

IMPROVED REGIONALLY LEAST-COST  
SYNTAX ERROR RECOVERY

By  
MARTIN L. TRAVIOLIA  
//

Bachelor of Science  
Oral Roberts University  
Tulsa, Oklahoma  
1980

Master of Science  
University of Tulsa  
Tulsa, Oklahoma  
1985

Submitted to the Faculty of the  
Graduate College of the  
Oklahoma State University  
in partial fulfillment of  
the requirements for  
the Degree of  
DOCTOR OF PHILOSOPHY  
December, 1991

Sheddy  
KPHB  
T1815L

IMPROVED REGIONALLY LEAST-COST  
SYNTAX ERROR RECOVERY

Thesis Approved:

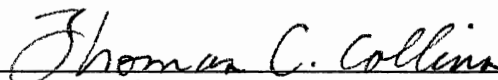


Thesis Adviser



Donald D. Fisher

Wayne B. Powell



Dean of the Graduate College

## ACKNOWLEDGMENTS

I wish to express my thanks to Dr. Hedrick for his encouragement and patience throughout my graduate program. Also, I would like to thank The Williams Companies, Doug Foster, Martin Eady, Ron Miller, and Rick Corgill for their support of my graduate work.

My family has made many sacrifices to support my efforts on this dissertation. My wife, Paula, has worked tirelessly at home to enable me to spend my time on this project. Courtney and Richard have also invested much of their time with me in this effort and have suffered many weekends without Dad. I cannot thank my family enough for their support.

Finally, thanks to my Father in Heaven.

“... there is nothing better for men than to be happy and do good while they live. That every man may eat and drink, and find satisfaction in all his toil — this is the gift of God.” —Ecclesiastes 3:8 (NIV)

## TABLE OF CONTENTS

Chapter	Page
I. INTRODUCTION . . . . .	1
1.1 LITERATURE REVIEW . . . . .	1
1.1.1 Local Repair Schemes . . . . .	3
1.1.2 Phrase-Level Schemes . . . . .	4
1.1.3 Forward Move Schemes . . . . .	6
1.1.4 Globally Least-Cost Schemes . . . . .	8
1.1.5 Locally Least-Cost Schemes . . . . .	9
1.1.6 Regionally Least-Cost Schemes . . . . .	10
1.2 SUMMARY OF RESULTS . . . . .	11
1.3 ORGANIZATION OF DISSERTATION . . . . .	12
II. THEORETICAL BACKGROUND . . . . .	13
2.1 FORMAL LANGUAGE THEORY . . . . .	13
2.2 LEAST-COST EDITING OF STRINGS . . . . .	14
2.3 LR( $k$ ) PARSING THEORY . . . . .	18
2.3.1 The LR( $k$ ) Parser . . . . .	18
2.3.2 LR( $k$ ) States and Parsing Functions . . . . .	20
2.3.3 The Explicitly Advancing LR( $k$ ) Parser . . . . .	29
2.4 EARLY'S ALGORITHM . . . . .	31
2.5 LYON'S ALGORITHM . . . . .	37
III. THE LR( $k$ ) EARLY'S ALGORITHM . . . . .	42
3.1 THE ALGORITHM . . . . .	42
3.2 PROOFS OF CORRECTNESS AND COMPLETENESS . . . . .	50
3.3 RUN TIME ANALYSIS . . . . .	59

Chapter	Page
IV. THE DEPTH-FIRST LR( $K$ ) EARLY'S ALGORITHM . . . . .	67
4.1 THE ALGORITHM . . . . .	67
4.2 PROOF OF CORRECTNESS . . . . .	70
4.3 PROOF OF COMPLETENESS . . . . .	78
4.4 RUN TIME ANALYSIS . . . . .	84
4.4.1 $\mathcal{O}(n^5)$ Time and $\mathcal{O}(n^2)$ Space Complexities . . . . .	84
4.4.2 $\mathcal{O}(n)$ Time and Space for LR( $k$ ) Grammars . . . . .	93
V. THE LEAST-COST LR( $k$ ) EARLY'S ALGORITHM . . . . .	107
5.1 THE ALGORITHM . . . . .	107
5.2 EXAMPLE EXECUTIONS . . . . .	110
5.3 PROOF OF CORRECTNESS . . . . .	112
5.4 PROOF OF COMPLETENESS . . . . .	130
5.4.1 Consistent Ordered Lists of Entries . . . . .	131
5.4.2 Monotonically Increasing Ordered Lists of Entries . . . . .	137
5.4.3 Completeness . . . . .	140
5.5 RUN TIME ANALYSIS . . . . .	151
5.5.1 $\mathcal{O}(n^5)$ Time and $\mathcal{O}(n^4)$ Space Complexities . . . . .	151
5.5.2 $\mathcal{O}(n)$ Time and Space for LR( $k$ ) Grammars . . . . .	152
5.6 REGIONALLY LEAST-COST ERROR RECOVERY . . . . .	153
VI. CONCLUSION . . . . .	159
6.1 SUMMARY . . . . .	159
6.2 FUTURE WORK . . . . .	160
BIBLIOGRAPHY . . . . .	161

## LIST OF FIGURES

Figure		Page
1.	Example Grammar . . . . .	23
2.	Sets of Valid $LR(k)$ Items for Example Grammar . . . . .	24
3.	GOTO Function for the Example Grammar . . . . .	26
4.	Derivation of Input String . . . . .	27
5.	Configurations for the Input String . . . . .	28
6.	Parse List for Early's Algorithm . . . . .	33
7.	Parse Lists for Lyon's Algorithm . . . . .	40
8.	Parse Lists for Early's Algorithm . . . . .	43
9.	Parse Lists for $LR(k)$ Early's Algorithm . . . . .	49
10.	Parse Lists for First Example . . . . .	111
11.	Parse Lists for Second Example . . . . .	112

## CHAPTER I

### INTRODUCTION

Modern compilers use syntax-directed parsing algorithms such as LL(1) and LALR(1). These algorithms have simplified the task of parsing programs greatly, but they have the undesirable property of stopping at the first syntax error in a program. This behavior is not acceptable because the program may contain several syntax errors, all of which should be reported to the programmer when the program is compiled. Therefore, researchers have developed syntax-error recovery schemes which allow syntax-directed parsers to recover from a syntax error and to continue parsing the program. Unfortunately, the syntax-error recovery schemes that have been developed are either costly in time or space, or fail to provide “good” error recovery for all errors. This dissertation presents a new syntax-error recovery scheme, for LR( $k$ ) parsers and their variants, which is less costly than similar schemes and can provide “good” error recovery on a wide range of errors.

#### 1.1 LITERATURE REVIEW

The literature on syntax-error recovery schemes is quite extensive. A large, but now dated, bibliography is provided by Ciesinger [7] and a relatively current survey of the field is provided by Hammond and Rayward-Smith [16].

In the literature, the terms “error recovery”, “error repair”, “error correction”, and “error handling” are given different, often conflicting, definitions by various authors. In this dissertation, the terms “error recovery scheme” and “error handling scheme” are taken to be synonymous and are used for any scheme that places a syntax-directed parser in a state that allows it to continue parsing after a syntax error is encountered. The terms “error repair scheme” and “error correction scheme” are also taken to be synonymous and are used for any error recovery scheme that operates by explicitly constructing a repair to the text of the program.

This review focuses on syntax-error recovery schemes that apply to LR( $k$ ) parsers and their variants and the reader is assumed to be familiar with the theory of LR( $k$ ) parsing as presented in a standard text such as Aho, Sethi, and Ullman [4]. In this section, the configuration of an LR( $k$ ) parser is represented, using Aho,



Sethi, and Ullman's notation, by

$$(s_0, X_1, s_1, \dots, X_m, s_m \mid a_i, a_{i+1}, \dots, a_n)$$

where the  $s_j$ 's are the states on the parse stack, the  $X_j$ 's are the terminal and non-terminal symbols which correspond to the states, and the  $a_k$ 's are the unconsumed input tokens.

For a configuration in which an  $LR(k)$  parser detects an error, the location of  $a_i$  is called the *parser-defined error location*. The parser-defined error locations may not be the actual location of the error because an  $LR(k)$  parser may not detect that a program is erroneous until after the point at which the error actually occurred. For example, in this PASCAL code fragment

...; I = J THEN ...

the error is the omission of an "IF" before the "I". However, an  $LR(k)$  parser will not detect an error until it encounters the "=". Thus, the parser-defined error location is at the "=" while the actual error location is at the "I".

For the purposes of this dissertation, syntax-error recovery schemes can be classified into two major categories: *ad hoc* schemes based on pragmatic principles; and *least-cost* schemes based on the minimum distance model for determining the location and nature of errors. These two classes of schemes both operate by determining the location and nature of the error and then restarting the parser in a state which enables it to continue parsing as if the erroneous section of the program text had been transformed into a correct section of program text. The primary difference between ad hoc schemes and least-cost schemes is that ad hoc schemes use heuristics, that depend on the grammar describing the language and the parsing algorithm, to determine the location and nature of an error; while least-cost schemes apply the minimum distance correction model which determines the location and nature of an error independently of any grammar for the language or any parsing algorithm.

The minimum distance correction model of syntax errors is the only model of syntax errors that has been studied. In the minimum distance correction model, errors are considered to be the result of the insertion, deletion, or substitution of tokens in the program. A more powerful model, that more accurately describes the process by which syntax errors enter into the text of a program, would be helpful to the development of error recovery schemes. The development of such a model is a tremendous task since it requires an understanding of the mental processes of programmers. In addition, the difficulties encountered in applying the minimum

distance model to the problem of syntax error recovery make it seem doubtful that a more powerful model could be applied in practice.

Both the ad hoc and least-cost schemes can be further subdivided. The ad hoc schemes can be broken into combinations of three basic schemes: local repair schemes, phrase-level schemes, and forward move schemes. The least-cost schemes can be subdivided into three distinct types: globally least-cost, locally least-cost, and regionally least-cost. The remainder of this section examines these six types of syntax-error recovery schemes.

### 1.1.1 Local Repair Schemes

Local repair schemes are based on the observation that most errors are single token errors. Ripley and Druseikis [26] report that 88 percent of all errors in their sample of PASCAL programs are single token insertions, deletions, or replacements.

A local repair scheme is a syntax-error recovery scheme where the insertion, deletion, or replacement of a single token is allowed as a repair. A simple local repair scheme considers single token repairs only at the parser-defined error location.

Local repair schemes can generate a large number of possible repairs so various methods are used to select a single repair. One method is to assign insertion, deletion, or replacement costs to each token and choose the repair with the lowest cost. A simple local repair scheme with this method is used by Sippu and Soisalon-Soininen [28] to augment their phrase-level scheme.

One local repair scheme, which is also a method for selecting among possible repairs, consists of attempting to parse a few tokens beyond a repair. If this *parse check* does not encounter another error, the repair is considered successful. Of course, when there is more than one repair that passes the parse check, some additional method must be used to select the repair. The parse check method is used by Graham, Haley, and Joy [14].

Another local repair scheme, takes into account the fact that the parser-defined error location may not be the actual error location. If no repair is successful at the parser-defined error location, then repairs are tried at previous locations. These previous locations may be at previous tokens if already parsed tokens are retained by the parser, or they may be locations on the parse stack. This scheme is used in conjunction with the parse check method by Feyock and Lazarus [12], and Burke and Fisher [6].

Finally, another local repair scheme is Burke and Fisher's scope recovery scheme [6] which attempts to insert a sequence of *scope closers* instead of just trying single token repairs. The scope closers are a language dependent set of

token sequences such as “END”, “END IF” and “END RECORD” that close recursive constructs in the language. Since omitting a scope closer is an error that is usually detected long after it’s location has been shifted onto the parse stack, scope recovery is tried at all locations that precede the parser defined error location.

The major drawback of local repair schemes is that they can be expected to fail for some errors. Thus local repair schemes must always be used in conjunction with some other error recovery scheme.

### 1.1.2 Phrase-Level Schemes

Phrase-level schemes do not attempt to repair any errors. Instead they attempt to place the parser in a state where it can continue parsing at a point beyond the error location. While phrase-level schemes can be used alone; good examples are Wirth’s follow set scheme [31] and YACC’s syntax-error recovery scheme [17]; they are most often used in conjunction with local repair schemes.

Some phrase-level schemes are based on the idea of isolating an error phrase and reducing it to a nonterminal. For example, given a configuration

$$(s_0, X_1, s_1, \dots, X_m, s_m \mid a_i, a_{i+1}, \dots, a_n)$$

where a parser-defined error is detected at  $a_i$  and an error recovery that places the parser in the configuration

$$(s_0, X_1, s_1, \dots, X_j, s_j, A, s_A \mid a_{i+k}, \dots, a_n)$$

where  $A$  is a nonterminal symbol, then the error phrase

$$X_j, \dots, X_m, a_i, \dots, a_{i+k}$$

has been reduced to the nonterminal  $A$ . Other phrase-level schemes simply change the error configuration to

$$(s_0, X_1, s_1, \dots, X_j, s_j \mid a_{i+k}, \dots, a_n)$$

where  $X_j$  is not required to be a nonterminal symbol. These schemes do not attempt to interpret the error recovery as the reduction of an error phrase to a nonterminal. They just pop the stack and delete input tokens until parsing can continue.

