CS143 Notes: Parsing [*]

David L. Dill

1 Errata

Lots of typos fixed 7:35 PM 4/19/2015. Thanks, Rui Ueyama.

Please email additional reports of errors or suggested improvements to dill@cs.stanford.edu.

2 Syntactic analysis

Programs have a tree-like structure. A node in the tree represents a part of the program, and the node's children represent subparts. For example, an "if-then-else" construct will have three children: an expressions for the if part, and statements for the then and else parts. Each of these parts may be arbitrarily complex (so the children may be the roots of arbitrarily large subtrees). However, program texts are flat. The structure is implicit, but the representation is a sequence of characters with no explicit structure other than the order of the characters.

Compilers need to recover the structure of the program from its textual representation. This process is called *parsing*, and the algorithm that does it is called a *parser*. The parser reads the program text and converts it to a tree structure. In many cases, the tree is stored explicitly. However, in some cases, compilation can proceed "on the fly" – processing can occur as the program is being parsed. However, the parser can still be thought of as recovering the program structure: as it parses, it systematically traverses an implicit tree structure and the processing is based on this traversal.

Parsing in program languages is based on the theory of *context-free languages*. Context-free languages were invented in an attempt to describe natural languages mathematically, and

[*]Copyright © by David L. Dill, 1998–2015. All rights reserved. This material may be freely distributed for educational purposes, so long as (1) this copyright notice remains, (2) the notes are not modified in any way, and (3) no fee is charged. Other uses require the express written consent of the author.

(apparently independently) invented to describe the structure of programming languages. The first use of context-free languages was to provide a precise definition of the structure of programming languages, since natural language specifications were subject to ambiguity, leading to misunderstandings about the definition of the programming language. However, once a formal notation became available, the possibility of using it to generate a parser automatically became irresistible.

Parsing theory is one of the major triumphs of computer science. It draws on a deep and independently interesting body of theory to solve important practical problems. The result is a method that can be used to describe languages formally and precisely, and to automate a major part of the processing of that language. Parsing theory has led to exceptionally efficient parsing algorithms (whether they are generated by hand or automatically). These algorithms run in time linear in the length of the input, with very small constant factors. Parsing is emphasized in CS143 because it is so generally useful, but also as a case study in computer science at its best. We will cover only the most widely used and useful algorithms, though. You should be aware that there is a huge body of literature on different algorithms and techniques.

Parsing theory has been so successful that it is taken for granted. For practical purposes, the problem is (almost) completely solved.

2.1 Context-free grammars

A context-free grammar (also called BNF for "Backus-Naur form") is a recursive definition of the structure of a context-free language.

Here is a standard example, for describing simple arithmetic expressions. This grammar will be referred to as the "expression grammar" in the rest of this subsection.

$$\begin{array}{lcl} E & \rightarrow & E + E \\ E & \rightarrow & E * E \\ E & \rightarrow & (E) \\ E & \rightarrow & Id \\ E & \rightarrow & Num \end{array}$$

In this grammar, $+$, $*$, (,), *Id*, and *Num* are symbols that can actually appear in expressions, while E is an symbol that stands for the set of all expressions. Note that, as in all good recursive definitions, there are some base cases which are not recursive ($E \rightarrow Id$ and $E \rightarrow Num$).

Theory

As with regular expressions, context-free grammars (CFGs) provide a way to define an infinite language with a finite set of rules. CFGs are more expressive than regular expressions (every regular language can be described by a CFG, but not *vice versa*). Basically, CFGs allow recursion to be used more freely than regular expressions.

A CFG is a four-tuple $\langle V, \Sigma, R, S \rangle$.

- V is a non-empty finite set of symbols, which we call *nonterminals*. Nonterminals are used to represent recursively defined languages. In the expression grammar above, the only nonterminal was E. In examples, nonterminals will usually be capital letters.
- Σ is a non-empty finite set of *terminal symbols*. Terminal symbols actually appear in the strings of the language described by the CFG. Σ and V are disjoint. In the expression grammar above, the terminals are '+', '$*$', '(', ')', *Id*, and *Num*. In examples, lower-case letters will often be terminal symbols.
- R is a non-empty finite set of *productions*. The productions are rules that describe how nonterminals can be expanded to sets of strings. A production has a *left-hand side (LHS)* consisting of a single nonterminal, and a *right-hand side (RHS)* consisting of a (possibly empty) string of terminals and nonterminals. In regular expression notation, a production is a member of $V \rightarrow (V \cup \Sigma)^*$. In the expression grammar above, there are five productions.
- S is the *start symbol* (otherwise known as the *sentence symbol*). It is a nonterminal representing the entire language of the CFG. The start symbol in expression grammar is E (obviously, since there is only one nonterminal to choose from). I often fail to mention explicitly what the start symbol is, if it seems obvious.

In addition to the notation above, we use the convention that lower-case letters late in the alphabet, like w, x, y, z, are used to represent strings of terminal symbols (i.e., members of Σ^*), while Greek letters early in the alphabet, like α, β, γ, represent strings of terminal and nonterminal symbols (i.e., members of $(V \cup \Sigma)^*$. However, the symbol ϵ always represents the empty string.

A production like $A \rightarrow \epsilon$ has a RHS that is the empty string. Such a production is called an ϵ-production.

The language of a context-free grammar

The purpose of a context-free grammar is to describe a language. A language that can be described by a context-free grammar is called a *context-free language*. The basic idea is to define a process whereby a terminal string can be *derived* from S by repeatedly replacing symbols that occur on the left-hand side of some production by the string of symbols on the right-hand side of the production. Making this precise requires some preliminary definitions of relations between strings of terminals and nonterminals.

Definition 1 *$\alpha A\beta$* immediately derives *$\alpha\gamma\beta$, (written $\alpha A\beta \Longrightarrow \alpha\gamma\beta$) if there is a production $A \rightarrow \gamma$.*

We can also say: $\alpha\gamma\beta$ is immediately derived from $\alpha A\beta$.

Note that α, β, or γ may be empty strings.

Example: In the expression grammar, $E + E \Longrightarrow E + E * E$, where α is $E+$, β is ϵ, A is E, and γ is $E * E$.

Definition 2 *A* derivation *is a non-empty sequence of strings over $(V \cup \Sigma)$ where each string in the sequence immediately derives the next.*

A derivation is often written as a sequence of strings separated by $\Longrightarrow$, for example $E+E \Longrightarrow E + E * E \Longrightarrow E + (E) * E \Longrightarrow E + (E + E) * E$.

The following definitions capture the relations between the first and last strings in a derivation.

Definition 3 *α* eventually derives *β (written $\alpha \overset{*}{\Longrightarrow} \beta$) if there exists a derivation with α as its first string and β as its last.*

Note that $\overset{*}{\Longrightarrow}$ is the *reflexive transitive closure* of $\Longrightarrow$. It allows the possibility of a derivation of length 0 (no steps).

Definition 4 *$\alpha \overset{+}{\Longrightarrow} \beta$ if there is some γ such that $\alpha \Longrightarrow \gamma$ and $\gamma \overset{*}{\Longrightarrow} \beta$.*

So, $\overset{+}{\Longrightarrow}$ says there is a derivation of at least one step from α to β.

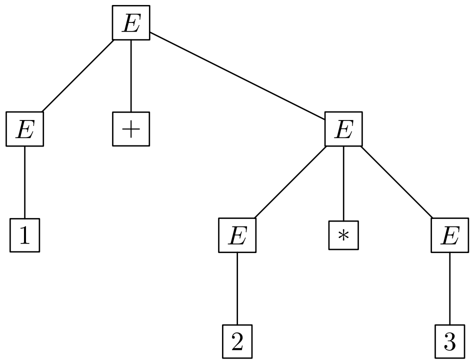


Figure 1: A parse tree

Definition 5 *The* language of a context free grammar G, *written* $L(G)$, *is the set of all terminal strings that can be derived from the sentence symbol. More symbolically, if* $G = \langle V, \Sigma, R, S\rangle$ *is a CFG,* $L(G)$ *is* $\{x \mid x \in \Sigma^* \wedge S \overset{+}{\Longrightarrow} x\}$.

(Of course, S is not in Σ^*, so it would be equivalent to say $L(G)$ is $\{x \mid x \in \Sigma^* \wedge S \overset{*}{\Longrightarrow} x\}$.

Here is another way to look at this definition: to prove that a string x is in the language of a CFG G, it is sufficient to exhibit a derivation of x from S using the productions of the grammar. Proving that x is *not* in $L(G)$ is another matter. (It happens to be possible to find a proof that x is in $L(G)$ or not *automatically.* In other words, the problem of deciding whether $x \in L(G)$ is *decidable* for all context-free grammars G.)

Parse trees

Parse trees provide a somewhat more abstract way of looking at derivations. Each node of a parse tree is labeled with a symbol. The root of a parse tree is labeled with a nonterminal. Whenever a node is labeled with a nonterminal A, it may have children. The children of the node, from left to right, are labeled with the symbols from α, where $A \rightarrow \alpha$ is a production in the grammar. If a node is labeled with a terminal symbol, it has no children. The *frontier* of a parse tree is the sequence of labels of its leaves, from left to right.

Figure 1 shows a parse tree based on the expression grammar. Obviously, there is a relationship between derivations and parse trees. Is there a one-to-one correspondence? Interestingly, the answer is "no." In general, there are many derivations corresponding to the same parse tree. In the example above, the derivations

$$E \Longrightarrow E + E \Longrightarrow 1 + E \Longrightarrow 1 + E * E \Longrightarrow 1 + 2 * E \Longrightarrow 1 + 2 * 3$$

and

$$E \Longrightarrow E + E \Longrightarrow E + E * E \Longrightarrow E + 2 * E \Longrightarrow E + 2 * 3 \Longrightarrow 1 + 2 * 3$$

both correspond to the parse tree of Figure 1 The frontier of this tree is $1 + 2 * 3$.

There is no *unique* derivation for a given parse tree because the productions of the grammar may be expanded in different orders, and the parse tree does not capture the history of the expansion, just the final result.

However, of the many derivations that correspond to a particular parse tree, we can *choose* a unique one by constraining the order in which nonterminals are expanded. One useful rule is: "always expand the leftmost nonterminal in the string." A derivation that adheres to this rule is called a *leftmost derivation*.

More specifically, suppose we have a string of terminals and nonterminals whose leftmost nonterminal is A. Such a string can be written $xA\alpha$, where x is a string of zero or more terminal symbols. If $A \to \beta$ is a production, we can write $xA\alpha \overset{L}{\Longrightarrow} x\beta\alpha$. Let's call this a "leftmost derivation step." A leftmost derivation is a sequence of leftmost derivation steps. The first of the two example derivations above is a leftmost derivation.

Of course, there are other rules that could select a unique derivation to go with a parse tree. For example, a *rightmost derivation* expands the rightmost nonterminal in every string.

Leftmost and rightmost derivations are interesting because certain parsing algorithms generate them. It is useful to know what kind of derivation a parsing algorithm generates if actions are associated with the productions, since the order of expansion affects the order of the actions.

To prove that a string x is in the language of a context-free grammar, we could produce a parse tree with S at the root and x as the frontier, instead of producing a derivation.

Ambiguous grammars

Definition 6 *A context-free grammar is* ambiguous *if, for some terminal string x, there exists more than one parse tree with S at the root and x as the frontier. parse tree with S*

Given the immediately forgoing discussion, it is obvious that "leftmost derivation" or "rightmost derivation" could be substituted for "parse tree" in this definition without changing the meaning.

Is the expression grammar ambiguous? Very much so! Figure 2 shows an alternative parse for $1+2*3$ to that in Figure 1. The parse of Figure 1 corresponds to the grouping $1+[2*3]$.

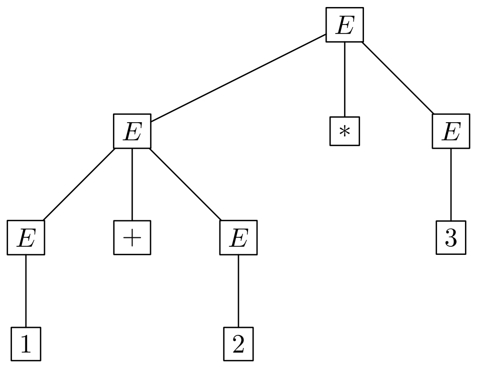


Figure 2: Another parse tree for $1 + 2 * 3$.

we would expect, while Figure 2 corresponds to $[1 + 2] * 3$. (Note: If there are actually parenthesis in the expression, the parse tree is different from either Figure 1 or Figure 2.) Although we know which parse tree is correct, the grammar doesn't specify which to use. The string $1 * 2 * 3$ can be parsed in two different ways also. The trees are the same as in Figures 1 and 2, except that $+$ is replaced by $*$. In this case, performing the arithmetic operations would yield the same result (6) for both expressions, but still there are two parse trees and the grammar is ambiguous.

