PRACTICAL, INCREMENTAL, NONCANONICAL PARSING: CELENTANO'S METHOD AND THE GENERALIZED PIECEWISE LR PARSING ALGORITHM

BY

JEFFREY L. OVERBEY

B.S., Southeast Missouri State University, 2004

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2006

Urbana, Illinois

Abstract

Parsers in modern integrated development environments (IDEs) for general-purpose languages are virtually all of the *ad hoc,* recursive descent variety. While such parsers have many disadvantages when compared to machine-generated LALR(1) parsers, they have two winning qualities: They are not restricted to any finite amount of lookahead, and in an IDE, they can re-parse small segments of a file as they change rather than re-parsing the entire file.

Theoretically, both of these capabilities can be achieved through variations on traditional LR parsing techniques. Schell's Generalized Piecewise LR (GPLR) parsing algorithm provides a particularly powerful method for allowing unbounded lookahead, and Celentano's approach to incremental LR parsing provides a simple and easily implementable technique to avoid re-parsing an entire file as changes are made.

In this thesis, we prove that Celentano's technique can be applied to GPLR parsers, despite their use of unbounded lookahead; furthermore, this does not require a change to either algorithm. This result is prefaced by intuitive developments of the LR and GPLR parsing algorithms and Celentano's construction. We conclude by proposing a method for constructing compact GPLR parsers from LALR(1) machines and identifying future directions for research in incremental, noncanonical parsing.

Acknowledgments

First and foremost, I would like to thank my thesis advisor, Prof. M. Dennis Mickunas, for his guidance, encouragement, and support. There are very few people in the world who find parsing interesting enough to agree to supervise an independent study and a master's thesis despite the fact that they are technically retired, finishing law school, and on the verge of moving to Florida where, I am told, they can spend time enjoying things like "ocean kayaking" and "good weather," the existence of which is still questioned by the rest of us in Illinois. While I will never be able to repay him for his many hours of help and guidance—not at his lawyer price, anyway—he will always have my eternal gratitude.

I would like to thank Richard Schell, who I have not met yet, for providing the foundation on which the present work is built. His dissertation is remarkably innovative and lucid. While his work was mostly theoretical in 1979, in the present era of multicore computers with gigabytes of memory and clock speeds approaching their theoretical limits, it has real practical potential. Without his work, this thesis would not exist.

Finally, I would like to thank my Ph.D. advisor, Prof. Ralph Johnson, for giving me an exciting project to work on and allowing me to combine my interests in parsing, refactoring, and program representation; IBM, for funding my work on Photran under a PERCS grant; Brian Foote, for encouraging me to pursue parsing despite his personal distaste for all things theoretical; my family, friends, and colleagues, who make life fun and help me keep everything in perspective; and God, for sticking with me through all of this and helping me out more than I deserve.

Table of Contents

List of Figures

List of Symbols

$G = (V_N, V_T, P, S)$	The context-free grammar being parsed (p. 3)
V_N	The nonterminal symbols of G (p. 3)
V_T	The terminal symbols of G (p. 3)
V	The vocabulary of G ($V = V_N \cup V_T$) (p. 3)
P	The productions of G (p. 3)
S	The start symbol (start nonterminal) of G (p. 3)
$	End-of-input marker (p. 3, p. 13)
A, B, C	Nonterminals (p. 4)
a, b, c	Terminals (p. 4)
X, Y, Z	Terminals or nonterminals (p. 4)
w, x, y, z	Strings in V_t^* (p. 4)
$\alpha, \beta, \gamma, \delta, \zeta, \eta, \theta, \iota$	Strings in V^* (p. 4)
ϵ	The empty string (p. 4)
a^n	n-fold repetition of a (p. 4)
$[A \rightarrow \alpha \cdot \beta]$	LR(0) item (p. 7)
q_0	Initial state of an LR parser (p. 8) or other automaton (p. 22)
$[A \rightarrow \alpha \cdot \beta,\ w]$	LR(k) item (p. 12)
First_k	(p. 13)
$\mathrm{Closure}_{\mathrm{LR}}$	(p. 14)
$\mathrm{Goto}_{\mathrm{LR}}$	(p. 14)
$\mathrm{Action}_{\mathrm{LR}}$	(p. 14)
$\perp$	Continuation marker (false bottom-of-stack marker) (p. 16)

$\langle A \to \alpha \cdot \beta \rangle$	Cancellation symbol (p. 17)
$[A \to \alpha \cdot \beta,\ B \to \gamma \cdot \delta]$	GPLR(1) item (p. 18)
χ	Terminal, nonterminal, or cancellation symbol (p. 21)
Σ	Input alphabet of a two-stack pushdown automaton (2PDA) (p. 22)
Γ_1	First stack alphabet of a 2PDA (p. 22)
Γ_2	Second stack alphabet of a 2PDA (p. 22)
Δ	Transition relation of a 2PDA (p. 22)
τ_0	Initial contents of the first stack of a 2PDA (p. 22)
F	Set of final states of a 2PDA (p. 22)
$(q, \alpha, \beta) \longrightarrow (q', \alpha', \beta')$	Δ-rule of a 2PDA (p. 23)
M	A 2PDA (p. 23)
$\alpha \textcircled{q} \beta$, $\alpha \odot \beta$	Configuration of a 2PDA (p. 23)
$\vdash$, $\vdash_M$	Moves-to relation on a 2PDA (p. 23)
$\mathscr{I}_{\text{GPLR}}$	Set of all GPLR(1) items for a grammar (p. 24, p. 27)
V_C	The cancellation symbols of a grammar (p. 24)
$S_{I,\chi}$, $R_{I,\chi}$, and $C_{I,\chi}$	GPLR item selectors (pp. 24–25)
I	Set of GPLR(1) items (p. 25)
Core	(p. 25)
$\text{Closure}_{\text{GPLR}}$	(p. 25)
$\text{Goto}_{\text{GPLR}}$	(p. 25)
$\text{Cont}_{\text{GPLR}}$	(p. 26)
$\mathscr{C}_{\text{GPLR}}$	Canonical collection of sets of GPLR(1) items (p. 26)
Π	Parse sequence (p. 31)
p	In a stack tree, the pointer to the top-of-stack node (p. 31)
p_k	Stack tree node pointer associated with the kth input symbol (p. 33)
has-lalr-lookahead	(pp. 44, 45)
lookback	(p. 45)
includes	(p. 45)

1

Introduction

Parsing is often touted as one of the great successes of computer science.[1] The problem is beautifully mathematical, the solutions are provably correct, and the results are intensely practical. The automatic generation of parsers from grammars is a rare case where specification is simpler than direct implementation and, by many measures, produces superior results.

Parsing has been studied almost exhaustively in the literature. Researchers have parsed everything from arithmetic expressions to natural language, on every computer from the DEC VAX to Dell PCs to Crays, reading input from left to right and right to left, usually once but sometimes twice or more, constructing syntax trees from top to bottom, bottom to top, all at once or in pieces.

This makes it especially ironic that the parsers in today's most advanced compilers and language tools are almost all ad hoc, recursive descent parsers that disregard almost the entire lot of post-1960s research. While parser generators such as `yacc` have made (theoretically superior) LALR(1) parsers practically viable, very few general-purpose programming languages have a syntax that can be easily formalized as an LALR(1) grammar; parser writers find it simpler to deal with the eccentricities of these languages operationally rather than by finagling a grammar which will mollify the parser generator.

The problem is that the grammars in many modern programming language specifications are not LALR(1) and are not easily made LALR(1). There are two reasons. First, some language features are not context-free, despite the fact that they are specified in a context-free grammar. In Fortran 2003, for example, `f(6)` can be either an array access or a function call, depending on how `f` was declared. Since function calls and array accesses can occur in the same contexts, there is an ambiguity in the specification grammar. Secondly, the remaining problems—those not manifested as ambiguities—are due to the fact that, while LALR(1) parsers have infinite left context, they only have a single token of right context (lookahead). In the first edition of the Java Language Specification [15], the authors dedicate an entire chapter to producing an LALR(1) grammar for the Java language, elaborating on the pecularities of the specification grammar

[1] Actually, the term *computer science* was not coined until 1961 [22], several years after the earliest parsing and compilation research.

that make it non-LALR(1) and how these are resolved; every problem is due in part to limited right context.

Returning to the original point, despite the fact that generated parsers can guarantee correctness and efficiency, programmers often resort to ad hoc parsing techniques because it is difficult to make an LALR(1) grammar for the language they want to parse. When the difficulties are due to the fact that some language features are not context-free, the ad hoc approach is obviously the right choice.[2] However, when the difficulties are due to the limited-lookahead restriction of LALR(1) parsers (as was the case with Java), reverting to ad hoc techniques seems a bit extreme; it would be preferable to simply remove this lookahead restriction in the parser generator.

There are a number of techniques for allowing infinite right context while producing a deterministic parse. But are they worth implementing, or are they just theoretical toys? Of course, it depends on the specific technique, but let us look at why LALR(1) became the algorithm of choice for generated parsers during the 1970s and 1980s. First, the class of LALR(1) grammars is relatively large (perhaps not large enough, but large nonetheless). Second, LALR(1) parsers are typically quite small. Third, they can parse in a single left-to-right scan of the input. It should not be surprising that infinite-right-context parsers tend to be larger and slower than LALR(1) parsers. Twenty or thirty years ago, they would have been considered prohibitively so. However, memory requirements are almost negligible now, when a gigabyte of memory costs little more than a tank of gasoline. And the algorithms under consideration are all sub-quadratic; on a multi-gigahertz machine, this should hardly be considered "too slow." More specific details will be given later, but the point is this: Although these algorithms may have been deemed impractical when they were developed, the reasoning is no longer valid, and their practicality should be reconsidered.

Ad hoc, recursive descent parsers have another advantage besides their ability to handle bizarre language features. Due to the way they are coded, it is easy to (re)parse a single function or a single statement, not necessarily an entire program. While this is virtually useless in a compiler, it is extremely useful in modern integrated development environments, where editors, search tools, and other components are more syntax-aware than ever before. This mechanism is stereotypically used as an ad hoc approach to *incremental parsing*, or updating the syntax tree of a program by re-parsing only parts that have changed rather than the entire program.

At present, the ability to parse incrementally is useful, but there is not

[2] Some might argue that context-sensitive grammars are the right choice, should a context-sensitive parser generator become realistic for implementation. Inasmuch as a grammar is a specification of parser behavior, neither ad hoc code nor a context-sensitive grammar will look anything like the actual (written) language specification, so the preference of one over the other is debatable and dependent upon the specific feature in question and the masochistic tendencies of the implementer.

enough evidence to call it mandatory. However, a cursory glance at the sophisticated, real-time analyses built into modern interactive development tools—and the plethora of work in that area—makes it very likely that incremental parsing will become a necessity in the future, if it is not considered so already.

There is a fairly efficient algorithm for incremental LR parsing due to Celentano [4] which is perfectly applicable to generated LALR(1) parsers. However, until now, no method for incremental parsing has been considered for deterministic parsing algorithms that use infinite right context. If incremental parsing will never be needed—in a typical compiler, perhaps—this is not a problem. But as parsers are finding their ways into more and more interactive tools, the inability to parse incrementally could legitimately be a reason to discount these techniques.

Fortunately, it is not. In this thesis, we will look at one LR-based parsing algorithm which uses infinite right context, Schell's Generalized Piecewise LR (GPLR) method, and show that GPLR parsers can be made incremental using Celentano's method *with no modification to either algorithm.* In addition to proving this formally, we will look at why this is the case, and how it might apply to similar parsing algorithms.

In Chapter 2, we will review shift-reduce and LR parsing. The GPLR parsing algorithm will be described in Chapter 3. In Chapter 4, we will describe Celentano's method for incremental LR parsing and its application to GPLR and other algorithms. Formalisms leading to a proof of correctness for the incremental GPLR parsing algorithm will be introduced as needed; the corresponding section titles will be marked with an asterisk and may be skipped if the reader simply wants an intuitive description of the algorithm.

Background and Notation

We assume that the reader is familiar with alphabets, languages, automata, context-free grammars, derivations, and the general concept of parsing as presented in a typical treatise on the theory of computation, such as [16]. LR parsing will be reviewed, but some prior familiarity with the topic is helpful, perhaps at the level of Aho, Sethi, and Ullman's *Compilers: Principles, Techniques, and Tools* [2].

Unless stated otherwise, we assume that we are parsing an underlying context-free grammar $G = (V_N, V_T, P, S)$ where V_N denotes a finite, non-empty set of nonterminal symbols, V_T denotes a finite set of terminal symbols, $V_N \cap V_T = \emptyset$, $P \subseteq V_N \times (V_N \cup V_T)^*$ denotes a finite set of productions, and $S \in V_N$ is the start symbol of the grammar. We will let V denote $V_N \cup V_T$. The distinguished terminal symbol \$ represents the end of input string. We will often use a list of productions as a shorthand notation for a grammar whose start symbol is the nonterminal on the left-hand side of the first production and whose terminal and nonterminal symbols can be inferred from the list of productions.

Capital letters toward the beginning of the alphabet (A, B, C) denote nonterminals, while the same letters in lowercase (a, b, c) denote terminals. Capital letters toward the end of the alphabet (X, Y, Z) denote symbols in $V = V_N \cup V_T$, while lowercase x, y, and z denote strings in V_T^*. Lower-case greek letters toward the beginning of the alphabet (α, β, γ) denote strings in V^*.

The symbol ϵ denotes the empty string. The expression a^n denotes the string consisting of n repetitions of the symbol a (so $a^3 = aaa$ and $a^0 = \epsilon$).

2 Foundations of GPLR Parsing

In this chapter, we will describe shift-reduce parsing in its most general form, then describe how LR parsing improves upon the shift-reduce algorithm. In the next chapter, we will describe Schell's Generalized Piecewise LR (GPLR) parsing algorithm, which further extends the LR parsing algorithm to parse a larger class of languages.

2.1 Shift-Reduce Parsing

A *parse* of a string w in a grammar G is simply a sequence of productions applied in a derivation of w; a *parser*, then, is a program which receives a string as input and produces a parse of that string in a particular grammar (or produces an error if that string is not in the language of that grammar) [25, p. 160].

One of the simplest parsers is a nondeterministic stack machine which behaves as follows. This is called the *shift-reduce parser* for G [25, p. 164].

- It starts with an empty stack and w as its input. It will output a parse, i.e., a sequence of productions; initially, this sequence is empty.
- It pushes the symbols of w onto its stack one at a time.
- If, at any point, the sequence of symbols on the top of its stack matches the right-hand side of one of the productions of G, it pops those symbols, replacing them with the nonterminal on the left-hand side of that production. That production is added to the output parse.
- The parse accepts if, and only if, it reaches a state where the only symbol on its stack is S, and no input remains.

The action of pushing an input symbol onto its stack is called *shifting;* that of popping the stack and replacing the symbols with the left-hand side of a production is called *reducing* by that production.

For example, suppose we have the balanced parentheses grammar

$$S \to (S) \mid () \mid SS$$

and the input (())(). The shift-reduce parser behaves as follows (symbols that are about to be popped because they match the right-hand side of a production are underlined; the nonterminal replacing them is displayed in boldface on the following line):

Stack	Remaining Input	Production
	(())()	
(	())()	
((	))()	
$(\underline{()}$	)()	S →()
(**S**	)()	
$\underline{(\mathrm{S})}$	()	S →(S)
S	()	
S(	)	
$\mathrm{S}\underline{()}$		S →()
$\underline{\mathrm{S}\mathbf{S}}$		S →SS
S		

The shift-reduce parser for a grammar is conceptually simple, but it is not used in practice because it is *nondeterministic.* Suppose we had the alternative balanced parentheses grammar

$$S \rightarrow (S) \ \mid \ \epsilon \ \mid \ SS;$$

then the shift-reduce parser could reduce ϵ to S at any time (since it can always find ϵ on the top of its stack); theoretically, the machine can even reduce ϵ several times in a row. In general, it cannot decide whether its next action is to shift the next input symbol or reduce by some production. It may even be able to reduce by several different productions at any given point.

The LR parsing method is a more sophisticated variation on the shift-reduce parsing technique which eliminates this nondeterminism. Conceptually, the parser still shifts symbols onto its stack and reduces them, but it is augmented with a state machine which controls its action. This state machine, coupled with the ability to “look ahead” at a finite prefix of the remaining input, guarantees that, at any point, the decision to shift or reduce by a particular production is uniquely determined.

2.2 LR Parsing

2.2.1 LR(0) Parsing

The most fundamental difference between the basic shift-reduce parsing algorithm and LR parsing is the addition of a finite state machine. While the basic shift-reduce parser decides what action to take (shift or reduce by a particular production) based solely on the parser's stack contents, in the LR parsing algorithm, this action is determined by the state of this machine. The state changes (1) as input symbols are read and (2) when a reduction is performed. The exact details will be described later; for now, let us describe the construction of this finite state machine.

For the time being, we will use the grammar

$$\begin{array}{lcl} A & \rightarrow & aA \mid B \\ B & \rightarrow & bb \end{array}$$

for the language a^*bb as a running example (where A is the start symbol of the grammar).

Recall that the goal of a parser is to produce a derivation of the input string from the start symbol (in this case, A). When a parser starts, then, it is expecting to see a string that is derivable from the start symbol (A). Since the two A-productions in our grammar are $A \rightarrow aA$ and $A \rightarrow B$, this means the first thing it should expect to see is either an aA or a B. Let us use

$$\begin{array}{c} [A \rightarrow \cdot aA] \\ [A \rightarrow \cdot B] \end{array}$$

to indicate this.

If we next were to shift the symbol a, we would use $[A \rightarrow a \cdot A]$ to indicate this new situation; intuitively, the dot indicates where the parser is in matching the right-hand side of a production. Equivalently, then, the symbol after the dot tells us what the parser is expecting to see next.

These productions with a dot in the middle or on either end are called *LR(0) items,* and each state in our state machine will be a set of these items. (We will usually just list the items, as above, rather than using the usual mathematical notation for sets.) So if we look at all of the symbols after the dots in every item in our state, we should have a complete list of what symbols the parser should expect to see next in its input.

When the symbol after the dot is a terminal, like a, the meaning is obvious: The next symbol in the input should be exactly that symbol. But what does it means for a parser to "expect" a nonterminal like B, since there are only terminals in the input string? Effectively, it means the parser should expect to see *anything derivable from B.* This means the parser may also see bb at this point (since $B \rightarrow bb$), so $[B \rightarrow \cdot bb]$ should also be included in the state (i.e., the set of items describing the state of our parser). So the complete initial state of our parser is

$$\begin{array}{c} [A \rightarrow \cdot aA] \\ [A \rightarrow \cdot B] \\ [B \rightarrow \cdot bb]. \end{array}$$

In general, any time a nonterminal appears after the dot, we should look at each production for that nonterminal and include an item in the state for that production with a dot on the left-hand side. $A \to \cdot B$ means we are expecting to see something derivable from B, and $B \to \cdot bb$ tells exactly what things derivable from B look like. This process of finding items with nonterminals after the dots and adding new items with their productions is called *closing the state.*

After we close a state, we have a complete list of every possible next input symbol: Just look at all of the terminals appearing after the dot in an item. Let's call the initial state of our parser—the one we constructed above—q_0. The terminals after the dots in q_0 are a and b, so the next input symbol should be either of these.

Suppose we are in state q_0 and the next input symbol is a. The symbol a should be shifted onto the stack, and our state machine should move to a new state. But what items are in this new state?

The answer is quite simple: All of the items in q_0 that had an a after the dot, except now the dot will be moved forward one position since we already matched the a. Since we didn't see a B or a b, none of the other items apply, so they will not be included in this new state. This just leaves $[A \to a \cdot A]$. As before, we need to close this state to get a complete picture of what terminals to expect, so this new state should also include $[A \to \cdot aA]$, $[A \to \cdot B]$, and $[B \to \cdot bb]$.