Phrase-level schemes that operate by reducing an error phrase to a nonterminal have been developed by Leinius [18], Peterson [25], and Sippu and Soisalon-Soininen [28]. Other phrase-level schemes have been developed by Burke and Fisher [6]; Feyock and Lazarus [12]; and Graham, Haley, and Joy [14].

The error recovery chosen by any phrase-level scheme is dependent on the order in which the stack states and input tokens are searched for a possible recovery. Sippu and Soisalon-Soininen introduce a very useful notation for describing search orders. Let  $(j, k)$  denote the segment

$$X_{j+1}, s_{j+1}, \dots, X_m, s_j \mid a_i, \dots, a_{i+k-1}$$

of the configuration

$$(s_0, X_1, s_1, \dots, X_m, s_m \mid a_i, \dots, a_n).$$

The following search order denotes a search that checks the whole stack from top to bottom for a state which can shift the current input token before deleting an input token:

$$\begin{array}{cccc} (m, 0) & (m-1, 0) & \dots & (0, 0) \\ (m, 1) & (m-1, 1) & \dots & (0, 1) \\ \vdots & & & \\ (m, n) & (m-1, n) & \dots & (0, n) \end{array}$$

Sippu and Soisalon-Soininen's notation for search orders in phrase-level schemes helps in classifying other phrase-level schemes, even when they have complicated search procedures. For instance, Burke and Fisher's secondary recovery [6] has many features such as parse checking possible recoveries and checking for the possibility of inserting scope closers. In addition, they do not use the concept of reducing an error phrase, but instead simply remove states from the parse stack until one is found that allows parsing to resume with the current token. However, when their secondary recovery scheme is examined, it is seen that they search for an error phrase in the following order:

$$\begin{array}{cccc} (m, 0) & (m, 1) & \dots & (m, n) \\ (m-1, 0) & (m-2, 0) & \dots & (0, 0) \\ (m-1, 1) & (m-2, 1) & \dots & (0, 1) \\ \vdots & & & \\ (m-1, n) & (m-2, n) & \dots & (0, n) \end{array}$$

Another example of a phrase-level scheme that has an unusual search order is YACC's syntax-error recovery scheme. This scheme is similar to the scheme used by Graham, Haley, and Joy [14] as the basis for their second level recovery scheme. In this scheme, the states on the stack are popped until one is found that has a shift on the special token *error* (these states are added by extra productions in

the grammar) and then the shift is performed. Next, the input tokens are skipped until one is found that is shiftable in the current state. This scheme corresponds to a search order of

$$(m, -) \ (m - 1, -) \ \dots \ (0, -)$$

which pops the stack independently of the input tokens; followed by a search order of

$$(m', 0) \ (m', 1) \ \dots \ (m', n)$$

where  $m'$  is the value of  $m$  determined by the first search.

While the search order notation is useful for describing phrase-level schemes, it also highlights a weakness of phrase-level schemes. The search order is fixed. If multiple recoveries are possible, the recovery selected will be the one which occurs first in the search order.

Another problem with phrase-level schemes is that they do not handle single token errors well. This problem with single token errors can be alleviated by using a local repair scheme in conjunction with a phrase-level scheme. But, Sippu and Soisalon-Soininen [28] report that a simple local repair scheme often chooses an inappropriate repair which causes extraneous errors to be generated. Burke and Fisher [6]; Feyock and Lazarus [12]; and Graham, Haley, and Joy [14] have apparently overcome this problem by using local repair schemes that parse check the repairs.

In general, researchers do not report the worst-case time complexity of their syntax-error recovery algorithms. However, Burke and Fisher's scheme, which is the most recent and powerful phrase-level scheme, is reported by Burke [5] to have a worst-case time complexity of  $\mathcal{O}(n^5)$ .

### 1.1.3 Forward Move Schemes

Forward move schemes are the result of attempts to extend the Graham-Rhodes method [13] for precedence parsers to  $\text{LR}(k)$  parsers. The Graham-Rhodes method for precedence parsers continues parsing after an error is detected until a reduction that involves the error is called for, or another error is encountered. When a reduction that involves the error is called for, the error phrase is repaired by modifying it to match the right-hand side of some production in the grammar and then performing the corresponding reduction. Of course, for  $\text{LR}(k)$  parsers, parsing beyond the parser defined error location is difficult since whenever an error is detected the parser is in a state that cannot lead to a shift of the next input token.

The problem of parsing beyond the parser-defined error location for  $\text{LR}(k)$  parsers has been solved by Druseikis and Ripley [10], Mickunas and Modry [22],

and Pennello and DeRemer [24]. All of these solutions involve restarting the parser at the parser-defined error location in a state that can shift the next input token. There may be several such states so separate parses must be carried out in parallel for each state. Pennello and DeRemer only carry out the parallel parses as long as the next action for each parse is the same.

Once the forward move has been carried out Michunas and Modry, and Pennello and DeRemer repair the single token error and reduce the error phrase so the parser can continue. Druseikis and Ripley do not attempt a repair or a reduction. Instead, they start another forward move at the point at which the previous forward move stopped. This allows the parser to continue and detect some subsequent errors, but it also lets other errors remain undetected because left context information is lost when a forward move is started.

Mickunas and Modry, and Pennello and DeRemer use single token repairs at the parser-defined error location in an attempt to find a repair that allows the parser to enter one of the states used to start the forward move. Mickunas and Modry recursively invoke their recovery scheme if another error is encountered during the forward move. Pennello and DeRemer restart the forward move at the location of any subsequent errors and then attempt to concatenate the pieces of the forward move together during the selection of repairs. For both schemes, if all repairs fail at the current error location then the entire forward move process is repeated at the previous location.

Forward move schemes share a common goal with the reduction oriented phrase-level schemes; both classes of schemes attempt to reduce an error phrase to a nonterminal. However, forward move schemes attempt to do this by recognizing and repairing single token errors in the error phrase, while phrase-level schemes just attempt to isolate the error phrase. Forward move schemes have the advantage that the right end of the error phrase is determined by contextual information; and it is not determined by a static search order.

Forward move schemes carry out a large number of parallel parses. Pennello and DeRemer show how their parallel forward moves can be combined into a single deterministic forward move by adding additional states to the  $LR(k)$  parser which are used during a forward move. This increases the efficiency of their forward moves at the cost of larger parse tables. They report the size of their parse tables for PASCAL increased by 55 percent.

While no worst-case time complexities are reported for forward moves schemes, it is clear that they are essentially backtracking schemes and are guided by heuristics. Thus, their time complexity should be at worst  $\mathcal{O}(c^n)$ . In fact, Graham, Haley, and Joy [14] cite the poor performance of forward moves schemes as motivation for their development of a phrase-level scheme.

#### 1.1.4 Globally Least-Cost Schemes

Globally least-cost schemes result from the straight forward application of the minimum distance model of errors to the problem of syntax-error recovery. These schemes are called globally least-cost because in principle they examine the entire program text when recovering from an error.

In the minimum distance model, a program is considered to be a string of tokens, and a (programming) language is simply a set of strings. The distance between two strings is the number of changes needed to transform one string into the other string, where a change is the insertion, deletion, or replacement of a token. The distance between a string and a language is the minimum of the distances between the string and each of the strings in the language. This minimum distance criterion allows the number of errors in a string for a language to be defined as the distance of the string from the language.

The minimum distance criterion can easily be extended to a least-cost criterion by assigning an insertion and deletion cost for each token and a replacement cost for every pair of tokens. The use of differing costs for the insertion, deletion, or replacement of different symbols allows a least-cost scheme to be *tuned* to reflect some practical features of syntax errors. For example, it is unlikely that a reserved word is misused in a program so the deletion cost for reserved words should be high.

While the minimum distance criterion determines the number of errors in a string, it does not fix their location. The following PASCAL code fragment illustrates this problem:

```
...IF A = 0 THEN BEGIN A := B ; ELSE ...
```

This fragment contains only one error, but it could be corrected either to

```
...IF A = 0 THEN A := B ; ELSE ...
```

or to

```
...IF A = 0 THEN BEGIN A := B ; END ELSE ...
```

To fix the location of errors in a string when more than one set of changes is possible, a rule for choosing the locations of the errors is needed. One rule, which accommodates the left to right bias of most parsing algorithms, is to choose a set of changes that places the errors at locations farthest to the right in the string. Under this rule, in the previous example, the second change would be preferred.

While the number and location of errors can be determined, there still may be several possible changes. For example, in the erroneous code fragment

...A := B C ; ...

it is obvious that an operator must be inserted between "B" and "C", but there are many possible operators that could be inserted.

Global least-cost recovery schemes have been developed by Aho and Peterson [2]; Lyon [19]; and Mauney [20]. These schemes are, in fact, strongly related to each other and they can all be viewed as extensions of Early's algorithm [11] for parsing arbitrary context free grammars (Mauney's algorithm is actually an extension of the Graham-Harrison-Ruzzo algorithm [15] which is closely related to Early's algorithm). All of these algorithms have a worst-case time complexity of  $\mathcal{O}(n^3)$ , where  $n$  is the length of the program.

Mauney reports that his algorithm readily achieves poor running times in practice. The other globally least-cost algorithms suffer from this defect as well. In fact, both Early's algorithm and the Graham-Harrison-Ruzzo algorithm are more costly than they need to be for syntax-error recovery because they proceed breadth-first (carrying all parses forward simultaneously) instead of depth-first (carrying only one parse forward at a time). The situation is made worse because globally least-cost recovery schemes can be interpreted as adding, to the grammar being used for parsing, error productions which represent the insertion, deletion or replacement of a token. The resulting grammar is highly ambiguous; just the type of grammar for which Early's algorithm and the Graham-Harrison-Ruzzo algorithm exhibit their worst-case running times.

The problems of these globally least-cost recovery schemes are illustrated in Chapter 2 where a parse of a string using Lyon's globally least-cost recovery scheme is presented.

### 1.1.5 Locally Least-Cost Schemes

Given the the poor run times of the globally least-cost schemes, one would hope that the situation could be improved by making some simplifying assumptions. Locally least-cost schemes arise from the assumption that the parser-defined error location and the actual error location are the same. Locally least-cost schemes are similar to local repair schemes except that locally least-cost schemes do not limit themselves to a single token repair. Thus, they always find a repair at the parser-defined error location.

The major drawback of a locally least-cost scheme is the limited quality of error recovery it can achieve since it uses only a small amount of context in selecting a repair. For example, consider the following PASCAL code fragment.

...; A := B C ...



A correct prefix parser will detect a syntax error on reading “C”, but there are several possible recoveries:

```
...; A := B + C ...
...; A := B ; C ...
...; A := B [ C ...
...; A := B ( C ...
```

However, because a locally least-cost scheme uses only the limited amount of information that exists at the point the error is detected, it will always make the same recovery in this circumstance. It could make a better recovery if it used more information such as the next input token:

```
...; A := B C ;
...; A := B C :=
...; A := B C ]
...; A := B C )
```

Another reason the quality of locally least-cost schemes is limited is that changes are allowed only at the point where the error is detected, but this point may not be the point at which the error exists. For example, given the PASCAL code fragment:

```
...; I = J THEN ...
```

an LR( $k$ ) parser will not detect an error until the “=” is read, since “I” could be the beginning of an assignment statement. Therefore, locally least-cost recovery replaces “=” with “:=” and a second, spurious error is found when the “THEN” is read.

### 1.1.6 Regionally Least-Cost Schemes

Regionally least-cost schemes attempt to address the weakness of locally least-cost error recovery by allowing changes in the region of the program surrounding the parser defined error location. Regionally least-cost schemes are very similar to forward move schemes in that both can look at regions of the program text. The primary difference between the two is that regionally least-cost schemes choose the least-cost parse after carrying out all possible parses of the region allowing for the insertion, deletion, and replacement of tokens (pruning of the parses is possible due to the least-cost criterion).

Regionally least-cost schemes have been developed by Mauney [20] and Tai [29]. Mauney's scheme is a globally least-cost scheme, based on the Graham-Harrison-Ruzzo algorithm, modified to work within a region; but it still has a worst-case time complexity of  $\mathcal{O}(d^3)$  where  $d$  is the length of the region. Tai's scheme is based on pattern matching and is only suitable for very small regions since the number of patterns increases rapidly with the size of the region.

One issue for regionally least-cost schemes is the determination of the region size. Mauney and Fischer [21] resolve this issue by defining a Moderate Phrase-Level Uniqueness (MPLU) property for tokens. They show that a regionally least-cost repair scheme can use the first MPLU token after the error location as the end marker for the region. If the least-cost repair of the region does not delete the end marker, then a larger region will not improve the recovery unless it contains additional errors. For a large PASCAL program, they report that the mean distance between MPLU tokens is less than 11 tokens.

The quality of syntax-error recovery provided by a regionally least-cost scheme should be as good or better than that provided by a forward move scheme over the same region. This follows from the observation that if a regionally least-cost scheme considers the same tokens as a forward move scheme, then it should select the same or a better repair than the forward move scheme because it is not limited to single token repairs.