Ambiguity is a problem for two reasons: First, as in our first example, it can represent a failure to specify an important property of the language being defined (such as how to evaluate an arithmetic expression). Second, even if the meaning of the construct is not affected, it tends to cause problems for parsing algorithms. Most efficient parsing algorithms simply fail for ambiguous grammars, and even if they don't, there remains the question of how to choose the correct parse tree if the parsing algorithm produces several.

There are several ways to cope with ambiguity. One is to find an equivalent but unambiguous grammar. (Here, "equivalent" means "describes the same language.") The an unambiguous grammar for expressions is

$$\begin{array}{lcl}
E & \rightarrow & E + T \\
E & \rightarrow & T \\
T & \rightarrow & T * F \\
T & \rightarrow & F \\
F & \rightarrow & (E) \\
F & \rightarrow & Id \\
F & \rightarrow & Num
\end{array}$$

Figure 3 shows the parse tree for $1 + 2 * 3$ that results from this grammar. Suppose we are generating a string with multiple $+$ symbols. This CFG forces the derivation to generate

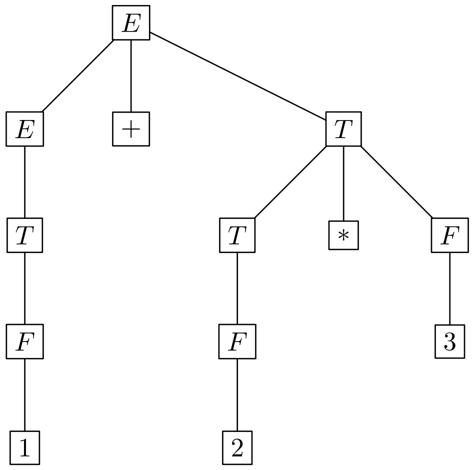


Figure 3: Parse tree for $1 + 2 * 3$ using the unambiguous expression grammar.

all of the $+$ symbols at the top of the parse tree by repeatedly expanding the $E \rightarrow E + T$ production. Once the $E \rightarrow T$ production is expanded, it is impossible to get another $+$ from the T unless the production $F \rightarrow (E)$ is expanded (as it is for an expression like $(1 + 2) * 3$. In addition, the grammar makes $+$ and $*$ group to the left (they are *left associative*). Study this grammar carefully: generate some parse trees and understand the intuition. How could you make $*$ right associative?

It is not always possible to find an unambiguous grammar for a particular context-free language. Some languages are *inherently ambiguous*. Also, when it is possible, it is not necessarily easy to find it. There is provably no universal method for find an unambiguous grammar if one exists, or even to determine if there is an unambiguous grammar (more precisely, the problem of determining whether a context-free language has an unambiguous CFG is undecidable).

2.2 Extended BNF

The context free grammar notation can be extended to make it more convenient and readable. As with extended regular expressions, this additional notation adds no new power – any extended context-free grammar can be reduced to an equivalent ordinary CFG by a series of transformations. Context-free grammar notation is sometimes called "BNF" for "Backus-Naur Form," and the extended notation is often called "Extended BNF" (we will refer to it as "EBNF").

EBNF notation allows the use of extended regular expressions on the right hand sides of productions. Hence, the productions can look like $A \rightarrow a(b \mid (c?\ B)^*)$. EBNF has lots of *meta-*

operators that are part of the EBNF notation, not terminal symbols. In the example above, only the letters are terminals or nonterminals; all the other symbols are meta-operators. Here is an longer EBNF example, based on the procedure call syntax of the Pascal language:

$$\begin{array}{rcl} \textit{proccall} & \rightarrow & \textit{ID} \text{ ('(' } \textit{arglist} \text{ ')')?} \\ \textit{arglist} & \rightarrow & \textit{expr} \text{ (',' } \textit{expr} \text{)}^* \end{array}$$

(), ?, and $*$ are symbols of the EBNF notation, not terminals, unless they are enclosed in single quotes. In English, this says that a procedure call is an `ID` followed by an optional argument list which is enclosed in parentheses. The argument list is a comma-separated list of one or more expressions.

An EBNF grammar can be converted to an ordinary CFG by a series of transformations that eliminate the extended regular expression constructs. Whenever we have an embedded $|$ in a production, which would have the structure $A \rightarrow \alpha(\beta \mid \gamma)\delta$, where α, β, γ, and δ are arbitrary extended regular expressions, we can eliminate the $|$ by choosing a new nonterminal that does not appear in the grammar, say B, and replacing the old production with three new productions:

$$\begin{array}{rcl} A & \rightarrow & \alpha B \delta \\ B & \rightarrow & \beta \\ B & \rightarrow & \gamma \end{array}$$

Similarly, $A \rightarrow \alpha\beta?\ \gamma$ can be replaced by new productions $A \rightarrow \alpha B \gamma$, $B \rightarrow \beta$ and $B \rightarrow \epsilon$. $A \rightarrow \alpha\beta^*\gamma$ can be replaced by

$$\begin{array}{rcl} A & \rightarrow & \alpha B \gamma \\ B & \rightarrow & B\beta \\ B & \rightarrow & \epsilon \end{array}$$

$A \rightarrow \alpha\beta^+\gamma$ can be treated similarly, but using $B \rightarrow \beta$ instead of $B \rightarrow \epsilon$. Both $*$ and $+$ can be converted to right-recursive grammars instead of left-recursive grammars by using the production $B \rightarrow \beta B$ instead of $B \rightarrow B\beta$. This is very useful to know, because some there are cases where left recursion or right recursion is strongly preferable.

These transformations can be applied to the Pascal procedure call example above, yielding:

$$\begin{array}{rcl}
\mathit{proccall} & \rightarrow & \mathit{ID}\ A \\
A & \rightarrow & \text{'('}\ \mathit{arglist}\ \text{')'} \\
A & \rightarrow & \epsilon
\end{array}$$

$$\begin{array}{rcl}
\mathit{arglist} & \rightarrow & \mathit{expr}\ B \\
B & \rightarrow & B\ \text{','}\ \mathit{expr} \\
B & \rightarrow & \epsilon
\end{array}$$

Of course, particular grammars can be rewritten into smaller or nicer CFGs. However, as with many problems in mathematical manipulation, using your brain to produce a nicer result produces a wrong result more frequently than you might expect. Mindless application of the transformations is less risky.

Useless symbols and productions

Sometimes a grammar can have symbols or productions that cannot be used in deriving a string of terminals from the sentence symbol. These are imaginatively named *useless symbols* and *useless productions*.

Example

Consider the following CFG:

$$\begin{array}{rcl}
S & \rightarrow & SAB \mid a \\
A & \rightarrow & AA \\
B & \rightarrow & b
\end{array}$$

(From now on, we occasionally adopt the convention of allowing | at "top-level" in the right-hand side of a production as an abbreviation for multiple productions with the same left-hand side symbol. This is *not* considered extended BNF.) A is a useless symbol: No terminal strings can be derived from A (because eliminating one A produces two more), so A can never appear in a derivation that ends in a string of all terminals. Because of this, the production $S \rightarrow SAB$ is also useless: it introduces an A, so if this production is used, the derivation cannot result in a terminal string. Also, the only way to introduce a B into a derivation is to use $S \rightarrow SAB$, so B is useless as well. If we delete all the useless symbols and productions from the grammar, we get an equivalent, but simpler, CFG:

$$S \rightarrow a$$

Here is a more formal definition:

Definition 7 *A terminal or nonterminal symbol X is* useless *if there do not exist strings of symbols α, β, and a string of terminals x such that $S \overset{*}{\Longrightarrow} \alpha X \beta \overset{*}{\Longrightarrow} x$.*

It is a lot simpler to state various properties of context-free grammars if there are no useless symbols. *From now to the end of the lectures on parsing, we assume there are no useless symbols in our CFGs, unless there is an explicit statement to the contrary.*

3 Top down parsing

We have defined the language of a CFG in a "generative" fashion: the language is the set of strings which can be derived from a sentence symbol via a set of rules. *Parsing* inverts the process of generating a string. The parsing problem is, given a CFG and a terminal string, to find a derivation of the string, or report that none exists.

There are two important styles of parsing: *top-down* and *bottom-up*. In actuality, very many parsing algorithms have been proposed, not all of which fit neatly into one of these categories. But the top-down/bottom-up distinction fits the parsing algorithms in CS143 very well, and is very helpful for understanding parsing algorithms.

Top-down parsing can be thought of as *expanding* a parse tree or derivation from the sentence symbol until the frontier matches the desired input string (or until it becomes clear that there is no way to make them match). Obviously, there are in general an infinite number of trees that can be expanded. The tree that is expanded depends on which productions are used to replace the nonterminals in the frontier of the tree at each step of the expansion. The differences among top-down parsing methods are in the methods used to choose this production.

3.1 Top-down parsing by guessing

To introduce the idea of top-down parsing, let's temporarily disregard the exact method use to pick the next production to expand. Instead, we'll do it by making a "lucky guess" at each step about which production to expand. Later, instead of guessing, we'll look up what to do in a table.

At any point during the parse, the state of the parser is characterized by two items: the *remaining input* and the *stack*. Once you know the values of these variables, you know what will happen with the rest of the parse. The input contains a sequence of terminal symbols.

Its initial value is the input string to be parsed, x, with the leftmost symbol first. The stack contains terminal and nonterminal symbols. Initially, the stack contains one symbol, which is S, the sentence symbol of the CFG.

Intuitively, the stack stores a partially expanded parse tree. The top of the stack is a *prediction* about what the parser will see next. If the top symbol is a terminal, it must *match* the next symbol in the input stream. If the top symbol is a nonterminal, there must be some way to *expand* it to a string that matches the beginning of the input (top-down parsing is sometimes called "predictive parsing").

Reflecting the discussion of the previous paragraph, there are two basic actions that occur during parsing. When there is a terminal symbol on top of the stack, it is *matched* with the next symbol on the input. Matching compares the two symbols; if they are not equal, the input string is *rejected*, meaning that the input string is not in the language of the CFG. If the symbols match, the top symbol is popped off of the stack and the input symbol is removed from the front of the input. When there is a nonterminal A on top of the stack, it is *expanded* by choosing a production $A \to \alpha$ from the CFG, popping A from the stack, and pushing the symbols in α, rightmost first, onto the stack (so that the leftmost symbol of α ends up on top of the stack). (Note: if $\alpha = \epsilon$, expanding has the effect of simply popping A off of the stack.)

Example

Let's parse the input "aab" using the CFG:

$$\begin{array}{lcl} S & \to & AS \mid B \\ A & \to & a \\ B & \to & b \end{array}$$

Here are the steps of the parse. \$ represents "end of file" and "bottom of stack." The actions explain how we get from one step to the next. The top of the stack is drawn on the left. Whether we expand or match is determined by whether the top symbol is a nonterminal or terminal; if it's a nonterminal, we have to guess which production with that symbol on the left-hand side should be expanded. At the end of the parse, we *accept* the input string if the input and stack are both empty. If there is a mismatch, we reject.

parse	(top) stack	action
aab$	S$	expand $S \to AS$
aab$	AS$	expand $A \to a$
aab$	aS$	match
ab$	S$	expand $S \to AS$
ab$	AS$	expand $A \to a$
ab$	aS$	match
b$	S$	expand $S \to B$
b$	B$	expand $B \to b$
b$	b$	match
$	$	accept

If this parse algorithm accepts, we can be sure that the input is in the language of the CFG. If the parse does not accept, the string may or may not be in the language. However, if we assume that the *best possible* guess is made at each step, the parsing algorithm will accept if it is possible to do so, and does not accept exactly when the input is not in the language.

A derivation and parse tree can be reconstructed from the history of expansions in the parse. The derivation in this case is

$$S \Longrightarrow AS \Longrightarrow aS \Longrightarrow aAS \Longrightarrow aaS \Longrightarrow aaB \Longrightarrow aab.$$

This is a *leftmost* derivation because we always expanded the top symbol on the stack, which was the leftmost symbol of the string derived up to that point.