We will call this state the *goto state from* q_0 *on* a. The goto state from q_0 on b is constructed similarly. We can also construct goto states on nonterminal symbols appearing after the dots in the items of a state, e.g., the goto state from q_0 on B; we will explain the utility of these later. If we construct the initial state, and its goto states, and the goto states' goto states, etc., until we cannot construct any more states, we have a finite state machine called the *LR(0) characteristic finite state machine,* or *cfsm.* The cfsm for our example grammar is shown in Figure 2.1.[1]

As we shift symbols, moving from state to state, the dots in our items keep moving to the right, so eventually we will reach a state where one of these items has a dot on the right end. Let's take $[B \to bb\,\cdot]$ as an example. This means that we have seen bb—something derivable from B—in its entirety, so we can safely reduce it to B. (Thus, we often refer to items with dots on the right end as *reduce items* and others as *shift items.*)

But after we reduce bb to B, the state containing $[B \to bb\,\cdot]$ no longer describes the state of our parser. What state should it be in?

Recall that $[A \to \cdot B]$ (or any item with B after the dot) meant that we were expecting to see something derivable from B, and $[B \to bb\,\cdot]$ (or any B-reduce

[1] Notice that q_1 loops back to itself on a. The items in the goto state from q_1 on a are exactly the items of q_1, and there should never be two states in the machine with identical sets of items. This is also why q_0 and q_1 lead to the same goto state on b (q_2).

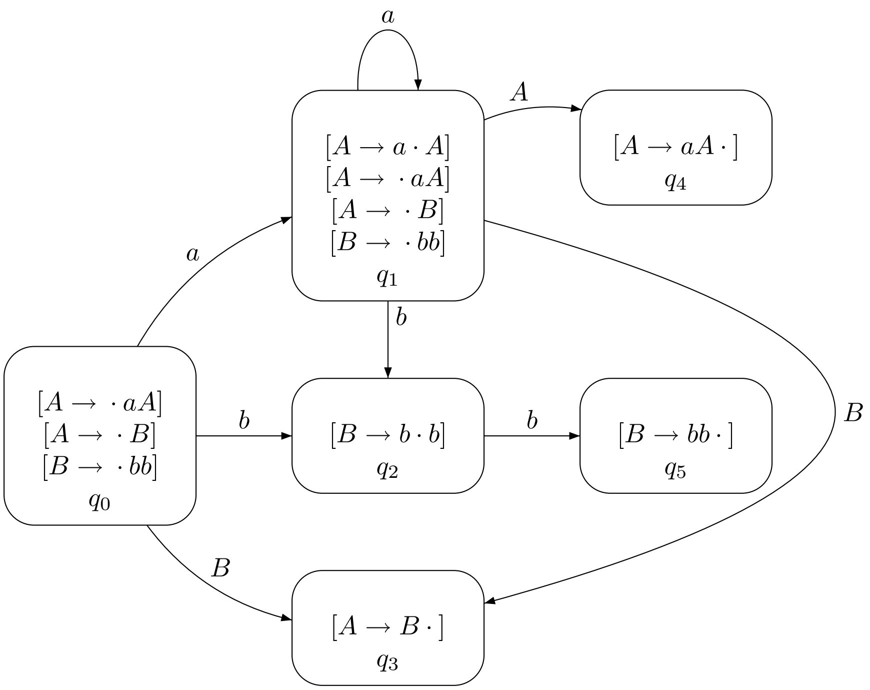

Figure 2.1: LR(0) cfsm for the non-augmented grammar $A \to aA \mid B; B \to bb$

item) means we just saw something derivable from B. So we want return to whatever state we were in just before we got sidetracked processing the input derivable from B.[2]

This is where the stack can be useful to us. Stacks have traditionally been used as a data structure to remember what a machine was doing before it got sidetracked doing something else.[3] So in addition to storing shifted symbols on the stack, we also need to store *what state the parser was in* after the symbol was shifted. We will write these symbol-state pairs as $\genfrac{}{}{0pt}{}{X}{q}$, where X is the symbol and q is the state. At the beginning of the parse, we will place $\genfrac{}{}{0pt}{}{-}{q_0}$ on the stack so that we have a record of our initial state.[4]

Suppose we are parsing the string $aaabb$ in the example grammar. Following the cfsm in Figure 2.1, our state machine will move through the states $q_0, q_1, q_1, q_1, q_2, q_5$, shifting the five symbols and their corresponding states onto the stack before reaching q_5 with the reduce item $[B \to bb \cdot]$. At that point, it will have

$$\begin{array}{cccccc} - & a & a & a & b & b \\ q_0 & q_1 & q_1 & q_1 & q_2 & q_5 \end{array}$$

[2] In our simple example, we know exactly which state this was, but in general, there may be several such states, and we need to know which one to jump back to.

As an aside, one can entertain the possibility of combining all of these states into a single new state and avoid using the stack altogether, but the net result would be an unnecessarily complex parser limited to regular languages.

[3] Activation records on the stack of an executing process are a well-known example.

[4] The procedure for reducing looks at the top stack symbol *after* the terminals being reduced have been popped; if we did not include this, we would be left with an empty stack, which we would have to treat as a special case.

on its stack. To reduce bb to B, it must pop the last two pairs off the stack. This leaves $\overset{a}{q_1}$ on top of the stack, which means that just before it started processing the input derivable from B, it had just shifted a and moved to state q_1. So we should not be surprised to find that q_1 includes the item $[A \rightarrow\ \cdot B]$. Now that we have seen the entire input derivable from b and popped those pairs off the stack, the state machine will move to state q_3—the goto state from q_1 on B—and we will push $\overset{B}{q_3}$ onto the stack. Intuitively, we are doing the same thing we did when we shifted a terminal: We are pushing a symbol-state pair onto the stack and going to the appropriate goto state. The only difference is that this time we are doing it for a string of symbols—the entire string derivable from B—rather than just a single symbol. Effectively, we "matched" a nonterminal in the input string, just as we "match" a terminal in the input string when we shift it.

The only problem that remains is that we do not know when to accept the input. Finishing out the parse of $aaabb$ in the example grammar, it would seem that if we have no input remaining, we are in state q_0, and only $\overset{-}{q_0}$ is on our stack, we can accept. But that is exactly the configuration of our parser when it is started, which means it can accept the empty string, which is clearly not derivable in the sample grammar![5]

To remedy this, we never construct LR parsers from "ordinary" grammars. Instead, we construct an *augmented grammar* by adding a new production $S' \rightarrow A$, where A is the original start symbol, and make S' the new start symbol for the grammar. (S' should be some new nonterminal not already in the grammar, of course.) When we construct the cfsm, then, the initial state will be the closure of $\{[S' \rightarrow\ \cdot A]\}$, and the machine will accept when it is about to reduce A to S' (i.e., when it reaches the state containing $[S' \rightarrow A \cdot]$).

Fortunately, for our example, the cfsm for the augmented grammar is nearly identical to the one shown earlier for the non-augmented grammar. It is shown in Figure 2.2: The start state q_0 is the same as before but also includes $[S' \rightarrow\ \cdot A]$. Also, the cfsm contains a new state q_6 consisting of the sole item $[S' \rightarrow A \cdot]$.

With this new cfsm, the complete parse of $aaabb$ is as follows.

[5] This is one example. The general problem we are trying to prevent is a *shift/accept* conflict: The parser cannot tell when it should shift the next symbol and when it should accept the input.

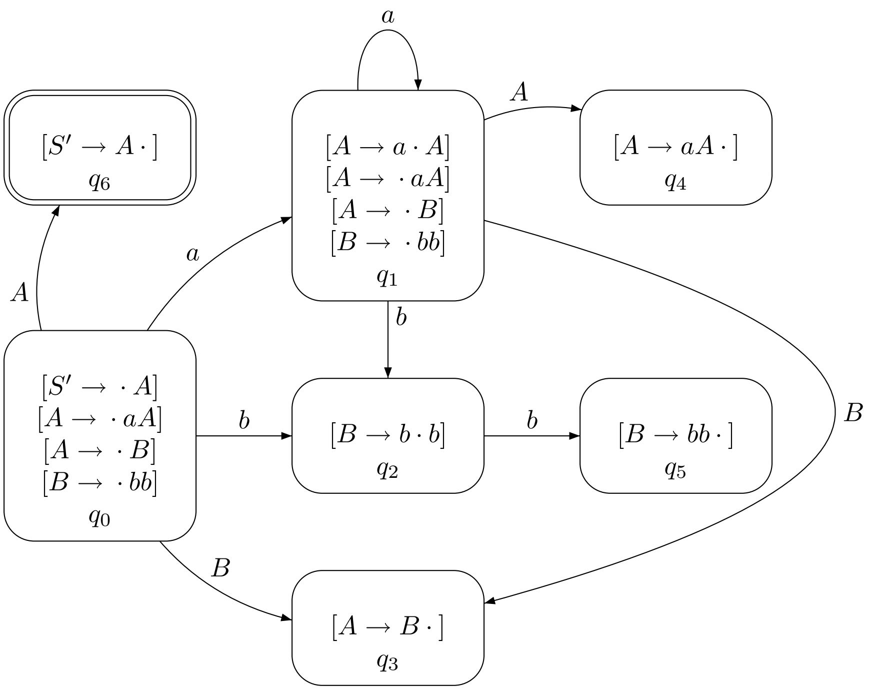

Figure 2.2: LR(0) cfsm for the augmented grammar $S \to A; A \to aA\,|\,B; B \to bb$

Stack	Remaining Input	Reduce Item
$\begin{matrix} - \\ q_0 \end{matrix}$	aaabb	
$\begin{matrix} - & a \\ q_0 & q_1 \end{matrix}$	aabb	
$\begin{matrix} - & a & a \\ q_0 & q_1 & q_1 \end{matrix}$	abb	
$\begin{matrix} - & a & a & a \\ q_0 & q_1 & q_1 & q_1 \end{matrix}$	bb	
$\begin{matrix} - & a & a & a & b \\ q_0 & q_1 & q_1 & q_1 & q_2 \end{matrix}$	b	
$\begin{matrix} - & a & a & a & b & b \\ q_0 & q_1 & q_1 & q_1 & \underline{q_2} & \underline{q_5} \end{matrix}$		$[B \to bb\,\cdot]$
$\begin{matrix} - & a & a & a & \mathbf{B} \\ q_0 & q_1 & q_1 & q_1 & \underline{\mathbf{q_3}} \end{matrix}$		$[A \to B\,\cdot]$
$\begin{matrix} - & a & a & a & \mathbf{A} \\ q_0 & q_1 & q_1 & \underline{q_1} & \underline{\mathbf{q_4}} \end{matrix}$		$[A \to aA\,\cdot]$
$\begin{matrix} - & a & a & \mathbf{A} \\ q_0 & q_1 & \underline{q_1} & \underline{\mathbf{q_4}} \end{matrix}$		$[A \to aA\,\cdot]$
$\begin{matrix} - & a & \mathbf{A} \\ q_0 & \underline{q_1} & \underline{\mathbf{q_4}} \end{matrix}$		$[A \to aA\,\cdot]$
$\begin{matrix} - & \mathbf{A} \\ q_0 & \underline{\mathbf{q_4}} \end{matrix}$		$[S' \to A\,\cdot]$ *(accept)*

2.2.2 Determinism

Although the last section completely describes the operation of the simplest LR parsing algorithm, we glossed over one important issue. The biggest improvement of LR parsers over basic shift-reduce parsers was supposed to be determinism, the unique selection of a shift or reduce action. Certainly, the LR(0) cfsm helps to eliminate some nondeterminism: For example, a parser will not reduce by an ϵ-production unless there is an item like $C \rightarrow \cdot$ in its current state, and it will not try to shift the next symbol if there are no shift items for that symbol in its current state. But what happens if there is both a shift item and a reduce item in the same state? Or what if there are two different reduce items?

The simple answer is, we cannot decide what to do, and so we cannot create an LR(0) parser for that grammar. The problematic state is said to have an irresolvable *shift/reduce* or *reduce/reduce* conflict.

The longer answer is that we should try to construct a more complicated parser and see if that will eliminate the problem. Conflicts can never be prevented entirely in deterministic parsing algorithm: There are context-free languages which are not deterministic, and some languages are inherently ambiguous (i.e., every grammar for that language is ambiguous). But it is not difficult to accept more than the LR(0) grammars.

2.2.3 Adding Lookahead: LR(k) Parsing

LR(0) parsers, described above, are actually the simplest form of a more general technique known as LR(k) parsing, where k is some natural number (usually 1, rarely 2, and almost never larger). In an LR(k) parser, rather than reducing any time we end up in a state with a reduce item, the items in the states will be augmented with k-symbol *lookahead strings.* A reduce action will only be performed when the next k symbols of the remaining input match the lookahead string in the reduce item. For example, the item $[C \rightarrow x \cdot,\ a]$ has the one-symbol lookahead string a. So if that item and $[D \rightarrow x \cdot,\ b]$ appear in the same state, we can still choose a single reduce action, since reduction by $C \rightarrow x$ should occur only when the lookahead is a, and reduction by $D \rightarrow x$ when it is b. Conceptually, this is simple; the construction is slightly more complex.

We already mentioned that we will add lookahead components to reduce items in our cfsm; in fact, we will add lookahead components to all of the items in the cfsm. The lookahead component is useless in shift items, but it will make the construction easier.

Recall that an LR(0) item $[E \rightarrow \alpha \cdot \beta]$ in a state intuitively indicates that the parser has seen (something derivable from) α and it is expecting to see (something derivable from) β next. The *LR(k) item* $[E \rightarrow \alpha \cdot \beta,\ w]$ intuitively indicates that the parser has seen (something derivable from) α, it is expecting to see (something derivable from) β next, and after that, it is expecting to see

w. That is, after the reduction to E is performed in some successor state, the next k input symbols will be w.

Let us use the distinguished symbol $ to represent "end of input." Recall that, in the LR(0) construction, the initial state was the closure of the state containing the single item $[S' \to \cdot S]$. In an LR(k) parser, the parser should expect to see something derivable from S followed immediately by the end of input, so the initial state will be the closure of the state containing the single LR(k) item $[S' \to \cdot S,\ \$^k]$.[6]

The notion of closure for LR(k) items is slightly different than for LR(0) items, of course, since lookahead components are involved. The basic idea of finding the nonterminals after the dots and including their productions is the same, but what should the lookaheads be?

Let us recall *why* we compute closures in the first place. If we have some item of the form $[E \to \alpha \cdot F\beta,\ w]$—that is, an item with a nonterminal (F) after the dot—then the parser is in a state where it should expect to see anything derivable from F, so we include items for each F-production in the grammar. If the parser *does* see something derivable from F (i.e., it uses one of these items to eventually reduce something to F), after it reduces, it will return to this state and move forward to the state containing $[E \to \alpha F \cdot \beta,\ w]$, where it will eventually see β and then w. So the first few symbols following the symbols derived from F will be the first few symbols of something derivable from βw.

Therefore, we compute LR(k) closures as follows. Each item with a nonterminal after the dot has the form

$$[E \to \alpha \cdot F\beta,\ w],$$

where E and F are nonterminals, α and β are strings in V^*, and w is a string in V_T^k, where k is the lookahead length. We find all of the productions for F in the grammar; call them $F \to \alpha_1$, $F \to \alpha_2$, …, $F \to \alpha_n$. Then we should ensure that the state contains each of the items

$$\begin{gathered}[F \to \cdot \alpha_1,\ \mathrm{First}_k(\beta w)],\\ [F \to \cdot \alpha_2,\ \mathrm{First}_k(\beta w)],\\ \ldots,\\ [F \to \cdot \alpha_n,\ \mathrm{First}_k(\beta w)].\end{gathered}$$

where $\mathrm{First}_k(\alpha) \overset{\mathrm{def}}{=} \{x \mid \alpha \overset{\mathrm{lm}}{\Longrightarrow}^* x\beta \text{ and } |x| = k, \text{ or } \alpha \Rightarrow^* x \text{ and } |x| < k\}$.

When we compute goto states, lookaheads are simply carried over unchanged from the original state. For example, suppose a state contains the item $[C \to a \cdot bc,\ de]$. Then the goto state on b will contain $[C \to ab \cdot c,\ de]$; the parser still expects to see de after it reduces abc to C, just like it did in the original state, so the lookahead component does not change.

[6] Requiring such an endmarker limits a parser to recognizing only the strict deterministic context-free languages.

2.2.4 Summary of the LR(k) Construction

We have described the entire LR(k) construction, but it is useful to summarize it in a more precise and compact form. In the next chapter, we will expand on this in describing the GPLR construction.

The LR(k) cfsm is constructed as follows.

1. The initial state is $\text{Closure}_{\text{LR}}(\{[S' \to\ \cdot S,\ \$^k]\})$, where

2. Given a set I of LR(k) items, the $\text{Closure}_{\text{LR}}(I)$ is defined recursively as the smallest set satisfying

$$\begin{aligned}\text{Closure}_{\text{LR}}(I) = I \quad \cup \quad \{[B \to\ \cdot\gamma,\ \text{First}_k(\beta w)] \mid \\ [A \to \alpha \cdot B\beta,\ w] \in \text{Closure}_{\text{LR}}(I) \text{ and} \\ B \to \gamma \in P\}.\end{aligned}$$

3. In a given state q, if there is an item $[A \to \alpha \cdot X\beta,\ w]$, then there is also a state $q' = \text{Goto}_{\text{LR}}(q, X)$, and there is a transition from q to q' on symbol X.

4. The function Goto_{LR} is defined to be the smallest set satisfying

$$\begin{aligned}\text{Goto}_{\text{LR}}(q, X) = \text{Closure}_{\text{LR}}(\quad \{ \quad [A \to \alpha X \cdot \beta,\ w] \mid \\ [A \to \alpha \cdot X\beta,\ w] \in q\}).\end{aligned}$$

5. The accepting state is the (unique) state containing $[S' \to S\, \cdot\, ,\ \$^k]$.

The parser is driven by two functions, $\text{Action}_{\text{LR}}$ and Goto_{LR}. The Goto_{LR} function was described above; $\text{Action}_{\text{LR}}$ is defined as follows. (If $\text{Action}_{\text{LR}}$ is multiply defined for any input, then the underlying grammar is not LR(k) and the construction fails.) Given a cfsm state q and a lookahead string $au \in V_T^k$,

$$\text{Action}_{\text{LR}}(q, au) \overset{\text{def}}{=} \begin{cases} \textit{shift and go to } \textit{Goto}_{\textit{LR}}(q, a) & \text{if } \exists[A \to \alpha \cdot a\beta,\ w] \in q \\ \textit{reduce by } A \to \alpha & \text{if } \exists[A \to \alpha\, \cdot\, ,\ au] \in q \\ \textit{accept} & \text{if } [S' \to S\, \cdot\, ,\ \$^k] \in q \\ \textit{error} & \text{otherwise.} \end{cases}$$

3

The GPLR Parsing Algorithm

We will now build on the informal, operational description of LR parsing in the previous chapter to present the Generalized Piecewise LR parsing algorithm. Originally presented in [23, p. 101–114], GPLR is much more sophisticated than the basic LR algorithm but is also much more powerful: The GPLR(1) grammars properly include the LR(1) grammars, and they also include many languages which are not LR(k) for any k [23, p. 111].[1]

This power is due to the fact that GPLR is a *noncanonical* parsing technique. Informally, this means that, where an LR(k) parser would encounter a shift/reduce or reduce/reduce conflict, a GPLR parser will temporarily ignore the problem and make some reductions on the remaining input. Eventually, it will return to the site of the conflict and make a decision, facilitated by its new knowledge of what nonterminal(s) the remaining input will reduce to. A more formal definition is that a *canonical* parsing algorithm (e.g., LR(k)) is one that will only reduce *handles,*[2] while a *noncanonical* parsing method is one that may also reduce phrases which are not handles.