Also, the quality of the syntax-error recovery provided by a regionally least-cost scheme should be as good or better than that provided by a phrase-level scheme; if the actual error phrase is in the region. This is because phrase-level schemes are restricted to a fixed search order when locating an error phrase, while a regionally least-cost scheme is driven only by the context of the error and the cost of repairs.

Regionally least-cost schemes have the potential to be superior syntax-error recovery schemes; however, they currently suffer from running times similar to those for globally least-cost schemes. Mauney notes that the  $\mathcal{O}(n^3)$  worst-case time complexity of his algorithm is achieved in practice because it uses a breadth-first search. The algorithm examines most of the possible repairs in the region before finding the least-cost repair. Mauney points out that a depth-first search would greatly increase the speed of a regionally least-cost algorithm for most errors encountered in practice.

## 1.2 SUMMARY OF RESULTS

In this dissertation, a new regionally least-cost syntax-error recovery scheme is developed for LR( $k$ ) parsers. This scheme is based on a new globally least-cost

algorithm, called the Least-Cost LR( $k$ ) Early's Algorithm. Unlike other globally least-cost error recovery schemes, the new global least-cost error recovery scheme is not based directly on Early's algorithm or the related Graham-Harrison-Ruzzo algorithm and it uses a depth-first search. The algorithm's worst-case time complexity is  $\mathcal{O}(n^5)$  which makes its worst-case performance superior to forward move schemes and the same as the most powerful phrase-level schemes. Furthermore, the depth-first nature of the algorithm enables it to perform linearly for correct input.

The Least-Cost LR( $k$ ) Early's Algorithm is developed in three steps. First, a new algorithm, related to Early's algorithm, is developed. This algorithm is called the LR( $k$ ) Early's Algorithm because it uses the states of an LR( $k$ ) parser instead of LR( $k$ ) items. The LR( $k$ ) Early's Algorithm has the same space complexity as Early's algorithm, but its time complexity is  $\mathcal{O}(n^4)$ . The LR( $k$ ) Early's Algorithm shares the same breadth-first bias as Early's algorithm and the Graham-Harrison-Ruzzo algorithm.

Second, the Depth-First LR( $k$ ) Early's Algorithm is developed to overcome the breadth-first bias of the LR( $k$ ) Early's Algorithm. This algorithm allows a single parse to be pursued; and backtracks to try alternate parses only if the need arises. The Depth-First LR( $k$ ) Early's Algorithm has the same space complexity as the LR( $k$ ) Early's Algorithm, but its time complexity is  $\mathcal{O}(n^5)$ .

Third, the Depth-First LR( $k$ ) Early's Algorithm is used as the basis for the development of the Least-Cost LR( $k$ ) Early's Algorithm. This algorithm finds the globally least-cost repair of a string. Its time complexity is  $\mathcal{O}(n^5)$ . However, for an LR( $k$ ) grammar its time complexity is  $\mathcal{O}(n)$  for correct inputs.

Finally, the application of the Least-Cost LR( $k$ ) Early's Algorithm to regionally least-cost error recovery is discussed.

### 1.3 ORGANIZATION OF DISSERTATION

This dissertation is organized into six chapters. Chapter 1 is this introduction. Chapter 2 presents theoretical background. Chapter 3 develops the LR( $k$ ) Early's Algorithm. Chapter 4 develops the Depth-First LR( $k$ ) Early's Algorithm. Chapter 5 develops the Least-Cost LR( $k$ ) Early's Algorithm. Chapter 6 is a summary and also discusses directions for future research.

## CHAPTER II

### THEORETICAL BACKGROUND

In this chapter, the necessary background material for formal language theory, least-cost editing of strings, LR( $k$ ) parsing theory, and Early's algorithm is presented.

#### 2.1 FORMAL LANGUAGE THEORY

This section presents the terminology and notation for the formal language theory used in this paper. For further details, the reader is referred to Aho and Ullman [1] from which this material is derived.

An *alphabet* is a finite set of symbols; a *string* is a sequence of symbols from an alphabet; and a *language* is a set of strings from an alphabet. The empty string is denoted by  $\epsilon$ . The concatenation of two strings,  $x$  and  $y$ , is denoted by the juxtaposition of  $x$  and  $y$ ,  $xy$ . The length of a string  $x$  is denoted by  $|x|$ . A substring of  $x$  that contains the  $i^{\text{th}}$  thru  $j^{\text{th}}$  symbols of  $x$  is denoted by  $x_{i:j}$ .

The operator  $\oplus_k$  concatenates the first  $k$  characters from two strings  $x$  and  $y$  and is defined as follows:

$$x \oplus_k y = (xy)_{1:k}.$$

The operator  $\oplus_k$  is extended to pairs of languages by the following definition:

$$L_1 \oplus_k L_2 = \{w \mid x \in L_1, y \in L_2, \text{ and } w = x \oplus_k y\}.$$

If  $\Sigma$  denotes an alphabet then  $\Sigma^*$  denotes the set of strings over  $\Sigma$ ;  $\Sigma^+$  denotes the set of nonempty strings over  $\Sigma$ ; and  $\Sigma^k$  denotes the set of strings of length  $k$  over  $\Sigma$ . For convenience,  $\{a\}^k$  is written as  $a^k$ .

A context-free grammar (CFG) is a quadruple  $(N, \Sigma, P, S)$  where  $N$  is an alphabet of *nonterminal symbols*,  $\Sigma$  is an alphabet of *terminal symbols*,  $P$  is a set of *productions* and is a subset of  $N \times (N \cup \Sigma)^*$ , and  $S$  is a symbol of  $N$  called the start symbol. A production is written as  $A \rightarrow \alpha$ , where  $A$  is a nonterminal symbol from  $N$  and  $\alpha$  is a string of symbols from  $(N \cup \Sigma)^*$ . The productions in  $P$

are numbered, starting with one. If  $p$  is the number of a production,  $A \rightarrow \alpha$ , then  $\text{LHS}(p) = A$  and  $\text{RHS}(p) = \alpha$ .

For a CFG  $G$ , a relation  $\Rightarrow$  can be defined on strings from  $(N \cup \Sigma)^*$  as follows: If  $\alpha A \beta \in (N \cup \Sigma)^*$  and there exists a production  $A \rightarrow \gamma \in P$  then  $\alpha A \beta \Rightarrow \alpha \gamma \beta$  and it is said that  $\alpha A \beta$  directly derives  $\alpha \gamma \beta$ . The transitive and reflexive closure of  $\Rightarrow$  is denoted by  $\Rightarrow^*$  and if  $\alpha \Rightarrow^* \beta$  then it is said that  $\alpha$  derives  $\beta$ . A sequence  $\alpha_0 \Rightarrow \alpha_1 \Rightarrow \dots \Rightarrow \alpha_n$  is called a derivation from  $\alpha_0$  to  $\alpha_n$  of length  $n$ . Finally, the language generated by  $G$  is the set of all strings of terminal symbols that can be derived from  $S$ .

Throughout this dissertation, the notational conventions of Aho and Ullman [1] for terminal symbols, nonterminal symbols, and strings of terminal or nonterminal symbols are followed. The lower case letters at the beginning of the alphabet;  $a, b, c, \dots$ ; represent terminal symbols. The upper case letters at the beginning of the alphabet;  $A, B, C, \dots$ ; represent nonterminal symbols. The upper case letters at the end of the alphabet;  $V, W, X, \dots$ ; represent terminal or nonterminal symbols. The lower case letters at the end of the alphabet;  $v, w, x, \dots$ ; represent strings of terminal symbols. The lower case Greek letters;  $\alpha, \beta, \dots$ ; represent strings of terminal or nonterminal symbols.

## 2.2 LEAST-COST EDITING OF STRINGS

This section presents the formal machinery for the concept of the least-cost edit of a string. Much of this material is taken from [30].

*Definition 2.2.1 (Edit Operation)* Given an alphabet  $\Sigma$ ,  $a, b \in (\Sigma \cup \epsilon)$  and  $ab \neq \epsilon$ , an edit operation  $(a \mapsto b)$  denotes the replacement of  $a$  with  $b$  except, when  $a = \epsilon$ , it denotes the insertion of  $b$  and, when  $b = \epsilon$ , it denotes the deletion of  $a$ .

*Definition 2.2.2 (Set of Edit Operations)* Given two alphabets  $\Delta$  and  $\Sigma$ ,  $\Delta$  is called a set of edit operations for  $\Sigma$  if  $\Delta = \{(a \mapsto b) \mid a, b \in \Sigma \cup \epsilon \text{ and } ab \neq \epsilon\}$ .

*Definition 2.2.3 (Edit Sequence)* Given a set of edit operations  $\Delta$ , a string  $S \in \Delta^*$  is called an edit sequence.

*Definition 2.2.4 (Editing a String)* Given an set of edit operations  $\Delta$  for  $\Sigma$ , an edit sequence,  $S \in \Delta^*$ , is said to edit  $x \in \Sigma^*$  to  $y \in \Sigma^*$ , denoted  $x \xrightarrow{S} y$ , if either  $S = x = y = \epsilon$  or  $S = (a \mapsto b)S'$ ,  $x = ax'$ ,  $y = by'$ , and  $x' \xrightarrow{S'} y'$ .

*Definition 2.2.5 (Edit Cost Function for Edit Operations)* Given a set of edit operations  $\Delta$ , a function  $W : \Delta \rightarrow N$ , where  $N$  is the set of nonnegative integers, is an edit cost function if  $W((a \mapsto a)) = 0$  for all  $a \in \Sigma$ , and if  $W((a \mapsto b)) + W((b \mapsto c)) \geq W((a \mapsto c))$  for any  $a, b, c \in \Sigma$ .

*Definition 2.2.6 (Edit Cost Function for Edit Sequences)* If  $W : \Delta \rightarrow N$  is an edit cost function and  $S \in \Delta^*$  then  $W(S) = W((a \mapsto b)) + W(S')$  where  $S = (a \mapsto b)S'$  and  $W(\epsilon) = 0$ .

Requiring the edit cost function  $W$  to satisfy the triangle inequality guarantees that a lower cost edit can not result from applying edit operations to the result of previous edit operations. This constraint is necessary since an edit sequence can not apply an edit operation to the result of a previous edit operation.

In order to guarantee that for any  $x, y \in \Sigma^*$  there exists an edit sequence  $S$  such that  $x \xrightarrow{S} y$ , it is assumed for the remainder of this paper that  $\Delta = \{(a \mapsto b) \mid \text{for any } a, b \in (\Sigma \cup \epsilon)\}$ . This assumption is not a major constraint since  $W((a \mapsto b))$  can be set to an arbitrarily large value which will effectively prohibit the use of  $(a \mapsto b)$  in a least-cost edit.

*Definition 2.2.7 (Least Cost Edit of a String into a Language)* If  $L$  is a language over  $\Sigma$  and  $x \in \Sigma^*$ , then the least cost edit of  $x$  into  $L$  is given by an edit sequence  $S$  such that  $W(S) = \min(\{W(T) \mid x \xrightarrow{T} y \text{ and } y \in L\})$ .

Finally, the following lemmas describe the effects of concatenating edit sequences and their strings.

**LEMMA 2.2.1** *If  $x' \xrightarrow{S'} y'$  and  $x'' \xrightarrow{S''} y''$  then  $x'x'' \xrightarrow{S'S''} y'y''$  and  $W(S') + W(S'') = W(S'S'')$ .*

Proof: Let

$$S' = (a'_1 \mapsto b'_1)(a'_2 \mapsto b'_2) \dots (a'_{n'} \mapsto b'_{n'})$$

and

$$S'' = (a''_1 \mapsto b''_1)(a''_2 \mapsto b''_2) \dots (a''_{n''} \mapsto b''_{n''}).$$

Then,

$$x' = a'_1 a'_2 \dots a'_{n'},$$

$$y' = b'_1 b'_2 \dots b'_{n'},$$

$$x'' = a''_1 a''_2 \dots a''_{n''}$$

and

$$y'' = b_1'' b_2'' \dots b_{n''}''$$

so

$$x' x'' = a_1' a_2' \dots a_{n'}' a_1'' a_2'' \dots a_{n''}'' ,$$

$$y' y'' = b_1' b_2' \dots b_{n'}' b_1'' b_2'' \dots b_{n''}'' ,$$

and

$$S' S'' = (a_1' \mapsto b_1') (a_2' \mapsto b_2') \dots (a_{n'}' \mapsto b_{n'}') (a_1'' \mapsto b_1'') (a_2'' \mapsto b_2'') \dots (a_{n''}'' \mapsto b_{n''}'') .$$

Therefore,  $x' x'' \xrightarrow{S' S''} y' y''$ . Furthermore,  $W(S') + W(S'') = W(S' S'')$  since

$$W(S') = \sum_{i=1}^{n'} W((a_i' \mapsto b_i'))$$

$$W(S'') = \sum_{i=1}^{n''} W((a_i'' \mapsto b_i''))$$

and

$$W(S' S'') = \sum_{i=1}^{n'} W((a_i' \mapsto b_i')) + \sum_{i=1}^{n''} W((a_i'' \mapsto b_i'')) .$$