3.2 LL(1) parsing

If a computer could make lucky guesses with 100% reliability, as we have assumed above, parsing would be one of the less important applications. Instead, it is possible to construct a table where the proper action can be looked up, based on the symbols on *the top of the stack* and at *the beginning of the input*. This algorithm is called *LL(1) parsing* (for "leftmost (parse) lookahead 1 symbol"). LL(1) parsing is almost the simplest top-down parsing scheme one could imagine. In essence, it says: "don't choose a production unless it at least has a chance of matching the next input symbol."

LL(1) parsing is very efficient (the running time is linear in the length of the input string, with a very low constant factor). There is a tradeoff, however. Not all CFGs can be handled by LL(1) parsing. If a CFG cannot be handled, we say it is "not LL(1)." There is an LL(1) parse table generation algorithm that either successfully builds a parse table (if the CFG is LL(1)) or reports that the CFG is not LL(1) and why.

The LL(1) parse table construction is somewhat involved, so it will take a little while to go through all the steps. Before doing so, we first describe a transformation that increases the chances that a CFG will be LL(1).

3.2.1 Removing left recursion

A CFG is said to be left recursive if there exists a nonterminal A such that $A \overset{+}{\Longrightarrow} A\alpha$ (i.e., A can expand to a string beginning with A). *Left recursion* in a CFG is fatal for LL(1) parsing, and for most other top-down parsing algorithms. To see why, imagine that the parse table says to expand $A \rightarrow \beta$ when a is at the beginning of the input. The parse algorithm will go through a sequence of expand steps until it has $A\alpha$ on the top of the stack, without matching any inputs. But now we have A on top of the stack and a at the beginning of the input, so we'll do exactly the same thing *ad infinitum.* (LL(k) algorithms generalize LL(1) parsing to use the first k input symbols to decide what to expand. Although more powerful than LL(1) parsing when $k > 1$, the same argument shows that they are also unable to deal with left recursion.)

Fortunately, there is an algorithm to eliminate left recursion from any CFG, without changing its language. The general transformation is fairly difficult and not really practical, so we'll consider the important special case of removing *immediate* left recursion, which is when there is a production of the form $A \rightarrow A\alpha$ in the grammar.

Suppose we have a production $A \rightarrow A\alpha$ in our CFG. First, collect all of the productions having A on the left-hand side, and combine them into a single production using EBNF notation of the form $A \rightarrow A\alpha \mid \beta$. For example, suppose the productions are

$$\begin{array}{lcl} A & \rightarrow & A\alpha_1 \\ A & \rightarrow & A\alpha_2 \\ A & \rightarrow & \beta_1 \\ A & \rightarrow & \beta_2 \end{array}$$

The EBNF production is $A \rightarrow A(\alpha_1 \mid \alpha_2) \mid (\beta_1 \mid \beta_2)$.

A production of the form $A \rightarrow A\alpha \mid \beta$ can be written equivalently and non-recursively as $A \rightarrow \beta\alpha^*$ (to see this, expand A repeatedly and notice the pattern: $A \Longrightarrow A\alpha \Longrightarrow A\alpha\alpha \Longrightarrow \ldots \Longrightarrow \beta \ldots \alpha\alpha$).

All that remains is to convert this back to a (non-EBNF) CFG. Here we exploit the earlier observation that α^* can be expanded left recursively *or right recursively*:

$$\begin{array}{lcl} A & \rightarrow & \beta B \\ B & \rightarrow & \epsilon \\ B & \rightarrow & \alpha B \end{array}$$

If $\beta = \beta_1 \mid \beta_2$ and $\alpha = \alpha_1 \mid \alpha_2$ as in the example above, this finally becomes

$$\begin{array}{lcl} A & \rightarrow & \beta_1 B \\ A & \rightarrow & \beta_2 B \\ B & \rightarrow & \epsilon \\ B & \rightarrow & \alpha_1 B \\ B & \rightarrow & \alpha_2 B \end{array}$$

Example

Consider the unambiguous expression grammar of the previous lecture, which had the left-recursive productions

$$\begin{array}{lcl} E & \rightarrow & E + T \\ E & \rightarrow & T \end{array}$$

which can be rewritten as $E \rightarrow E + T \mid T$. In turn, this is $E \rightarrow T(+ T)^*$, which can be re-expanded right recursively into

$$\begin{array}{lcl} E & \rightarrow & TA \\ A & \rightarrow & +TA \\ A & \rightarrow & \epsilon \end{array}$$

IMPORTANT NOTE: A real understanding of this method requires a hands-on approach. Try generating some strings from each set of productions and see why they do the same thing. Try working through this on the whole grammar. Try making up some other grammars and trying it. Otherwise, you won't learn it!

Left factoring

A CFG has common left factors if the right-hand sides of two productions with the same left-hand symbol start with the same symbol. The presence of common left factors CFGs sabotages LL(1) parsing.

Suppose we have productions

$$\begin{array}{lcl} A & \rightarrow & \alpha\beta \\ A & \rightarrow & \alpha\gamma \end{array}$$

We can convert these to EBNF, also: $A \rightarrow \alpha(\beta \mid \gamma)$, and convert back:

$$\begin{array}{lcl} A & \rightarrow & \alpha B \\ B & \rightarrow & \beta \\ B & \rightarrow & \gamma \end{array}$$

Example

Suppose we had a CFG with productions

$$\begin{array}{lcl} E & \rightarrow & T + E \\ E & \rightarrow & T \end{array}$$

This converts to $E \rightarrow T(+ E \mid \epsilon)$, which converts back to

$$\begin{array}{lcl} E & \rightarrow & TA \\ A & \rightarrow & +E \\ A & \rightarrow & \epsilon \end{array}$$

A CFG may still not be LL(1) even after eliminating left recursion and left factors, but it *certainly* will not be LL(1) if they are not eliminated.

Unfortunately, while these transformations preserve the language of a CFG, *they do not preserve the structure of the parse tree.* The structure can be recovered from the parse with some trouble, but it is a distinct disadvantage of LL(1) parsing that the original grammar must be rewritten into a form that is probably not as natural as the original.

LL(1) parsing example

Before discussing in detail the construction the LL(1) parse tables, let's seen an example of how the parsing algorithm uses one. This is the grammar that results when left recursion is removed from the unambiguous expression grammar, as described above. We will call this our *LL(1) expression grammar.* It will be referred to frequently below.

$$\begin{array}{lcl} E & \to & TA \\ A & \to & +TA \mid \epsilon \\ T & \to & FB \\ B & \to & *FB \mid \epsilon \\ F & \to & (E) \mid a \end{array}$$

Here is the LL(1) parse table for the grammar:

	a	$+$	$*$	(	)	\$
E	$E \to TA$			$E \to TA$		
T	$T \to FB$			$T \to FB$		
F	$F \to a$			$F \to (E)$		
A		$A \to +TA$			$A \to \epsilon$	$A \to \epsilon$
B		$B \to \epsilon$	$B \to *FB$		$B \to \epsilon$	$B \to \epsilon$

The parsing algorithm is as before, except that when the top symbol on the stack is a nonterminal, we don't guess. Instead, we look in the table, in the row for the top-of-stack symbol and column for the next input symbol (which can be \$ if the entire input has been consumed). The table entry uniquely determines which production to expand.

Here is what happens when we parse the string $a + a * a$ using this table.

(top) stack	parse	action
E\$	a+a*a\$	expand $E \to TA$
TA\$	a+a*a\$	expand $T \to FB$
FBA\$	a+a*a\$	expand $F \to a$
aBA\$	a+a*a\$	match
BA\$	+a*a\$	expand $B \to \epsilon$
A\$	+a*a\$	expand $A \to +TA$
+TA\$	+a*a\$	match
TA\$	a*a\$	expand $T \to FB$
FBA\$	a*a\$	expand $F \to a$
aBA\$	a*a\$	match
BA\$	*a\$	expand $B \to *FB$
*FBA\$	*a\$	match
FBA\$	a\$	expand $F \to a$
aBA\$	a\$	match
BA\$	\$	expand $B \to \epsilon$
A\$	\$	expand $A \to \epsilon$
\$	\$	accept

LL(1) Parse Table Construction

The basic idea behind the LL(1) parse table is to find the set of terminal symbols that can appear at the beginning of a string expanded from a production. If the nonterminal on top of the stack is A and the next input is a, and there are two productions $A \to \alpha$ and $A \to \beta$, we choose $A \to \alpha$ if α can expand to something beginning with a. The choice must be unique; if β also expands to a string beginning with a, the grammar is not LL(1) (LL(1) parsing won't work, so we either have to change the grammar or use a different parsing algorithm).

Constructing the LL(1) parse table requires some preliminary definitions and computations. The first is the set of *nullable nonterminals*. To compute them, we need the more general concept of a nullable string:

Definition 8 *A string α in $(V \cup \Sigma)^*$ is* nullable *if $\alpha \overset{*}{\Longrightarrow} \epsilon$.*

We would like to determine the set of nullable symbols in a context-free grammar. Instead of giving an algorithm to compute these sets, we give a set of rules that can be applied iteratively to compute them. The order of application of the rules does not matter, so long as each rule is eventually applied whenever it can change the computed function (and as long is the rule is not applied to unnecessary strings). The rules for nullable strings compute a Boolean function *Nullable*. The domain of the function is all the strings consisting of single nonterminal symbols, or right-hand-sides of productions. Initially, we assume that *Nullable* is *false* for all strings; the rules can be applied to set it to *true* for some strings, whereupon other rules can set it to *true* for other strings. The process terminates when no rule can change *Nullable* from *false* to *true* for any string.

1. $\mathit{Nullable}(X_1X_2 \ldots X_n) = \mathit{true}$ if, for all $1 \leq i \leq n$, $\mathit{Nullable}(X_i)$.
2. $\mathit{Nullable}(A) = \mathit{true}$ if there is a production in the CFG $A \to \alpha$ and $\mathit{Nullable}(\alpha)$.

In particular, rule 1 implies that ϵ is nullable, since for all $i \leq i \leq n$ applies to no i whatsoever, so all (zero) symbols are nullable.

Here is an example:

$$\begin{array}{lcl} S & \to & ABC \\ A & \to & \epsilon \\ B & \to & \epsilon \\ C & \to & AB \end{array}$$

Initially, *Nullable* is *false* for everything. Then

$\mathit{Nullable}(\epsilon) = \mathit{true}$	rule 1 ($X_1X_2 \ldots X_n = \epsilon$)
$\mathit{Nullable}(A) = \mathit{true}$	rule 2 ($A \to \epsilon$)
$\mathit{Nullable}(B) = \mathit{true}$	rule 2 ($B \to \epsilon$)
$\mathit{Nullable}(AB) = \mathit{true}$	rule 1
$\mathit{Nullable}(C) = \mathit{true}$	rule 2 ($C \to AB$)
$\mathit{Nullable}(ABC) = \mathit{true}$	rule 1
$\mathit{Nullable}(S) = \mathit{true}$	rule 2 ($S \to ABC$)

In this case, everything is nullable, which is unusual.

By a much easier computation, the nullable symbols in the LL(1) expression grammar are A and B.

The next function that needs to be computed is the *FNE* set. *FNE* stands for "First, no ϵ". It is unique to this course, but I think it is easier to understand than the standard approach, which I will also describe. $\mathit{FNE}(\alpha)$ is the set of terminal symbols which can appear at the beginning of a terminal string derived from α.

Definition 9 *If α is a string in $(V \cup \Sigma)^*$, then $\mathit{FNE}(\alpha)$ is*
$\{a \mid a \in \Sigma \ \wedge \ \alpha \overset{*}{\Longrightarrow} ax,$ *for some* $x \in \Sigma^*.\}$.

(Note that since we've assumed there are no useless symbols in the CFG being analyzed, the definition would be equivalent if we relaxed the requirement that x be a terminal string.)

The domain of *FNE* consists of the set of all nonterminals and *suffixes* (i.e., tails) of strings appearing on the right-hand sides of productions, but the codomain of *FNE* consists of the subsets of Σ (i.e., $\mathit{FNE}(\alpha)$ is a set of terminal symbols). As with *Nullable*, we start with a "small" initial value for *FNE*, the function that maps every string to the empty set, and apply a set of rules in no particular order until no rules can change *FNE*.

