the definitions*.

This document assumes that the reader is familiar with lexical analysis and with the structure of phase-structure grammars, as well as a small amount of programming.

II. Motivation

It is necessary to understand a program's structure in order to execute or analyze it. Even after a lexical analysis has been performed, the token stream is simply a sequence. For example, the if keyword in Java is related to a condition and to subsequent statements. These statements may themselves be or contain if statements. In this case, the relationship between each if statement and its condition and branch or branches may not be immediately obvious. Parsing provides the structure that clarifies these relationships.

This document concerns *automated* LL(1) parsing. The first L in LL(1) specifies a left-to-right scan of the tokens; that is, tokens are read in the order they are produced, from the beginning of the document to the end. The second L in LL(1) states that the derivation produced is a leftmost derivation: only the leftmost nonterminal is reduced. The 1 in LL(1) states that only one token of lookahead is required; in other words, it is always possible to determine the next action by looking at the next token in the stream. In practice, most programming languages use LL(2) parsing. LL(2) parsing, however, adds complications that are not necessary for an early discrete math class.

The grammar in Figure 1 will serve as a running example throughout the document. $ will mark the end of input. ϵ represents the empty string. The symbols $\mathbb{V}$, $\mathbb{S}$, $\mathbb{T}$, and $\mathbb{P}$ are the vocabulary, start symbol, set of terminals, and set of productions in a phase-structure grammar. α and β are used throughout this document as arbitrary strings comprising terminals and/or nonterminals; in other words, $\alpha, \beta \in \mathbb{V}^\star$.

$$
\begin{aligned}
E &\to E + T \mid T \\
T &\to T * F \mid F \\
F &\to (E) \mid \mathsf{int}
\end{aligned}
$$

Fig. 1. An arithmetic grammar

The goal of all of the formalisms introduced in this document is to allow for the creation of a **parse table**. A parse table contains a grammar's nonterminals on the left and its terminals (as well as the end-of-input pseudoterminal $) along the top. An LL(1) parse is performed by identifying the leftmost nonterminal and the next token from input and looking up the appropriate production in the parse table. This process is presented in detail in Section VI.

III. Restructuring the grammar

Many grammars are amenable to LL(1) parsing, but only after they have been restructured. This section demonstrates the most common methods of restructuring grammars to this end.

A. Removal of left recursion

When E reduces to $E + T$, it produces a string that begins with E. As a result, it is possible for a parse to loop indefinitely instead of terminating, even on a finite input. The same thing occurs when T reduces to $T * F$.

In the more general case, a left-recursive production takes the form $N \to N\alpha \mid \beta$, where $\alpha, \beta \in \mathbb{V}^\star$ (that is, α and β are arbitrary strings comprising terminals and/or nonterminals). β necessarily starts with something besides N; otherwise, it would be impossible for N to reduce to a string in any language (as these strings must comprise only terminals). In this grammar, the first production produces a copy of α to the right of N. This rule may be applied any number of times, resulting in strings of the form $N\alpha^\star$. The second production $N \to \beta$ must eventually be used to reduce the string in a way that removes N. When this occurs, the string produced is of the form $\beta\alpha^\star$.

These productions may be rewritten to produce the same language with right recursion as follows:

$$
\begin{aligned}
N &\to \beta N' \\
N' &\to \alpha N' \mid \epsilon \ .
\end{aligned}
$$

Even more generally, there may be multiple productions of either form. In this more general case, the resulting string takes the form $\beta_i {\alpha_i}^\star$, where the choice of i may change for each

$$N \to N\alpha_1 \mid N\alpha_2 \mid \ldots \mid N\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$$

becomes

$$N \to \beta_1 N' \mid \beta_2 N' \mid \ldots \mid \beta_m N'$$
$$N' \to \alpha_1 N' \mid \alpha_2 N' \mid \ldots \mid \alpha_n N' \mid \epsilon$$

Fig. 2. Removal of left recursion

$$E \to TE'$$
$$E' \to +TE' \mid \epsilon$$
$$T \to FT'$$
$$T' \to *FT' \mid \epsilon$$
$$F \to (E) \mid \mathsf{int}$$

Fig. 3. The arithmetic grammar without left recursion

instance of some α_i. Figure 2 shows the transformation in this case.

Restructuring the grammar in Figure 1 yields the grammar in Figure 3.

B. Left factoring

In some grammars, different productions from the same nonterminal may begin with a common prefix or factor. In order to ensure that only one token of lookahead is required, these grammars must be left factored to produce an LL(1) grammar.

Consider the grammar in Figure 4, which contains a right recursive variant of the example grammar. Notice that this structure enforces right associativity on addition and multiplication. From a mathematical perspective, addition and multiplication are associative, so this makes no difference as far as correctness. However, this is contrary to the C specification. More importantly, this grammar would result in incorrect substitution and division, as these operations are *not* associative.

Although this grammar is not left recursive (and therefore is amenable to LL parsing), it is not clear which production should be used when examining a single token of input. For example, E may reduce to $T + E$ or to T. In either case, the next token could be (or an integer.