■

**LEMMA 2.2.2** *If  $x \xrightarrow{S} y$  and  $x = x' x''$  then there exists  $y', y'', S',$  and  $S''$  such that  $x' \xrightarrow{S'} y', x'' \xrightarrow{S''} y'', y = y' y''$  and  $S = S' S''$ .*

**Proof:** Let

$$S = (a_1 \mapsto b_1) (a_2 \mapsto b_2) \dots (a_n \mapsto b_n)$$

where  $x = a_1 a_2 \dots a_n$  and  $y = b_1 b_2 \dots b_n$ . Now, for any  $x'$  and  $x''$ , where  $x' x'' = x$ , there exists  $k$  such that  $0 \leq k \leq n$  and

$$x' = a_1 a_2 \dots a_k$$

and

$$x'' = a_{k+1} a_{k+2} \dots a_n .$$

Let

$$S' = (a_1 \mapsto b_1) (a_2 \mapsto b_2) \dots (a_k \mapsto b_k) ,$$

$$S'' = (a_{k+1} \mapsto b_{k+1})(a_{k+2} \mapsto b_{k+2}) \dots (a_n \mapsto b_n),$$

$$y' = b_1 b_2 \dots b_k$$

and

$$y'' = b_{k+1} b_{k+2} \dots b_n.$$

Obviously,  $S = S'S''$  and  $y = y'y''$ . Furthermore,  $x' \xrightarrow{S'} y'$  and  $x'' \xrightarrow{S''} y''$ . ■

**LEMMA 2.2.3** *If  $x \xrightarrow{S} y$  and  $y = y'y''$  then there exists  $x', x'', S'$ , and  $S''$  such that  $x' \xrightarrow{S'} y'$ ,  $x'' \xrightarrow{S''} y''$ ,  $x = x'x''$  and  $S = S'S''$ .*

Proof: Let

$$S = (a_1 \mapsto b_1)(a_2 \mapsto b_2) \dots (a_n \mapsto b_n)$$

where  $x = a_1 a_2 \dots a_n$  and  $y = b_1 b_2 \dots b_n$ . Now, for any  $y'$  and  $y''$ , where  $y'y'' = y$ , there exists  $k$  such that  $0 \leq k \leq n$  and

$$y' = b_1 b_2 \dots b_k$$

and

$$y'' = b_{k+1} b_{k+2} \dots b_n.$$

Let

$$S' = (a_1 \mapsto b_1)(a_2 \mapsto b_2) \dots (a_k \mapsto b_k),$$

$$S'' = (a_{k+1} \mapsto b_{k+1})(a_{k+2} \mapsto b_{k+2}) \dots (a_n \mapsto b_n),$$

$$x' = a_1 a_2 \dots a_k$$

and

$$x'' = a_{k+1} a_{k+2} \dots a_n.$$

Obviously,  $S = S'S''$  and  $x = x'x''$ . Furthermore,  $x' \xrightarrow{S'} y'$  and  $x'' \xrightarrow{S''} y''$ . ■

**LEMMA 2.2.4** *If  $x' \xrightarrow{S'} y'$ ,  $x'' \xrightarrow{S''} y''$  and  $W(S'S'') = \min(\{W(T) \mid x'x'' \xrightarrow{T} y'y''\})$  then  $W(S') = \min(\{W(T) \mid x' \xrightarrow{T} y'\})$  and  $W(S'') = \min(\{W(T) \mid x'' \xrightarrow{T} y''\})$ .*

Proof: The lemma is proved by contradiction. It is assumed that there exists an edit sequence  $S$  such that  $x' \xrightarrow{S} y'$  and  $W(S) < W(S')$ , or  $x'' \xrightarrow{S} y''$  and  $W(S) < W(S'')$ . The argument for either case is the same so let  $x' \xrightarrow{S} y'$  and  $W(S) < W(S')$ . But, then  $x'x'' \xrightarrow{SS''} y'y''$  and  $W(SS'') < W(S'S'')$ ; which is a contradiction. ■



## 2.3 LR( $k$ ) PARSING THEORY

This section reviews the principle definitions and results of LR( $k$ ) parsing theory; establishes the notation and terminology used in this dissertation; and lays the ground work for the development of the LR( $k$ ) Early's Algorithm. For further details about LR( $k$ ) parsing theory, the reader is referred to Aho and Ullman [1] from which most of the material in this section is derived.

### 2.3.1 The LR( $k$ ) Parser

The LR( $k$ ) parsing algorithm presented here is a nondeterministic parser when its grammar is not LR( $k$ ). No proofs of any properties of this nondeterministic parser are provided as the nondeterministic parser is used only to motivate some of the discussion of the LR( $k$ ) Early's Algorithm. For more details on the theory of nondeterministic LR( $k$ ) parsing, the reader is referred to Sippu and Soisalon-Soininen [27], which uses a different approach than is used here, and to Dehnert [8] from which this material is derived.

Algorithm 2.3.1 is the LR( $k$ ) parser. The LR( $k$ ) parser uses a set of states denoted by  $Q$ . Lower case letters  $q, r, s, \dots$  are used to represent elements of  $Q$ . The initial state of the LR( $k$ ) parser is represented by 0. The distinguished final state of  $Q$  is represented by  $f$ . The LR( $k$ ) parser accepts its input string by halting in  $f$ .

Associated with each  $q \in Q$  are two functions:  $f_q$  which is called the parsing action function; and  $g_q$  which is called the goto function. The function  $f_q$  maps strings from  $(\Sigma \cup \{\$\})^k$  to sets with elements of the form **shift**, **reduce 1**, **reduce 2**,  $\dots$  or **reduce  $|P|$** , where **reduce  $p$**  means to reduce using the  $p^{\text{th}}$  production of  $P$ . The function  $g_q$  maps symbols from  $(N \cup \Sigma)$  to elements of  $\{R \mid R \subseteq Q \text{ and } |R| \leq 1\}$ . The functions  $f_q$  and  $g_q$  must meet the following three restrictions for the states 0 and  $f$ :

- $f_f(u) = \emptyset$  for all  $u \in (\Sigma \cup \{\$\})^k$ ;
- $g_f(X) = \emptyset$  for all  $X \in (N \cup \Sigma)$ ; and
- $0 \notin g_q(X)$  for all  $q \in Q$  and  $X \in (N \cup \Sigma)$ .

These restrictions are required so that some of the proofs in this dissertation may be carried out. It should be noted that these restrictions are satisfied by the LR( $k$ ) parsers used in practice.

One feature of Algorithm 2.3.1 is that it allows an LR( $k$ ) parser to use an almost arbitrary  $Q$ ,  $f_q$ , and  $g_q$ . Consequently, the results of this dissertation may be

### ALGORITHM 2.3.1 *The LR(k) Parser*

Let  $G$  be a reduced CFG  $(N, \Sigma, P, S)$  and  $Q$  be a set of states as described previously. The input for all copies of the algorithm is  $w \in \Sigma^*$ . The output from a copy of the algorithm is a sequence of production numbers which represents a right parse of  $w$  for  $G$ , if the algorithm halts in the final state  $f$ . Three variables are used:  $q$  is the current state;  $i$  is the current position in the input string; and  $\alpha$  is a string of states which is the current stack of the parser. The algorithm proceeds as follows:

- I. Create the initial copy of the algorithm; set  $i = 1$ ; set  $q = 0$ ; set  $\alpha$  to 0; and append  $\$^{k+1}$  to  $w$ .
- II. Repeat this step as long as  $f_q(w_{i:i+k-1})$  contains at least one element. If  $f_q(w_{i:i+k-1})$  contains more than one element, create a new copy of the current algorithm for each additional element beyond the first one. Then, for each element, perform whichever of the following two cases applies in the copy of the algorithm for that element:
  - A. If **shift**  $\in f_q(w)$  then if  $g_q(w_{i:i})$  is empty, halt; otherwise, let  $i = i + 1$ ,  $q = g_q(w_{i:i})$ ,  $\alpha = \alpha q$ .
  - B. If **reduce**  $p \in f_q(w)$  then let  $m = |\text{RHS}(p)|$  and perform whichever of the following two cases applies for  $m$ :
    1.  $m > |\alpha|$ : Let  $r = \alpha_{(|\alpha|-m):(|\alpha|-m)}$ . If  $g_r(\text{LHS}(p))$  is empty, halt; otherwise, let  $q = g_r(\text{LHS}(p))$  and  $\alpha = \alpha_{1:(|\alpha|-m)}q$ .
    2.  $m \leq |\alpha|$ : If  $g_0(\text{LHS}(p))$  is empty, halt; otherwise, let  $q = g_0(\text{LHS}(p))$  and  $\alpha = \alpha_{1:1}q$ .

applied to any of the optimized LR( $k$ ) parsers which are used in practice. However, this flexibility requires the restrictions on the functions  $f_q$  and  $g_q$  and the addition of step II.B.2 to Algorithm 2.3.1. This step handles stack underflows and preserves the property that 0 is always the bottom state of the stack.

The actions of an LR( $k$ ) parser are described in terms of their affects on the *configurations* of the parser. A configuration of the LR( $k$ ) parser is an ordered pair  $(\alpha, w_{i:n})$  where  $w_{i:n}$  is the input remaining to be parsed and  $\alpha$  is the stack of the parser with  $\alpha_{|\alpha|-|\alpha|}$  the top of the stack and  $\alpha_{1:1}$  the bottom of the stack. When the input remaining to be parsed is not of any interest, a configuration is denoted

by  $(\alpha, \dots)$ .

A move of the  $LR(k)$  parser is the application of either step II.A, a shift, or step II.B, a reduction, to a configuration of the parser. A move results in a new configuration of the parser and is written as  $(\alpha, w_{i:n}) \vdash (\beta, w_{j:n})$  in which  $\vdash$  is read as moves directly to. Thus, the moves a parser can make form a relation on the configurations of the parser. The transitive closure of this relation is denoted by  $\vdash^+$  and the transitive and reflexive closure of this relation is denoted by  $\vdash^*$ . A sequence of moves of length  $l$  is denoted by  $\vdash^l$ .

A special notation

$$(\alpha \wr q, w_{i:n}) \vdash^* (\alpha q \gamma, w_{j:n})$$

is used through out this paper to denote a sequence of moves

$$(\alpha q, w_{i:n}) \vdash (\beta_1, \dots) \vdash (\beta_2, \dots) \vdash \dots \vdash (\beta_m, \dots) \vdash (\alpha q \gamma, w_{j:n})$$

where  $|\beta_l| > |\alpha q|$  for  $1 \leq l \leq m$ . Such a sequence of moves by the  $LR(k)$  parser has the property that none of the states in  $\alpha q$  is removed from the stack. Furthermore, none of the moves in the sequence depend on the states in  $\alpha$ . Thus, if

$$(\alpha \wr q, w_{i:n}) \vdash^* (\alpha q r, w_{l:n})$$

and

$$(\beta \wr r, w_{l:n}) \vdash^* (\beta r \delta, w_{j:n})$$

then

$$(\alpha \wr q, w_{i:n}) \vdash^* (\alpha q r \delta, w_{j:n}).$$

### 2.3.2 $LR(k)$ States and Parsing Functions

While the  $LR(k)$  parser and the notation for describing its actions have been introduced, the set  $Q$  and the parsing functions  $f_q$  and  $g_q$  have not been defined for a context-free grammar  $G$ . The following definitions introduce the basic concepts of  $LR(k)$  parsing theory and lead to the definition of  $Q$ ,  $f_q$ , and  $g_q$  for a context-free grammar  $G$ .

*Definition 2.3.1* Given a CFG  $G = (N, \Sigma, P, S)$ ,  $G$  is called a *reduced* CFG if for each  $X \in (N \cup \Sigma)$  there exists a derivation  $S \xRightarrow{*} wXy \xRightarrow{*} wxy$  where  $w, x, y \in \Sigma$ .

*Definition 2.3.2* Given a reduced CFG  $G = (N, \Sigma, P, S)$ ,

$$\text{FIRST}_k(\alpha) = \{x \mid \alpha \xRightarrow{*} x\beta \text{ and } |x| = k \text{ or } \alpha \xRightarrow{*} x \text{ and } |x| < k\}.$$

*Definition 2.3.3* Given a reduced CFG  $G = (N, \Sigma, P, S)$ , a derivation  $S \Rightarrow \alpha_0 \Rightarrow \alpha_1 \Rightarrow \dots \Rightarrow \alpha_n$  is called a *rightmost derivation* and is denoted  $S \xRightarrow{rm*} \alpha_n$ , if, for each step  $\alpha_i \Rightarrow \alpha_{i+1}$  in the derivation, the rightmost nonterminal  $A$  in  $\alpha_i$  is replaced using a production  $A \rightarrow \beta$  to obtain  $\alpha_{i+1}$ .

*Definition 2.3.4* Given a reduced CFG  $G = (N, \Sigma, P, S)$ , if  $S \xRightarrow{rm*} \alpha$  then  $\alpha$  is called a *right sentential form* of  $G$ .

*Definition 2.3.5* Given a reduced CFG  $G = (N, \Sigma, P, S)$  and a string  $\gamma \in (N \cup \sigma)^*$ , then  $\gamma$  is a *viable prefix* of  $G$ , if there exists a rightmost derivation  $S \xRightarrow{rm*} \alpha A w \xRightarrow{rm*} \alpha \beta_1 \beta_2 w$  and  $\gamma = \alpha \beta_1$ .