Rule 2 below is partially obvious: the *FNE* set of a string $X_1X_2 \ldots X_n$ always includes the *FNE* of X_1. However, it may include the *FNE* of X_2 and later symbols, also: If X_1 is nullable, there is a derivation $X_1X_2 \ldots X_n \overset{*}{\Longrightarrow} \epsilon a \ldots = a \ldots$, where X_1 "expands" into the empty string, so the first terminal symbol actually derives from X_2. However, even if X_1 is nullable, it may end up contributing terminals to the *FNE* of $X_1X_2 \ldots X_n$ because there may also be derivations where X_1 expands to a non-empty string (nullable symbols *may* expand to ϵ, but they may expand to other strings as well). Clearly, if $X_1 \ldots X_k$ are all nullable, the first terminal may come from X_{k+1}. The first rule also implies that $\mathit{FNE}(\epsilon) = \emptyset$.

The justification for rule 3 below is simple: If $\alpha \overset{*}{\Longrightarrow} ax$, and $A \to \alpha$ is a production in the CFG, then $A \Longrightarrow \alpha \overset{*}{\Longrightarrow} ax$, so $a \in \mathit{FNE}(A)$.

1. $FNE(a) = \{a\}$

2. $FNE(X_1X_2 \dots X_n) = $ if $Nullable(X_1)$ then $FNE(X_1) \cup FNE(X_2 \dots X_n)$ else $FNE(X_1)$.

3. $FNE(A) = \cup\{FNE(\alpha) \mid A \to \alpha \in P\}$

Let's use these rules to compute the *FNE* sets for our LL(1) expression CFG. These are purposely done in a suboptimal order to show that rules sometimes have to be used on the same string multiple times before the correct answer is reached.

String	Added	Reason
$+$	$+$	rule 1
$*$	$*$	rule 1
(	(	rule 1
a	a	rule 1
$+TA$	$+$	rule 2
A	$+$	rule 3
$*FB$	$*$	rule 2
B	$*$	rule 3
F	a	rule 3
FB	a	rule 2
T	a	rule 3
TA	a	rule 2
(E)	(	rule 2
F	(	rule 3
FB	(	rule 2
T	(	rule 3
TA	(	rule 2
E	$a,($	rule 3

Note: You should work through these in detail to see how the rules work, and try another order to see if you get the same result.

The resulting FNE sets for the nonterminals in the grammar are:

E	{ a, (}
T	{ a, (}
F	{ a, (}
A	{ + }
B	{ * }

Note: You should check the definition of FNE *above and re-inspect the LL(1) expression grammar to see why this makes sense.*

This CFG didn't use rule 2 in its full generality. Here is a simple example that does.

$$\begin{array}{lcl} S & \to & ABC \\ A & \to & aA \mid \epsilon \\ B & \to & b \mid \epsilon \\ C & \to & c \mid d \end{array}$$

Note that A and B are nullable. Applying the rules gives:

String	Added	Reason
a	a	rule 1
b	b	rule 1
c	c	rule 1
d	d	rule 1
C	c, d	rule 3
B	b	rule 2
aA	a	rule 2
A	a	rule 3
ABC	a, b, c, d	rule 2
S	a, b, c, d	rule 3

FNE(ABC) includes terminals from A but also from B (because A is nullable) and C (because B is nullable). *Suggestion: Show for each of member of FNE*(S) *that there is a way to derive a string from* S *beginning with that terminal.*

I will also describe the "standard" approach that appears in all textbooks on the subject, in case you need to talk to someone else about this material who didn't take this class. The standard approach is not to define *FNE* sets, but *FIRST* sets. The only difference is that *FIRST*(α) includes ϵ if α is nullable. Rules for computing *FIRST* can be given that are very similar to the rules for *FNE. FIRST* sets are defined the way they are because the concept generalizes to LL(k) parse table construction for $k > 1$. But we don't care about that, and inclusion of ϵ seems to lead to confusion, so we use *FNE* sets, instead.

The last set that needs to be defined before constructing the LL(1) parse table is the *FOLLOW* set for each nonterminal. The *FOLLOW* sets are only used for nullable productions. Recall that the LL(1) parse table selects a production based on the nonterminal on top of the stack and the next input symbol. We want to choose a production that is consistent with the next input. *FNE* sets tell us what we want to know if the next terminal is going to be derived from the top nonterminal on the stack (we just pick the production whose right-hand side has the next input in it's *FNE* set). But, suppose the nonterminal on top of the stack "expands" to ϵ! Then the next nonterminal is going to come from some symbol *after* that nonterminal. The *FOLLOW* sets say exactly which terminals can occur immediately after a nonterminal.

Definition 10 $FOLLOW(A) = \{a \mid S\$ \overset{*}{\Longrightarrow} \alpha A a \beta\}$, *for some* $\alpha \in (V \cup \Sigma)^*$, $\beta \in (V \cup \Sigma)^*$, *and* $a \in (\Sigma \cup \{\$\})$.

This definition derives strings from $S\$$ because we want \$ to be in the *FOLLOW* set of a nonterminal when the next input can be the end-of-file marker. The domain of *FOLLOW* is just the set of nonterminals (not strings, in contrast to the previous functions). The *FOLLOW* sets are computed in the same style as *Nullable* and *FNE*. Initially, the *FOLLOW* sets are all empty, then the following rules are applied in any order until no more changes are possible.

1. $\$ \in FOLLOW(S)$
2. $FOLLOW(B) \supseteq FNE(\beta)$ when $A \to \alpha B \beta$ appears in the CFG.
3. $FOLLOW(B) \supseteq FOLLOW(A)$ when $A \to \alpha B \beta$ appears in the CFG and β is nullable.

Rule 1 is justified since S derive a complete input string, which will be followed by \$. Rule 2 looks for productions where B is followed by something that can have a at the beginning (so a immediately follows whatever B expands to). Rule 3 deals with a more subtle case. If a is in $FOLLOW(A)$, and β is nullable, the follow derivation exists: $S \overset{*}{\Longrightarrow} \ldots Aa \ldots \Longrightarrow \ldots \alpha B \beta a \ldots \overset{*}{\Longrightarrow} \ldots \alpha B a \ldots$, so a is in $FOLLOW(B)$, too.

Let's compute the *FOLLOW* sets for the LL(1) expression grammar.

String	Added	Reason
E	\$	rule 1
E	)	rule 2 ($F \to (E)$)
T	+	rule 2 ($E \to TA$)
F	*	rule 2 ($T \to FB$)
A	\$,)	rule 3 ($E \to TA$, ϵ nullable)
T	\$,)	rule 3 ($E \to TA$, A nullable)
B	+,\$,)	rule 3 ($T \to FB$, ϵ nullable)
F	+,\$,)	rule 3 ($T \to FB$, B nullable)

The resulting *FOLLOW* sets are:

E	$\{ \$, \ ) \}$
T	$\{ +, \ \$, \ ) \}$
F	$\{ *, \ +, \ \$,) \}$
A	$\{ \$, \ ) \}$
B	$\{ +, \ \$, \ ) \}$

The LL(1) parse table is a two-dimensional array, which we will call *Table*. The rows are indexed by nonterminals (for what is on the top of the stack) and the columns by terminals (the next input symbol). Given A on the top of the stack and a next in the input, we need to choose a production $A \to \alpha$ to expand. $\mathit{Table}[A, a]$ should be $A \to \alpha$ only if there is some way to expand $A \to \alpha$ that can match a next in the input.

There are two ways that expanding $A \to \alpha$ can expand to something that matches a. One way is if α expands to something beginning with a, so we set $\mathit{Table}[A, a] = A \to \alpha$ whenever $a \in \mathit{FNE}(\alpha)$. The other way is if α expands to ϵ, and some symbol after α expands to a string beginning with a, so we also set $\mathit{Table}[A, a] = A \to \alpha$ when α is nullable and $a \in \mathit{FOLLOW}(A)$.

The LL(1) parse table for the LL(1) expression grammar appears in full, above, but let's look at a few examples. $E \to TA$ appears in $\mathit{Table}[E, a]$ because $a \in \mathit{FNE}(TA)$. The only productions with nullable right-hand sides in this grammar are $A \to \epsilon$ and $B \to \epsilon$. $\mathit{Table}[B, +] = B \to \epsilon$ because $+ \in \mathit{FOLLOW}(B)$.

There is one more extremely important point to make about the LL(1) parse table: If the rules above set $\mathit{Table}[A, a]$ to two different productions, the parse table construction fails. In this case, the CFG is *not LL(1).* LL(1) parsing demands that we be able to choose exactly the next production to expand based solely on whether it is consistent with the next input symbol. So the LL(1) parse table construction not only builds parse tables, it is a test for whether a CFG is LL(1) or not.

What about the entries that have *nothing* in them? They are *error* entries: if the parser ever looks at them, the input string can be rejected immediately. There is no way to expand anything that can match the next input.

In spite of its limitations, LL(1) parsing is one of the two most widely used parsing algorithms. The parsers can be built automatically, and the parsing algorithm is easy to understand. It is usually simple to do processing associated with individual grammar rules, during the parsing process (this is called *syntax-directed translation*). Furthermore, it can also be used as the basis for simple hand-written parsers, which allow a great deal of flexibility in handling special cases and in error recovery and reporting (see below). However, LL(1) parsing has some limitations that can be annoying. One of them is that it is not as general as the other common parsing algorithm, so there are some common grammatical constructs it cannot handle. Another is the requirement to eliminate left recursion and left factors, which can require rearranging a grammar in ways that make it less clear.

LL(1) parsing and recursive descent

LL(1) parsing can be used with an automatic parser generator. It is also appropriate as a basis for a simple hand-written parsing style, called *recursive descent*. The idea behind recursive descent parsing is to write a collection of recursive functions, one associated with each nonterminal symbol in the grammar. The function for a nonterminal is responsible for reading and parsing the part of the input string that that nonterminal expands to.

The parsing function for B in our LL(1) expression grammar might be:

```
int parse_B() {
  next = peektok();   /* look at next input, but don’t remove it */
  if (next == ’*’) {
    gettok();   /* remove ’*’ from input */
    parse_F();  /* should check error returns for these, */
    parse_B();  /* but I want to keep the code short */
    return 1;   /* successfully parsed B */
  }
  else if ((next == ’+’) || (next == ’)’) || (next == EOF)) {
    return 1;      /* successfully parsed B */
  }
  else {
    error("got %s, but expected *, +, ) or EOF while parsing B\n", next);
    return 0;
  }
}
```

Recursive descent parsing is very widely used, because it requires no special parser generation tools, it can be extended in *ad hoc* ways (for example, looking ahead to several inputs, or looking at other context, when the next input does not uniquely determine the production choice), and it allows the user great freedom in generating error messages and doing error recovery.

A looser style allows parsing of EBNF directly. For example:

```
int parse_T() {
  parse_F();  /* again, I should check the return code */
  while ((next = peektok()) == '*') {
    gettok();   /* remove '*' from input */
    parse_F();
  }
}
```

While substantially different from the previous code, this is still basically the same thing (it chooses whether to parse F or not at each point based on one lookahead symbol.

4 Bottom-up parsing

The other major class of parsing methods are the *bottom-up* algorithms. As the name suggests, bottom-up methods work by building a parse tree from the leaves up. This involves reading the input until the right-hand side of a production is recognized, then *reducing* the production instead of expanding productions until they match the input.

Shift-reduce parsing algorithms

The bottom-up algorithms we will study are all *shift-reduce* algorithms. Shift-reduce parsing uses the same data structures as LL parsing: a stack and the input stream. However, the stack is used in a different way. Terminal symbols are *shifted* onto the stack, that is, a symbol is removed from the beginning of the input and pushed onto the stack. If a sequence of symbols on the top of the stack matches the right-hand side of some production, they can be *reduced* by popping them and then pushing the symbol from the left-hand side of the same production. The parse is successful when the entire input has been shifted on the stack and reduced to the sentence symbol of the grammar.

As with LL parsing, there are choices to be made during this algorithm: there may be several productions whose right-hand sides match the stack at any time. Which one should be reduced? As with LL parsing, the choice is made by doing a table lookup with information from the current state of the parse.

To show how the basic parsing algorithm works, we do a simple example where we hide the details of the parse table and, instead, make the choices by guessing. Consider the simple CFG:

$$\begin{array}{lcl} S & \rightarrow & (S) \\ S & \rightarrow & a \end{array}$$

Here is the sequence of parser configurations that happens when parsing "$((a))$." The top of stack will be on the right to make the shifting more obvious.