GPLR is a *two-stack* parsing algorithm. The first stack serves the same function as the stack in an ordinary LR parser; we will simply refer to it as "the stack." The second stack, which we will call the "input stack," initially contains the input string, with the first symbol on the top of the stack and the endmarker ($) at the bottom. Symbols will be popped off the input stack, just as an ordinary LR parser reads its input from left to right. However, a GPLR parser will also push symbols back onto the input stack, at which point they will serve as the next input symbol.

[1] However, there are some LR(k) *grammars*, $k \geq 2$, which are not GPLR(1) [23, p. 111].

[2] Given a context-free grammar (V_N, V_T, P, S) and a rightmost derivation

$$S \overset{\text{rm}}{\Longrightarrow}^* \alpha A w \overset{\text{rm}}{\Longrightarrow} \alpha \beta w \overset{\text{rm}}{\Longrightarrow} xw,$$

we say that β is a *handle* of the right-sentential form $\alpha\beta w$. In English, "a handle of a right-sentential form is any substring which is the right side of some production and which can be replaced by the left side of that production so that the resulting string is also a right-sentential form" [3, p. 179].

3.1 Shift-Reduce-Cancel-Continue Parsing

While an ordinary LR parser has four actions—*shift, reduce, accept,* and *error*—a GPLR parser can also *cancel* or *continue. Continuation* is the action which allows the parser to temporarily ignore a conflict; *cancellation* is used in returning to the site where the conflict occurred.

3.1.1 Continuation

A GPLR parser continues when there are at least two different (shift and/or reduce) actions it could have taken. To *continue,* it places a special marker on the stack (denoted $\perp$) and changes the current state to a *continuation state.* Like goto states, continuation states are constructed to follow from an existing state on a particular input symbol, so we can speak of, say, *the continuation state from* q_0 *on* B. After entering a continuation state, the parser proceeds to make reductions on the remaining input, but it only makes reductions that would have been made *no matter which of the possible prior actions was taken.* This is ensured by two means.

First, the stack marker ($\perp$) is treated as the bottom of the stack; the parser is not allowed to look at any of the symbols below $\perp$ until the $\perp$ symbol is explicitly removed from the stack by a *cancel* action.[3] This will be described further in the discussion of cancellation, but for now, suffice it to say that this will prevent the continuation site from accidentally being "absorbed" into a large reduction.

Second, the continuation state is *constructed* to guarantee that only conservative reductions are made. Suppose one shift and two reduce actions were possible in the original state on a particular lookahead symbol. Then there are several shift items that should be carried over to the continuation state; if it made either reduction, there are several states it could have moved to, and the items from these states are carried over as well. In other words, the continuation state is constructed to include all of the items that would be valid no matter which of the original actions was taken.[4] As symbols are shifted, the goto states from this continuation state will be followed, and if a state is ever reached where only one reduce action is possible—i.e., there is only a single reduce item—then we can be assured that, no matter which of the original actions was taken, the parser would have ended up in a state with only this single reduce item.

[3] The intuition is that we are "spawning a new parser" to handle the remaining input. As we will see, this "new parser" will eventually pass a cancellation symbol and a nonterminal back to the original parser so that it can decide which of the possible actions was appropriate.

[4] The logic is similar to that for the removal of ϵ-productions in a finite automaton: The continuation state is really the union of several other states, and its goto successors are the unions of the original states' goto successors, so these states are really tracking the progress of several paths through the original automaton. The difference between this construction and the ϵ-removal construction is that we reduce only when we would reduce in *all* of the original states, while an automaton with ϵ-productions removed will accept when it would have accepted along *any* of the original paths.

3.1.2 Cancellation

Suppose the stack of a GPLR parser contains the following symbols (from bottom to top)

$$a \ \perp \ b \ c$$

and it has just reached a state with a single reduce item allowing it to reduce abc to some nonterminal, say F. In other words, after it shifted the a, it was not sure whether to reduce a to something or shift the b, so it *continued.* But now that it has seen the b and c following it, it knows for sure that shifting b was correct. Unfortunately, since it is not allowed to look at symbols below $\perp$, the parser cannot reduce abc, as it can only see bc.

The solution is something called an *insufficient stack depth reduction.* The parser will pop as many symbols as possible (b and c), remove the $\perp$ marker, and place a *cancellation symbol* at the front of the input stream, followed by the nonterminal it attempted to reduce to (F). Cancellation symbols indicate (1) which production the parser wants to reduce by, and (2) what suffix of that production's right-hand side was actually visible on the stack. In this case, we wanted to reduce by $F \rightarrow abc$, but only bc was visible on the stack, so the cancellation symbol will be $\langle F \rightarrow a \cdot bc \rangle$.

Now that the $\perp$ marker and the stack contents above it are gone, the parser will return to the state is was in before it continued (the state on top of the stack), but it will use this cancellation symbol as the lookahead token. It will recognize that there is, in fact, an a on top of its stack, so it will *cancel* a, i.e., pop a (and the corresponding state) off the stack and remove the cancellation symbol from the input stream.

It is also possible to *cancel* with insufficient stack depth. In this case, if the parser needs to *cancel* $\langle A \rightarrow \alpha\beta \cdot \gamma \rangle$ but only β is visible on the stack, the cancellation symbol is popped from the input stack, the symbols of β and the $\perp$ symbol below them are popped from the parser stack, and a new cancellation symbol $\langle A \rightarrow \alpha \cdot \beta\gamma \rangle$ is pushed onto the input stack. As with an insufficient stack depth reduction, the parser's current state is set to the new state on top of the stack, and this new cancellation symbol serves as the next input symbol.

3.1.3 Cancellation Symbols and Nonterminals as Input

Recall that, in an insufficient stack depth reduction, both the cancellation symbol and the reduced nonterminal are placed at the beginning of the input stream. The presence of these new types of symbols in the input stream causes some complication.

Consider a grammar with the productions $A \rightarrow a$ and $B \rightarrow Ab$, and suppose for illustrative purposes that a parser has $a{\perp}b$ on its stack. It wants to reduce by $B \rightarrow Ab$, and, seeing b on the stack, it pops it and places $\langle B \rightarrow A \cdot b \rangle \ B$ at the beginning of the input stream. Then the parser cannot *cancel,* since A is not at the top of its stack. It must first reduce a to A; then it can *cancel*.

Similarly, when the nonterminal B appears at the front of the input stream, if the current state contains an item with a B after the dot, it can simply shift the nonterminal B onto its stack and go to the appropriate goto state, as it did for shifting terminal symbols. But if the current state does not contain an item with a B after the dot, it will need to reduce first.

To summarize, when the lookahead symbol is a cancellation symbol, the parser can either *cancel* or *reduce*. If it is a nonterminal, it can either *shift* or *reduce*.

3.2 The GPLR CFSM

Armed with a general understanding of how a GPLR parser should work, we will now focus on the characteristic finite state machine (cfsm) which drives it. Just as the states of an LR(k) cfsm were sets of LR(k) items, the states of a GPLR cfsm will be sets of *GPLR items.*

3.2.1 GPLR Items and Cancellation Lookahead

A GPLR item consists of two component LR(0) items and is written, e.g.,

$$[A \to aB\cdot,\ C \to D \cdot e].$$

The second component item cannot have a dot at the right or left end. In some cases, there will not be a second component, e.g.,

$$[A \to aB\cdot,\ ].$$

The first component item has the exact same meaning as in the LR(0) cfsm; it is called the *core* of the GPLR item. We refer to the GPLR item as a *shift item* or *reduce item* if its core is an LR(0) shift item or reduce item, respectively. The second component is the *cancellation lookahead*; it serves a function similar to the lookahead component in LR(k) items. Specifically, if a reduce item such as $[A \to aB\cdot,\ C \to D \cdot e]$ appears in a state, then if the parser reduces aB to A (and possibly makes a few more reductions immediately after that), it *may* be possible for it to end up in a state where it can *cancel* $\langle C \to D \cdot e\rangle$—i.e., a state containing an item of the form $[C \to D \cdot e,\ \textit{(something)}]$. As in the LR(0) cfsm, if a state contains an item with the core $[C \to D \cdot e]$, then we can be assured that D will appear on top of the stack, since such an item will only appear if the machine started with $[C \to \cdot De]$ and then recognized a D. So, consequently, a parser can *cancel* anything that appears as a core in its current state.

The purpose of GPLR lookahead, then, is to determine whether a reduction may lead to a state where a particular cancellation is possible. This will be used to determine the parser's action when a cancellation symbol appears in the input stream. So, for example, if $[A \to aB\cdot\,,\ C \to D\cdot e]$ appears in the parser's current state, and the cancellation symbol $\langle C \to D\cdot e\rangle$ appears next in the input, then it should *reduce* by $A \to aB$, since it is possible (though not guaranteed) that it will end up in a state where cancellation by $\langle C \to D\cdot e\rangle$ is feasible.

3.2.2 Constructing the Machine

The construction of the GPLR cfsm is very similar to the construction of the LR(k) cfsm, adjusted appropriately for cancellation lookahead.

The initial state of the GPLR cfsm is the closure of $\{[S' \to\ \cdot S\$,\ ]\}$; the final state is the (unique) state containing $[S' \to S\cdot\$,\ ]$.

Before describing how closures are computed, we will explain how goto states are computed. As with the LR(k) construction, to compute the goto state from a (closed) state q on a terminal or nonterminal X, all of the items in q with an X after the dot in the core component are incorporated into the new state with the dot moved forward one position, and lookaheads are simply carried over unchanged from the original state. To see why, suppose a state contains the item $[C \to a\cdot bc,\ D \to e\cdot F]$. Then the goto state on b will contain $[C \to ab\cdot c,\ D \to e\cdot F]$. In the original state, the lookahead $D \to e\cdot F$ meant that, after it becomes possible to reduce abc to C, doing so (and possibly making further reductions) could put the parser in a state containing the core $D \to e\cdot F$. This statement is just as true in the new (goto) state as it was in the original state: It does not depend on the position of the dot in the core.

The closure of a set of GPLR items is computed as follows. Each item with a nonterminal after the dot has the form

$$[A \to \alpha\cdot B\beta,\ C \to \gamma\cdot\delta].$$

We find all of the productions for B in the grammar; call them $B \to \alpha_1$, $B \to \alpha_2$, …, $B \to \alpha_n$. We have two cases to consider.

If β derives at least one string other than ϵ, we should ensure that the state contains each of the items

$$\begin{gathered}[B \to\ \cdot\alpha_1,\ A \to \alpha B\cdot\beta],\\ [B \to\ \cdot\alpha_2,\ A \to \alpha B\cdot\beta],\\ \ldots,\\ [B \to\ \cdot\alpha_n,\ A \to \alpha B\cdot\beta].\end{gathered}$$

Intuitively, the item $[A \to \alpha\cdot B\beta,\ C \to \gamma\cdot\delta]$ indicates that the parser is in a state where it needs to recognize a B, but it will get sidetracked doing this (it will have to go through several more states and eventually *reduce* something to B), but after it does so, it will return to this state and jump forward to the goto state on B. Then it may be possible to *cancel* $\langle A \to \alpha B\cdot\beta\rangle$ in that state.

If $\beta \Rightarrow^* \epsilon$ (or $\beta = \epsilon$), we should ensure that the state contains each of the items

$$\begin{gathered}[B \rightarrow \ \cdot \alpha_1,\ C \rightarrow \gamma \cdot \delta], \\ [B \rightarrow \ \cdot \alpha_2,\ C \rightarrow \gamma \cdot \delta], \\ \ldots, \\ [B \rightarrow \ \cdot \alpha_n,\ C \rightarrow \gamma \cdot \delta].\end{gathered}$$

As before, the parser will get sidetracked recognizing a B but will then return to this state and jump forward to the goto state on B. The goto state on B will contain the item

$$[A \rightarrow \alpha B \cdot \beta,\ C \rightarrow \gamma \cdot \delta].$$

In the event that ϵ is reduced to β and the item

$$[A \rightarrow \alpha B \beta \cdot,\ C \rightarrow \gamma \cdot \delta]$$

becomes valid, the parser should be prepared to *cancel* $\langle C \rightarrow \gamma \cdot \delta \rangle$ without consuming any further input.

When it is possible to *continue* in a state q (we will describe how to determine this momentarily) on a particular symbol, we will also need to compute a *continuation state* from q on that symbol. The symbol will be either a terminal or nonterminal; call it X. The continuation state from q on X should contain the closure of all of the following items.

- All of the items in q that have X after the dot in the core should be included; i.e., if $[A \rightarrow \alpha \cdot X\beta,\ B \rightarrow \gamma \cdot \delta]$ is in q, then it is also included in the continuation state from q on X. Since there is an X after the dot, it is possible to *shift* X; by including these items and closing the state, we are retaining the ability to *shift* X after we *continue.*

- We should determine all of the *reduce* items in q that have an X after the dot in the *lookahead* item. Each of these items has the form $[A \rightarrow \alpha Y \cdot,\ B \rightarrow \gamma \cdot X\delta]$ for some A, α, Y, B, γ, and δ. For each of these items, we should find every production whose right-hand side includes a nonterminal which derives a string ending in B; that is, we should find every production $C \rightarrow \zeta D \eta$ (for some C, ζ, D, η) such that $D \Rightarrow^* \theta B$ for some θ.[5] Then the continuation state should include $[B \rightarrow \gamma \cdot X\delta,\ C \rightarrow \zeta D \cdot \eta]$. Effectively, this is including every possible item that could appear if we had reduced $\alpha X \beta$ to A in q (and then moved to the goto state on A). Since $B \rightarrow \gamma \cdot \delta$ was the lookahead in the original item, through some sequence of reductions, we could end up in a state with that core. We then need to find every lookahead that could be paired with that core, so we form all of the lookaheads which have a dot immediately after a B (or something that derives something ending in B)—all of the lookaheads

[5] Remember that $B \Rightarrow^* B$ when computing this.

corresponding to an core that could appear immediately after a B was recognized.

After a continuation state has been added to the cfsm, goto states from that continuation state will need to be added, and goto states from those goto states, etc. Some of these states may already exist in the cfsm, i.e., there may already be a state consisting of the desired items.

The initial state, its goto states, the goto states from those goto states, etc. are called *unprimed states.* All of the other states—continuation states and their goto states that were not already in the cfsm—are called *primed states.*

3.2.3 The GPLR Action Function

The GPLR Action function is straightforward: Shift and reduce items induce *shift* and *reduce* actions, as expected. The contexts in which a parser may *cancel* or *continue* have already been described and should not be surprising.

Let q be a state in the cfsm (i.e., a set of GPLR items) and χ an input symbol (terminal, nonterminal, or cancellation symbol).

- If χ is a terminal or nonterminal and q contains an item of the form $[A \to \alpha \cdot \chi\beta,\ B \to \gamma \cdot \delta]$, then a parser in state q with input χ should *shift* χ (note that we must have $\chi \in V$).
- If q contains an item of the form $[A \to \alpha \cdot ,\ B \to \gamma \cdot \delta]$, then a parser in state q with input χ should *reduce* by $A \to \alpha$ if one of the following holds:
  - $\delta \Rightarrow^* \chi\iota$ for some $\iota \in V^*$ (note that this requires $\chi \in V$), or
  - $\chi = \langle B \to \gamma \cdot \delta \rangle$.
- If $\chi = \langle A \to \alpha \cdot \beta \rangle$ and q contains a item of the form $[A \to \alpha \cdot \beta,\ B \to \gamma \cdot \delta]$ (so $\alpha, \beta \neq \epsilon$), then a parser in state q with input $\chi = \langle A \to \alpha \cdot \beta \rangle$ should *cancel* $\langle A \to \alpha \cdot \beta \rangle$.
- If more than one of the above applies, the parser should *continue,* with one exception. If both a *cancel* and a *reduce* action are indicated for an *unprimed* state, the conflict is unresolvable, and parser construction fails.

3.2.4 Summary of the GPLR CFSM Construction

We will now summarize the GPLR cfsm construction and describe when to construct continuation states. The cfsm is constructed as follows.

1. The initial state is constructed as the closure of $[S' \to \cdot S\$,\ ]$.
2. In a given state q, if there is an item $[A \to \alpha \cdot X\beta,\ B \to \gamma \cdot \delta]$, then the goto state from q on X is constructed, and there is a transition from q to this state on symbol X.

3. As each state q is constructed, if a *continue* action is indicated for any symbol χ, then a continuation state from q on χ is constructed.

4. The previous two steps are repeated until no new states can be added to the machine.

5. The accepting state is the (unique) state containing $[S' \to S \cdot \$,\ ]$.

3.3 Formalizing the GPLR Construction*

To facilitate the proof of correctness in the following chapter, we will now give a formal specification of the GPLR construction. GPLR parsers will be modeled as two-stack pushdown automata; we will begin with a description of such automata, and then describe the construction of the cfsm, and end with with the automaton's semantics. The reader uninterested in the formal specification and proof of correctness may skip this and all subsequent sections whose titles are marked with asterisks.

3.3.1 Two-Stack Pushdown Automata

A GPLR recognizer, which we will formally define later, is modeled as a two-stack pushdown automaton. To simplify the presentation, we will use a nonstandard definition of such an automaton which allows the two stacks to have different stack alphabets.

Definition 1 *A* two-stack pushdown automaton *is an octuple*

$$(Q, \Sigma, \Gamma_1, \Gamma_2, \Delta, q_0, \tau_0, F)$$

where

- Q *is a finite, non-empty set of states;*
- Σ *is a finite set of input symbols;*
- Γ_1 *is a finite, non-empty set of symbols that can appear on the first stack;*
- Γ_2 *is a finite set of symbols that can appear on the second stack, where* $\Sigma \subseteq \Gamma_2$*;*
- $q_0 \in Q$ *is the initial state;*
- $\tau_0 \in \Gamma_1^*$ *is the initial contents of the first stack;*
- $F \subseteq Q$ *is the set of final states; and*
- Δ*, the transition relation, is a finite subset of*

$$(Q \times \Gamma_1^* \times \Gamma_2^*) \times (Q \times \Gamma_1^* \times \Gamma_2^*).$$

The transition relation will usually be given by an explicit listing of rules of the form

$$(q, \alpha, \beta) \longrightarrow (q', \alpha', \beta').$$

Such a rule intuitively means that an automaton in state q with α on top of its first stack and β on top of its second stack will make a transition to state q', replacing α with α' on the first stack and β with β' on the second stack. This will be defined more precisely below.

Definition 2 *A* configuration *or* snapshot *of a two-stack pushdown automaton $M = (Q, \Sigma, \Gamma_1, \Gamma_2, \Delta, q_0, \tau_0, F)$ is an ordered triple, written*

$$\alpha \textcircled{q} \beta$$

where

- *$\alpha \in \Gamma_1^*$ is the contents of the first stack, written with the topmost symbol appearing on the right;*
- *$q \in Q$ is the current state; and*
- *$\beta \in \Gamma_2^*$ is the contents of the second stack, written with the topmost symbol appearing on the left.*

When the current state is not important, it will be written $\odot$.

Definition 3 *Let $M = (Q, \Sigma, \Gamma_1, \Gamma_2, \Delta, q_0, \tau_0, F)$ be a two-stack pushdown automaton. The $\vdash$ relation (often read "moves to" or "derives") is defined on configurations of M according to the following. Let $\alpha\beta \textcircled{q} \gamma\delta$ be a configuration of M. Then*

$$\alpha\beta \textcircled{q} \gamma\delta \;\vdash_M\; \alpha\beta' \textcircled{q'} \gamma'\delta.$$

iff Δ contains a rule

$$(q, \beta, \gamma) \longrightarrow (q', \beta', \gamma')$$

for some $q' \in Q$, $\beta' \in \Gamma_1^*$, and $\gamma' \in \Gamma_2^*$. The subscript M will be omitted when it is clear from context.