*Definition 2.3.6* Given a reduced CFG  $G = (N, \Sigma, P, S)$ , then  $[A \rightarrow \beta_1 \cdot \beta_2, u]$  is an *LR( $k$ ) item* (for  $k$  and  $G$ ), if  $A \rightarrow \beta_1 \beta_2$  is a production in  $P$  and  $u \in (\cup_{i=1}^k \Sigma^i \oplus_k \$^k)$ .

*Definition 2.3.7* Given a reduced CFG  $G = (N, \Sigma, P, S)$ , a viable prefix  $\alpha \beta_1$  for  $G$ , and an LR( $k$ ) item  $[A \rightarrow \beta_1 \cdot \beta_2, u]$  for  $G$ , then  $[A \rightarrow \beta_1 \cdot \beta_2, u]$  is a *valid item* for  $\alpha \beta_1$ , if there is a rightmost derivation  $S \xRightarrow{rm*} \alpha A w \xRightarrow{rm*} \alpha \beta_1 \beta_2 w$  such that  $u = \text{FIRST}_k(w) \oplus_k \$^k$ .

*Definition 2.3.8* Given a reduced CFG  $G = (N, \Sigma, P, S)$  and a viable prefix  $\alpha$  of  $G$ ,  $V_k^G(\alpha)$  is the set of all valid LR( $k$ ) items for  $\alpha$  and the *collection of sets of valid LR( $k$ ) items* is  $\{V_k^G(\gamma) \mid \gamma \text{ is a viable prefix of } G\}$ .

*Definition 2.3.9* Given a reduced CFG  $G = (N, \Sigma, P, S)$ , the *augmented grammar derived from  $G$*  is  $G' = (S' \cup N, \Sigma, \{S' \rightarrow S\$ \} \cup P, S')$ , where  $S'$  and  $\$$  are not in  $(N \cup \Sigma)$ .

The previous definition uses the production  $S' \rightarrow S\$$  to augment the grammar instead of the more standard  $S' \rightarrow S$ . This is done so that the resulting  $\text{LR}(k)$  parser will ‘shift’ on the end marker  $\$$  and enter a unique final state. The existence of a unique final state simplifies subsequent algorithms and proofs. This trick is borrowed from Early [11].

*Definition 2.3.10* Given a reduced CFG  $G = (N, \Sigma, P, S)$ , the collection,  $Q$ , of sets of valid  $\text{LR}(k)$  items for its augmented grammar  $G'$  is called the *canonical collection* of sets of  $\text{LR}(k)$  items for  $G$ .

*Definition 2.3.11* Given a reduced CFG  $G = (N, \Sigma, P, S)$  and the canonical collection  $Q$  of sets of  $\text{LR}(k)$  items for  $G$ , the function  $\text{GOTO}$  maps  $Q \times (N \cup \Sigma)$  to  $Q$  such that  $\text{GOTO}(V_k^G(\alpha), X) = V_k^G(\alpha X)$ .

The next definition is unusual in  $\text{LR}(k)$  parsing theory. It is the standard definition for a consistent set of items but the set is called a deterministic set of items. The term deterministic is used here to emphasize that a set of items lacking this property can still be used by the nondeterministic  $\text{LR}(k)$  parser. Dehnert [8] shows that a right parse can still be generated under these circumstances.

*Definition 2.3.12* Given a reduced CFG  $G = (N, \Sigma, P, S)$ , the set of valid  $\text{LR}(k)$  items for a viable prefix  $\gamma$  of  $G$  is *deterministic* if there do not exist two items  $[A \rightarrow \cdot \alpha, u]$  and  $[B \rightarrow \beta_1 \cdot \beta_2 \beta_3, v]$  in the set such that  $|\beta_2| \leq 1$ ,  $|\beta_2| |\beta_3| = |\beta_3|$ ,  $\beta_2 \in \Sigma^*$ , and  $u \in (\text{FIRST}_k(\beta_2 \beta_3 v) \oplus_k \$^k)$ .

The following definition defines a  $\text{LR}(k)$  grammar in a nonstandard way. Instead of the traditional definition, an  $\text{LR}(k)$  grammar is defined in terms of a canonical collection of deterministic sets of  $\text{LR}(k)$  items. This approach is the opposite of Aho and Ullman’s [1] in which this definition is proved as a theorem. The definition is used here to emphasize the validity of those aspects of  $\text{LR}(k)$  parsing theory that apply to non- $\text{LR}(k)$  grammars.

*Definition 2.3.13* A reduced CFG  $G = (N, \Sigma, P, S)$  is called an  *$\text{LR}(k)$  grammar*, if each set of items in its canonical collection of sets of  $\text{LR}(k)$  items is deterministic.

*Definition 2.3.14* Given a reduced CFG  $G = (N, \Sigma, P, S)$ , the *canonical  $\text{LR}(k)$  parser* for  $G$  is an  $\text{LR}(k)$  parser for which  $Q$  is the canonical collection of  $\text{LR}(k)$  items for  $G$ ;  $0$  is the set of items which contains  $[S' \rightarrow S\$, \$^k]$ ;  $f$  is the set of items which contains  $[S' \rightarrow S\$, \$^k]$ ;  $g_q(X) = \text{GOTO}(q, X)$ ; and  $f_q$  is defined as follows:

$$\begin{aligned}
S' &\rightarrow E \\
E &\rightarrow E + T \\
E &\rightarrow T \\
T &\rightarrow T * F \\
T &\rightarrow F \\
F &\rightarrow (E) \\
F &\rightarrow a
\end{aligned}$$

Figure 1: Example Grammar

- **shift**  $\in f_q(u)$ , if  $[A \rightarrow \beta_1 \cdot a\beta_2, v] \in q$  and  $u \in (\text{FIRST}_k(a\beta_2v) \oplus_k \$^k)$
- **reduce**  $i \in f_q(u)$ , if  $[A \rightarrow \beta \cdot, u] \in q$  and  $A \rightarrow \beta$  is the  $i^{\text{th}}$  production in  $P$ .

Note that the canonical LR( $k$ ) parser for an LR( $k$ ) grammar is deterministic but, in general, a canonical LR( $k$ ) parser is nondeterministic. Of course, if the theory is restricted to LR( $k$ ) grammars, all the results from Aho and Ullman [1] apply to LR( $k$ ) parsers as they are defined here.

To illustrate the construction of the canonical collection of sets of LR(1) items and parsing functions for an LR(1) grammar, the grammar in Figure 1 is used. The canonical collection of sets of LR(1) items and the GOTO function for the grammar are given in Figure 2 and Figure 3.

Each set of items in the canonical collection of sets of items corresponds to a state of the canonical LR(1) parser for the grammar. Except for the initial and final state, each state is labeled with the unique symbol that precedes the “.” in the set of items for the state. These symbols are also subscripted in order to provide a unique identifier for each state.

To illustrate the actions of the LR(1) parser,

$$a * (a + a * a)$$

is used as the input string. The rightmost derivation of this input string is given in Figure 4. The configurations of the parser as it parses the input strings are given in Figure 5.

0:	$[S' \rightarrow \cdot E \$, \$]$ $[E \rightarrow \cdot E + T, \$]$ $[E \rightarrow \cdot T, \$]$ $[E \rightarrow \cdot E + T, +]$ $[E \rightarrow \cdot T, +]$ $[T \rightarrow \cdot T * F, \$]$ $[T \rightarrow \cdot F, \$]$ $[T \rightarrow \cdot T * F, +]$ $[T \rightarrow \cdot F, +]$ $[T \rightarrow \cdot T * F, *]$ $[T \rightarrow \cdot F, *]$ $[F \rightarrow \cdot (E), \$]$ $[F \rightarrow \cdot a, \$]$ $[F \rightarrow \cdot (E), +]$ $[F \rightarrow \cdot a, +]$ $[F \rightarrow \cdot (E), *]$ $[F \rightarrow \cdot a, *]$	( <sub>1</sub> :	$[F \rightarrow (\cdot E), \$]$ $[F \rightarrow (\cdot E), +]$ $[F \rightarrow (\cdot E), *]$ $[E \rightarrow \cdot E + T, )]$ $[E \rightarrow \cdot T, )]$ $[E \rightarrow \cdot E + T, +]$ $[E \rightarrow \cdot T, +]$ $[T \rightarrow \cdot T * F, )]$ $[T \rightarrow \cdot F, )]$ $[T \rightarrow \cdot T * F, +]$ $[T \rightarrow \cdot F, +]$ $[T \rightarrow \cdot T * F, *]$ $[T \rightarrow \cdot F, *]$ $[F \rightarrow \cdot (E), )]$ $[F \rightarrow \cdot a, )]$ $[F \rightarrow \cdot (E), +]$ $[F \rightarrow \cdot a, +]$ $[F \rightarrow \cdot (E), *]$ $[F \rightarrow \cdot a, *]$	( <sub>2</sub> :	$[F \rightarrow (\cdot E), )]$ $[F \rightarrow (\cdot E), +]$ $[F \rightarrow (\cdot E), *]$ $[E \rightarrow \cdot E + T, )]$ $[E \rightarrow \cdot T, )]$ $[E \rightarrow \cdot E + T, +]$ $[E \rightarrow \cdot T, +]$ $[T \rightarrow \cdot T * F, )]$ $[T \rightarrow \cdot F, )]$ $[T \rightarrow \cdot T * F, +]$ $[T \rightarrow \cdot F, +]$ $[T \rightarrow \cdot T * F, *]$ $[T \rightarrow \cdot F, *]$ $[F \rightarrow \cdot (E), )]$ $[F \rightarrow \cdot a, )]$ $[F \rightarrow \cdot (E), +]$ $[F \rightarrow \cdot a, +]$ $[F \rightarrow \cdot (E), *]$ $[F \rightarrow \cdot a, *]$
$a_1$ :	$[F \rightarrow a \cdot, \$]$ $[F \rightarrow a \cdot, +]$ $[F \rightarrow a \cdot, *]$	$*_1$ :	$[T \rightarrow T * \cdot F, \$]$ $[T \rightarrow T * \cdot F, +]$ $[T \rightarrow T * \cdot F, *]$ $[F \rightarrow \cdot (E), \$]$ $[F \rightarrow \cdot a, \$]$ $[F \rightarrow \cdot (E), +]$ $[F \rightarrow \cdot a, +]$ $[F \rightarrow \cdot (E), *]$ $[F \rightarrow \cdot a, *]$	$*_2$ :	$[T \rightarrow T * \cdot F, )]$ $[T \rightarrow T * \cdot F, +]$ $[T \rightarrow T * \cdot F, *]$ $[F \rightarrow \cdot (E), )]$ $[F \rightarrow \cdot a, )]$ $[F \rightarrow \cdot (E), +]$ $[F \rightarrow \cdot a, +]$ $[F \rightarrow \cdot (E), *]$ $[F \rightarrow \cdot a, *]$
$a_2$ :	$[F \rightarrow a \cdot, )]$ $[F \rightarrow a \cdot, +]$ $[F \rightarrow a \cdot, *]$				
$)_1$ :	$[F \rightarrow (E) \cdot, \$]$ $[F \rightarrow (E) \cdot, +]$ $[F \rightarrow (E) \cdot, *]$				
$)_2$ :	$[F \rightarrow (E) \cdot, )]$ $[F \rightarrow (E) \cdot, +]$ $[F \rightarrow (E) \cdot, *]$	$F_1$ :	$[T \rightarrow F \cdot, \$]$ $[T \rightarrow F \cdot, +]$ $[T \rightarrow F \cdot, *]$	$F_2$ :	$[T \rightarrow F \cdot, )]$ $[T \rightarrow F \cdot, +]$ $[T \rightarrow F \cdot, *]$