Stack (top)	input	action
\$	((a))\$	shift
\$(	(a))\$	shift
\$((	a))\$	shift
\$((a	))\$	reduce $S \to a$
\$((S	))\$	shift
\$((S)	)\$	reduce $S \to (S)$
\$(S	)\$	shift
\$(S)	\$	reduce $S \to (S)$
\$S	\$	accept

We can extract a derivation and parse tree from a shift-reduce parse. The sequence of reductions represents the *reverse* of a derivation. In this case, it is $S \Longrightarrow (S) \Longrightarrow ((S)) \Longrightarrow ((a))$. Although the example grammar doesn't show it, it is also a *rightmost* derivation. To see this, consider the CFG:

$$\begin{array}{lcl} S & \to & AB \\ A & \to & a \\ B & \to & b \end{array}$$

The only string in the language is ab. Here is a parse:

Stack (top)	input	action
\$	ab\$	shift
\$a	b\$	reduce $A \to a$
\$A	b\$	shift
\$Ab	\$	reduce $B \to b$
\$AB	\$	reduce $S \to AB$
\$S	\$	accept

Although $A \to a$ is reduced before $B \to b$, the derivation is reversed, so we end up with $S \Longrightarrow AB \Longrightarrow Ab \Longrightarrow ab$, a rightmost derivation.

4.1 LR(0) parsing

The first shift-reduce parsing algorithm we will discuss is *LR(0) parsing.* It is an instance of LR(k) parsing (which stands for "left-to-right (parsing), rightmost (derivation) (with lookahead) of k symbols"). LR(0) parsing is pretty much useless as a stand-alone parsing algorithm, but it is the basis for other extremely useful algorithms.

The key idea in all LR parsing algorithms is to run a finite-state automaton from the bottom of the parse stack to the top. The state of this automaton (at the top of the stack) is used, along with some lookahed symbols, to choose whether to shift another symbol or reduce a production, and (if the latter) which production to reduce. The construction of the finite automaton is somewhat involved.

The states of the automaton (which we will call the *LR(0) state machine*) consist of *LR(0) items.* An LR(0) item is a production with a position marked in it.

Definition 11 *An* LR(0) item *is a pair consisting of a production $A \to \alpha$ and a position i, where $0 \leq i \leq |\alpha|$, where $|\alpha|$ is the length of α.*

An item is written $A \to \alpha \bullet \beta$, where $\bullet$ marks the position. For example, $A \to \bullet ab$, $A \to a \bullet b$, and $A \to ab\bullet$ are all items. An ϵ production has only one item, which is written $A \to \bullet\epsilon$. Note that this would be equivalent to $A \to \epsilon\bullet$, if we ever wrote that, which we don't.

The finite-state automaton in this case is called the *LR(0) machine.* Each state of the LR(0) machine is a set of items. If two states have the same set of items, they aren't two states – they are the same state. Intuitively, the LR(0) machine keeps track of the productions that might eventually be reduced when more stuff is pushed on the stack. The positions keep track of how much of the production is already on the stack.

For convenience, the first step of any LR parsing algorithm is to add a new sentence symbol S' and a new production $S' \to S$ to the CFG. Let's use the first example CFG above.

$$\begin{array}{rcl} S' & \to & S \\ S & \to & (S) \\ S & \to & a \end{array}$$

The first state starts with the item that says: "we are parsing a sentence, and we haven't seen anything yet:" $S' \to \bullet S$.

The construction of a state starts with a *kernel*, which is a core set of items. The kernel of the our first state is $S' \to \bullet S$. The *kernel* is extended via the *closure* operation. The closure operation takes into account that whenever we are in the middle of an item $A \to \alpha \bullet B\beta$ (meaning "I might eventually be able to reduce $A \to \alpha B\beta$, and α is already on the stack"), it could be that the parser will see something that can be reduced to B. The items that reflect this are those of the form $B \to \bullet\gamma$ (meaning "I might eventually be able to reduce $B \to \gamma$, and I haven't seen anything in γ yet"). These are called *closure items.*

Definition 12 *The* LR(0) closure step *adds all items of the form $B \to \bullet\gamma$ to a state whenever the state contains an item $A \to \alpha \bullet B\beta$.*

There are two important points to make about the closure step:

- When a closure item begins with a nonterminal, adding it to the state may cause additional closure items to be added.
- The state is a *set* of items, which means it has no duplicates – "adding" items that already there has no effect.

To be explicit, the procedure for closure is:

```
repeat until no change
    if there is an item $A \to \alpha \bullet B\beta$ in the state add $B \to \bullet\gamma$ to the state
    for all productions $B \to \gamma$ in the grammar
```

Our first LR(0) state has a kernel of $S' \to \bullet S$. Closure introduces items $S \to \bullet(S)$ and $S \to \bullet a$. No additional items can or should be added (there are no more dots in front of non-terminals). So the first state is:

$$\boxed{\begin{array}{rcl} S' & \to & \bullet S \\ S & \to & \bullet(S) \\ S & \to & \bullet a \end{array}}$$

A box is drawn around the state to indicate that the closure operation has been performed.

To complete the state machine, we need to define the transitions between the states. This is done by the *goto* function. If our state is q, whenever there is at least one item of the form $A \to \alpha \bullet X\beta$, where X is a terminal or nonterminal, $goto(q, X)$ is defined. It is the set of all items $A \to \alpha X \bullet \beta$ where $A \to \alpha \bullet X\beta$ is an item in q. *Important note: Never do this to items like $A \to \bullet\epsilon$, which is a reduce item. The goto function doesn't apply to such* *productions, because ϵ is contains neither terminal nor nonterminal symbols. I would love to have a quarter where no students add transitions from $A \to \bullet\epsilon$ to $A \to \epsilon\bullet$.*

For each symbol X, *goto* generates the kernel of a successor state to q. There are several important points to notice:

- For a given symbol X, *goto* operates on *all* of the items where X is the next symbol. There is only one successor on each X.
- If $q' = goto(q, X)$, all of the items in q' are of the form $A \to \alpha X \bullet \beta$. I.e., the $\bullet$ is always immediately after an X.

Once the kernels of the successor states have been generated, the closure operation is applied to each to complete the set of items in the state. Given an LR(0) state, it is possible to determine which of its items were in its kernel and which were introduced by closure by checking whether the $\bullet$ is at the beginning of the item (closure) or not (kernel) (the one exception to this rule is the item that started everything, $S' \to \bullet S$, which is introduced by neither operation).

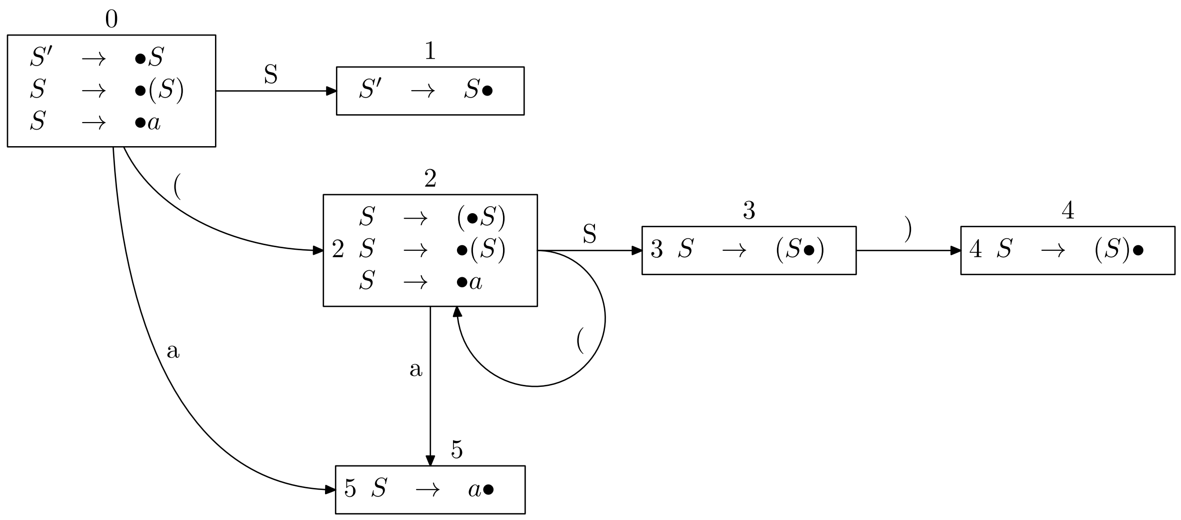
If, after applying closure, the set of items is exactly the same as some other set of items, the two states are actually the same state. (An optimization to this is to look at whether the kernel generated by *goto* is the same as the kernel of an existing state, in which case the states can be merged before wasting a closure operation.)

Let's apply the *goto* operation to the one state we have generated so far (let us call it 0). There are three symbols of interest: "S", "(", and "a". In each case, there is only one item with that symbol next.

$$\begin{aligned} goto(0, S) &= \{S' \to S\bullet\} \\ goto(0, \text{“(”}) &= \{S \to (\bullet S)\} \\ goto(0, a) &= \{S \to a\bullet\} \end{aligned}$$

In the resulting DFA, there is a transition from state q_i to q_j on symbol X if and only if $q_j = closure(goto(q_i, X))$. The initial state of the DFA is the one whose kernel is $S' \to \bullet S$.

The complete LR(0) machine for the current example is:



Let's consider state 2 in more detail. The state was first generated by $\mathit{goto}(0, '(') = \{S \to (\bullet S)\}$; this is the kernel of state 2. Because of the $\ldots \bullet S \ldots$ in the kernel item, *closure* adds items $S \to \bullet(S)$ and $S \to \bullet a$, completing the state. $\mathit{goto}(2, '('))$ also generates $\{S \to (\bullet S)\}$, so at this point, we know we are going to state 2 (looping back to it, actually). Or we wait until we generate the closure of our "new" state, and then notice that we have exactly the same items we had in state 2 and merge the states. State 2 is different from state 0 because the kernel items are different (even though the closure items are the same).

It is illuminating to see how this machine directs parsing. When an item has $\ldots \bullet a \ldots$ for some *terminal* a, we call it a *shift item.* It says that a should be shifted onto the stack if it appears as the next input symbol.

An item of the form $A \to \alpha\bullet$ is a *reduce item.* It indicates that, when this state is reached, the production $A \to \alpha$ should be reduced (α will be guaranteed to be on top of the stack if the parser gets to this state). Reducing the item $S' \to S\bullet$ *accepts* the input string. *Important: an item like $A \to \bullet\epsilon$ is a reduce item; since the RHS is the empty string, position 0 is the end of the item.*

Unlike the example of shift-reduce parsing, an LR(0) parser does not actually shift symbols onto the stack. Instead, it shifts states. No information is lost because of the special structure of the LR(0) machine. Notice that every transition into a state has exactly the same symbol labelling it. If you see a state q on the stack, it is as though a were shifted onto the stack. The stack will also have states corresponding to nonterminal symbols.

In more detail, the LR(0) parsing algorithm starts with the first state (0 in our example) and executes the following steps repeatedly:

shift If the next input is a and there is a transition on a from the top state on the stack (call it q_i) to some state q_j, push q_j on the stack and remove a from the input.

reduce If the state has a reduce item $A \to \alpha\bullet$

1. Pop one state on the stack for every symbol in α (note: symbols associated with these states will *always* match symbols in α).
2. Let the top state on the stack now be q_i. There will be a transition in the LR(0) machine on A to a state q_j. Push q_j on the stack

error If the state has no reduce item, the next input is a, and there is no transition on a, report a parse error and halt.

accept When the item $S' \to S\bullet$ is reduced accept if the next input symbol is \$, otherwise report an error and halt. (This rule is a bit weird. The remaining LR-style parsing algorithms don't need to check if the input is empty. We define it this way so LR(0) parsing can do our simple example grammar.)

LR(0) parsing requires that each of these steps be uniquely determined by the LR(0) machine and the input. Therefore, if a state has a reduce item, it *must not* have any other reduce items or shift items. With this restriction, the current state determines whether to shift or reduce, and which production to reduce, without looking at the next input. If it shifts, it can read the next input to see which state to shift.

Let's parse the input "((a))" using this LR(0) machine.