This transformation relies on the observation that the grammar may be rewritten in a way that parses the T common

$$E \to T + E \mid T$$
$$T \to F * T \mid F$$
$$F \to (E) \mid \mathsf{int}$$

Fig. 4. A right recursive arithmetic grammar

$$E \to TX$$
$$X \to +E \mid \epsilon$$
$$T \to FY$$
$$Y \to *T \mid \epsilon$$
$$F \to (E) \mid \mathsf{int}$$

Fig. 5. The grammar in Figure 4 left factored

to both productions and then decides what to do with the remainder of the input.

More generally, a nonterminal N may reduce to any number of strings, which can be partitioned into those that start with some common factor M and those that do not. The productions that begin with M may be condensed into a single production $N \to MX$, where X is some new nonterminal. X can produce the various suffixes that appear after M in the original grammar:

$$N \to M\alpha_1 \mid M\alpha_2 \mid \ldots \mid M\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$$

becomes

$$N \to MX \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$$
$$X \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \ .$$

Applying this process to the grammar in Figure 4 produces the grammar in Figure 5.

IV. $FIRST$ and $FOLLOW$ sets

The first step towards creating a parse table is to compute the $FIRST$and $FOLLOW$ sets for each nonterminal in the grammar. Strictly speaking, the $FOLLOW$ sets are only required for nonterminals whose $FIRST$ sets contain ϵ; however, the $FOLLOW$ set for one nonterminal is frequently useful for the computation of other $FOLLOW$ sets.

A. $FIRST$ sets

The $FIRST$ set of a nonterminal N is the set of terminals that may appear in some string derived from N. It includes ϵ whenever the empty string may be derived from N. Intuitively, it is the set of all tokens that may appear when parsing N; a parser that encounters a token not in the $FIRST$ set of its leftmost nonterminal will never parse successfully and can immediately report an error. The definition of a $FIRST$ set is given in Definition 1.

Definition 1:

$$FIRST(N) = \left\{ t \in \mathbb{T} \mid N \overset{*}{\to} t\alpha \right\} \cup ML(N)$$
$$ML(N) = \begin{cases} \{\epsilon\} & N \overset{*}{\to} \epsilon \\ \emptyset & \text{otherwise} \ . \end{cases}$$

$FIRST$ sets are usually easiest to compute from the bottom of a grammar to the top, as the bottommost productions usually

produce more terminals. Nonterminals defined towards the top tend to reference $FIRST$ sets for nonterminals defined below.

The grammar in Figure 3 has five nonterminals. The bottommost terminal is F, which has two productions. As this is a context-free grammar, terminals produced can never be removed. Accordingly, every string derived from F must either begin with (or int. Therefore, $FIRST(F) = \{(, \mathsf{int}\}$.

Since the first production for T' begins with $*$, $* \in FIRST(T')$. Since T' can reduce to the empty string, ϵ is also in $FIRST(T')$. $FIRST(T') = \{*, \epsilon\}$.

The computation of $FIRST(T)$ makes use of the already-computed $FIRST(T)$. Everything reduced from T begins with F, so $FIRST(T)$ includes all of the terminals in $FIRST(F)$. Since F cannot reduce to the empty string, neither can T. Accordingly, $FIRST(T) = \{(, \mathsf{int}\}$.

The other $FIRST$ sets are computed in the same way, resulting in the following values:

$$\begin{aligned} FIRST(E) &= \{(, \mathsf{int}\} \\ FIRST(E') &= \{+, \epsilon\} \\ FIRST(T) &= \{(, \mathsf{int}\} \\ FIRST(T') &= \{*, \epsilon\} \\ FIRST(F) &= \{(, \mathsf{int}\} \ . \end{aligned}$$

B. $FOLLOW$ sets

The $FOLLOW$ set of a nonterminal N is the set of terminals that may appear after N in some string derived from the start symbol. For convenience, $ is treated as a terminal for this purpose and may appear in a $FOLLOW$ set. (In a language whose lexical specification produces an explicit $token, this special case is unnecessary.) $FOLLOW$ sets are formally defined in Definition 2.

Definition 2:

$$FOLLOW(N) = \left\{ t \in \mathbb{T} \mid \mathbb{S} \overset{*}{\rightarrow} \alpha N t \beta \right\} \cup MB(N)$$

$$MB(N) = \begin{cases} \{\$\} & \mathbb{S} \overset{*}{\rightarrow} \alpha N \\ \emptyset & \text{otherwise} \ . \end{cases}$$

$FOLLOW$ sets are usually easiest to compute from the top of the grammar down. In the case of the grammar in Figure 3, the $FOLLOW$ set of E necessarily includes $, as E (the start symbol in this grammar) reduces after zero steps to a string ending in E. E also appears in parenthetical statements, so) may also appear after E. Because E can appear in no other circumstances (i.e., it appears nowhere else in the grammar), nothing else may follow it and $FOLLOW(E) = \{), \$\}$.