Figure 2: Sets of Valid LR( $k$ ) Items for Example Grammar

$F_3:$	$[T \rightarrow T * F., \$]$ $[T \rightarrow T * F., +]$ $[T \rightarrow T * F., *]$	$+_1:$	$[E \rightarrow E + .T, \$]$ $[E \rightarrow E + .T, +]$ $[T \rightarrow .T * F, \$]$ $[T \rightarrow .F, \$]$ $[T \rightarrow .T * F, +]$ $[T \rightarrow .F, +]$ $[T \rightarrow .T * F, *]$ $[T \rightarrow .F, *]$	$+_2:$	$[E \rightarrow E + .T, )]$ $[E \rightarrow E + .T, +]$ $[T \rightarrow .T * F, )]$ $[T \rightarrow .F, )]$ $[T \rightarrow .T * F, +]$ $[T \rightarrow .F, +]$ $[T \rightarrow .T * F, *]$ $[T \rightarrow .F, *]$
$F_4:$	$[T \rightarrow T * F., )]$ $[T \rightarrow T * F., +]$ $[T \rightarrow T * F., *]$				
$T_1:$	$[E \rightarrow T., \$]$ $[E \rightarrow T., +]$ $[T \rightarrow T . * F, \$]$ $[T \rightarrow T . * F, +]$ $[T \rightarrow T . * F, *]$		$[F \rightarrow .(E), \$]$ $[F \rightarrow .a, \$]$ $[F \rightarrow .(E), +]$ $[F \rightarrow .a, +]$ $[F \rightarrow .(E), *]$ $[F \rightarrow .a, *]$		$[F \rightarrow .(E), )]$ $[F \rightarrow .a, )]$ $[F \rightarrow .(E), +]$ $[F \rightarrow .a, +]$ $[F \rightarrow .(E), *]$ $[F \rightarrow .a, *]$
$T_2:$	$[E \rightarrow T., )]$ $[E \rightarrow T., +]$ $[T \rightarrow T . * F, )]$ $[T \rightarrow T . * F, +]$ $[T \rightarrow T . * F, *]$	$E_1:$	$[F \rightarrow (E.), \$]$ $[F \rightarrow (E.), +]$ $[F \rightarrow (E.), *]$ $[E \rightarrow .E + T, )]$ $[E \rightarrow .E + T, +]$	$f:$	$[S' \rightarrow E \$., \$]$
$T_3:$	$[E \rightarrow E + T., \$]$ $[E \rightarrow E + T., +]$ $[T \rightarrow T . * F, \$]$ $[T \rightarrow T . * F, +]$ $[T \rightarrow T . * F, *]$	$E_2:$	$[F \rightarrow (E.), )]$ $[F \rightarrow (E.), +]$ $[F \rightarrow (E.), *]$ $[E \rightarrow .E + T, )]$ $[E \rightarrow .E + T, +]$		
$T_4:$	$[E \rightarrow E + T., )]$ $[E \rightarrow E + T., +]$ $[T \rightarrow T . * F, )]$ $[T \rightarrow T . * F, +]$ $[T \rightarrow T . * F, *]$	$E_3:$	$[S' \rightarrow E . \$, \$]$ $[E \rightarrow E . + T, \$]$ $[E \rightarrow E . + T, +]$		

Figure 2: continued



GOTO	$a$	(	)	*	+	$F$	$T$	$E$	\$
0	$a_1$	( <sub>1</sub>				$F_1$	$T_1$	$E_3$	
$a_1$									
$a_2$									
( <sub>1</sub>	$a_2$	( <sub>2</sub>				$F_2$	$T_2$	$E_1$	
( <sub>2</sub>	$a_2$	( <sub>2</sub>				$F_2$	$T_2$	$E_2$	
) <sub>1</sub>									
) <sub>2</sub>									
* <sub>1</sub>	$a_1$	( <sub>1</sub>				$F_3$			
* <sub>2</sub>	$a_2$	( <sub>2</sub>				$F_4$			
+ <sub>1</sub>	$a_1$	( <sub>1</sub>				$F_1$	$T_3$		
+ <sub>2</sub>	$a_2$	( <sub>2</sub>				$F_4$	$T_4$		
$F_1$									
$F_2$									
$F_3$									
$F_4$									
$T_1$				* <sub>1</sub>					
$T_2$				* <sub>2</sub>					
$T_3$				* <sub>1</sub>					
$T_4$				* <sub>2</sub>					
$E_1$			) <sub>1</sub>		+ <sub>2</sub>				
$E_2$			) <sub>2</sub>		+ <sub>2</sub>				
$E_3$					+ <sub>1</sub>				$f$

Figure 3: GOTO Function for the Example Grammar

$$\begin{aligned}
E &\Rightarrow T \\
&\Rightarrow T * F \\
&\Rightarrow T * (E) \\
&\Rightarrow T * (E + T) \\
&\Rightarrow T * (E + T * F) \\
&\Rightarrow T * (E + T * a) \\
&\Rightarrow T * (E + F * a) \\
&\Rightarrow T * (E + a * a) \\
&\Rightarrow T * (T + a * a) \\
&\Rightarrow T * (F + a * a) \\
&\Rightarrow T * (a + a * a) \\
&\Rightarrow F * (a + a * a) \\
&\Rightarrow a * (a + a * a)
\end{aligned}$$

Figure 4: Derivation of Input String

0	$a * (a + a * a) \$$
$0a_1$	$*(a + a * a) \$$
$0F_1$	$*(a + a * a) \$$
$0T_1$	$*(a + a * a) \$$
$0T_1 *_1$	$(a + a * a) \$$
$0T_1 *_1 ($	$a + a * a) \$$
$0T_1 *_1 ( _1 a_1$	$+ a * a) \$$
$0T_1 *_1 ( _1 F_1$	$+ a * a) \$$
$0T_1 *_1 ( _1 T_1$	$+ a * a) \$$
$0T_1 *_1 ( _1 E_1$	$+ a * a) \$$
$0T_1 *_1 ( _1 E_1 +_2$	$a * a) \$$
$0T_1 *_1 ( _1 E_1 +_2 a_2$	$* a) \$$
$0T_1 *_1 ( _1 E_1 +_2 F_2$	$* a) \$$
$0T_1 *_1 ( _1 E_1 +_2 T_4$	$* a) \$$
$0T_1 *_1 ( _1 E_1 +_2 T_4 * _2$	$a) \$$
$0T_1 *_1 ( _1 E_1 +_2 T_4 * _2 a_2$	$) \$$
$0T_1 *_1 ( _1 E_1 +_2 T_4 * _2 F_4$	$) \$$
$0T_1 *_1 ( _1 E_1 +_2 T_4$	$) \$$
$0T_1 *_1 ( _1 E_1$	$) \$$
$0T_1 *_1 ( _1 E_1 ) _1$	$\$$
$0T_1 * _1 F_3$	$\$$
$0T_1$	$\$$
$0E_3$	$\$$
$0E_3 f$	

Figure 5: Configurations for the Input String

### 2.3.3 The Explicitly Advancing LR( $k$ ) Parser

This section presents a modification of the standard LR( $k$ ) parser that emphasizes the role of lookahead in the parser. The modified LR( $k$ ) parser is called the explicitly advancing LR( $k$ ) parser and is Algorithm 2.3.2. The explicitly advancing LR( $k$ ) parser is important to the development of the Least-Cost LR( $k$ ) Early's Algorithm in Chapter 5.

The explicitly advancing LR( $k$ ) parser is best described by the difference between its configurations and those of the standard LR( $k$ ) parser. For the standard LR( $k$ ) parser, a configuration of the parser is written as  $(\alpha, w_{i:n})$ , where  $\alpha$  is the stack and  $w_{i:n}$  is the remaining input. Implicit in this configuration is the fact that  $w_{i,i+k-1}$  is the  $k$  symbols of lookahead used by the LR( $k$ ) parser. For the explicitly advancing LR( $k$ ) parser, the equivalent configuration would be written as  $(\alpha, u, w_{i+k:n})$ , where  $\alpha$  is the stack,  $u$  is the lookahead, and  $w_{i+k:n}$  is the remaining input which has not been scanned.

Since the lookahead component of the input is made explicit in the explicitly advancing LR( $k$ ) parser, there must a parsing action which causes the parser to scan an input symbol and add it to the lookahead. This parsing action is called an *advance* and the corresponding move is written as  $(\alpha, u, aw) \vdash (\alpha, ua, w)$ . When a move is required to be an advance, it is written as  $(\alpha, u, x) \stackrel{a}{\vdash} (\alpha, v, y)$ . When a sequence of zero or more moves are required to be advances, they are written as  $(\alpha, u, x) \stackrel{a^*}{\vdash} (\alpha, v, y)$ .

The explicitly advancing LR( $k$ ) parser uses the same set of states,  $Q$ , and the same goto function,  $g_q$ , as the standard LR( $k$ ) parser. It also accepts its input string by halting in the distinguished final state  $f$  just like the standard LR( $k$ ) parser. However, the parsing action function,  $f_q$ , is extended so that it maps strings from  $\bigcup_{i=0}^k (\Sigma \cup \$)^i$  instead of just from  $(\Sigma \cup \$)^k$ . This is achieved by defining

$$f_q(u) = \{\text{advance}\} \text{ for any } q, \text{ expect } f, \text{ and all } u \in \bigcup_{i=0}^{k-1} (\Sigma \cup \$)^i.$$

With the addition of advances, the same notation for describing a sequence of moves is used for the explicitly advancing LR( $k$ ) parser as is used for the LR( $k$ ) parser.

It is obvious that the explicitly advancing LR( $k$ ) parser and the LR( $k$ ) parser for a grammar accept the same strings since for any sequence of moves in one there is a corresponding sequence of moves in the other.

### ALGORITHM 2.3.2 *The Explicitly Advancing LR(k) Parser*

Let  $G$  be a reduced CFG  $(N, \Sigma, P, S)$  and  $Q$  be a set of states as described previously. The input for all copies of the algorithm is  $z \in \Sigma^*$ . The output from a copy of the algorithm is a sequence of production numbers which represents a right parse of  $z$  for  $G$ , if the algorithm halts in the final state  $f$ .

Three variables are used:  $i$  keeps track of the current position in the input string;  $q$  is the current state; and  $w$  stores the  $k$  symbol lookahead string. The algorithm proceeds as follows:

- I. Create the initial copy of the algorithm; set  $i = 0$ ; set  $q = 0$ ; initialize the stack to 0; let  $w = \epsilon$ ; and append  $\$^k$  to  $z$ .
- II. Repeat this step as long as  $f_q(w)$  contains at least one element. If  $f_q(w)$  contains more than one element, create a new copy of the current algorithm for each of the additional elements. Then, for the element for which this copy of the algorithm was created, perform whichever of the following three cases applies.
  - A. If **advance**  $\in f_q(w)$  then let  $i = i + 1$ ,  $w = wz_{i,i}$ .
  - B. If **shift**  $\in f_q(w)$  then, if  $g_q(w)$  is empty, halt. Otherwise let  $q = g_q(w_{1,1})$ , push  $q$  onto the stack, and let  $w = w_{2,k}$ .
  - C. If **reduce**  $p \in f_q(w)$  then let  $m = |\text{RHS}(p)|$ ,  $l$  = the number of states on the stack, and perform whichever of the following two cases applies for  $m$ :
    1. If  $m > l$  then pop  $m$  states from the stack and let  $r$  be the top state left on the stack. Then, if  $g_r(\text{LHS}(p))$  is empty, halt; otherwise, let  $q = g_r(\text{LHS}(p))$  and push  $q$  onto the stack.
    2. If  $m \leq l$  then pop all the states from the stack until 0 is the only state left on the stack. Then, if  $g_0(\text{LHS}(p))$  is empty, halt; otherwise, let  $q = g_0(\text{LHS}(p))$  and push  $q$  onto the stack.

## 2.4 EARLY'S ALGORITHM

This section presents Early's algorithm (Algorithm 2.4.1) in order to provide background and motivation for the development of the LR( $k$ ) Early's Algorithm. The form of the algorithm presented here is that of Early [11], but the notation and method used are from Aho and Ullman [1].

Early's algorithm solves the problem of recognizing strings in the language generated by an arbitrary CFG. It does so by pursuing possible derivations of the string as it proceeds through a left to right scan of the string. All possible derivations of the string from the start symbol are not pursued because some grammars produce an infinite number of derivations for a particular string. The multiple derivations are kept track of on  $n + 2$  parse lists  $I_0, I_1, \dots, I_n, I_{n+1}$  where  $n$  is the length of the string  $w$  to be parsed. Parse list  $I_{j-1}$  is the state of the parse before  $w_{j,j}$  is parsed and the parse list  $I_j$  is the state of the parse after  $w_{j,j}$  is parsed. Parse list  $I_{n+1}$  is needed because Early's algorithm also 'scans' the end of string symbol  $\$$  in order to simplify its test for successful termination.

Each parse list  $I_j$  is a set of entries of the form  $[i, A \rightarrow \alpha \cdot \beta, u]$ , where  $i$  is the number of a parse list,  $A \rightarrow \alpha\beta \in P$ , and  $u \in (\Sigma \cup \$)^k$ . The second and third components of an entry form an LR( $k$ ) item. This is not accidental and is a result of the approach Early [11] used to develop the algorithm.

An entry on a parse list is used to track the attempted derivation of a substring of the input string. The critical property of entries is that an entry  $[i, A \rightarrow \alpha \cdot \beta, u]$  is on a parse list  $I_j$  if and only if  $S \Rightarrow^* \gamma A \delta$ ,  $\gamma \Rightarrow^* w_{1:j}$ , and  $\alpha \Rightarrow^* w_{j+1:j}$ . This property can be viewed as the loop invariant that Algorithm 2.4.1 is designed to maintain.

Algorithm 2.4.1 starts by placing the entry  $[0, S' \rightarrow \cdot S \$, \$^k]$  on  $I_0$ . The algorithm terminates when all the parse lists have been processed. The input string is in the language generated by  $G$  if and only if  $[0, S' \rightarrow S \$ \cdot, \$^k]$  is on  $I_{n+1}$  (this is the only entry that can possibly be on  $I_{n+1}$ ).

To illustrate the actions of Early's algorithm, the grammar from Figure 1 and the input string

$$a * (a + a * a)$$

are used. Figure 6 shows the parse lists calculated by Early's algorithm.