Definition 4 *The* action *of a two-stack pushdown automaton M on input $w \in \Sigma^*$ is any sequence*[6] *s of configurations that satisfies*

- *$s_0 = \tau_0 \textcircled{q_0} w$ and*
- *$s_i \vdash s_{i+1}$ for every i.*

When only one such sequence exists for every w, the automaton is said to be deterministic*; otherwise, it is* nondeterministic.

[6] Given a set S (in our case, the set of configurations of M), a *sequence* over s is a function mapping the natural numbers {0, 1, ...} to elements of S. The *terms* of the sequence are the elements $s(0), s(1), \ldots$, which will will denote by $s_0, s_1, \ldots$.

3.3.2 Constructions on Sets of GPLR Items

In the forthcoming formalisms, $\alpha, \beta, \gamma, \delta, \zeta, \eta$, and θ denote strings in V^*; A, B, C, and D denote nonterminals; X and Y denote symbols in $V = V_N \cup V_T$. We will use $\mathscr{I}_{\text{GPLR}}$ to refer to the set of all GPLR items for a grammar, that is, the set consisting of every item appearing in every state of the GPLR cfsm (this is formalized in Definition 15). Free variables are existentially quantified unless stated otherwise. So, for example,

$$f(I, X) \stackrel{\text{def}}{=} \{[A \to \alpha \cdot X\beta,\ B \to \gamma \cdot \delta] \in I\}$$

abbreviates the more precise but cumbersome

$$\begin{aligned} f(I, X) \stackrel{\text{def}}{=} \{[A \to \alpha \cdot X\beta,\ B \to \gamma \cdot \delta] \ \mid \ & [A \to \alpha \cdot X\beta,\ B \to \gamma \cdot \delta] \in I, \\ & \text{for some } \alpha, \beta, \gamma, \delta \in V^*,\ A, B \in V_N\}. \end{aligned}$$

Cancellation Symbols

Definition 5 *Given an augmented grammar $G = (V_N, V_T, P, S)$, the set V_C of cancellation symbols for G is*

$$V_C \stackrel{\text{def}}{=} \{\langle A \to \alpha \cdot \beta\rangle \ \mid \ A \to \alpha\beta \in P,\ |\alpha| > 0,\ \textit{and}\ |\beta| > 0\}.$$

Subset Selectors

We begin by defining some notation: $S_{I,\chi}$, $R_{I,\chi}$, and $C_{I,\chi}$ will be used to refer to the shift, reduce, and cancellation items, respectively, in a state I on lookahead χ.

Definition 6 *Given a set $I \subseteq \mathscr{I}_{\text{GPLR}}$ of GPLR items and a symbol $\chi \in V \cup V_C$, the* GPLR shift items in I for lookahead χ *are precisely the elements of the set*

$$S_{I,\chi} \stackrel{\text{def}}{=} \begin{cases} \{[A \to \alpha \cdot \chi\beta,\ B \to \gamma \cdot \delta] \in I\} & \textit{if } \chi \in V_N \cup V_T \\ \emptyset & \textit{if } \chi \in V_C. \end{cases}$$

Definition 7 *Given a set $I \subseteq \mathscr{I}_{\text{GPLR}}$ of GPLR items and a symbol $\chi \in V \cup V_C$, the* GPLR reduce items in I for lookahead χ *are precisely the elements of the set*

$$R_{I,\chi} \stackrel{\text{def}}{=} \begin{cases} \{[A \to \alpha \cdot,\ B \to \gamma \cdot \delta] \in I \ \mid \ \delta \Rightarrow^* \chi w,\ \exists w \in V_T^*\} & \textit{if } \chi \in V_N \cup V_T \\ \{[A \to \alpha \cdot,\ B \to \gamma \cdot \delta] \in I\} & \textit{if } \chi = \langle B \to \gamma \cdot \delta\rangle. \end{cases}$$

Definition 8 *Given a set $I \subseteq \mathscr{I}_{\text{GPLR}}$ of GPLR items and a symbol $\chi \in V \cup V_C$, the* GPLR cancellation items in I for lookahead χ *are precisely the elements of* *the set*

$$C_{I,\chi} \overset{\text{def}}{=} \begin{cases} \emptyset & \text{if } \chi \in V_N \cup V_T \\ \{[A \to \alpha \cdot \beta,\ B \to \gamma \cdot \delta] \in I\} & \text{if } \chi = \langle A \to \alpha \cdot \beta \rangle. \end{cases}$$

When considering the cancellation items in I, often we are not concerned with the lookahead components. Thus, we define a function Core which extracts the first component of a GPLR item.

Definition 9 *Given a GPLR item, the* core *of the item is the LR(0) item*

$$\mathit{Core}([A \to \alpha \cdot \beta,\ B \to \gamma \cdot \delta]) \overset{\text{def}}{=} [A \to \alpha \cdot \beta].$$

As usual, we can extend this definition to operate on a set $I \subseteq \mathscr{I}_{\scriptscriptstyle GPLR}$ of GPLR items:

$$\mathit{Core}(I) \overset{\text{def}}{=} \{\mathit{Core}(i) \mid i \in I\}.$$

Successor Item Sets

We will now formalize the notions of closure, goto successor, and continuation successor.

Definition 10 *Given a set $I \subseteq \mathscr{I}_{\scriptscriptstyle GPLR}$ of GPLR items for a grammar $G = (V_N, V_T, P, S)$, the* GPLR closure of I *is defined as the smallest set satisfying*

$$\begin{aligned} \mathit{Closure}_{\scriptscriptstyle GPLR}(I) \overset{\text{def}}{=}\ & I \\ \cup\ & \{[C \to\ \cdot \zeta,\ A \to \alpha C \cdot \beta] \mid \\ & \quad [A \to \alpha \cdot C\beta,\ B \to \gamma \cdot \delta] \in \mathit{Closure}_{\scriptscriptstyle GPLR}(I) \\ & \quad \text{and } C \to \zeta \in P,\ \text{where } \beta \Rightarrow^* w \text{ for some } w \in V_T^+\} \\ \cup\ & \{[C \to\ \cdot \zeta,\ B \to \gamma \cdot \delta] \mid \\ & \quad [A \to \alpha \cdot C\beta,\ B \to \gamma \cdot \delta] \in \mathit{Closure}_{\scriptscriptstyle GPLR}(I) \\ & \quad \text{and } C \to \zeta \in P,\ \text{where } \beta \Rightarrow^* \epsilon\}. \end{aligned}$$

Definition 11 *Given a set $I \subseteq \mathscr{I}_{\scriptscriptstyle GPLR}$ of GPLR items and a symbol $X \in V$, the* goto successor on lookahead X *is given by*

$$\begin{aligned} \mathit{Goto}_{\scriptscriptstyle GPLR}(I) \overset{\text{def}}{=} \mathit{Closure}_{\scriptscriptstyle GPLR}(\{&[A \to \alpha X \cdot \beta,\ B \to \gamma \cdot \delta] \mid \\ & [A \to \alpha \cdot X\beta,\ B \to \gamma \cdot \delta] \in I\}). \end{aligned}$$

Definition 12 *Given a set $I \subseteq \mathscr{I}_{\scriptscriptstyle GPLR}$ of GPLR items for a grammar $G = (V_N, V_T, P, S)$ and a symbol $X \in V$, the* continuation successor on lookahead X

is given by

$$\begin{array}{rcl} \mathit{Cont}_{\scriptscriptstyle GPLR}(I) & \overset{\text{def}}{=} & \{[B \to \gamma \cdot X\delta,\ C \to \zeta \cdot \eta] \mid \\ & & \qquad [B \to \gamma \cdot X\delta,\ C \to \zeta \cdot \eta] \in I\} \\ & \cup & \{[B \to \gamma \cdot Y\delta,\ C \to \zeta Z \cdot \eta] \mid \\ & & \qquad [A \to \alpha X \cdot\, ,\ B \to \gamma \cdot Y\delta] \in I, \\ & & \qquad C \to \zeta Z \cdot \eta \in P,\ \text{and } Z \Rightarrow^* \theta B\}. \end{array}$$

Definition 13 *Given a set $I \subseteq \mathscr{I}_{\scriptscriptstyle GPLR}$ of GPLR items and a symbol $\chi \in V \cup V_C$, the* GPLR action function *is defined such that*

$$\mathit{Action}_{\scriptscriptstyle GPLR}(I,\chi) \overset{\text{def}}{=}$$

$$\begin{cases} \textit{shift and go to } \mathit{Goto}_{\scriptscriptstyle GPLR}(I,\chi) & \textit{if } |S_{I,\chi}| \geq 1 \textit{ and } |R_{I,\chi}| = 0 \\ & \textit{(note that this requires } \chi \in V\textit{)} \\ \textit{reduce by } A \to \alpha & \textit{if } |R_{I,\chi}| = 1, |S_{I,\chi}| = 0, \textit{ and } |C_{I,\chi}| = 0, \\ & \ \textit{where } [A \to \alpha,\ B \to \gamma \cdot \delta] \in R_{I,\chi} \\ \textit{cancel } \langle A \to \alpha \cdot \beta \rangle & \textit{if } |R_{I,\chi}| = 0 \textit{ and } |\mathit{Core}(C_{I,\chi})| = 1 \\ & \textit{where } [A \to \alpha \cdot \beta] \in \mathit{Core}(C_{I,\chi}) \\ \textit{continue in } \mathit{Cont}_{\scriptscriptstyle GPLR}(I,\chi) & \textit{if any of the following hold:} \\ & |S_{I,\chi}| \geq 1 \textit{ and } |R_{I,\chi}| \geq 1 \\ & |R_{I,\chi}| \geq 1 \textit{ and } |\mathit{Core}(C_{I,\chi})| \geq 1 \\ & |R_{I,\chi}| \geq 2 \\ & |\mathit{Core}(C_{I,\chi})| \geq 2 \\ \textit{accept} & \textit{if } [S' \to S \cdot \$,\ ] \in I \\ \textit{error} & \textit{otherwise.} \end{cases}$$

3.3.3 The GPLR Recognizer

Definition 14 *Given an augmented grammar $G = (V_N, V_T, P, S')$, the* canonical collection of sets of GPLR items, *$\mathscr{C}_{\scriptscriptstyle GPLR}$, is defined inductively as the smallest set satisfying the following.*

- *$\mathit{Closure}_{\scriptscriptstyle GPLR}(\{[S' \to\ \cdot S\$,\ ]\})$ is in $\mathscr{C}_{\scriptscriptstyle GPLR}$.*
- *If $I \in \mathscr{C}_{\scriptscriptstyle GPLR}$, then for every $X \in V$, $\mathit{Goto}_{\scriptscriptstyle GPLR}(I,X)$ is in $\mathscr{C}_{\scriptscriptstyle GPLR}$.*
- *If $I \in \mathscr{C}_{\scriptscriptstyle GPLR}$ and $\mathit{Action}_{\scriptscriptstyle GPLR}(I,\chi) =$ continue for some $\chi \in V \cup V_C$, then $\mathit{Cont}_{\scriptscriptstyle GPLR}(I,X)$ is in $\mathscr{C}_{\scriptscriptstyle GPLR}$.*

Definition 15 *Given the canonical collection $\mathscr{C}_{\scriptscriptstyle GPLR}$ of sets of GPLR items for a grammar, the set $\mathscr{I}_{\scriptscriptstyle GPLR}$ of all GPLR items in a grammar is defined such that*

$$\mathscr{I}_{\scriptscriptstyle GPLR} \overset{\text{def}}{=} \bigcup_{I \in \mathscr{C}_{GPLR}} I.$$

Theorem 1 *$\mathscr{C}_{\scriptscriptstyle GPLR}$ contains a finite number of sets, and each set in $\mathscr{C}_{\scriptscriptstyle GPLR}$ contains a finite number of GPLR items.*

Proof This follows from the fact that there are a finite number of productions in the grammar, and each production has finite length, so the number of distinct GPLR items is finite. Consequently, the number of sets of items is also finite. □

Definition 16 *Each set of items in $\mathscr{C}_{\scriptscriptstyle GPLR}$ is classified as either* primed *or* unprimed *according to the following.*

- *$Closure_{\scriptscriptstyle GPLR}(\{[S' \to \cdot S\$,\ ]\})$ is unprimed.*
- *If $I \in \mathscr{C}_{\scriptscriptstyle GPLR}$ is unprimed, then, for every $X \in V$, $Goto_{\scriptscriptstyle GPLR}(I, X)$ is unprimed.*
- *All other sets are primed.*

In other words, primed sets are continuation successors and goto successors that are constructed solely due to continuation.

Definition 17 *The* GPLR recognizer *for an augmented grammar $G' = (V_N, V_T, P, S')$ is the two-stack pushdown automaton such that*

- *$Q = \mathscr{C}_{\scriptscriptstyle GPLR}$;*
- *$\Sigma = T$;*
- *$\Gamma_1 = (V \cup V_C \cup \{\bot\}) \times Q$, where $\bot$ is a distinguished symbol (we will usually write these pairs as $\overset{X}{q}$ rather than (X, q));*
- *$\Gamma_2 = V \cup V_C$;*
- *$q_0 = Closure_{\scriptscriptstyle GPLR}(\{[S' \to \cdot S\$,\ ]\})$;*
- *$\tau_0 = (\bot, q_0)$;*
- *$F = \{\, q \in Q \mid Action_{\scriptscriptstyle GPLR}(q) = accept\}$; and*
- *$\Delta = \bigcup_{q \in Q, \chi \in (V \cup V_C)} \Delta_{q,\chi}$. The definition of $\Delta_{q,\chi}$ is displayed in Figure 3.1.*

Theorem 2 *GPLR recognizers are deterministic.*

Proof A simple induction on sequences of configurations. □

$$
\Delta_{q,\chi} \overset{\mathrm{def}}{=} \left\{ \begin{array}{lr}
\Big\{ (q,\ \epsilon,\ \chi) \longrightarrow \Big(q',\ \genfrac{}{}{0pt}{}{\chi}{q'},\ \epsilon\Big) \Big\} & \\
\qquad\qquad \text{if } \mathrm{Action}_{\textsc{gplr}}(q,\chi) = \text{shift and go to } q' & \textit{(shift)} \\[2ex]
\Big\{ \Big(q,\ \genfrac{}{}{0pt}{}{Y}{q'} \genfrac{}{}{0pt}{}{X_1}{q_1} \genfrac{}{}{0pt}{}{X_2}{q_2} \ldots \genfrac{}{}{0pt}{}{X_n}{q_n},\ \chi\Big) \longrightarrow \Big(q'',\ \genfrac{}{}{0pt}{}{Y}{q'} \genfrac{}{}{0pt}{}{A}{q''},\ \chi\Big) \ \Big|\ Y \in V \cup V_C; q', q_1, \ldots, q_n \in Q; q'' = \mathrm{Goto}_{\textsc{gplr}}(q', A) \Big\} & \textit{(reduce)} \\
\cup \Big\{ \Big(q,\ \genfrac{}{}{0pt}{}{Y}{q'} \genfrac{}{}{0pt}{}{\perp}{q''} \genfrac{}{}{0pt}{}{X_i}{q_i} \ldots \genfrac{}{}{0pt}{}{X_n}{q_n},\ \chi\Big) \longrightarrow \Big(q',\ \genfrac{}{}{0pt}{}{Y}{q'},\ \langle A \to X_1 \ldots X_{i-1} \cdot X_i \ldots X_n \rangle\, A\, \chi\Big) \ \Big|\ 1 < i < n, Y \in V \cup V_C; q', q'', q_1, \ldots, q_n \in Q \Big\} & \textit{(reduce-c)} \\
\qquad\qquad \text{if } \mathrm{Action}_{\textsc{gplr}}(q,\chi) = \text{reduce by } A \to X_1 X_2 \ldots X_n & \\[2ex]
\Big\{ \Big(q,\ \genfrac{}{}{0pt}{}{Y}{q'} \genfrac{}{}{0pt}{}{X_1}{q_1} \genfrac{}{}{0pt}{}{X_2}{q_2} \ldots \genfrac{}{}{0pt}{}{X_n}{q_n},\ \chi\Big) \longrightarrow \Big(q',\ \genfrac{}{}{0pt}{}{Y}{q'},\ \epsilon\Big) \ \Big|\ Y \in V \cup V_C; q', q_1, \ldots, q_n \in Q \Big\} & \textit{(cancel)} \\
\cup \Big\{ \Big(q,\ \genfrac{}{}{0pt}{}{Y}{q'} \genfrac{}{}{0pt}{}{\perp}{q''} \genfrac{}{}{0pt}{}{X_i}{q_i} \ldots \genfrac{}{}{0pt}{}{X_n}{q_n},\ \chi\Big) \longrightarrow \Big(q',\ \genfrac{}{}{0pt}{}{Y}{q'},\ \langle A \to X_1 \ldots X_{i-1} \cdot X_i \ldots X_n \beta \rangle\Big) \ \Big|\ 1 < i < n, Y \in V \cup V_C; q', q'', q_1, \ldots, q_n \in Q \Big\} & \textit{(cancel-c)} \\
\qquad\qquad \text{if } \chi = \langle A \to X_1 \ldots X_n \cdot \beta \rangle \text{ and } \mathrm{Action}_{\textsc{gplr}}(q,\chi) = \text{cancel} & \\[2ex]
\Big\{ \Big(q,\ \genfrac{}{}{0pt}{}{Y}{q},\ \chi\Big) \longrightarrow \Big(q',\ \genfrac{}{}{0pt}{}{Y}{q} \genfrac{}{}{0pt}{}{\perp}{q'},\ \chi\Big) \ \Big|\ Y \in V \cup V_C; q' = \mathrm{Cont}_{\textsc{gplr}}(q,\chi) \Big\} & \textit{(continue)} \\
\qquad\qquad \text{if } \mathrm{Action}_{\textsc{gplr}}(q,\chi) = \text{continue.} &
\end{array} \right.
$$

Figure 3.1: Definition of $\Delta_{q,\chi}$ (see Definition 17)

4 Adding Incrementality

The parsing algorithms described thus far start at the beginning of the input, process it from left to right, and then terminate after reaching the end of input. Sometimes this is not desirable. For example, when a document is being edited in a “smart” syntax-highlighting editor, it may be desirable to re-parse the document frequently, perhaps even on every keystroke. However, re-parsing the entire document (and reconstructing an entire syntax tree) is time-consuming. In particular, when only a small portion of the document has been changed, it is desirable to re-parse only the portion that has changed and modify the appropriate part of the syntax tree, leaving the unaffected parts alone.

That is precisely the goal of *incremental parsing.* Augusto Celentano’s “Incremental LR Parsers” [4] is the traditional reference on the subject’s application to LR parsers, although his work was improved on by Agrawal and Detro [1] and is largely based on previous work by Ghezzi and Mandrioli [11, 12].