Stack (top)	input	action
0 \$	((a))\$	shift 2
02 \$(	(a))\$	shift 2
022 \$((	a))\$	shift 5
0225 \$((a	))\$	reduce $S \to a$
0223 \$((S	))\$	shift 4
02234 \$((S)	))\$	reduce $S \to (S)$
023 \$(S	)\$	shift 4
0234 \$(S)	\$	reduce $S \to (S)$
01 \$S	\$	accept

At each step, we have listed the symbols associated with the states on the stack (associating the "bottom of stack" symbol, \$, with state 0). Let's look at the reductions of $S \to (S)$ in more detail. When the first such reduction occurs, the stack is 02234; three symbols are popped of (because the length of "(S)" is 3), leaving a stack of 02. There is a transition from the top state, 2, on S to state 3, so we push a 3, leaving 023 on the stack. The second time it reduces $S \to (S)$, the stack is 0234. When three states are popped, this leaves a stack with just 0 on it(so top-of-stack state is different this time). There is a transition from state 0 to state 1 on S, so the new stack is 01. At this point, the action is to reduce $S' \to S$ and the input has been consumed, so the parser accepts. (It helps to visualize parsing by tracing the top-of-stack state in the diagram with your finger as you step through the parse.)

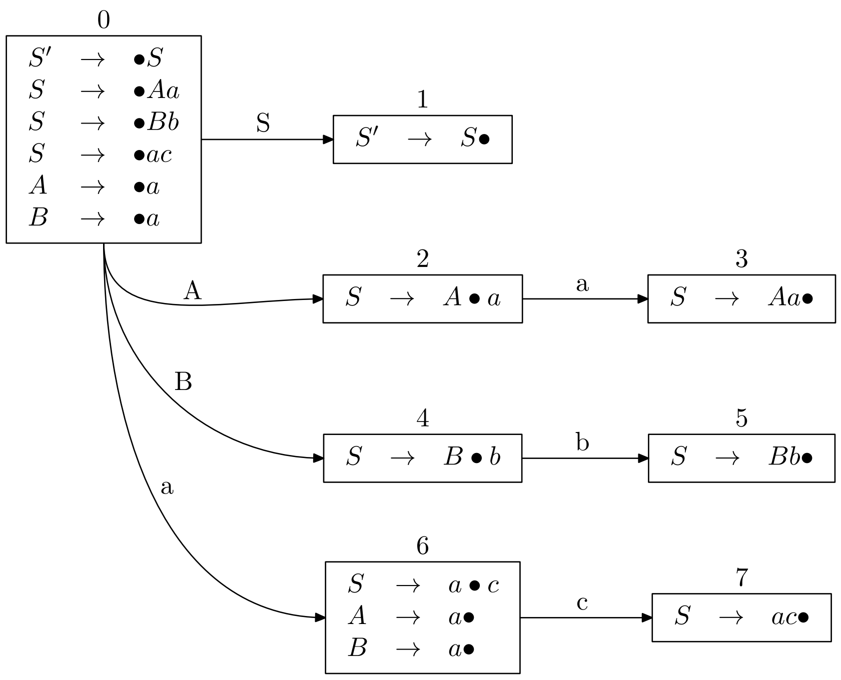
SLR(1) parsing

Here is an example of a CFG that is not LR(0):

$$\begin{array}{llll}
0 & S' & \to & S \\
1 & S & \to & Aa \\
2 & S & \to & Bb \\
3 & S & \to & ac \\
4 & A & \to & a \\
5 & B & \to & a
\end{array}$$

The productions are numbered because the numbers are used in the parse table construction below.

Here is the LR(0) machine that results



The machine is not LR(0) because of shift/reduce and reduce/reduce conflicts in state 6 (there is a shift item and two reduce items in the state, so the parser doesn't know whether to shift or reduce, and if it decided to reduce, anyway, it wouldn't know which production to reduce). Hence, this grammar is not LR(0).

However, if we allowed the parser to base its choice on the next input symbol, the correct choice could be made reliably. If you examine the grammar carefully, you can see that $A \to a$ should only be reduced when the next input is a, $B \to a$ should only be reduced when the next input is b, and, if the next input is c, the parser should shift.

How could we determine this algorithmically? The next three parsing algorithms all do it in different ways. The simplest method is SLR(1) parsing, which uses FOLLOW sets to compute lookaheads for actions. Using the rules for computing FOLLOW sets in LL(1) parsing, we compute $\mathit{FOLLOW}(S) = \{\$\}$, $\mathit{FOLLOW}(A) = \{a\}$, and $\mathit{FOLLOW}(B) = \{b\}$. We can then associate each reduce item with a *lookahead set* consisting of the FOLLOW set of the symbol on the left-hand side of the item. State 6 would then look like:

$$\begin{array}{lcll} S & \to & a \bullet c & \\ A & \to & a\bullet, & \{a\} \\ B & \to & b\bullet, & \{b\} \end{array}$$

Since the lookahead sets for each shift item are disjoint from each other, and disjoint from the symbols that can be shifted, this state has no *SLR(1) conflicts.*

The parse table for SLR(1) has the same format as for the two other shift-reduce parsing algorithms that are going to be discussed, LR(1) and LALR(1). The parse table consists of two two-dimensional arrays: an *ACTION* table and a *GOTO* table. Here is the SLR(1) parse table for the CFG above.

	ACTION				*GOTO*		
	a	b	c	$	S	A	B
0	s6				1	2	4
1				acc			
2	s3						
3				r1			
4		s5					
5				r2			
6	r4	r5	s7				
7				r3			

The rows of both tables are indexed by the states of the LR(0) machine. The columns of the *ACTION* table are indexed by terminal symbols and the end-of-file marker $. The columns of the *GOTO* table are indexed by nonterminals.

The entries of the *ACTION* table are *shift actions*, of the form sn, where n is index of an LR(0) state, or rp, where p is the index of a production in the CFG. At each step, the parser looks in *ACTION*$[q, a]$, where q is the current LR(0) state and a is the next input symbol. If the entry is "sn," it shifts state number n onto the stack. If the entry is "rp," it reduces production p. If there is no entry in the *ACTION* table, a parse error is reported (the input is not in the language of the CFG).

The *GOTO* table gives the next-state transition function for nonterminals (given a current LR state and a nonterminal, it gives the next LR state). It is used during reductions: after popping states for the right-hand side of a production, it designates the state to be pushed for the left-hand side symbol. Empty entries in the *GOTO* table are never referenced, even if the input string is not parse-able (if there were a parse error, it would have been caught earlier when an empty entry of the *ACTION* table was referenced).

As with LL(1) parsing, all LR parsing algorithms require that the table *uniquely* determine the next parse action: there must be at most one action in each entry of the parse table, or the CFG cannot be handled by the parsing algorithm, in which case the CFG is said to be "not SLR(1)" (or LR(1) or LALR(1)). When there are multiple actions in a table entry, there is said to be a conflict. There can be *shift/reduce* conflicts (if there is a shift and a reduce action in the same table entry), or *reduce/reduce conflicts* (if there are several reduce actions in the same table entry). There is no such thing as a *shift/shift* conflict, because the

LR machine *goto* operation ensure that each state has at most one successor. Conflicts only show up in the *ACTION* table, not the *GOTO* table.

The above table has no conflicts. The most interesting row is for state 6 (the one with the LR(0) conflicts). Observe that the row has two different reduce actions and a shift action, yet they do not conflict because they are in different columns (because their lookahead symbols are disjoint).

LR parse algorithm

The table-driven version of the SLR(1) parsing algorithm is exactly the same for LR(1) and LALR(1) parsing. The differences among these algorithms are in the table constructions.

When production p is reduced, the parser looks up the production, and pops one state off of the stack for each symbol on the right-hand side of production p. Then it looks up an entry in the *GOTO* table. The columns of the *GOTO* table are indexed by nonterminal symbols, and the entries in the table are the indexes of LR(0) states. The parser pushes onto the parse stack the state in $GOTO[n, A]$, where n is the LR(0) state on top of the stack (after popping one state for each right-hand symbol), and A is the nonterminal on the left-hand side of production p.

We assume state 0 is the *start state*, whose kernel is $\{S' \to \bullet S\}$. Initially, the stack has state 0 and nothing else on it, and the input is the input string followed by the end-of-file marker, \$.

1. Let q to the top state on the stack, let a be the next input symbol, and let *act* be $ACTION[q, a]$.
2. If *act* is "sn", push n on the stack and remove a from the input;
3. Else, if *act* is "rp", and supposing production p is $A \to \alpha$,
   (a) Pop $length(\alpha)$ states off of the stack. Let q' be the top of stack symbol immediately after doing this.
   (b) Push $GOTO[q', A]$ onto the stack;
4. Else, if *act* is "acc", accept the input;
5. Else, if *act* is "error" (an empty entry), report an error (the input string is not in the language of the CFG).

The "accept" action only ever appears in the column \$, so it will only be found when the input has been exhausted. Unlike LR(0) parsing, it is not necessary to have a separate check to see if the input is empty before accepting.

Example SLR(1) parse

As an example, let us parse the input "ab" using our CFG and table.

Stack (top)	input
0 \$	ab\$
06 \$a	b\$
04 \$B	b\$
045 \$Bb	\$
01 \$S	\$
accept	

Filling in the parse tables

The *ACTION* table is filled in according to the following rules.

- If there is a transition from state q to state number n on terminal symbol a, set $ACTION[q, a] = sn$.
- If there is a reduce item $A \to \alpha\bullet$ in state q and a is in $FOLLOW(A)$, set $ACTION[q, a] = rp$, where p is the number of $A \to \alpha$, unless the item is $S' \to S\bullet$, in which case set $ACTION[q, a] = acc$ (in this case $a = \$$).
- If $ACTION[q, a]$ already has an entry when one of the rules above applies, report that the CFG is "not SLR(1)" and halt.

The definition *GOTO* table is quite simple: if there is a transition from state q to state n on nonterminal A in the LR(0) machine, set $GOTO[q, A] = n$.

You should reconstruct the table above to see how the rules apply to it. We also recommend that you try doing the SLR(1) parse table construction for the CFG $S \to Sa \mid \epsilon$.

Using ambiguous grammars

The family of shift-reduce parse algorithms describe here all fail for ambiguous grammars. Ambiguity means that, in at least one place in a parse, more than one shift or reduce action can lead to a successful parse, so multiple conflicting actions will appear at some point in the parse table.

However, there is a trick for resolving conflicts in LR parse tables which sometimes "works," meaning that it results in a conflict-free parse table that still parses the language of the original grammar. The trick is especially useful for dealing with certain ambiguous grammars, and can lead to parsers that are smaller and more efficient than a parser built from an unambiguous grammar. However, the trick is dangerous, because it is not obvious whether the resulting parser actually parses the intended language.

The basic idea is this: whenever there are multiple actions in a particular entry of the *ACTION* table, delete all but one of them so that the correct precedence and associativity is enforced in the parse tree.

Example

The first expression grammar we gave was highly ambiguous:

$$\begin{array}{lcl} E & \rightarrow & E + E \\ E & \rightarrow & E * E \\ & & \ldots \end{array}$$

If you try to construct the SLR(1) parser for this language, there will be many conflicts (*try it!*). One SLR(1) state is:

$$\begin{array}{lcl} E & \rightarrow & E + E\bullet \\ E & \rightarrow & E \bullet + E \\ E & \rightarrow & E \bullet * E \end{array}$$

$FOLLOW(E) = \{+, *,), \$\}$, so the SLR(1) lookahead symbols don't help.

The row from the *ACTION* table would be:

+	*	)	$
r1/s6	r1/s7	r1	r1

The "s6" and "s7" are made-up state numbers, representing the next states for $+$ and $*$. The SLR(1) parsing algorithm will report two shift/reduce conflict in this state, with the reductions and lookahead symbols involved.

Intuitively, the r1/s6 conflict is whether to make $+$ left associative or right associative. If we reduce $E \to E + E$, and we're parsing $1 + 2 + 3$, it is going to group it as $[1 + 2] + 3$. If we shift instead, we'll reduce the $2 + 3$ later, *then* $1 + [2 + 3]$, so we'll get the right-associative grouping. The conflict stems directly the ambiguity in the grammar.

Similarly, the conflict on $*$ is about the relative precedence of $+$ and $*$. If we reduce, $1+2*3$ will be grouped $[1 + 2] * 3$, while shifting will group it as $1 + [2 * 3]$.