ALGORITHM 2.4.1 *Early's Algorithm*

- I. Let  $n = |w|$ ,  $j = 0$ ,  $w_{n+1:n+k+1} = \$^{k+1}$ , and place  $[0, S' \rightarrow \cdot S \$, \$^k]$  on  $I_0$ .
- II. While  $j \leq n$ , perform the following steps:
  - A. Perform the following steps until no new entries are added to  $I_j$ :
    1. Let  $[i, A \rightarrow \alpha \cdot B \beta, u]$  be an entry on  $I_j$ . For each  $B \rightarrow \gamma$  in  $P$  and each  $v \in FIRST_k(\beta u)$ , add the entry  $[j, B \rightarrow \cdot \gamma, v]$  to  $I_j$ , if it is not already on the parse list.
    2. Let  $[i, A \rightarrow \gamma \cdot, u]$  be an entry on  $I_j$  with  $u = w_{j+1:j+k}$ . For each entry  $[l, B \rightarrow \alpha \cdot A \beta, v]$  on a parse list  $I_l$ , add the entry  $[l, B \rightarrow \alpha A \cdot \beta, v]$  to  $I_j$ , if it is not already on the parse list.
  - B. For each  $[i, A \rightarrow \alpha \cdot a \beta, u]$  on  $I_j$  such that  $a = w_{j+1:j+1}$ , add the entry  $[i, A \rightarrow \alpha a \cdot \beta, u]$  to  $I_{j+1}$ , if it is not already on the parse list.
  - C. Let  $j = j + 1$ .

$I_0$	$I_1$
$[0, S' \rightarrow \cdot E \$, \$]$	$[0, F \rightarrow a \cdot, \$]$
$[0, E \rightarrow \cdot E + T, \$]$	$[0, F \rightarrow a \cdot, +]$
$[0, E \rightarrow \cdot T, \$]$	$[0, F \rightarrow a \cdot, *]$
$[0, E \rightarrow \cdot E + T, +]$	$[0, T \rightarrow F \cdot, \$]$
$[0, E \rightarrow \cdot T, +]$	$[0, T \rightarrow F \cdot, +]$
$[0, T \rightarrow \cdot T * F, \$]$	$[0, T \rightarrow F \cdot, *]$
$[0, T \rightarrow \cdot F, \$]$	$[0, E \rightarrow T \cdot, \$]$
$[0, T \rightarrow \cdot T * F, +]$	$[0, E \rightarrow T \cdot, +]$
$[0, T \rightarrow \cdot F, +]$	$[0, T \rightarrow T \cdot * F, \$]$
$[0, T \rightarrow \cdot T * F, *]$	$[0, T \rightarrow T \cdot * F, +]$
$[0, T \rightarrow \cdot F, *]$	$[0, T \rightarrow T \cdot * F, *]$
$[0, F \rightarrow \cdot (E), \$]$	
$[0, F \rightarrow \cdot a, \$]$	
$[0, F \rightarrow \cdot (E), +]$	
$[0, F \rightarrow \cdot a, +]$	
$[0, F \rightarrow \cdot (E), *]$	
$[0, F \rightarrow \cdot a, *]$	

Figure 6: Parse List for Early's Algorithm



$I_2$	$I_3$
$[0, T \rightarrow T * \cdot F, \$]$	$[2, F \rightarrow (\cdot E), \$]$
$[0, T \rightarrow T * \cdot F, +]$	$[2, F \rightarrow (\cdot E), +]$
$[0, T \rightarrow T * \cdot F, *]$	$[2, F \rightarrow (\cdot E), *]$
$[2, F \rightarrow \cdot (E), \$]$	$[3, E \rightarrow \cdot E + T, )]$
$[2, F \rightarrow \cdot a, \$]$	$[3, E \rightarrow \cdot T, )]$
$[2, F \rightarrow \cdot (E), +]$	$[3, E \rightarrow \cdot E + T, +]$
$[2, F \rightarrow \cdot a, +]$	$[3, E \rightarrow \cdot T, +]$
$[2, F \rightarrow \cdot (E), *]$	$[3, T \rightarrow \cdot T * F, )]$
$[2, F \rightarrow \cdot a, *]$	$[3, T \rightarrow \cdot F, )]$
	$[3, T \rightarrow \cdot T * F, +]$
	$[3, T \rightarrow \cdot F, +]$
	$[3, T \rightarrow \cdot T * F, *]$
	$[3, T \rightarrow \cdot F, *]$
	$[3, F \rightarrow \cdot (E), )]$
	$[3, F \rightarrow \cdot a, )]$
	$[3, F \rightarrow \cdot (E), +]$
	$[3, F \rightarrow \cdot a, +]$
	$[3, F \rightarrow \cdot (E), *]$
	$[3, F \rightarrow \cdot a, *]$

$I_4$	$I_5$
$[3, F \rightarrow a \cdot, \$]$	$[3, E \rightarrow E + \cdot T, )]$
$[3, F \rightarrow a \cdot, +]$	$[3, E \rightarrow E + \cdot T, +]$
$[3, F \rightarrow a \cdot, *]$	$[5, T \rightarrow \cdot T * F, )]$
$[3, T \rightarrow F \cdot, \$]$	$[5, T \rightarrow \cdot F, )]$
$[3, T \rightarrow F \cdot, +]$	$[5, T \rightarrow \cdot T * F, +]$
$[3, T \rightarrow F \cdot, *]$	$[5, T \rightarrow \cdot F, +]$
$[3, E \rightarrow T \cdot, \$]$	$[5, T \rightarrow \cdot T * F, *]$
$[3, E \rightarrow T \cdot, +]$	$[5, T \rightarrow \cdot F, *]$
$[3, T \rightarrow T \cdot * F, \$]$	$[5, F \rightarrow \cdot (E), )]$
$[3, T \rightarrow T \cdot * F, +]$	$[5, F \rightarrow \cdot a, )]$
$[3, T \rightarrow T \cdot * F, *]$	$[5, F \rightarrow \cdot (E), +]$
$[2, F \rightarrow (E \cdot), \$]$	$[5, F \rightarrow \cdot a, +]$
$[2, F \rightarrow (E \cdot), +]$	$[5, F \rightarrow \cdot (E), *]$
$[2, F \rightarrow (E \cdot), *]$	$[5, F \rightarrow \cdot a, *]$
$[3, E \rightarrow E \cdot + T, )]$	
$[3, E \rightarrow E \cdot + T, +]$	

$I_6$	$I_7$
$[5, F \rightarrow a \cdot, )]$	$[5, T \rightarrow T * \cdot F, )]$
$[5, F \rightarrow a \cdot, +]$	$[5, T \rightarrow T * \cdot F, +]$
$[5, F \rightarrow a \cdot, *]$	$[5, T \rightarrow T * \cdot F, *]$
$[5, T \rightarrow F \cdot, )]$	$[7, F \rightarrow \cdot (E), )]$
$[5, T \rightarrow F \cdot, +]$	$[7, F \rightarrow \cdot a, )]$
$[5, T \rightarrow F \cdot, *]$	$[7, F \rightarrow \cdot (E), +]$
$[3, E \rightarrow E + T \cdot, )]$	$[7, F \rightarrow \cdot a, +]$
$[3, E \rightarrow E + T \cdot, +]$	$[7, F \rightarrow \cdot (E), *]$
$[5, T \rightarrow T \cdot * F, )]$	$[7, F \rightarrow \cdot a, *]$
$[5, T \rightarrow T \cdot * F, +]$	
$[5, T \rightarrow T \cdot * F, *]$	

$I_8$	$I_9$
$[7, F \rightarrow a\cdot, )]$	$[2, F \rightarrow (E)\cdot, \$]$
$[7, F \rightarrow a\cdot, +]$	$[2, F \rightarrow (E)\cdot, +]$
$[7, F \rightarrow a\cdot, *]$	$[2, F \rightarrow (E)\cdot, *]$
$[5, T \rightarrow T * F\cdot, )]$	$[0, T \rightarrow T * F\cdot, \$]$
$[5, T \rightarrow T * F\cdot, +]$	$[0, T \rightarrow T * F\cdot, +]$
$[5, T \rightarrow T * F\cdot, *]$	$[0, T \rightarrow T * F\cdot, *]$
$[3, E \rightarrow E + T\cdot, )]$	$[0, E \rightarrow T\cdot, \$]$
$[3, E \rightarrow E + T\cdot, +]$	$[0, E \rightarrow T\cdot, +]$
$[3, T \rightarrow T \cdot * F, )]$	$[0, T \rightarrow T \cdot * F, \$]$
$[3, T \rightarrow T \cdot * F, +]$	$[0, T \rightarrow T \cdot * F, +]$
$[3, T \rightarrow T \cdot * F, *]$	$[0, T \rightarrow T \cdot * F, *]$
$[2, F \rightarrow (E\cdot), \$]$	$[0, S' \rightarrow E \cdot \$, \$]$
$[2, F \rightarrow (E\cdot), +]$	$[0, E \rightarrow E \cdot + T, \$]$
$[2, F \rightarrow (E\cdot), *]$	$[0, E \rightarrow E \cdot + T, +]$
$[3, E \rightarrow \cdot E + T, )]$	
$[3, E \rightarrow \cdot E + T, +]$	

$$I_{10}$$

$$[0, S' \rightarrow E\$ \cdot, \$]$$

## 2.5 LYON'S ALGORITHM

Early's algorithm can be extended to perform globally least-cost syntax-error recovery. One such extension is Lyon's algorithm [19]. Algorithm 2.5 and Algorithm 2.5.2 are Lyon's algorithm but its form has been changed from Lyon's presentation so that it more closely resembles the form of Early's algorithm presented in this dissertation.

Figure 7 gives the parse lists produced by Lyon's algorithm for the input string  $a * a\$$  when no lookahead is used and the cost of all edit operations is one. The lookahead component of the entries is replaced with a cost variable. This variable contains the cost of the edits of the input string that lead to the addition of the entry to its parse list. It is important to note that an entry with a lower cost replaces a similar entry with a higher cost when the entry is added to a parse list.

In Figure 7, a double bar is used to separate the entries that would be present if Early's algorithm were used. The additional entries are added by Lyon's algorithm. The large increase in the number of entries, for a string with no syntax errors, illustrates the performance problems caused by the breadth-first nature of Early's algorithm when it is used for globally least-cost error recovery.

ALGORITHM 2.5.1 *Lyon's Algorithm*

I. Let  $n = |w|$ ,  $j = 0$ ,  $w_{n+1} \cdot n+k+1 = \$^{k+1}$ , and place  $[0, S' \rightarrow \cdot S \$, 0]$  on  $I_0$ .

II. While  $j \leq n$ , perform the following steps:

A. While there are unscanned entries of the form  $[l, A \rightarrow \alpha \cdot \beta, c]$  on  $I_j$ , where  $l \neq j$ , scan such an entry by applying Algorithm 2.5.2 to  $([l, A \rightarrow \alpha \cdot \beta, c], j)$ .

B. Let  $l = j - 1$ .

C. While  $l \geq 0$  perform the following steps:

1. While there are unstable entries of the form  $[l, A \rightarrow \alpha \cdot, c]$  on  $I_j$ , stabilize a least-cost, unstable entry  $[l, A \rightarrow \alpha \cdot, c]$  using the following steps:

i. For each entry  $[i, B \rightarrow \delta \cdot A\sigma, d]$  on  $I_l$ , add the entry  $[i, B \rightarrow \delta A\sigma \cdot, c + d + \min(\{W(T) \mid \epsilon \xrightarrow{T} \sigma\})]$  to  $I_j$ .

ii. While there are unscanned entries of the form  $[l, A \rightarrow \alpha \cdot \beta, c]$  on  $I_j$ , where  $l \neq j$ , scan such an entry by applying Algorithm 2.5.2 to  $([l, A \rightarrow \alpha \cdot \beta, c], j)$ .

2. For each  $A \in N$  find a least-cost entry of the form  $[l, A \rightarrow \alpha \cdot, c]$  on  $I_j$  and perform the following steps:

i. For each entry  $[i, B \rightarrow \delta \cdot A\sigma, d]$  on  $I_l$ , add  $[i, B \rightarrow \delta A \cdot \sigma, c + d]$  to  $I_j$ .

ii. While there are unscanned entries of the form  $[l, A \rightarrow \alpha \cdot \beta, c]$  on  $I_j$ , where  $l \neq j$ , scan such an entry by applying Algorithm 2.5.2 to  $([l, A \rightarrow \alpha \cdot \beta, c], j)$ .

3. Let  $l = l - 1$ .

D. For each entry  $[l, A \rightarrow \alpha \cdot B\beta, c]$  on  $I_j$  and each  $B \rightarrow \delta \in P$ , add the entry  $[j, B \rightarrow \cdot \delta, 0]$  to  $I_j$ .

E. While there are unscanned entries of the form  $[j, A \rightarrow \alpha \cdot \beta, c]$  on  $I_j$ , perform the following steps for such an entry:

1. Apply Algorithm 2.5.2 to  $([j, A \rightarrow \alpha \cdot \beta, c], j)$ .

2. If  $\beta = B\delta$ , for each  $B \rightarrow \gamma \in P$ , add  $[j, B \rightarrow \cdot \gamma, 0]$  to  $I_j$ .