In this chapter, we will review Celentano’s algorithm for incremental LR(1) parsing and, more importantly, illustrate and prove formally that it applies to GPLR parsers *without modification.*

4.1 Incremental LR(1) Parsing: Celentano’s Algorithm

4.1.1 Theory

An LR(1) parser consists of just three things: (1) a stack, (2) a state machine, and (3) an input stream. Thus, we can give a complete “snapshot” of an LR(1) parser at any point during its parse by describing the entire contents of its stack, its current state, and the remaining input. Such a snapshot is called a *configuration,* and the entire sequence of configurations exhibited by a successful parse is called a *parse sequence.*

We will write configurations as $\alpha \textcircled{q} w$, where α is the stack contents, q is the state in the cfsm, and w is the remaining input. So, for example, every LR(1) parse sequence has the form

$$\underset{q_0}{\overset{-}{}}\ \textcircled{\scriptsize q_0}\, w, \ldots, \underset{q_f}{\overset{S}{}}\ \textcircled{\scriptsize q_f}\, \epsilon.$$

Celentano's algorithm relies on an important observation: At any point during a parse, the next action of an LR(1) parser is completely described by (1) the state of its cfsm, (2) the contents of its stack, and (3) *only the first symbol of the remaining input.* In other words, if an LR parser in state q has α on its stack and a is the next input symbol, it will always make exactly the same move, so the next state and stack contents will also be the same. By induction, that statement applies to strings as well: If a parser is in state q with α on its stack, its state and stack contents after consuming the input $a_1 a_2 \ldots a_n$ are uniquely determined.

Once this is realized, the actual algorithm is not hard to understand.

Suppose we have already parsed an input string (call it z); that is, we know a parse sequence for z. Now, part of that input has changed, so we have a new input (call it z') which we want to parse. We can divide z and z' into three parts: The substring preceding the changed region, the substring that changed, and the substring following the changed region. More formally, $z = xwy$ and $z' = xw'y$ for some $x, w, w', y \in V_T^*$.

Now, let us consider how to parse z'. There is no reason to re-parse x since it appears unchanged at the beginning of the input. A parser will always start in the initial state with $\underset{q_0}{\overset{-}{}}$ on its stack; as discussed above, after consuming x, its state and stack contents will be uniquely determined. So all the parser's configurations will be the same through the point where it shifts the last symbol of x. (Actually, since a configuration includes the *entire* remaining input, we need to substitute w' for w in constructing the new parse sequence, but since the first symbol of the remaining input is a symbol in x, the state and stack contents will be the same as before.)

Now, the parser should parse the symbols of w' as an ordinary LR(1) parser, since they are different from w.

As soon as the last symbol of w is shifted, the parser will change its course of action. It is now in familiar territory again, since all of the remaining input is the same. However, since it has just finished processing the changed region, its state and stack contents are likely not the same as the were during the previous parse. So it should continue to behave like a normal LR(1) parser as it proceeds through the symbols of y *until it reaches a known configuration,* i.e., a configuration where the state, stack contents, and remaining input are exactly the same as they were at some point during the previous parse. Once such a configuration is found, all of the remaining configurations can be copied directly from the previous parse.

That is the intuition; Celentano's much more precise description is as follows.

Algorithm 1 ***Incremental GPLR Parsing*** *[4, Algorithm 1, p. 312–313]*

Input: Two strings $z = xwy$ and $z' = xw'y$

A parse sequence $\Pi = S_0 S_1 \ldots S_n$ for z

Output: A parse sequence $\Pi' = S'_0 S'_1 \ldots S'_m$ for z'

Let $x = x_1 x_2 \ldots x_{|x|}$, $y = y_1 y_2 \ldots y_{|y|}$, $w = w_1 w_2 \ldots w_{|w|}$; *let* t *denote a generic suffix of* z. *Let* p *and* r *be two indices such that*

$$\begin{aligned} S_p &= T_0 \ldots T_p \odot wy\$, \\ S_{p-1} &= T_0 \ldots T_{p-1} \odot x_{|x|} wy\$, \\ S_r &= T_0 \ldots T_r \odot y\$, \text{ and} \\ S_{r-1} &= \begin{cases} T_0 \ldots T_{r-1} \odot w_{|w|} y\$ & \text{if } w \neq \epsilon \\ T_0 \ldots T_{r-1} \odot x_{|x|} y\$ & \text{if } w = \epsilon. \end{cases} \end{aligned}$$

Initially Π' *is the empty sequence.*

1. *Let* $S_i = \alpha_i \textcircled{q} \beta_i t_i$; *for all* i, $0 \leq i \leq p$, *append to* Π' *the configurations* $S'_i = \alpha_i \textcircled{q} \beta_i t'_i$ *where* t' *is obtained from* t *by replacing* w *with* w'.

2. *Append to* Π' *the configurations* S'_{p+i} *such that* $S'_{p+i-1} \vdash S'_{p+i}$ $0 < i \leq k$, *where* k *is the smallest value such that* $S'_{p+k} = \alpha \odot y\$$. *(If* $w' = \epsilon$ *this step is not executed; i.e.,* $k = 0$.*)*

3. *Let* S'_s *be the last configuration appended to* Π'. *If there exists an index* q, $0 \leq q \leq n - r$ *such that* $S'_s = S_{n-q}$, *go to step 5; otherwise go to step 4.*

4. *Append to* Π' *a new configuration obtained by performing a move on* z', *and go back to step 3.*

5. *Let* $S'_{s+j} = S_{n-q+j}$ *for all* j, $0 < j \leq q$. *Let* $m = s + q$.

4.1.2 Implementation

If one were to literally implement Celentano's algorithm as described above—by dumping the entire stack, state, and remaining input at every step—the results would be disappointing, to say the least. Perhaps the most ingenious part of Celentano's paper is not the algorithm itself but the effective implementation he suggests.

The Stack Tree

Rather than using a typical implementation of a stack, the parser's stack is replaced with a *stack tree.* A stack tree consists of a tree and a pointer p to a node of the tree. Since the stack of an LR parser always has at least one element on it (the initial state), we can assume that, at the beginning of the parse, the stack tree has a single node (labeled with the initial state), and p points to this node. The stack tree implements the usual interface of a stack as follows.

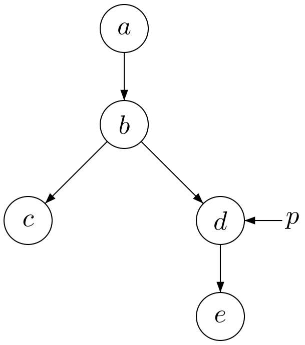

Figure 4.1: A sample stack tree

- *Push(x).* If the node pointed to by p already has a child node labeled x, change p to point at this child. If not, create a new child node labeled x, and point p at this child.
- *Top.* Return the label of the node pointed at by p.
- *Pop.* Point p at the parent of the node to which it is currently pointing.

Note that nodes are never removed from the tree; furthermore, given a pointer to a node in the stack tree, the "entire contents of the stack" can be determined by tracing the nodes back to the root. As an example, after executing the instructions

push(a) push(b) push(c) pop push(c) pop push(d) push(e) pop,

the stack tree will appear as shown in Figure 4.1 and, as expected, the sequence of nodes from the root to p spells *abd*.

Using a stack tree, then, provides an efficient way to store the entire contents of the stack at any point in a parse: We can simply note which node p was pointing at.

Input Labeling

Recall that, to store a complete parser configuration, we must store the stack contents, the current state, and the remaining input; our ultimate reasons for doing this are (1) to restart the parser at the beginning of the changed region and (2) to detect when the parser has reached a known configuration. The remainder of the implementation relies on three important observations.

1. There is no need to explicitly store the current state, since the current state of the parser is always the state on top of the stack.
2. It suffices to store only configurations immediately after a symbol is shifted. Between any two such configurations, there may be zero

or more reductions. The only step of Algorithm 1 affected by this is step 3, where the parser attempts to find a known configuration. If the known configuration was one of these “reduce configurations” which is not stored, then of course the parser will not find a matching configuration, so it will proceed to perform reductions until it reaches its next “shift configuration,” which will match. These reductions would not be performed if every configuration were stored, so extra work is being done, but correctness is not affected. Furthermore, it significantly reduces the number of configurations we must store: It is equal to the number of input symbols, since each symbol is shifted exactly once. That leads to the following:

3. If only shift configurations are stored, then we can associate with each input symbol a pointer to the stack contents for the configuration after which that symbol was shifted. By so labeling each symbol in the input string, we effectively have a complete parser configuration: The remaining input is obvious, and the associated stack tree pointer gives the corresponding stack contents, which also gives the current state of the parser.

Summary of Incremental LR(1) Parsing Implementation

The efficient implementation of Celentano's algorithm, then, involves replacing the parser's stack with a stack tree and labeling each input symbol with a pointer to the stack tree node immediately after that symbol is shifted. Restarting the parser at the beginning of the changed region amounts to resetting the stack tree pointer p and the current state, and detecting a known configuration amounts to a pointer comparison against p. The complete algorithm using a stack tree is as follows.

1. During the first parse, the parser proceeds as usual, but immediately after the parser shifts each symbol of the input, that symbol is labeled with a pointer to the stack tree node to which p points.

2. Suppose symbols i through j of the input have changed (where i and j denote positive integers). We will let p_k denote the pointer to the stack tree node associated with input symbol k. An incremental parse proceeds as follows.

   - Retain the labels (i.e., the stack tree pointers) associated with all of the unchanged symbols; the changed symbols are unlabeled.
   - Point parser's stack tree pointer p at the same node as p_i.
   - Set the parser's current state to the state now on top of the parser's stack.
   - Set the remaining input to symbol i through the end of input.

- Parse symbols i through j as in item 1 above, labeling each symbol with the stack tree nodes for future incremental parses.
- For each symbol $k > j$ through the end of input, proceed as follows. After symbol k is shifted, compare the stack tree pointer p to p_k (the node labeling symbol k). If $p = p_k$, report success and terminate the parse; a known configuration has been reached. If $p \neq p_k$, then relabel symbol k by setting p_k equal to p, and continue parsing (i.e., reduce as necessary, and repeat this step when the next symbol is shifted).

Note that the number of stack tree nodes is bounded by the length of the input, and only a single pointer is associated with each input symbol, so the space overhead is relatively modest: It is linear in the length of the input. Furthermore, the speed of the initial parse is virtually the same as for a non-incremental parser, since the only overhead is associating a stack tree node pointer with each input symbol.

4.2 Incremental GPLR Parsing

4.2.1 Overview

A GPLR parser consists of three things: (1) a parse stack, (2) a state machine, and (3) an input stack. As we did with LR parsers, we can give a complete "snapshot" of a GPLR parser at any point during its parse by describing the entire contents of its parse stack, its current state, and the contents of the input stack. Configurations of GPLR parsers will be written similarly to configurations of LR(1) parsers, and parse sequences defined similarly.

Recall that Celentano's algorithm relies heavily on the observation that at any point during a parse, the next action of an LR(1) parser is completely described by (1) the state of its cfsm, (2) the contents of its stack, and (3) only the first symbol of the remaining input. This is not true for GPLR parsers. In fact, the biggest advantage of GPLR parsers—the property that gives them their parsing power—is that they can use an unlimited amount of lookahead to resolve conflicts that would occur in a normal LR parser.[1] There is a similar statement which is true of GPLR parsers, however. Each input terminal is only consumed once, i.e., only nonterminals and cancellation symbols are ever pushed back onto the input stack. So at any point during a parse, a GPLR parser's next action is completely described by (1) the state of its cfsm, (2) the contents of its parse stack, and (3) *the remaining input through the first terminal symbol.*

This means that, as far as implementation is concerned, Celentano's algorithm can be implemented on a GPLR parser exactly as it would be for

[1] Technically, this is not quite accurate. They still use a single lookahead symbol, but since this symbol may be a nonterminal or cancellation symbol, the number of terminals symbols processed before committing to a particular reduction is unlimited.

a regular LR(1) parser. We can use a stack tree to record the contents of the parser stack throughout the parse (including all $\perp$ symbols, of course) and associate a stack tree node with each input terminal immediately before it is shifted. Since a terminal will only be shifted when it is on top of the stack—when there are no cancellation symbols or nonterminals preceding it—the above statement assures us that any reductions, cancellations, and continuations that should follow the shift action *will* occur.

We will now prove this formally. A full example of incremental GPLR parsing follows the proof of correctness.

4.2.2 Proof of Correctness*

We will begin by defining an *incremental GPLR parse sequence* as the sequence of configurations generated by applying Celentano's algorithm to a GPLR parse. This is simply a formalization of the incremental parsing algorithm described informally above. We will then give two lemmas which lead to a proof of Theorem 3, the GPLR analog of the theorem Celentano proves in [4] to demonstrate the correctness of his algorithm.

Definition 18 ***Incremental GPLR Parse Sequence***

Input: *Two strings* $z = xwy$ *and* $z' = xw'y$

A parse sequence $\Pi = S_0 S_1 \ldots S_n$ *for* z

Output: *A parse sequence* $\Pi' = S'_0 S'_1 \ldots S'_m$ *for* z'

Let $x = x_1 x_2 \ldots x_{|x|}$, $y = y_1 y_2 \ldots y_{|y|}$, $w = w_1 w_2 \ldots w_{|w|}$*; let* t *denote a generic suffix of* z*. Let* p *and* r *be two indices such that*

$$\begin{aligned} S_p &= T_0 \ldots T_p \odot wy\$, \\ S_{p-1} &= T_0 \ldots T_{p-1} \odot x_{|x|} wy\$, \\ S_r &= T_0 \ldots T_r \odot y\$, \text{ and} \\ S_{r-1} &= \begin{cases} T_0 \ldots T_{r-1} \odot w_{|w|} y\$ & \text{if } w \neq \epsilon \\ T_0 \ldots T_{r-1} \odot x_{|x|} y\$ & \text{if } w = \epsilon. \end{cases} \end{aligned}$$

Let k *be the smallest integer such that* $p \leq k$ *and* $S'_k = \alpha \odot y\$$*. Let* q' *be the smallest integer greater than* k *such that there exists* q *with* $r \leq q \leq n$ *and* $S'_{q'} = S_q$*.*

$$S'_i \overset{\text{def}}{=} \begin{cases} \alpha_i \text{\textcircled{@}} \beta_i t'_i, \text{ where } S_i = \alpha_i \text{\textcircled{@}} \beta_i t_i \text{ and} & \\ \qquad t'_i \text{ is } t \text{ with } w' \text{ substituted for } w & \text{if } 0 \leq i < p \qquad \text{(pre)} \\ \vdash (S'_{i-1}) & \text{if } p \leq i \leq k < q' \qquad \text{(diff)} \\ S_{q+(i-q')} & \text{if } i \geq q' \qquad \text{(post)} \end{cases}$$

Lemma 1 *The contents of the second stack in a GPLR recognizer is always of the form* αw*, where* $\alpha \in (V_N \cup V_C)^*$ *and* $w \in V_T^*$*. Furthermore, if* $\alpha \text{\textcircled{@}} \beta w \vdash_M$

$\alpha' \textcircled{$q'$} \beta' w'$ (where $\beta \in (V \cup V_C)^*$), then w' is a suffix of w. In other words, terminals are never pushed back onto the second stack.

Proof The proof is a simple induction on sequences of configurations following from the observation that, in each rule, the contents of the second stack is either unmodified, the topmost symbol is popped, or a nonterminal and cancellation symbol are pushed onto the stack. It proceeds as follows.

Base case. A sequence consisting of only a single configuration must contain only the initial configuration of a GPLR parser,

$$\begin{matrix}\perp \\ q_0\end{matrix}\ \textcircled{q_0}\, z,$$

where $z \in V_T^*$ denotes the initial input string. Clearly, z is of the desired form. The second statement ("Furthermore...") holds due to a false antecedent.

Inductive case. Suppose $\alpha \textcircled{$q$} \gamma \ \vdash_M \ \alpha' \textcircled{q'} \gamma'$ where $\gamma = \beta w$ for some $\beta \in (V_N \cup V_C)^*$ and $w \in V_T^*$. We will show that $\gamma' = \beta' w'$ for some $\beta' \in (V_N \cup V_C)^*$ and $w' \in V_T^*$, where w' is a suffix of w, by case analysis according to the definition of Δ in Definition 17.

- Suppose the move is due to (shift) or (cancel). Both of these rules are of the form

  $$(\cdot,\ \cdot,\ \chi) \longrightarrow (\cdot,\ \cdot,\ \epsilon)$$

  where $\chi \in V \cup V_C$. By the induction hypothesis, $\gamma \in (V_N \cup V_C)^* V_T^*$; since γ' is γ with the first symbol removed, $\gamma' \in (V_N \cup V_C)^* V_T^*$, and the portion matching V_T^* must be a suffix of the corresponding portion of γ.

- Suppose the move is due to (reduce) or (continue). Both of these rules are of the form

  $$(\cdot,\ \cdot,\ \chi) \longrightarrow (\cdot,\ \cdot,\ \chi),$$

  and so the second stack is unaffected.

- Suppose the move is due to (reduce-c). Then $\gamma' = \chi' A \gamma$, where $\chi' \in V_C$ and $A \in V_N$. Since $\chi' A \in (V_N \cup V_C)^*$, $\gamma \in (V_N \cup V_C)^*$ by the induction hypothesis, and the terminal portion of γ is unaffected, the lemma holds.

- Suppose the move is due to (cancel-c). Then $\gamma = \beta w = \chi \beta_2 w$, where $\chi \in V_C$, $\beta_2 \in (V_N \cup V_C)^*$, and $w \in V_T^*$; and $\gamma' = \chi' \beta_2 w$, where $\chi' \in V_C$. Since $\chi' \beta_2 \in (V_N \cup V_C)^*$, $w \in V_T^*$, and w is a suffix of itself, the lemma holds. □

Lemma 2 *Given an arbitrary configuration $\alpha \textcircled{$q$} \beta a w$ of a GPLR recognizer, where $\beta, \beta' \in (V_N \cup V_C)^*$ and $a \in V_T$,*

$$\alpha \textcircled{q} \beta a w \ \vdash_M \ \alpha' \textcircled{q'} \beta' w \implies \alpha \textcircled{q} \beta a v \ \vdash_M \ \alpha' \textcircled{q'} \beta' v \textit{ for every } v \in V_T^*.$$

Similarly,

$$\alpha \textcircled{q} \beta aw \vdash_M \alpha' \textcircled{q'} \beta' aw \implies \alpha \textcircled{q} \beta av \vdash_M \alpha' \textcircled{q'} \beta' av \textit{ for every } v \in V_T^*.$$

In other words, each move of a GPLR parser is determined by at most one unconsumed terminal from the original input string; the remainder of the input plays no role in determining the next configuration.

Proof Observe that all six rules comprising the transition relation (function) Δ in Definition 17 are contingent upon the pattern

$$(\cdot,\ \cdot,\ \chi)$$

being matched, where $\chi \in V \cup V_C$. According to the definition of Δ, then, only the first symbol of the second stack (χ) is used to determine which rule applies, and Lemma 1 guarantees that terminals are only removed from the stack, never pushed back onto it. Therefore, no terminals below the topmost terminal on the second stack can ever be used to determine the next move of the parser, and the lemma follows. □

Theorem 3 *In Definition 18,*

1. S'_0 *is the initial configuration;*
2. *for* $1 \le i \le m$*;* $S'_{i-1} \vdash S'_i$*; and*
3. $S'_m = T_0 T_f \odot \$$.

Proof (We will use the expressions (pre), (diff), and (post) from Definition 18.) To prove Part 1, we note that $S_0 = \underset{q_0}{\perp} \textcircled{q_0} xwy$ by Definition 17. Thus, (pre) gives $S'_0 = \underset{q_0}{\perp} \textcircled{q_0} xw'y$, which is the initial configuration for the recognizer for z'.

Part 2 is proved by induction on i. There are three cases, each of which will be conducted by a separate induction on i.

- **Case 1:** $1 \le i < p$**.** The arguments for the base and inductive cases are similar. By Lemma 1 and the definition of p, the second stack in each configuration S_i must have the form $\beta\hat{x}wy$ where $\hat{x}$ is a suffix of x. Since Π is a valid parse sequence, $S_{i-1} \vdash S_i$. Now $|\beta\hat{x}| \ge 1$ since $i < p$, and from Lemma 2 we know that $S'_{i-1} \vdash S'_i$.
- **Case 2:** $p \le i \le k < q'$**.** The proof of this case is immediate due to (diff).
- **Case 3:** $i \ge q'$**.** For the base case, suppose $i = q'$. Now q' was chosen such that $\vdash (S'_{q'-1}) = S_q$ for some q, and $S_q = S_{q+(i-q')} = S'_i$. Now suppose $i > q'$ and $S'_{j-1} \vdash S'_j$ for all $j < i$. Since Π is a valid parse sequence, $S_{q+((i-1)-q')} \vdash S_{q+(i-q')}$, and therefore $S'_{i-1} \vdash S'_i$.