In this case, the conflicts can be *resolved* by the user by selectively removing table entries. To make $+$ left associative and make the precedence of $+$ greater than $*$, the row from the table should be:

$+$	$*$	)	\$
r1	s7	r1	r1

Another LR(0) state that arises is:

$$\begin{array}{lll} E & \to & E * E\bullet \\ E & \to & E \bullet +E \\ E & \to & E \bullet *E \end{array}$$

The lookahead symbols for $E \to E * E$ are the same as for $E \to E + E$, because they are based on $FOLLOW(E)$, so the table will have conflicts between both shift items and the reduce item. In this case, we should favor reducing $E \to E * E$ over shifting $+$, because $*$ has higher precedence than $+$ (when parsing $1 * 2 + 3$, it will reduce after it sees $1 * 2$, which will group as $[1 * 2] + 3$). We should also favor reducing $E \to E * E$ over shifting $*$, if we want $*$ to be left associative.

Parser generators of the YACC family have a way for the user to declare for each operator whether it is left or right associative, and its precedence relative to other operators. Shift/reduce conflicts are resolved automatically by comparing the next terminal in the shift item with the rightmost terminal in the reduce item[1], to see which has the highest precedence, or, if the operators are the same, whether they are left or right associative. The reduce item is favored if its precedence is greater than the shift item or if the precedences are the same and its operator is left associative.

[1]There is also a mechanism to declare the precedence of a production to be the same as a token explicitly.

There are also efficiency advantages, compared with writing an unambiguous grammar. The unambiguous grammar is usually much larger than the ambiguous grammar, and its LR state machine larger still. So the parse table for the ambiguous grammar may be much smaller than that for the unambiguous grammar. This is a space advantage, which may translate into a speed advantage when certain types of table compaction are used. A more direct reason that the ambiguous grammar may be faster to parse is that it requires a lot fewer reductions of *unit productions*. A unit production is one whose left-hand side is a single nonterminal. Our unambiguous expression grammar had two of these $E \to T$ and $T \to F$. In general, unambiguous expression grammars make heavy use of unit productions, which may double or triple the number of reductions that happen during parsing. On the other hand, although these efficiency advantages are real, it is not clear how important they are given that computers are so much faster and and so much more memory than when these parsing algorithms were invented. For most application, LR parsing is so blindingly fast that doubling the speed is not noticeable.

The use of these rules is dangerous. They can be used safely and advantageously in some circumstances. For example, there is a long tradition of defining expression syntax with prefix and postfix unary operators and binary operators, with explicit precedence rules. The precedence rules can be implemented very naturally using the above mechanism. Otherwise, you need to think through the consequences of conflict resolution *very, very carefully.* But thinking carefully might not be sufficient to get it right, as I know from a few personal experiences.

Note that all uses of the conflict resolution mechanism above resolve shift-reduce conflicts. Legitimate uses of this mechanism for reduce-reduce conflicts are extremely rare.

LR(1) parsing

The most powerful parsing method I will discuss is LR(1) parsing. LR(1) parsing uses the same parsing algorithms as SLR(1) parsing, but uses a different state machine: *the LR(1) machine.* The state machine keeps more information in its states and computes lookaheads for items more accurately, so the resulting LR(1) parse table is less likely to have conflicts. Hence, some CFGs are LR(1) (meaning that the parse table has no conflicts), that are not SLR(1). LR(1) parsing is not widely used because there is not much practical difference in power between LR(1) and the next parsing method we'll discuss, LALR(1), and LALR(1) parse tables are much smaller than LR(1). However, LALR(1) parsing is based on LR(1) parsing.

Here is a CFG that is not SLR(1) but is LR(1):

$$\begin{array}{lcl} S & \rightarrow & Aa \\ S & \rightarrow & Bb \\ S & \rightarrow & bAb \\ A & \rightarrow & a \\ B & \rightarrow & a \end{array}$$

The first state of the LR(0) machine and state reached from it via goto on a are:

$$\boxed{\begin{array}{rcl} S' & \rightarrow & \bullet S \\ S & \rightarrow & \bullet Aa \\ S & \rightarrow & \bullet Bb \\ S & \rightarrow & \bullet bAb \\ A & \rightarrow & \bullet a \\ B & \rightarrow & \bullet a \end{array}} \qquad \boxed{\begin{array}{rcl} A & \rightarrow & a\bullet \\ B & \rightarrow & a\bullet \end{array}}$$

Note that the second state has an LR(0) conflict, because there are two reductions. In this case, $FOLLOW(A) = \{a, b\}$ and $FOLLOW(B) = \{b\}$, so b is in the SLR(1) lookahead of both reduce items. When we try to build the table, there will be two reduce actions in the row for this state, column b. When the SLR(1) parse table has more than one action in an entry, the actions are *SLR(1) conflicts*.

SLR(1) lookahead sets are not as accurate as they could be. In fact, the lookahead set for an item really depends on the symbols that were shifted before the current state. The CFG above has two occurrences of A. If A occurs at the beginning of the input (after some reductions, of course), it must be followed by a if the parse is to succeed. However, if A occurs right after b, it must be followed by b. If the LR states kept track of this context, we would notice that there could never be a b after A *in the particular state where the conflict occurs.* This would remove the conflict.

The LR(1) machine construction keeps track of context-dependent lookahead information by putting the lookahead symbols in the items during the *closure* operation.

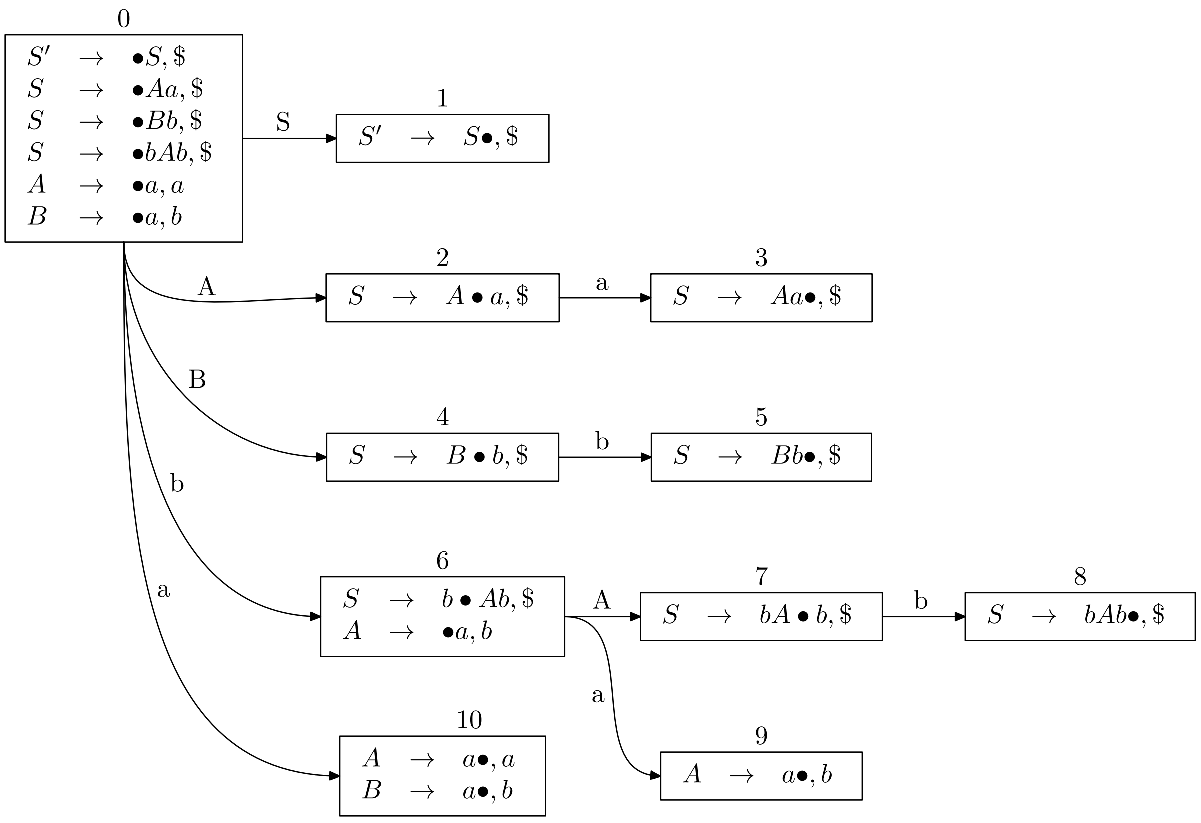
Definition 13 *An* LR(1) item *is a triple consisting of a production, a position in the right-hand side of the production, and a* lookahead symbol*, which is a terminal or \$.*

LR(1) items look like $[A \rightarrow \alpha \bullet \beta, a]$, where a is the lookahead symbol. Intuitively, this means "we have seen α and we expect possibly to see β, followed by a." The LR(1) state machine construction is similar to the LR(0) construction, except that the LR(1) states are sets of LR(1) items. *This means that one state in the LR(0) machine may be split into several states* *in the LR(1) machine, with the same sets of LR(0) items but different lookaheads.* Hence, the LR(1) state machine is more accurate, but potentially larger, than the LR(0) machine.

Here are the changes to the LR(0) state machine construction:

1. The first item is $[S' \to \bullet S, \$]$. Obviously, the only symbol that can occur after S *in this context* is end-of-file.
2. The *CLOSURE* operation is modified to compute lookaheads of items: Whenever $[A \to \alpha \bullet B\beta, a]$ appears in a state and $b \in FNE(\beta a)$, add $[B \to \bullet\gamma, b]$ to the state for every production $B \to \gamma$ in the CFG. Intuitively, $A \to [\alpha \bullet B\beta, a]$ means "we have seen α and expect possibly to see $B\beta a$. Therefore, we should also expect to see γ followed by the first terminal derived from βa.
3. The *GOTO* operation is unchanged, except that it copies LR(1) items, with their lookaheads unchanged, instead of LR(0) items.

Here is the full LR(1) machine construction. Note that the more precise lookaheads have eliminated all conflicts. In this case, the states correspond exactly to LR(0) states, but this is unusual. Once we have the machine, the parse table construction is exactly as in SLR(1) parsing, so the parse table is not shown.



LALR(1) parsing

LALR(1) parsing is probably the most widely used automatically generated bottom-up parsing algorithm. It is almost as powerful as LR(1) parsing, but has parse tables that are much smaller than full LR(1) tables. The LALR(1) machine can be constructed by first building the full LR(1) machine, then *merging states that have identical sets of LR(0) items.* The result is an LR(0) machine with more precise lookahead information than SLR(1).

The following CFG is LR(1) but *not* LALR(1).

$$\begin{array}{rcl} S' & \rightarrow & S \\ S & \rightarrow & Aa \\ S & \rightarrow & Bb \\ S & \rightarrow & bAb \\ S & \rightarrow & bBa \\ A & \rightarrow & a \\ B & \rightarrow & a \end{array}$$

The leftmost state below is the initial state of the LR(1) machine for this CFG. The middle state is the state that is reached via *goto* on a from the initial state, and the rightmost state is reached by doing *goto* on b, then *goto* on a.

$$\boxed{\begin{array}{rcl} S' & \rightarrow & \bullet S, \$ \\ S & \rightarrow & \bullet Aa\$ \\ S & \rightarrow & \bullet Bb\$ \\ S & \rightarrow & \bullet bAb\$ \\ S & \rightarrow & \bullet bBa\$ \\ A & \rightarrow & \bullet a, a \\ B & \rightarrow & \bullet a, b \end{array}} \qquad \boxed{\begin{array}{rcl} A & \rightarrow & a\bullet, a \\ B & \rightarrow & a\bullet, b \end{array}} \qquad \boxed{\begin{array}{rcl} A & \rightarrow & a\bullet, b \\ B & \rightarrow & a\bullet, a \end{array}}$$

In this case, the sets of LR(0) items in the middle and right states are identical. Merging two states with identical LR(0) item sets computes the union of the LR(1) items of the states. If states 1 and 2 are merged, all predecessors of 1 and 2 *goto* the merged state in the LALR(1) state machine.