$FOLLOW(E')$ makes use of $FOLLOW(E)$, as E reduces to a string ending in E'. As a result, any terminal that can occur after E may also appear after E': $\alpha E t \beta \rightarrow \alpha T E' t \beta$. The production $E' \rightarrow +TE'$ can never change what follows E'; $\alpha E' t \beta \rightarrow \alpha + T E' t \beta$. Lastly, the production $E' \rightarrow \epsilon$ removes E', producing a string where no terminal follows that particular instance of E'. As a result, $FOLLOW(E') = FOLLOW(E) = \{), \$\}$.

	+	*	int	(	)	$
E			T E'	T E'		
E'	+ T E'				ϵ	ϵ
T			F T'	F T'		
T'	ϵ	* F T'			ϵ	ϵ
F			int	(E)		

TABLE I
THE PARSE TABLE FOR THE GRAMMAR IN FIGURE 3

T can only ever be produced with E' following it. Accordingly, any terminal that can appear at the beginning of a string derived from E' may appear after T: $(FIRST(E') - \{\epsilon\}) \subseteq FOLLOW(T)$. Additionally, E' can reduce to the empty string (since $\epsilon \in FIRST(E')$). As a result, $FOLLOW(E') \subseteq FOLLOW(T)$. The result is that $FOLLOW(T) = (FIRST(E') - \{\epsilon\}) \cup FOLLOW(E') = \{+,), \$\}$.

T and T' have a similar relationship to that of E and E'. Once their $FOLLOW$ sets are computed, the $FOLLOW(F)$ is straightforward. The resulting $FOLLOW$ sets follow:

$$\begin{aligned} FOLLOW(E) &= \{), \$\} \\ FOLLOW(E') &= \{), \$\} \\ FOLLOW(T) &= \{+,), \$\} \\ FOLLOW(T') &= \{+,), \$\} \\ FOLLOW(F) &= \{+, *,), \$\} \ . \end{aligned}$$

V. CREATING A PARSE TABLE

With the $FIRST$ and $FOLLOW$ sets defined for a grammar's nonterminals, the grammar's LL(1) parse table can be created. The parse table contains the grammar's nonterminals at the head of the rows and the grammar's terminals (including $ but *not* including ϵ) at the head of the columns. Each cell in the table is either empty or contains the right-hand side of some production in the grammar whose left-hand side is the nonterminal at the head of its row. Nothing besides the right-hand side of a production whose left-hand side is at the head of the row may ever be placed in the table.

To populate the row in the parse table with N at its head, examine all of the productions in $\mathbb{P}$ with N on the left-hand side. For each terminal $t \in FIRST(N)$, choose the production that reduces N in that direction. (For every grammar at this level, this choice should be trivial.) Place the right-hand side of this production in the cell belonging to row N and column t. Whenever ϵ appears in $FIRST(N)$, place the right-hand side of the production that moves in the direction of ϵ (in this grammar, this always happens directly: $N \rightarrow \epsilon$) in the cell for each terminal in $FOLLOW(N)$.

The LL(1) parse table for the grammar in Figure 3 is Table I.

VI. USING A PARSE TABLE

Parsing begins with the start symbol $\mathbb{S}$ followed by $ in the stack and the entire input. It proceeds by matching the leftmost symbol in the stack against the leftmost symbol t in the input. If the leftmost symbol in the stack is a terminal and it matches

Stack	Input	Action
$E$$	(1 + 2) * 3$	$E \to TE'$
$TE'$$	(1 + 2) * 3$	$T \to FT'$
$FT'E'$$	(1 + 2) * 3$	$F \to (E)$
$(E)T'E'$$	(1 + 2) * 3$	consume
$E)T'E'$$	1 + 2) * 3$	$E \to TE'$
$TE')T'E'$$	1 + 2) * 3$	$T \to FT'$
$FT'E')T'E'$$	1 + 2) * 3$	$F \to$ int
int $T'E')T'E'$$	1 + 2) * 3$	consume
$T'E')T'E'$$	+ 2) * 3$	$T' \to \epsilon$
$E')T'E'$$	+ 2) * 3$	$E' \to +TE'$
$+TE')T'E'$$	+ 2) * 3$	consume
$TE')T'E'$$	2) * 3$	$T \to FT'$
$FT'E')T'E'$$	2) * 3$	$F \to$ int
int $T'E')T'E'$$	2) * 3$	consume
$T'E')T'E'$$	) * 3$	$T' \to \epsilon$
$E')T'E'$$	) * 3$	$E' \to \epsilon$
$)T'E'$$	) * 3$	consume
$T'E'$$	* 3$	$T' \to *FT'$
$*FT'E'$$	* 3$	consume
$FT'E'$$	3$	$F \to$ int
int $T'E'$$	3$	consume
$T'E'$$	$	$T' \to \epsilon$
$E'$$	$	$T' \to \epsilon$
$	$	accept

TABLE II
PARSING (1 + 2) * 3 USING TABLE I

the leftmost symbol in the input, the terminal is consumed; in other words, it is removed from both the stack and the input. In practice, consuming an input token often means storing in an object that is being parsed. If the leftmost symbol in the stack is a nonterminal N, the action specified in the parse table for row N and column t is performed, reducing N. If there is no action, the input is not in the language and the input string is rejected. Table II demonstrates this process on the input (1 + 2) * 3.