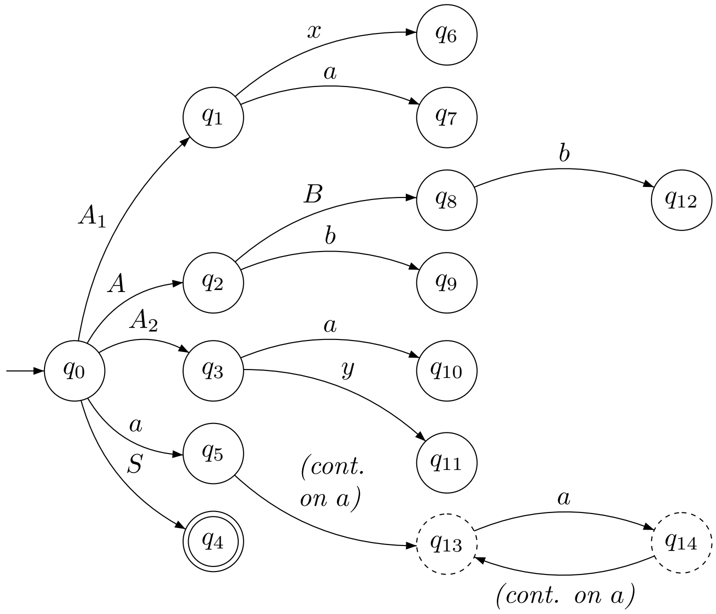

Figure 4.2: GPLR(1) cfsm for the example grammar

The proof of Part 3 is straightforward. Clearly, if the final configuration is due to (post), it will be identical to the final configuration in Π. If the final configuration S'_f is due to (pre) or (diff), then $S'_{f-1} \vdash S'_f$ by Part 2, and S'_i is not defined for any $i > f$, and thus S'_f must be the halting configuration $T_0 T_f \odot \$$. $\Box$

4.2.3 An Example of Incremental GPLR Parsing

To illustrate incremental GPLR parsing, we will use the (somewhat contrived) grammar

$$\begin{array}{rcl} S & \rightarrow & AB \\ A & \rightarrow & A_1 x \mid A_2 y \\ A_1 & \rightarrow & A_1 a \mid a \\ A_2 & \rightarrow & A_2 a \mid a \\ B & \rightarrow & Bb \mid b \end{array}$$

for the (regular) language $a^+(x|y)b^+$. Notice, however, that the exact parse of a^+, which may be arbitrarily long, depends on whether x or y follows. Thus, we cannot reduce any of the as until we have seen all of the as; since the number of as is arbitrarily long, this grammar is not LR(k) for any k. The GPLR(1) cfsm for this grammar is depicted in Figure 4.2; due to space considerations, the complete list of items in each state and the corresponding actions are displayed in the appendix. Primed states are shown in dotted circles.

The initial parse of *aaaaxbbb* is shown in Figure 4.3; the resulting stack tree and input labels are shown in Figure 4.4. The symbol $\langle * \rangle$ is used to denote

$\langle A_1 \to A_1 \cdot a \rangle$. As before, symbols that are about to be reduced are underlined, and the resulting nonterminal and cancellation symbol (if applicable) are displayed in boldface on the following line.

Now, given this parse of *aaaaxbbb*, suppose the middle *axb* is changed to *ybbb*. We retain the labels for the portion of the input that is the same; that is, the original input

1	2	3	4	5	6	7	8
●	●	●	●	●	●	●	●
a	a	a	**a**	**x**	**b**	b	b

is revised, yielding the new input

1	2	3	4	5	6	7	8	9
●	●	●					●	●
a	a	a	**y**	**b**	**b**	**b**	b	b.

Since symbols 1 through 3 are unchanged, the parser is restarted as follows. The stack is reset according to pointer labeling symbol 3; according to Figure 4.4, this will effectively place

$$\begin{array}{cccccc} - & a & \perp & a & \perp & a \\ q_0 & q_5 & q_{13} & q_{14} & q_{13} & q_{14} \end{array}$$

on the stack. Then, the current state is reset to the state on top of the stack (q_{14}). The remaining input consists of symbol 4 (the start of the changed region) through the end of input:

4	5	6	7	8	9
				●	●
y	**b**	**b**	**b**	b	b.

The incremental parse proceeds as shown in Figure 4.5. We use $\langle \dagger \rangle$ to denote $\langle A_2 \to A_2 \cdot a \rangle$; the unmodified *b*s which still have stack tree labels are denoted $\dot{b}$. After shifting in step 17, the current stack tree node will be the same one labeling the shifted *b*, and so the parser can accept the input.

Step	Parser Stack	Input Stack	Action
1.	$\begin{array}{c} - \\ q_0 \end{array}$	aaaaxbbb	Shift and go to q_5
2.	$\begin{array}{cc} - & a \\ q_0 & q_5 \end{array}$	aaaxbbb	Continue in $q_1$3
3.	$\begin{array}{ccc} - & a & \perp \\ q_0 & q_5 & q_{13} \end{array}$	aaaxbbb	Shift and go to q_{14}
4.	$\begin{array}{cccc} - & a & \perp & a \\ q_0 & q_5 & q_{13} & q_{14} \end{array}$	aaxbbb	Continue in $q_1$3
5.	$\begin{array}{ccccc} - & a & \perp & a & \perp \\ q_0 & q_5 & q_{13} & q_{14} & q_{13} \end{array}$	aaxbbb	Shift and go to q_{14}
6.	$\begin{array}{cccccc} - & a & \perp & a & \perp & a \\ q_0 & q_5 & q_{13} & q_{14} & q_{13} & q_{14} \end{array}$	axbbb	Continue in q_{13}
7.	$\begin{array}{ccccccc} - & a & \perp & a & \perp & a & \perp \\ q_0 & q_5 & q_{13} & q_{14} & q_{13} & q_{14} & q_{13} \end{array}$	axbbb	Shift and go to q_{14}
8.	$\begin{array}{cccccccc} - & a & \perp & a & \perp & a & \perp & a \\ q_0 & q_5 & q_{13} & q_{14} & q_{13} & q_{14} & q_{13} & \underline{q_{14}} \end{array}$	xbbb	Reduce by $A_1 \to A_1 a$
9.	$\begin{array}{cccccc} - & a & \perp & a & \perp & a \\ q_0 & q_5 & q_{13} & q_{14} & q_{13} & \underline{q_{14}} \end{array}$	$\langle * \rangle$ $\mathbf{A_1}$ xbbb	Reduce by $A_1 \to A_1 a$
10.	$\begin{array}{cccc} - & a & \perp & a \\ q_0 & q_5 & q_{13} & \underline{q_{14}} \end{array}$	$\langle * \rangle$ $\mathbf{A_1}$ $\langle * \rangle$ A_1 xbbb	Reduce by $A_1 \to A_1 a$
11.	$\begin{array}{cc} - & a \\ q_0 & \underline{q_5} \end{array}$	$\langle * \rangle$ $\mathbf{A_1}$ $\langle * \rangle$ A_1 $\langle * \rangle$ A_1 xbbb	Reduce by $A_1 \to a$
12.	$\begin{array}{cc} - & A_1 \\ q_0 & q_1 \end{array}$	$\langle * \rangle$ A_1 $\langle * \rangle$ A_1 $\langle * \rangle$ A_1 xbbb	Cancel $\langle A_1 \to A_1 \cdot a \rangle$
13.	$\begin{array}{c} - \\ q_0 \end{array}$	A_1 $\langle * \rangle$ A_1 $\langle * \rangle$ A_1 xbbb	Shift A_1
14.	$\begin{array}{cc} - & A_1 \\ q_0 & q_1 \end{array}$	$\langle * \rangle$ A_1 $\langle * \rangle$ A_1 xbbb	Cancel $\langle A_1 \to A_1 \cdot a \rangle$
15.	$\begin{array}{c} - \\ q_0 \end{array}$	A_1 $\langle * \rangle$ A_1 xbbb	Shift A_1
16.	$\begin{array}{cc} - & A_1 \\ q_0 & q_1 \end{array}$	$\langle * \rangle$ A_1 xbbb	Cancel $\langle A_1 \to A_1 \cdot a \rangle$
17.	$\begin{array}{c} - \\ q_0 \end{array}$	A_1 xbbb	Shift A_1
18.	$\begin{array}{cc} - & A_1 \\ q_0 & q_1 \end{array}$	xbbb	Shift x
19.	$\begin{array}{ccc} - & A_1 & x \\ q_0 & \underline{q_1} & \underline{q_6} \end{array}$	bbb	Reduce by $A_1 \to A_1 x$
20.	$\begin{array}{cc} - & \mathbf{A_1} \\ q_0 & \underline{\mathbf{q_1}} \end{array}$	bbb	Reduce by $A \to A_1$
21.	$\begin{array}{cc} - & \mathbf{A} \\ q_0 & \mathbf{q_2} \end{array}$	bbb	Shift b
22.	$\begin{array}{ccc} - & A & b \\ q_0 & q_2 & \underline{q_9} \end{array}$	bb	Reduce by $B \to b$
23.	$\begin{array}{ccc} - & A & \mathbf{B} \\ q_0 & q_2 & \mathbf{q_8} \end{array}$	bb	Shift b and go to q_{12}
24.	$\begin{array}{cccc} - & A & B & b \\ q_0 & q_2 & \underline{q_8} & \underline{q_{12}} \end{array}$	b	Reduce by $B \to Bb$
25.	$\begin{array}{ccc} - & A & \mathbf{B} \\ q_0 & q_2 & \mathbf{q_8} \end{array}$	b	Shift b and go to q_{12}
26.	$\begin{array}{cccc} - & A & B & b \\ q_0 & q_2 & \underline{q_8} & \underline{q_{12}} \end{array}$	*(end of input)*	Reduce by $B \to Bb$
27.	$\begin{array}{ccc} - & A & \mathbf{B} \\ q_0 & \underline{q_2} & \underline{\mathbf{q_8}} \end{array}$	*(end of input)*	Reduce by $S \to AB$
28.	$\begin{array}{cc} - & \mathbf{S} \\ q_0 & \mathbf{q_4} \end{array}$	*(end of input)*	Accept

Figure 4.3: Complete GPLR(1) parse of *aaaaxbbb* in the example grammar.

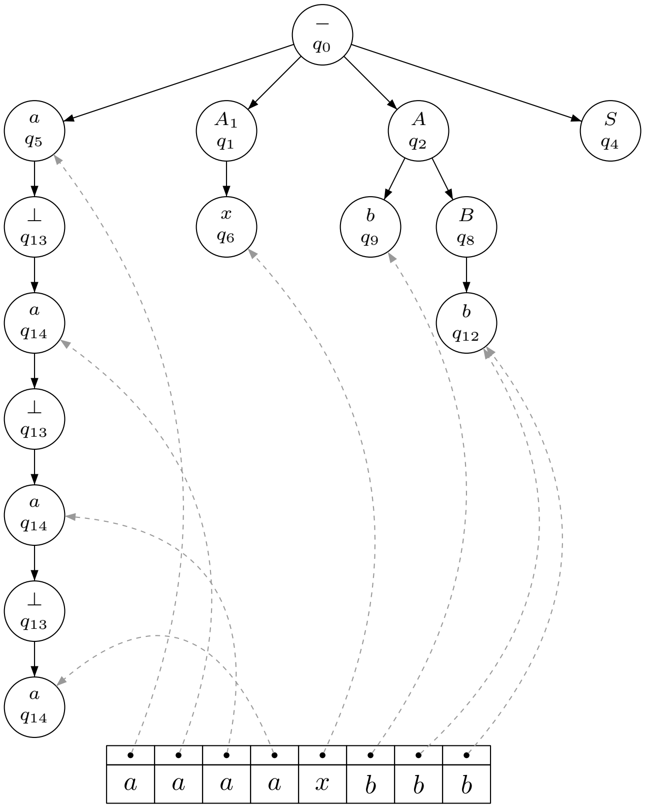

Figure 4.4: Stack tree and input labels for the parse of *aaaaxbbb*

Step	Parser Stack	Input Stack	Action
1.	$\overset{-}{q_0}\ \overset{a}{q_5}\ \overset{\perp}{q_{13}}\ \overset{a}{q_{14}}\ \overset{\perp}{q_{13}}\ \underline{\overset{a}{q_{14}}}$	ybbbḃḃ	Reduce by $A_2 \rightarrow A_2a$
2.	$\overset{-}{q_0}\ \overset{a}{q_5}\ \overset{\perp}{q_{13}}\ \underline{\overset{a}{q_{14}}}$	$\langle\dagger\rangle$ $\mathbf{A_2}$ ybbbḃḃ	Reduce by $A_2 \rightarrow A_2a$
3.	$\overset{-}{q_0}\ \underline{\overset{a}{q_5}}$	$\langle\dagger\rangle$ $\mathbf{A_2}$ $\langle\dagger\rangle$ $\mathbf{A_2}$ ybbbḃḃ	Reduce by $A_2 \rightarrow a$
4.	$\overset{-}{q_0}\ \overset{\mathbf{A_2}}{\mathbf{q_3}}$	$\langle\dagger\rangle$ $\mathbf{A_2}$ $\langle\dagger\rangle$ $\mathbf{A_2}$ ybbbḃḃ	Cancel $\langle A_2 \rightarrow A_2 \cdot a\rangle$
5.	$\overset{-}{q_0}$	$\mathbf{A_2}$ $\langle\dagger\rangle$ $\mathbf{A_2}$ ybbbḃḃ	Shift A_2
6.	$\overset{-}{q_0}\ \overset{A_2}{q_3}$	$\langle\dagger\rangle$ $\mathbf{A_2}$ ybbbḃḃ	Cancel $\langle A_2 \rightarrow A_2 \cdot a\rangle$
7.	$\overset{-}{q_0}$	$\mathbf{A_2}$ ybbbḃḃ	Shift A_2
8.	$\overset{-}{q_0}\ \overset{A_2}{q_3}$	ybbbḃḃ	Shift and go to q_{11}
9.	$\overset{-}{q_0}\ \underline{\overset{A_2}{q_3}\ \overset{y}{q_{11}}}$	bbbḃḃ	Reduce by $A \rightarrow A_2y$
10.	$\overset{-}{q_0}\ \overset{\mathbf{A}}{\mathbf{q_2}}$	bbbḃḃ	Shift and go to q_9
11.	$\overset{-}{q_0}\ \overset{A}{q_2}\ \overset{b}{q_9}$	bbḃḃ	Shift and go to q_9
12.	$\overset{-}{q_0}\ \overset{A}{q_2}\ \underline{\overset{b}{q_9}}$	bbḃḃ	Reduce by $B \rightarrow b$
13.	$\overset{-}{q_0}\ \overset{A}{q_2}\ \overset{\mathbf{B}}{\mathbf{q_8}}$	bbḃḃ	Shift and go to q_{12}
14.	$\overset{-}{q_0}\ \overset{A}{q_2}\ \underline{\overset{B}{q_8}\ \overset{b}{q_{12}}}$	bḃḃ	Reduce by $B \rightarrow Bb$
15.	$\overset{-}{q_0}\ \overset{A}{q_2}\ \overset{\mathbf{B}}{\mathbf{q_8}}$	bḃḃ	Shift and go to q_{12}
16.	$\overset{-}{q_0}\ \overset{A}{q_2}\ \underline{\overset{B}{q_8}\ \overset{b}{q_{12}}}$	ḃḃ	Reduce by $B \rightarrow Bb$
17.	$\overset{-}{q_0}\ \overset{A}{q_2}\ \overset{B}{q_8}$	ḃḃ	Shift and go to q_{12} Leads to pointer match: Accept

Figure 4.5: Incremental GPLR(1) parse of *aaa***ybbb***bb* in the example grammar.

5 Future Work & Conclusions

We conclude by proposing a method to reduce the size of GPLR parsers and discussing related work.

5.1 Lookahead Generalized Piecewise LR Parsing

While GPLR is a very powerful parsing technique, the parsers produced tend to be very large. For example, the LALR(1) grammar used to produce the Fortran 95 parser in Photran [21] produces an LALR(1) cfsm with approximately 2,700 states, while the corresponding GPLR(1) cfsm has almost 90,000. Furthermore, in an LALR(1) parser, the $\text{Action}_{\text{LR}}$ function (usually stored in tabular form) contains only terminals in its domain, the $\text{Action}_{\text{GPLR}}$ function is defined on terminals, nonterminals, and cancellation symbols (of which there are thousands). Even with gigabytes of memory and advanced compression techniques, this disparity in parser size is enough to warrant concern.

Though we will leave a proof of correctness and empirical study to future work, we will briefly suggest a method for producing more reasonably-sized GPLR parsers. We call the method *lookahead GPLR,* or LAGPLR, as it compresses the GPLR cfsm in the same way that DeRemer's LALR(k) method [7] compresses the LR(k) cfsm. Equivalently, it allows a GPLR parser to be driven from the LR(0) cfsm. (The ideas and portions of the exposition are borrowed from the author's previous work on parallel LALR parsing [20].)

5.1.1 LALR Parsers and the DeRemer-Pennello Lookahead Computation

The concept behind LALR parsing is simple. The set of cores in each state of the LR(k) cfsm will always be identical to one of the states in the LR(0) cfsm. That is, the LR(k) cfsm may contain several states with the same cores but different lookaheads. In theory, the LALR(k) cfsm is constructed from the

LR(k) cfsm by merging states with identical cores. For example, the two states

$$\begin{aligned} q_2 &= \{[A \rightarrow ab \cdot c,\ x], [B \rightarrow\ \cdot cd,\ y]\} \text{ and} \\ q_3 &= \{[A \rightarrow ab \cdot c,\ z], [B \rightarrow\ \cdot cd,\ y]\} \end{aligned}$$

would be merged into a single state

$$q_{2,3} = \{[A \rightarrow ab \cdot c,\ x], [A \rightarrow ab \cdot c,\ z], [B \rightarrow\ \cdot cd,\ y]\}.$$

However, the original purpose of the LALR construction was to avoid constructing the entire LR(k) cfsm due to its large size, so in practice, a different method is needed.

In [8], DeRemer and Pennello provide a method for constructing LALR(1) parsers *from the LR(0) cfsm.* (Our presentation follows the more succinct description in [26, p. 125–135].) As described in Chapter 2, the lookahead components of LR(k) items are used only to determine *reduce* actions. This method proceeds by defining several relations on the LR(0) machine. The ultimate result is the **has-lalr-lookahead** relation: If a reduce item $[A \rightarrow \alpha \cdot]$ appears in a state q of the LR(0) cfsm, then the parser should *reduce* by $A \rightarrow \alpha$ on every lookahead a such that $(q, A \rightarrow \alpha)$ **has-lalr-lookahead** a.

The definition of **has-lalr-lookahead** relation and its component relations are shown in Figure 5.1. Juxtaposition denotes relational composition, and Goto_{LR} is extended in the natural way to operate on strings in V^*:

$$\text{Goto}_{\text{LR}}(q, X\alpha) \stackrel{\text{def}}{=} \text{Goto}_{\text{LR}}(\text{Goto}_{\text{LR}}(q, X), \alpha).$$

R^* denotes the reflexive, transitive closure of the relation R. Note that a successful implementation depends on an efficient algorithm for computing reflexive, transitive closures. DeRemer and Pennello suggested one due to Eve and Kurki-Suonio [9], although many others suffice as well.