In this example, the merged states are:

$$\boxed{\begin{array}{rcl} A & \rightarrow & a\bullet, a \\ A & \rightarrow & a\bullet, b \\ B & \rightarrow & a\bullet, a \\ B & \rightarrow & a\bullet, b \end{array}}$$

As a notational convenience, we generally combine the lookaheads for identical LR(1) items. This state contains four items, but is written as:

$$\boxed{\begin{array}{lcl} A & \rightarrow & a\bullet, \{a, b\} \\ B & \rightarrow & a\bullet, \{a, b\} \end{array}}$$

In the merged state, we have reduce/reduce conflicts on both a and b (the lookahead sets for the reduce items are no longer disjoint). There were no conflicts in the original LR(1) machine, so this example shows that LR(1) parsing can handle more grammars than LALR(1) parsing. However, in practice, the difference is almost never important. Generally, when a CFG for a real programming language fails to be LALR(1), it also fails to be LR(1). LALR(1) parse tables are much smaller than LR(1) parse tables (by a factor of 10 in typical examples), so LALR(1) has emerged as the dominant LR parsing algorithm. It represents good engineering compromises between efficiency and generality, and matches the needs of programming languages very well.

The previous CFG was not only LR(1) but also LALR(1). The LR(1) machine has no two states with the same LR(0) items, so no states will be merged.

The relative power of LR parsing algorithms

LR(0) parsing is a weak algorithm that handles almost nothing. Any LR(0) CFG is also SLR(1), because SLR(1) parsing cannot introduce new conflicts in an LR(0) grammar (it just puts in lookahead information). However, the CFG we gave at the beginning of the description of SLR(1) parsing was not LR(0), so we have a proof that SLR(1) parsing is more powerful than LR(0) parsing. Here is the CFG again:

$$\begin{array}{lcl} S & \rightarrow & Aa \\ S & \rightarrow & Bb \\ S & \rightarrow & ac \\ A & \rightarrow & a \\ B & \rightarrow & a \end{array}$$

Every SLR(1) grammar is also LALR(1), because LALR(1) uses the same (LR(0)) state machine as SLR(1) but has more refined lookaheads. So LALR(1) will never have conflicts that SLR(1) does not have. The CFG we gave at the beginning of the discussion of LR(1) parsing was not SLR(1), so it proves that LALR(1) is more powerful than SLR(1):

$$\begin{array}{lcl} S & \rightarrow & Aa \\ S & \rightarrow & Bb \\ S & \rightarrow & bAb \\ A & \rightarrow & a \\ B & \rightarrow & a \end{array}$$

If there are conflicts in an LR(1) grammar, there will certainly be conflicts in the LALR(1) parser for the same grammar, because LALR(1) just merges some of the LR(1) states. So LALR(1) can never handle a CFG that LR(1) cannot handle. We just gave a grammar that was LR(1) and not LALR(1), which proves that LR(1) is more powerful than LALR(1).

5 Syntax-directed translation

In a broad sense, *syntax-directed translation* is the idea of using the structure of a language to organize processing and translation. Compilation is a complicated process, but it can often be made simpler by breaking the processing into small parts associated with language constructs. The most general approach to syntax-directed translation is to build a tree structure during parsing, then traverse one or more times, calling special functions for each type of node in the tree.

We are not going to study this subject in detail. Instead, we will look specifically at the processing that can be done during shift-reduce parsing. In many cases, a tree never needs to be built. The parsing process can be thought of as traversing a "virtual tree" in a some way. LL(1) or recursive-descent parsing does a top-down left-to-right traversal: each node of the parse tree is visited before its children, and the children are visited in left-to-right order. Shift-reduce parsing does a bottom-up, left-to-right traversal of the virtual parse tree. If the processing of a compiler can be done in the same order that the parser traverses the tree, it is not in general necessary to build the tree.

YACC and its clones (such as byacc and bison) provide a method to associate C or C++ code with reductions in the grammar. Here is an example of a simple calculator, based on our ambiguous expression grammar (this is, of course, not YACC syntax):

$$\begin{array}{lcll} E & \rightarrow & E * E & \texttt{\{ \$\$=\$1*\$3; \}} \\ E & \rightarrow & E + E & \texttt{\{ \$\$=\$1+\$3; \}} \\ E & \rightarrow & (E) & \texttt{\{ \$\$=\$2; \}} \\ E & \rightarrow & \mathit{num} & \texttt{\{ \$\$=\$1; \}} \end{array}$$

A calculator works by bottom-up evaluation of arithmetic expressions. To compute $E + E$, we first compute the value of the right and left expressions, then add them. Hence, it is perfectly matched to shift-reduce parsing.

In YACC, there is a value potentially associated with each symbol in a production. $$ stands for the value associated with the left-hand side symbol, while $\$i$ stands for the value of the ith symbol on the right-hand side. Bottom-up evaluation means the the right-hand values are computed before the left-hand values. In this case, the lexer converts numbers like "123" to their numerical values, which are associated with the token *num* that appears in the CFG.

In the grammar above, when $E \rightarrow num$ is reduced, the value associated with *num* ($1) is copied to the E on the left-hand side ($$). When $E \rightarrow E * E$ is reduced, the values associated with the two Es on the right-hand side are multiplied and associated with the E on the left-hand side.

How is this implemented? YACC has a second stack, called the *value stack*, which parallels the parse stack and contains the values associated with symbols during the parse. The value stack is represented as an array in C, which we will call *values*. The special symbols $\$i$ and $$ are translated into C code that reads or write locations in the value stack. Suppose that, just before reducing $A \rightarrow \alpha$, the index of the top value on the value stack is t and $|\alpha|$ is the number of symbols in α. Then $\$i$ is translated into $values[t - (|\alpha| - i)]$. Hence, $1 is the value associated with the leftmost symbol of the right-hand side, as it should be. When $A \rightarrow \alpha$ is reduced, the symbols of α are popped and A is pushed, so the value associated with A will occupy the same place in the value stack as $1. Hence, in YACC, $$ is actually the same array expression as $1.

This implementation has several natural consequences. First, `{ $$=$1; }` is redundant, since $$ and $1 are the same location (I personally prefer to put it in for clarity). Second, it is possible to use locations like "$0" and "$-1," which access values associated with other symbols not in the current production. This can be used to good effect if you are absolutely sure you know what is there. For example, suppose that whenever B appears in the grammar, it is immediately after A (e.g., in productions like $S \rightarrow AB$). Then, in a production with B on the left-hand side, $0 will always be the value associated with A. *Use this sort of trick with extreme caution, if at all.* You have been warned.

In the shift-reduce parse below, the parse stack is shown with symbols on it instead of states, and the value stack is shown underneath it. Values that are not of interest are shown as $-$. *num* is compressed to n.

$Stack(top)$	$input$
\$	$1+2*3\$$
-	
$\$n$	$+2*3\$$
- 1	
$\$E$	$+2*3\$$
- 1	
$\$E+$	$2*3\$$
- 1 -	
$\$E+n$	$*3\$$
- 1 - 2	
$\$E+E$	$*3\$$
- 1 - 2	
$\$E+E*$	$3\$$
- 1 - 2 -	
$\$E+E*\ n$	\$
- 1 - 2 - 3	
$\$E+E$	\$
- 1 - 6	
$\$E$	\$
- 7	

Actions in the middle of productions

It is often useful to do some processing in a production before the end of the right-hand side (say, to do a computation with side effects before processing that will happen when parsing the remainder of the production). YACC offers an obvious way to include such actions in the grammar: just insert some C code, in braces, between the symbols in the right-hand side of the production. For example,

```
S : A { printf("%d\n", $1); } B { $$=$3; }
```

to print the value associated with A before parsing B.

The implementation of this feature is a little less obvious. YACC generates a fresh nonterminal symbol inserts (let's say *M5* in this case), which is guaranteed not to appear elsewhere in the grammar, and inserts in the production where the action appeared. YACC also adds an ϵ production (e.g. $M5 \to \epsilon$). The new action is associated with the new production, and is executed when that production is reduced. In the one-line example above, the value associated with B becomes \$3 because the new nonterminal *M5* has \$2.

YACC fiddles with the \$ variables to make things more convenient (and a little confusing). Suppose we have something like

```
S : A { $$=$1; } B { $$=$2+$3; }
```

In this example, \$1 is the value of A, and the first \$\$ is the same as \$2, the value associated with the "middle" action. In the action at the end of the production, \$2 is the value of the middle action, and \$3 is the value of B. It is very useful to be able to associate values with middle actions, as we shall see below.

There is one pitfall of using middle actions. If you have an LALR(1) grammar and add middle actions, there is a significant risk of introducing new conflicts into the grammar, because the grammar has changed. Adding a middle action at the left-hand end of a production is especially likely to introduce conflicts. Usually, these problems are solved by rearranging the grammar or moving the middle action somewhere else.

Top-down propagation of information

Even in bottom-up parsing, information can be passed down the "virtual parse tree" without actually building the tree, *if the information is flowing from left to right.* The need for this arises all the time, typically when something has been defined before it is going to be used.

In practice, top-down propagation is frequently handled through an auxilliary data structure, such as a stack or symbol table (this will be the case in programming problem 3). However, it is often convenient to be able to deal with it directly in the parser.

Let us add a somewhat contrived feature to the expression grammar to illustrate this. Following languages like ML which allow binding of variables to values in a "let" expression, we will have a single variable, x, in our new, improved calculator. x can be bound to a value via the new construct $let\ x = E\ in\ E$, where the first E is the new value for x, and the second E is an expression that may refer to x, which will have its current bound value. *let* expressions can be nested, in which case old bindings are restored when a let ends. For example,

$$let\ x = 10\ in\ (let\ x = 20\ in\ 3 * x) + x$$

returns $3 * 20 + 10 = 70$ (ok, I admitted it was contrived). If x hasn't been bound, it has the value 0.

Here is an extended grammar:

$$
\begin{array}{lll l}
E & \rightarrow & \mathit{let}\ \mathrm{x} = \mathrm{E}\ \texttt{\{ \$\$=g; g=\$4; \}}\ \mathit{in}\ \ \mathrm{E} & \texttt{\{ g=\$5; \$\$=\$7; \}} \\
E & \rightarrow & E * E & \texttt{\{ \$\$=\$1*\$3; \}} \\
E & \rightarrow & E + E & \texttt{\{ \$\$=\$1+\$3; \}} \\
E & \rightarrow & (E) & \texttt{\{ \$\$=\$2; \}} \\
E & \rightarrow & \mathit{num} & \texttt{\{ \$\$=\$1; \}} \\
E & \rightarrow & x & \texttt{\{ \$\$=g; \}}
\end{array}
$$

This code is subtle, but worth understanding. Assume that g is a new global variable that is declared elsewhere, which stores the current bound value of x. Whenever an x appears in an expression, the last production in the grammar is reduced and the action associates the current value of g with E, achieving the same effect as if the actual number had appeared in that point of the parse.

We are using the value stack to save and restore the old bindings of x at the end of a *let* expression. In the first production, we first save the old value of g in \$\$ (which is also \$5), then assign the new value of x, \$4, to g. When the E later in the same production is parsed, g will have the current bound value of x. At the end of the production, the old value of g is restored (it is still in \$5), and the computed value of the second E is "returned" as the value of the entire *let* expression. \$5 has the correct value, even though other instances of the same production may have occurred in the second E, because the value stack can keep multiple copies at the same time.

We could avoid the use of g by changing the first and last productions:

$$
\begin{array}{lll l}
E & \rightarrow & \mathit{let}\ \ \mathrm{x} = \mathrm{E}\ \mathit{in}\ \ \texttt{\{ \$\$=\$4; \}}\ E & \texttt{\{ \$\$=\$7; \}} \\
E & \rightarrow & E *\ \texttt{\{ \$\$=\$0; \}}\ E & \texttt{\{ \$\$=\$1*\$4; \}} \\
E & \rightarrow & E +\ \texttt{\{ \$\$=\$0; \}}\ E & \texttt{\{ \$\$=\$1+\$4; \}} \\
E & \rightarrow & (\ \texttt{\{ \$\$=\$0; \}}\ E) & \texttt{\{ \$\$=\$3; \}} \\
E & \rightarrow & \mathit{num} & \texttt{\{ \$\$=\$1; \}} \\
E & \rightarrow & x & \texttt{\{ \$\$=\$0; \}}
\end{array}
$$

In this case, we have arranged for the current bound value of x to be \$0 whenever we are in the right-hand side of a production with E on the left-hand side. Whenever we have E that is not the leftmost symbol on the right-hand side of a production, we insert a "middle" action that copies the value into a point just below the next E (where it will be \$0 in the right-hand side of the production).

I do not endorse writing YACC actions that are this tricky. To do it, we've had to scatter extra code through many different productions, and the results are probably not very obvious. However, it is worth understanding this example because it truly enhances ones intuition about how parsing works.