While precise, these definitions do not give much insight into why **has-lalr-lookahead** is correct. We will give a bit of intuition and then illustrate it with a concrete example.

First, recall the definition of $\text{Closure}_{\text{LR}}$, which defines how LR(1) lookaheads are computed in the first place:

$$\begin{aligned} \text{Closure}_{\text{LR}}(I) = I \quad \cup \quad \{[B \rightarrow\ \cdot \gamma,\ \text{First}_k(\beta w)] \mid \\ [A \rightarrow \alpha \cdot B\beta,\ w] \in \text{Closure}_{\text{LR}}(I) \\ \text{and } B \rightarrow \gamma \in P\}. \end{aligned}$$

Intuitively, when we are closing a state, we are adding the item $[B \rightarrow\ \cdot \gamma,\ \ldots]$ to that state because, while we're in the middle of processing $[A \rightarrow \alpha \cdot B\beta,\ w]$, we will have to temporarily work our way through some other production and

$$
\begin{array}{rcl}
\textbf{has-lalr-lookahead} & \overset{\text{def}}{=} & \textbf{lookback includes}^{*}\ \textbf{reads}^{*}\ \textbf{directly-reads} \\
(\mathrm{Goto}_{\mathrm{LR}}(q,\alpha), A \rightarrow \alpha)\ \textbf{lookback}\ (q,A) & \text{iff} & \text{state } q \text{ has an outgoing transition on } A \\
(\mathrm{Goto}_{\mathrm{LR}}(q,\alpha), A)\ \textbf{includes}\ (q,B) & \text{iff} & \text{state } q \text{ has an outgoing transition on } B \text{ and} \\
 & & \text{there is a production } B \rightarrow \alpha A \beta \text{ such that } \beta \Rightarrow^{*} \epsilon \\
\textbf{reads} & \overset{\text{def}}{=} & \textbf{goes-to has-null-transition} \\
(q,A)\ \textbf{goes-to}\ \mathrm{Goto}_{\mathrm{LR}}(q,A) & \text{iff} & \text{state } q \text{ has an outgoing transition on } A \\
q\ \textbf{has-null-transition}\ (q,A) & \text{iff} & \text{state } q \text{ has an outgoing transition on } A \text{ and } A \Rightarrow^{*} \epsilon \\
\textbf{directly-reads} & \overset{\text{def}}{=} & \textbf{goes-to has-transition-on terminal} \\
q\ \textbf{has-transition-on}\ X & \text{iff} & \text{state } q \text{ has an outgoing transition on } X \\
X\ \textbf{terminal}\ X & \text{iff} & X \in V_T
\end{array}
$$

Figure 5.1: Definitions comprising the **has-lalr-lookahead** relation

reduce it to B before we can move forward to $[A \to \alpha B \cdot \beta,\ w]$. After that, the next symbol to process will be something that can start βw. So all such symbols should be lookaheads for the $[B \to\ \cdot \gamma,\ \ldots]$ item. Also, recall that Goto_{LR} takes existing items and moves the dot forward but does not change the lookahead. So several states away, we will have a reduce item $[B \to \gamma\, \cdot\, ,\ \ldots]$ with the same lookahead.

The DeRemer-Pennello method starts at the corresponding state *in the LR(0) cfsm* containing this reduce item $[B \to \gamma\, \cdot\,]$ and works *backward* through the cfsm, finding all the states the parser could be in after reducing by $B \to \gamma$. Each of those states will contain items of the form $[A \to \alpha \cdot B\beta,\ w]$, i.e., items that were being processed before the parser got side-tracked processing the B-production. If we follow the B-transition out of this state, we will reach the state corresponding to $[A \to \alpha B \cdot \beta,\ w]$, and all of the outgoing terminal transitions will correspond to symbols the parser would expect to see next—i.e., $\text{First}_k(\beta w)$. (There is a slight complication to handle the case where the first symbol(s) in β are nullable nonterminals; we will describe this below.)

To illustrate the method concretely, suppose we have the augmented grammar

$$\begin{array}{rcl} S' & \to & S \\ S & \to & AdB \mid Ce \\ A & \to & aB \\ B & \to & b \\ C & \to & BD \\ D & \to & \epsilon \end{array}$$

which generates the language $\{abdb,\ be\}$. The canonical LR(0) cfsm for this grammar is shown in Figure 5.2.

Let us calculate the LALR(1) lookahead component for the reduce item $[B \to b\, \cdot\,]$ in state q_{11}.

First, we calculate

$$(q_{11}, B \to b) \quad \textbf{lookback} \quad \{(q_0, B), (q_4, B), (q_8, B)\}.$$

If our parser was in state q_{11} and popped b (the right-hand side of $B \to b$) off its stack, these are the states it could end up in. So we can just follow the B-transitions out of q_0, q_4, and q_8, and look at the outgoing terminals to find the lookahead, right?

Not quite. Remember, we are attempting to answer the question, "What could be the next symbol that we shift?" It is possible that we may perform *several* reductions, including reducing some nonterminals to ϵ, before we shift

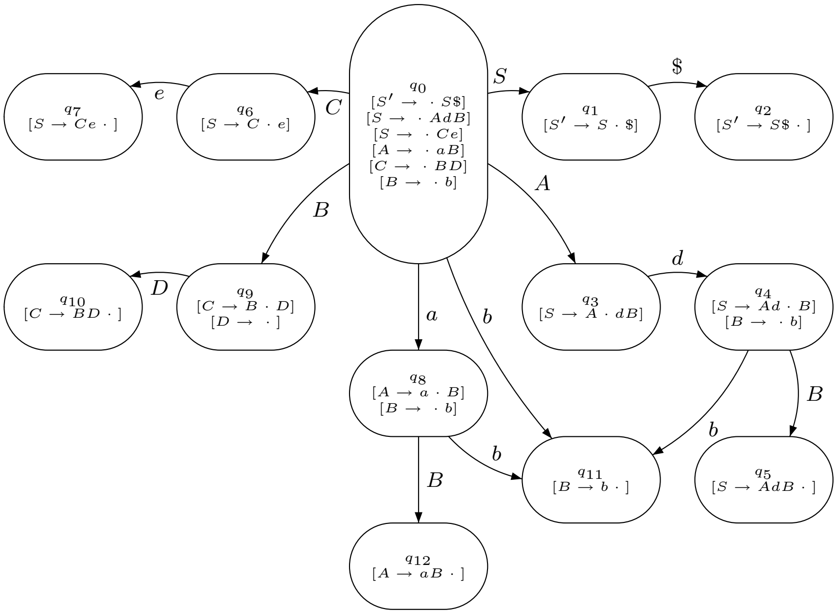

Figure 5.2: LR(0) cfsm for the LALR lookahead demonstration grammar

the next symbol.

First, if the symbol we reduced to (B) is the rightmost symbol in a production, the parser could perform another reduction, and if that symbol were the rightmost in a production, it could perform another reduction, etc. The **includes*** relation accounts for this. In our example, after reducing by $B \to b$, if it ended up in q_4, it would reduce AdB to S, putting us in q_0. Thus, the parser could end up in state q_0 with S on top of its stack; so when we consider lookaheads, we need to follow the S-transition out of q_0 and look at the outgoing terminals there as well. So the lookahead set should include all of S's lookaheads coming out of q_0.

There is yet another situation to consider. Recall from above that we plan to follow the B-transition out of q_0. State q_0 contains the item $[C \to BD]$, but D is nullable. That effectively makes B the rightmost symbol of this C-production in q_0, and so we also need to follow the C-transition out of q_0 and include its lookahead (e). The **reads*** relation detects strings of nullable nonterminals (like D in this example) and ensures that we include their lookaheads as well.

Finally, notice that **directly-reads** simply follows the given nonterminal transition out of a state and looks for outgoing terminal transitions from there.

So now we have

$$\begin{array}{l}
(q_{11}, B \to b) \\
\qquad \begin{array}{ll}
\textbf{lookback} & \{(q_0, B), (q_4, B), (q_8, B)\} \\
\textbf{includes}^* & \{(q_0, B), (q_4, B), (q_8, B), (q_0, S), (q_0, A), \\
 & \qquad (q_0, C), (q_4, C), (q_8, C)\} \\
\textbf{reads}^* & \{(q_0, B), (q_4, B), (q_8, B), (q_0, S), (q_0, A), \\
 & \qquad (q_0, C), (q_4, C), (q_8, C), (q_9, D)\} \\
\textbf{directly-reads} & \{d, \$, e\}.
\end{array}
\end{array}$$

Note that the lookahead d was obtained from following the A-transition out of q_0; similarly, \$ was obtained from (q_0, S) and e from (q_0, C).

5.1.2 Constructing LAGPLR Parsers

We can now use the ideas behind the LALR construction and the DeRemer-Pennello lookahead computation to construct a GPLR ("LAGPLR") parser from the LR(0) machine. Essentially, we will use a variation on the **has-lalr-lookahead** relation to allow nonterminals and cancellation symbols in the lookahead; then we will describe the computation of continuation states.

Nonterminal Lookaheads

Extending the **has-lalr-lookahead** relation to compute nonterminals in addition to terminals is easy. Recall that **directly-reads** is a relation on $(Q \times V_N) \times V_T$, and

$$\textbf{directly-reads} = \textbf{goes-to has-transition-on terminal}.$$

By eliminating the **terminal** relation from this composite, we have the desired relation on $(Q \times V_N) \times (V_N \cup V_T)$, which can replace **directly-reads** in the lookahead computation to yield both terminals and nonterminals:

$$\textbf{directly-reads-sym} \overset{\text{def}}{=} \textbf{goes-to has-transition-on}.$$

Cancellation Lookaheads

Given a state q, we can *cancel* when the lookahead symbol is $\langle A \to \alpha \cdot \beta \rangle$ iff $[A \to \alpha \cdot \beta]$ is a core in q. But when do we want to *reduce* on that lookahead symbol? After we perform the reduction, we want to end up in (1) a state that contains an item with the core $[A \to \alpha \cdot \beta]$, i.e., a state where we can *cancel*; or (2) a state where we can reduce again, which will in turn lead us to a state where we can *cancel*.

Recall from Section 5.1.1 that the states to which we may reduce are given by the composite relation **lookback includes***. More formally, let q denote a state in the LR(0) cfsm containing a reduce item $[A \to \alpha \cdot]$; if the parser were to reduce by $A \to \alpha$ in state q, the states to which the parser could transition before shifting the next input symbol and the nonterminals to which it could reduce are given by the relation **reduces-to**, which is defined such that

$$(q, A \to \alpha)\,\textbf{reduces-to}\,(q', B) \quad \text{iff} \quad (q, A \to \alpha)\,\textbf{lookback includes}^*(q', B).$$

Now, recall how lookahead items were computed in the GPLR construction. If a nonterminal B appeared after the dot in the core of an item, then when that state was closed, that core was used as the lookahead for all of the the B-items that were added to the state (with the dot moved one position forward). We will define a relation **contains-cancellation-core** which essentially "does this backwards:" Given a state and a nonterminal B, it selects all of the items with a B after the dot. These will become the cancellation lookaheads. Letting S denote the start symbol of the (augmented) grammar, we define

$$\begin{aligned}&(q', B)\,\textbf{contains-cancellation-core}\,\langle C \to \alpha B \cdot \beta\rangle \text{ iff}\\ &\qquad [C \to \alpha \cdot B\beta] \in q \text{ and } (|\alpha| > 0 \text{ or } B = S) \text{ and } |\beta| > 0.\end{aligned}$$

So to find the cancellation lookaheads for a reduce item $[A \to \alpha \cdot]$ in a state q, we simply find every $\langle B \to \gamma \cdot \delta\rangle$ such that

$$(q, A \to \alpha)\,\textbf{reduces-to contains-cancellation-core}\,\langle B \to \gamma \cdot \delta\rangle\,.$$

The complete LAGPLR lookahead computation can be summarized as the relation

$$\begin{aligned}&\textbf{has-lagplr-lookahead} \overset{\text{def}}{=}\\ &\qquad \textbf{lookback includes}^*\,\textbf{reads}^*\,\textbf{directly-reads-sym}\\ &\qquad \cup\,\textbf{reduces-to contains-cancellation-core}.\end{aligned}$$

Continuation States

As with the original GPLR construction, if a unique *shift* or *reduce* action cannot be determined for a given state, the parser should *continue.* The intuition used to construct GPLR continuation states can be applied to the LAGPLR construction as well: We carry shift items over to the continuation state, and we carry the cancellation lookaheads of reduce items over as cores in the continuation state. Formally, given a state q in the LR(0) cfsm and a lookahead symbol X, the continuation state q' from q on X is constructed as the closure of the following set of items.

- Every shift item $[A \to \alpha \cdot X\beta]$ in q is included in q'.
- If a *reduce* action is indicated for an item $[A \to \alpha \cdot]$ in q and

$$(q, A \to \alpha)\,\textbf{reduces-to contains-cancellation-core}\,\langle C \to \beta \cdot X\gamma\rangle\,,$$

then $[C \to \beta \cdot X\gamma]$ is included in q'.

One problem remains: How do we determine lookaheads for items in the continuation state and other primed states? Since continuation states only have outgoing transitions—they are not reachable from the start state—the lookahead relations defined above will not work. For example, a continuation state could reasonably include $[A \to a \cdot b]$, but that state cannot be $\mathrm{Goto}_{\mathrm{LR}}(q, a)$ for any q since it does not have any incoming transitions.

There are a number of possible solutions. The "best" solution would be to modify the relations to accommodate primed states. A simpler solution—the one we will propose—is the following: When we create a continuation state, we will add "temporary" transitions to the cfsm which are used solely for computing the relations described above.

When constructing a continuation state q' from q on X, items are copied over from several states. We need to record what those states were. Define a set $Q_{q',X}$ as the following:

- If there is one or more shift item $[A \to \alpha \cdot X\beta]$ in q (the state from which we are continuing), then q is included in $Q_{q',X}$.
- If a *reduce* action is indicated for an item $[A \to \alpha \cdot]$ in q and

$$\begin{aligned}&(q, A \to \alpha)\\ &\qquad \textbf{reduces-to}\ (q'', B)\\ &\qquad \textbf{contains-cancellation-core}\ \langle C \to \beta \cdot X\gamma \rangle\,,\end{aligned}$$

  then q'' is included in $Q_{q',X}$.

Now, repeat the following for every state q_2 in $Q_{q',X}$: For every state q_1 such that there is a transition from q_1 to q_2 labeled Y, create a temporary transition from q_1 to q' labeled Y. (The transition from q_1 to q_2 may be temporary.)

These temporary transitions are used solely for computing the lookahead relations defined above; they should be ignored when determining the action of the parser. (Clearly, when a parser shifts a, it should move to the appropriate goto state, not to some continuation state due to a temporary transition.)

An Example LAGPLR Parser

We will briefly describe what happens if the example in Section 4.2.3 is reworked as an LAGPLR parser. The LR(0) cfsm has the same unprimed states as the GPLR cfsm in Figure 4.2 (although the items do not have lookaheads, of course).

Computing the **has-lagplr-lookahead** relation on the LR(0) cfsm initially produces

$$\begin{array}{rll}
(q_5, A_1 \to a) & \textbf{has-lagplr-lookahead} & \{a, x, \langle A_1 \to A_1 \cdot a\rangle, \langle A \to A_1 \cdot x\rangle\} \\
(q_5, A_2 \to a) & \textbf{has-lagplr-lookahead} & \{a, y, \langle A_2 \to A_2 \cdot a\rangle, \langle A \to A_2 \cdot y\rangle\} \\
(q_{12}, B \to Bb) & \textbf{has-lagplr-lookahead} & \{\langle B \to B \cdot b\rangle\} \\
(q_8, S \to AB) & \textbf{has-lagplr-lookahead} & \{\textit{(end of input)}\} \\
(q_7, A_1 \to A_1 a) & \textbf{has-lagplr-lookahead} & \{a, x, \langle A_1 \to A_1 \cdot a\rangle, \langle A \to A_1 \cdot x\rangle\} \\
(q_{10}, A_2 \to A_2 a) & \textbf{has-lagplr-lookahead} & \{a, y, \langle A_2 \to A_2 \cdot a\rangle, \langle A \to A_2 \cdot y\rangle\} \\
(q_{11}, A \to A_2 y) & \textbf{has-lagplr-lookahead} & \{B, b, \langle S \to A \cdot B\rangle\} \\
(q_9, B \to b) & \textbf{has-lagplr-lookahead} & \{\textit{(end of input)}, \langle B \to B \cdot b\rangle\} \\
(q_6, A \to A_1 x) & \textbf{has-lagplr-lookahead} & \{B, b, \langle S \to A \cdot B\rangle\}.
\end{array}$$

There is a reduce-reduce conflict in state q_5 on lookahead a, so we need to compute a continuation state. We will call this state q_{13}, as it turns out to be the analog of state q_{13} in Figure 4.2. There are no a-shift items to carry over from q_5. We have

$$\begin{array}{l}
(q_5, A_1 \to a) \\
\quad \textbf{reduces-to}\{(q_0, A_1)\} \\
\quad \textbf{contains-cancellation-core}\{\langle A \to A_1 \cdot x\rangle, \langle A \to A_1 \cdot x\rangle\}
\end{array}$$

and

$$\begin{array}{l}
(q_5, A_2 \to a) \\
\quad \textbf{reduces-to}\{(q_0, A_2)\} \\
\quad \textbf{contains-cancellation-core}\{\langle A \to A_2 \cdot y\rangle, \langle A \to A_2 \cdot y\rangle\}.
\end{array}$$

Two of these cancellation symbols have as after the dot, and so

$$q_{13} \overset{\text{def}}{=} \{[A_1 \to A_1 \cdot a], [A_2 \to A_2 \cdot a]\}.$$

Now $Q_{q^{13},a} = \{q_0\}$, and q_0 has no incoming transitions, so we do not need to form any temporary transitions to q_{13}. The goto successor for q_{13} on a is q_5 (unlike in Section 4.2.3, where it produced a new primed state q_{14}). At this point, we can recompute the **has-lagplr-lookahead** relation and check for missing continuation states; there are none, and so the cfsm construction is complete.

An LAGPLR parse of *aaaaxbbb* is shown in Figure 5.3. It is identical to Figure 4.3 except that every occurrence of q_{14} has been replaced with q_5. As before, the symbol $\langle * \rangle$ is used to denote $\langle A_1 \to A_1 \cdot a\rangle$.

5.2 Related Work

While this thesis is, to the author's knowledge, the first discussion of a parsing algorithm that is both incremental and noncanonical, the topics of incremental and noncanonical parsing both have extensive but independent treatments in the literature.

Of the techniques for incrementality in LR parsers, Celentano's is perhaps the simplest, as it comprises an easy change in parser implementation, with no modification to the underlying parsing algorithm. An alternative implementation is given by Yeh and Kastens [32]. Both methods are improvements on the seminal work by Ghezzi and Mandrioli [11, 12]. If a parse tree or abstract syntax tree will be generated by the parse (as opposed to some other semantic action), the method of Wagner and Graham [30] may be preferable, as it calculates an incremental parse from the syntax tree rather than storing state information explicitly. In a similar vein, Larchevêque [19] requires a *threaded* syntax tree, as introduced in Ghezzi and Mandrioli [13, 14]. Degano, Mannucci, and Mojana [6] take a different approach entirely, defining a new parsing algorithm ("jump-shift-reduce") based on splitting the parsing tables produced by the LR parsing algorithm.

Many noncanonical parsing techniques have been proposed. Some could be made incremental using Celentano's algorithm, while others cannot; those that do admit a lemma similar to Lemma 1, that is, terminals are scanned from left to right, and each terminal is "read" exactly once. One of the earliest works on noncanonical parsing is due to Szymanski [28] (also described in [27]), who built on Knuth's [18] LR(k, t) algorithm. His general framework allows symbols to be "poured" back and forth between the two stacks an arbitrary number of times, making incrementality *à la* Celentano infeasible. However, more restricted versions—including his FSPA and LR(k, ∞) algorithms—scan the input once, left-to-right, pushing only nonterminals back onto the input stack. Tai's noncanonical SLR(1) parsers [29] and Schmitz's noncanonical LALR(1) parsers [24] enjoy the same property, and thus are candidates for incrementalization. Čulik and Cohen's LR-regular parsers [17] do not, as the input must be scanned twice, first from right to left and then from left to right. Other noncanonical parsing techniques include total precedence [5], BCP [31], and Full SPM [10].

5.3 Conclusion

The Generalized Piecewise LR parsing algorithm is one of the most powerful, deterministic parsing techniques known. By allowing unbounded right context, it can alleviate many of the problems typically encountered when designing LALR(1) grammars, and the use of cancellation allows it to accept many grammars that cannot be handled by other noncanonical techniques that only allow terminal and nonterminal lookaheads. Following an intuitive development of the LR(k) and GPLR algorithms, we have illustrated and proved that the GPLR parsers can be made incremental, allowing them to re-parse small segments of a document as they are changed (for example, in a language-based editor). Future research includes a techniques for reducing the size of GPLR parsers and applications of incremental parsing techniques to other noncanonical parsing algorithms.

Step	Parser Stack	Input Stack	Action
1.	$\begin{array}{l} - \\ q_0 \end{array}$	aaaaxbbb	Shift and go to q_5
2.	$\begin{array}{ll} - & a \\ q_0 & q_5 \end{array}$	aaaxbbb	Continue in $q_1$3
3.	$\begin{array}{lll} - & a & \perp \\ q_0 & q_5 & q_{13} \end{array}$	aaaxbbb	Shift and go to q_5
4.	$\begin{array}{llll} - & a & \perp & a \\ q_0 & q_5 & q_{13} & q_5 \end{array}$	aaxbbb	Continue in $q_1$3
5.	$\begin{array}{lllll} - & a & \perp & a & \perp \\ q_0 & q_5 & q_{13} & q_5 & q_{13} \end{array}$	aaxbbb	Shift and go to q_5
6.	$\begin{array}{llllll} - & a & \perp & a & \perp & a \\ q_0 & q_5 & q_{13} & q_5 & q_{13} & q_5 \end{array}$	axbbb	Continue in q_{13}
7.	$\begin{array}{lllllll} - & a & \perp & a & \perp & a & \perp \\ q_0 & q_5 & q_{13} & q_5 & q_{13} & q_5 & q_{13} \end{array}$	axbbb	Shift and go to q_5
8.	$\begin{array}{llllllll} - & a & \perp & a & \perp & a & \perp & a \\ q_0 & q_5 & q_{13} & q_5 & q_{13} & q_5 & q_{13} & \underline{q_5} \end{array}$	xbbb	Reduce by $A_1 \rightarrow A_1 a$
9.	$\begin{array}{llllll} - & a & \perp & a & \perp & a \\ q_0 & q_5 & q_{13} & q_5 & q_{13} & \underline{q_5} \end{array}$	$\langle * \rangle$ $\mathbf{A_1}$ xbbb	Reduce by $A_1 \rightarrow A_1 a$
10.	$\begin{array}{llll} - & a & \perp & a \\ q_0 & q_5 & q_{13} & \underline{q_5} \end{array}$	$\langle * \rangle$ $\mathbf{A_1}$ $\langle * \rangle$ A_1 xbbb	Reduce by $A_1 \rightarrow A_1 a$
11.	$\begin{array}{ll} - & a \\ q_0 & \underline{q_5} \end{array}$	$\langle * \rangle$ $\mathbf{A_1}$ $\langle * \rangle$ A_1 $\langle * \rangle$ A_1 xbbb	Reduce by $A_1 \rightarrow a$
12.	$\begin{array}{ll} - & A_1 \\ q_0 & q_1 \end{array}$	$\langle * \rangle$ A_1 $\langle * \rangle$ A_1 $\langle * \rangle$ A_1 xbbb	Cancel $\langle A_1 \rightarrow A_1 \cdot a \rangle$
13.	$\begin{array}{l} - \\ q_0 \end{array}$	A_1 $\langle * \rangle$ A_1 $\langle * \rangle$ A_1 xbbb	Shift A_1
14.	$\begin{array}{ll} - & A_1 \\ q_0 & q_1 \end{array}$	$\langle * \rangle$ A_1 $\langle * \rangle$ A_1 xbbb	Cancel $\langle A_1 \rightarrow A_1 \cdot a \rangle$
15.	$\begin{array}{l} - \\ q_0 \end{array}$	A_1 $\langle * \rangle$ A_1 xbbb	Shift A_1
16.	$\begin{array}{ll} - & A_1 \\ q_0 & q_1 \end{array}$	$\langle * \rangle$ A_1 xbbb	Cancel $\langle A_1 \rightarrow A_1 \cdot a \rangle$
17.	$\begin{array}{l} - \\ q_0 \end{array}$	A_1 xbbb	Shift A_1
18.	$\begin{array}{ll} - & A_1 \\ q_0 & q_1 \end{array}$	xbbb	Shift x
19.	$\begin{array}{lll} - & A_1 & x \\ q_0 & \underline{q_1} & \underline{q_6} \end{array}$	bbb	Reduce by $A_1 \rightarrow A_1 x$
20.	$\begin{array}{ll} - & \mathbf{A_1} \\ q_0 & \underline{\mathbf{q_1}} \end{array}$	bbb	Reduce by $A \rightarrow A_1$
21.	$\begin{array}{ll} - & \mathbf{A} \\ q_0 & \mathbf{q_2} \end{array}$	bbb	Shift b
22.	$\begin{array}{lll} - & A & b \\ q_0 & q_2 & \underline{q_9} \end{array}$	bb	Reduce by $B \rightarrow b$
23.	$\begin{array}{lll} - & A & \mathbf{B} \\ q_0 & q_2 & \mathbf{q_8} \end{array}$	bb	Shift b and go to q_{12}
24.	$\begin{array}{llll} - & A & B & b \\ q_0 & q_2 & \underline{q_8} & \underline{q_{12}} \end{array}$	b	Reduce by $B \rightarrow Bb$
25.	$\begin{array}{lll} - & A & \mathbf{B} \\ q_0 & q_2 & \mathbf{q_8} \end{array}$	b	Shift b and go to q_{12}
26.	$\begin{array}{llll} - & A & B & b \\ q_0 & q_2 & \underline{q_8} & \underline{q_{12}} \end{array}$	*(end of input)*	Reduce by $B \rightarrow Bb$
27.	$\begin{array}{lll} - & A & \mathbf{B} \\ q_0 & \underline{q_2} & \underline{\mathbf{q_8}} \end{array}$	*(end of input)*	Reduce by $S \rightarrow AB$
28.	$\begin{array}{ll} - & \mathbf{S} \\ q_0 & \mathbf{q_4} \end{array}$	*(end of input)*	Accept

Figure 5.3: Complete LAGPLR(1) parse of *aaaaxbbb* in the example grammar.

A

Supplement to Figure 4.2

Below is a complete listing of the items in each state of the canonical finite state machine in Figure 4.2 and the corresponding parser actions.

```
q0:
    On <A1> go to q1
    On <A2> go to q3
    On <S> go to q4
    On a go to q5
    On <A> go to q2

    Shift <A1>
    Shift <A>
    Shift <A2>
    Shift <S>
    Shift a
    Cancel <@start> ::= <S> .
    Cancel <S> ::= <A> . <B>
    Cancel <S> ::= <A> . <B>
    Cancel <A> ::= <A1> . x
    Cancel <A> ::= <A1> . x
    Cancel <A> ::= <A2> . y
    Cancel <A> ::= <A2> . y
    Cancel <A1> ::= <A1> . a
    Cancel <A1> ::= <A1> . a
    Cancel <A2> ::= <A2> . a
    Cancel <A2> ::= <A2> . a

    [    <@start> ::= . <S>    ,        ] (basis item)
    [    <S> ::= . <A> <B>    ,    <@start> ::= <S> .    ]
    [    <A> ::= . <A1> x    ,    <S> ::= <A> . <B>    ]
    [    <A> ::= . <A2> y    ,    <S> ::= <A> . <B>    ]
    [    <A1> ::= . <A1> a    ,    <A> ::= <A1> . x    ]
    [    <A1> ::= . a    ,    <A> ::= <A1> . x    ]
    [    <A2> ::= . <A2> a    ,    <A> ::= <A2> . y    ]
    [    <A2> ::= . a    ,    <A> ::= <A2> . y    ]
    [    <A1> ::= . <A1> a    ,    <A1> ::= <A1> . a    ]
    [    <A1> ::= . a    ,    <A1> ::= <A1> . a    ]
    [    <A2> ::= . <A2> a    ,    <A2> ::= <A2> . a    ]
    [    <A2> ::= . a    ,    <A2> ::= <A2> . a    ]
```

```
q1:
    On x go to q6
    On a go to q7

    Shift x
    Shift a
    Cancel <S> ::= <A> . <B>
    Cancel <A> ::= <A1> . x
    Cancel <A1> ::= <A1> . a

    [    <A> ::= <A1> . x    ,    <S> ::= <A> . <B>    ] (basis item)
    [    <A1> ::= <A1> . a    ,    <A> ::= <A1> . x    ] (basis item)
    [    <A1> ::= <A1> . a    ,    <A1> ::= <A1> . a    ] (basis item)

q2:
    On <B> go to q8
    On b go to q9

    Shift <B>
    Shift b
    Cancel <@start> ::= <S> .
    Cancel <@start> ::= <S> .
    Cancel <@start> ::= <S> .
    Cancel <B> ::= <B> . b
    Cancel <B> ::= <B> . b

    [    <S> ::= <A> . <B>    ,    <@start> ::= <S> .    ] (basis item)
    [    <B> ::= . <B> b    ,    <@start> ::= <S> .    ]
    [    <B> ::= . b    ,    <@start> ::= <S> .    ]
    [    <B> ::= . <B> b    ,    <B> ::= <B> . b    ]
    [    <B> ::= . b    ,    <B> ::= <B> . b    ]

q3:
    On y go to q11
    On a go to q10

    Shift a
    Shift y
    Cancel <S> ::= <A> . <B>
    Cancel <A> ::= <A2> . y
    Cancel <A2> ::= <A2> . a

    [    <A> ::= <A2> . y    ,    <S> ::= <A> . <B>    ] (basis item)
    [    <A2> ::= <A2> . a    ,    <A> ::= <A2> . y    ] (basis item)
    [    <A2> ::= <A2> . a    ,    <A2> ::= <A2> . a    ] (basis item)

q4:


    [    <@start> ::= <S> .    ,        ] (basis item)

q5:
    On a continue to q13
    Reduce on x
    Continue on a
    Reduce on y
    Reduce on <A> ::= <A1> . x
    Reduce on <A> ::= <A2> . y
    Reduce on <A1> ::= <A1> . a
    Reduce on <A2> ::= <A2> . a

    [    <A1> ::= a .    ,    <A> ::= <A1> . x    ] (basis item)
    [    <A2> ::= a .    ,    <A> ::= <A2> . y    ] (basis item)
    [    <A1> ::= a .    ,    <A1> ::= <A1> . a    ] (basis item)
    [    <A2> ::= a .    ,    <A2> ::= <A2> . a    ] (basis item)

q6:

    Reduce on <B>
    Reduce on b
    Reduce on <S> ::= <A> . <B>

    [    <A> ::= <A1> x .    ,    <S> ::= <A> . <B>    ] (basis item)

q7:

    Reduce on x
    Reduce on a
    Reduce on <A> ::= <A1> . x
    Reduce on <A1> ::= <A1> . a

    [    <A1> ::= <A1> a .    ,    <A> ::= <A1> . x    ] (basis item)
    [    <A1> ::= <A1> a .    ,    <A1> ::= <A1> . a    ] (basis item)

q8:
    On b go to q12

    Shift b
    Reduce on <@start> ::= <S> .
    Reduce on <@start> ::= <S> .
    Cancel <B> ::= <B> . b

    [    <S> ::= <A> <B> .    ,    <@start> ::= <S> .    ] (basis item)
    [    <B> ::= <B> . b    ,    <@start> ::= <S> .    ] (basis item)
    [    <B> ::= <B> . b    ,    <B> ::= <B> . b    ] (basis item)

q9:

    Reduce on b
    Reduce on <@start> ::= <S> .
    Reduce on <B> ::= <B> . b

    [    <B> ::= b .    ,    <@start> ::= <S> .    ] (basis item)
    [    <B> ::= b .    ,    <B> ::= <B> . b    ] (basis item)

q10:
    Reduce on a
    Reduce on y
    Reduce on <A> ::= <A2> . y
    Reduce on <A2> ::= <A2> . a

    [    <A2> ::= <A2> a .    ,    <A> ::= <A2> . y    ] (basis item)
    [    <A2> ::= <A2> a .    ,    <A2> ::= <A2> . a    ] (basis item)

q11:

    Reduce on <B>
    Reduce on b
    Reduce on <S> ::= <A> . <B>

    [    <A> ::= <A2> y .    ,    <S> ::= <A> . <B>    ] (basis item)

q12:

    Reduce on b
    Reduce on <@start> ::= <S> .
    Reduce on <B> ::= <B> . b

    [    <B> ::= <B> b .    ,    <@start> ::= <S> .    ] (basis item)
    [    <B> ::= <B> b .    ,    <B> ::= <B> . b    ] (basis item)

q13: (Continuation State)
    On a go to q14

    Shift a
    Cancel <A> ::= <A1> . x
    Cancel <A1> ::= <A1> . a
    Cancel <A> ::= <A2> . y
    Cancel <A2> ::= <A2> . a

    [    <A1> ::= <A1> . a    ,    <A> ::= <A1> . x    ] (basis item)
    [    <A1> ::= <A1> . a    ,    <A1> ::= <A1> . a    ] (basis item)
    [    <A2> ::= <A2> . a    ,    <A> ::= <A2> . y    ] (basis item)
    [    <A2> ::= <A2> . a    ,    <A2> ::= <A2> . a    ] (basis item)

q14: (Continuation State)
    On a continue to q13

    Reduce on x
    Continue on a
    Reduce on y
    Reduce on <A> ::= <A1> . x
    Reduce on <A1> ::= <A1> . a
    Reduce on <A> ::= <A2> . y
    Reduce on <A2> ::= <A2> . a

    [    <A1> ::= <A1> a .    ,    <A> ::= <A1> . x    ] (basis item)
    [    <A1> ::= <A1> a .    ,    <A1> ::= <A1> . a    ] (basis item)
    [    <A2> ::= <A2> a .    ,    <A> ::= <A2> . y    ] (basis item)
    [    <A2> ::= <A2> a .    ,    <A2> ::= <A2> . a    ] (basis item)
```

References

[1] Rekesh Agrawal and Keith D. Detro. An efficient incremental LR parser for grammars with epsilon productions. *Acta Informatica*, 19:369–376, 1983.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Prentice Hall, Upper Saddle River, NJ, 1986.

[3] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling*, volume I: Parsing. Prentice Hall, Englewood Cliffs, NJ, 1972.

[4] Augusto Celentano. Incremental LR parsers. *Acta Inf.*, 10:307–321, 1978.

[5] Alain Colmerauer. Total precedence relations. *Journal of the Association for Computing Machinery*, 17(1):14–30, January 1970.

[6] Pierpaolo Degano, Stefano Mannucci, and Bruno Mojana. Efficient incremental lr parsing for syntax-directed editors. *ACM Transactions on Programming Languages and Systems*, 10(3):345–373, July 1988.

[7] Frank DeRemer. *Practical Translators for LR(k) Languages.* PhD thesis, M.I.T., Cambridge, MA, 1969.

[8] Frank DeRemer and Thomas Pennello. Efficient computation of LALR(1) look-ahead sets. *ACM Trans. Program. Lang. Syst.*, 4(4):615–649, 1982.

[9] J. Eve and R. Kurki-Suonio. On computing the transitive closure of a relation. *Acta Informatica*, 8:303–314, 1977.

[10] Charles Nicholas Fischer. *On Parsing Context Free Languages in Parallel Environments.* PhD thesis, Cornell University, Ithaca, NY, 1975.

[11] Carlo Ghezzi and Dino Mandrioli. Incremental parsing. Technical Report Internal Report IEEPM n. 76–15, Politeenico di Milano, Istituto di Elettrotecnica ed Elettronica, 1976.

[12] Carlo Ghezzi and Dino Mandrioli. Augmenting parsers to support incrementality. Technical Report Internal Report IEEPM n. 77–6, Politeenico di Milano, Istituto di Elettrotecnica ed Elettronica, 1977.

[13] Carlo Ghezzi and Dino Mandrioli. Incremental parsing. *ACM Transactions on Programming Languages and Systems*, 1(1):58–70, July 1979.

[14] Carlo Ghezzi and Dino Mandrioli. Augmenting parsers to support incrementality. *Journal of the Association for Computing Machinery*, 27(3):564–579, July 1980.

[15] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison Wesley, Boston, MA, 1996.

[16] John Hopcroft, Rajeev Motwani, and Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, Boston, MA, 2000.

[17] Karel Cŭlik II and Rina Cohen. LR-Regular grammars—an extension of LR(k) grammars. *Journal of Computer and System Sciences*, 7:66–96, 1973.

[18] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8:607–639, 1965.

[19] J.M. Larchevêque. Optimal incremental parsing. *ACM Transactions on Programming Languages and Systems*, 17(1):1–15, January 1995.

[20] Jeffrey L. Overbey. Parallel parsing for programming languages. Manuscript in preparation. Available at http://jeff.over.bz.

[21] Photran: An integrated development environment for Fortran. http://www.eclipse.org/photran/.

[22] Professor George Forsythe. http://www-db.stanford.edu/pub/voy/museum/pictures/display/floor1.htm.

[23] Richard Marion Schell. *Methods for Constructing Parallel Compilers for Use in a Multiprocessor Environment*. PhD thesis, University of Illinois at Urbana-Champaign, 1979.

[24] Sylvain Schmitz. Noncanonical LALR(1) parsing. In Zhe Dang and Oscar H. Ibarra, editors, *DLT'06*, pages 95–107. Springer, 2006.

[25] Seppo Sippu and Eljas Soisalon-Soininen. *Parsing Theory*, volume I: Languages and Parsing. Springer-Verlag, New York, NY, 1988.

[26] Seppo Sippu and Eljas Soisalon-Soininen. *Parsing Theory*, volume II: LR(k) and LL(k) Parsing. Springer-Verlag, New York, NY, 1990.

[27] Thomas G. Szymanski and John H. Williams. Noncanonical extensions of bottom-up parsing techniques. *SIAM Journal on Computing*, 5(2), June 1976.

[28] Thomas Gregory Szymanski. *Generalized Bottom-Up Parsing*. PhD thesis, Cornell University, Ithaca, NY, 1973.

[29] Kuo-Chung Tai. Noncanonical SLR(1) grammars. *ACM Transactions on Programming Languages and Systems*, 1(2):295–320, October 1979.

[30] Tim A. Wagner and Susan L. Graham. Efficient and flexible incremental parsing. *ACM Transactions on Programming Languages and Systems*, 20(5):980–1013, September 1998.

[31] John H. Williams. Bounded context parsable grammars. *Information and Control*, 28:314–334, 1975.

[32] Dashing Yeh and Uwe Kastens. Automatic construction of incremental LR(1)-parsers. *ACM SIGPLAN Notices*, 23(3), March 1988.