F. Let  $j = j + 1$ .

ALGORITHM 2.5.2 *The Scanner for Lyon's Algorithm*

Scan  $([l, A \rightarrow \alpha \cdot \beta, c], j)$  by applying the following steps to it:

- I. If  $\beta = \epsilon$ ,  $w_{j+1:j+1} \neq \$$  and  $d = W((w_{j+1:j+1} \mapsto \epsilon))$  then add the entry  $[l, A \rightarrow \alpha \cdot, c + d]$  to  $I_{j+1}$ .
- II. If  $\beta = b\delta$  and  $b = w_{j+1:j+1}$  then add the entry  $[l, A \rightarrow \alpha b \cdot \delta, c]$  to  $I_{j+1}$ .
- III. If  $\beta = b\delta$ ,  $b \neq w_{j+1:j+1}$ ,  $w_{j+1:j+1} \neq \$$ ,  $b \neq \$$  and  $d = W((w_{j+1:j+1} \mapsto b))$  then add the entry  $[l, A \rightarrow \alpha b \cdot \delta, c + d]$  to  $I_{j+1}$ .
- IV. If  $\beta = X\delta$ ,  $w_{j+1:j+1} \neq \$$ ,  $d = W((w_{j+1:j+1} \mapsto \epsilon))$  and for all  $y \in \Sigma^*$  there does not exist a derivation  $X \xRightarrow{*} w_{j+1:j+1}y$  then add the entry  $[l, A \rightarrow \alpha \cdot X\delta, c + d]$  to  $I_{j+1}$ .
- V. If  $\beta = X\delta$ ,  $d = \min(\{W(T) \mid X \xRightarrow{*} x \text{ and } \epsilon \xrightarrow{T} x\})$  and for all  $y \in \Sigma^*$  there does not exist a derivation  $X \xRightarrow{*} w_{j+1:j+1}y$  then add the entry  $[l, A \rightarrow \alpha X \cdot \delta, c + d]$  to  $I_j$ .

$I_0$	$I_1$	$I_2$
$[0, S' \rightarrow \cdot E \$, 0]$	$[0, F \rightarrow a \cdot, 0]$	$[0, T \rightarrow T * \cdot F, 0]$
$[0, E \rightarrow \cdot E + T, 0]$	$[0, T \rightarrow F \cdot, 0]$	$[2, F \rightarrow \cdot (E), 0]$
$[0, E \rightarrow \cdot T, 0]$	$[0, E \rightarrow T \cdot, 0]$	$[2, F \rightarrow \cdot a, 0]$
$[0, T \rightarrow \cdot T * F, 0]$	$[0, T \rightarrow T \cdot * F, 0]$	$[0, E \rightarrow E \cdot + T, 1]$
$[0, T \rightarrow \cdot F, 0]$	$[0, S' \rightarrow E \cdot \$, 0]$	$[0, E \rightarrow E + \cdot T, 1]$
$[0, F \rightarrow \cdot (E), 0]$	$[0, E \rightarrow E \cdot + T, 0]$	$[0, E \rightarrow E + T \cdot, 2]$
$[0, F \rightarrow \cdot a, 0]$	$[0, F \rightarrow \cdot (E), 1]$	$[0, F \rightarrow \cdot (E), 2]$
$[0, F \rightarrow (\cdot E), 1]$	$[0, F \rightarrow (\cdot E), 1]$	$[0, F \rightarrow (\cdot E), 2]$
	$[0, T \rightarrow T * F \cdot, 2]$	$[0, F \rightarrow (E \cdot), 2]$
	$[0, E \rightarrow E + T \cdot, 2]$	$[0, F \rightarrow (E) \cdot, 3]$
	$[0, F \rightarrow (E \cdot), 2]$	$[0, S' \rightarrow E \cdot \$, 1]$
	$[0, F \rightarrow (E) \cdot, 3]$	$[1, E \rightarrow \cdot E + T, 1]$
	$[1, E \rightarrow \cdot E + T, 0]$	$[1, E \rightarrow E \cdot + T, 1]$
	$[1, E \rightarrow E \cdot + T, 1]$	$[1, E \rightarrow E + \cdot T, 2]$
	$[1, E \rightarrow E + \cdot T, 2]$	$[1, E \rightarrow E + T \cdot, 3]$
	$[1, E \rightarrow E + T \cdot, 3]$	$[1, E \rightarrow \cdot T, 1]$
	$[1, E \rightarrow \cdot T, 0]$	$[1, E \rightarrow T \cdot, 1]$
	$[1, E \rightarrow T \cdot, 1]$	$[1, T \rightarrow \cdot T * F, 1]$
	$[1, T \rightarrow \cdot T * F, 0]$	$[1, T \rightarrow T \cdot * F, 1]$
	$[1, T \rightarrow T \cdot * F, 1]$	$[1, T \rightarrow \cdot F, 1]$
	$[1, T \rightarrow \cdot F, 0]$	$[1, T \rightarrow F \cdot, 1]$
	$[1, T \rightarrow F \cdot, 1]$	$[1, F \rightarrow \cdot (E), 1]$
	$[1, F \rightarrow \cdot (E), 0]$	$[1, F \rightarrow (\cdot E), 1]$
	$[1, F \rightarrow (\cdot E), 1]$	$[1, F \rightarrow (E \cdot), 2]$
	$[1, F \rightarrow (E \cdot), 2]$	$[1, F \rightarrow (E) \cdot, 3]$
	$[1, F \rightarrow (E) \cdot, 3]$	$[1, F \rightarrow \cdot a, 1]$
	$[1, F \rightarrow \cdot a, 0]$	$[1, F \rightarrow a \cdot, 1]$
	$[1, F \rightarrow a \cdot, 1]$	$[0, T \rightarrow T \cdot * F, 1]$
		$[2, F \rightarrow (\cdot E), 1]$
		$[2, E \rightarrow \cdot E + T, 0]$
		$[2, E \rightarrow \cdot T, 0]$
		$[2, T \rightarrow \cdot T * F, 0]$
		$[2, T \rightarrow \cdot F, 0]$

Figure 7: Parse Lists for Lyon's Algorithm

$I_3$	$I_4$
$[2, F \rightarrow a\cdot, 0]$	$[0, S' \rightarrow E\$, 1]$
$[2, T \rightarrow F\cdot, 0]$	
$[2, E \rightarrow T\cdot, 0]$	
$[2, T \rightarrow T \cdot * F, 0]$	
$[2, E \rightarrow E \cdot + T, 0]$	
$[0, T \rightarrow T * F\cdot, 0]$	
$[0, S' \rightarrow E \cdot \$, 0]$	
$[0, E \rightarrow E \cdot + T, 0]$	
<hr/>	<hr/>
$[0, F \rightarrow a\cdot, 2]$	$[2, F \rightarrow \cdot(E), 1]$
$[0, T \rightarrow F\cdot, 2]$	$[2, F \rightarrow (\cdot E), 1]$
$[0, E \rightarrow T\cdot, 2]$	$[2, F \rightarrow (E\cdot), 2]$
$[0, T \rightarrow T * F\cdot, 4]$	$[2, F \rightarrow (E)\cdot, 3]$
$[0, E \rightarrow E \cdot + T, 2]$	$[2, T \rightarrow T * F\cdot, 2]$
$[0, E \rightarrow E + \cdot T, 2]$	$[2, E \rightarrow E + T\cdot, 2]$
$[0, E \rightarrow E + T\cdot, 3]$	$[3, E \rightarrow \cdot E + T, 0]$
$[0, F \rightarrow \cdot(E), 3]$	$[3, E \rightarrow E \cdot + T, 1]$
$[0, F \rightarrow (\cdot E), 3]$	$[3, E \rightarrow E + \cdot T, 2]$
$[0, F \rightarrow (E\cdot), 3]$	$[3, E \rightarrow E + T\cdot, 3]$
$[0, F \rightarrow (E)\cdot, 3]$	$[3, E \rightarrow \cdot T, 0]$
$[0, S' \rightarrow E \cdot \$, 2]$	$[3, E \rightarrow T\cdot, 1]$
$[1, E \rightarrow E \cdot + T, 2]$	$[3, T \rightarrow \cdot T * F, 0]$
$[1, E \rightarrow E + \cdot T, 2]$	$[3, T \rightarrow T \cdot * F, 1]$
$[1, E \rightarrow E + T\cdot, 4]$	$[3, T \rightarrow T * \cdot F, 2]$
$[1, E \rightarrow T\cdot, 2]$	$[3, T \rightarrow T * F\cdot, 3]$
$[1, T \rightarrow T \cdot * F, 2]$	$[3, T \rightarrow \cdot F, 0]$
$[1, T \rightarrow T * \cdot F, 3]$	$[3, T \rightarrow F\cdot, 1]$
$[1, T \rightarrow F\cdot, 2]$	$[3, F \rightarrow \cdot(E), 0]$
$[1, F \rightarrow \cdot(E), 2]$	$[3, F \rightarrow (\cdot E), 1]$
$[1, F \rightarrow (\cdot E), 2]$	$[3, F \rightarrow (E\cdot), 2]$
$[1, F \rightarrow (E\cdot), 3]$	$[3, F \rightarrow (E)\cdot, 3]$
$[1, F \rightarrow (E)\cdot, 3]$	$[3, F \rightarrow \cdot a, 0]$
$[1, F \rightarrow a\cdot, 1]$	$[3, F \rightarrow a\cdot, 1]$

Figure 7: continued



## CHAPTER III

### THE LR( $k$ ) EARLY'S ALGORITHM

In this chapter, a new algorithm, the LR( $k$ ) Early's Algorithm, is presented. This algorithm is similar to Early's algorithm but, unlike Early's algorithm, it makes use of the states of an LR( $k$ ) parser. The LR( $k$ ) Early's Algorithm is shown to simulate the nondeterministic LR( $k$ ) parser. Also, the LR( $k$ ) Early's Algorithm is shown to have a time complexity of  $\mathcal{O}(n^4)$  in general, where  $n$  is the length of the input string.

#### 3.1 THE ALGORITHM

The idea behind the LR( $k$ ) Early's Algorithm can be seen in the illustrations of the LR(1) parser and Early's algorithm in Chapter 2. For these illustrations, the grammar in Figure 1 is used to compute the canonical collection of LR(1) items (i.e. the states of the LR(1) parser) given in Figure 2. This grammar is also used with the input string

$$a * (a + a * a)$$

to illustrate the configurations of the LR( $k$ ) parser in Figure 5 and the parse lists computed by Early's algorithm in Figure 6.

A strong relationship exists between the states and configurations of the LR( $k$ ) parser in Figures 2 and 5 and the contents of the parse lists in Figure 6. The entries on the parse lists can be grouped so that, looking at only their second and third components, they correspond to the LR(1) items in the states of the LR(1) parser. Figure 8 shows this correspondence. Additionally, the states which correspond to entries on a parse list also appear on the top of the stack in configurations of the LR(1) parser which correspond to the same position in the input string. Thus, it appears that it may be possible to use the states of an LR( $k$ ) parser in an algorithm similar to Early's algorithm.

$$\begin{array}{c}
I_0 \\
\left. \begin{array}{l}
[0, S' \rightarrow \cdot E \$, \$] \\
[0, E \rightarrow \cdot E + T, \$] \\
[0, E \rightarrow \cdot T, \$] \\
[0, E \rightarrow \cdot E + T, +] \\
[0, E \rightarrow \cdot T, +] \\
[0, T \rightarrow \cdot T * F, \$] \\
[0, T \rightarrow \cdot F, \$] \\
[0, T \rightarrow \cdot T * F, +] \\
[0, T \rightarrow \cdot F, +] \\
[0, T \rightarrow \cdot T * F, *] \\
[0, T \rightarrow \cdot F, *] \\
[0, F \rightarrow \cdot (E), \$] \\
[0, F \rightarrow \cdot a, \$] \\
[0, F \rightarrow \cdot (E), +] \\
[0, F \rightarrow \cdot a, +] \\
[0, F \rightarrow \cdot (E), *] \\
[0, F \rightarrow \cdot a, *]
\end{array} \right\} 0
\end{array}
\qquad
\begin{array}{c}
I_1 \\
\left. \begin{array}{l}
[0, F \rightarrow a \cdot, \$] \\
[0, F \rightarrow a \cdot, +] \\
[0, F \rightarrow a \cdot, *]
\end{array} \right\} a_1 \\
\left. \begin{array}{l}
[0, T \rightarrow F \cdot, \$] \\
[0, T \rightarrow F \cdot, +] \\
[0, T \rightarrow F \cdot, *]
\end{array} \right\} F_1 \\
\left. \begin{array}{l}
[0, E \rightarrow T \cdot, \$] \\
[0, E \rightarrow T \cdot, +] \\
[0, T \rightarrow T \cdot * F, \$] \\
[0, T \rightarrow T \cdot * F, +] \\
[0, T \rightarrow T \cdot * F, *]
\end{array} \right\} T_1
\end{array}$$

Figure 8: Parse Lists for Early's Algorithm

$$\begin{array}{c}
I_2 \\
\left. \begin{array}{l}
[0, T \rightarrow T * \cdot F, \$] \\
[0, T \rightarrow T * \cdot F, +] \\
[0, T \rightarrow T * \cdot F, *] \\
[2, F \rightarrow \cdot (E), \$] \\
[2, F \rightarrow \cdot a, \$] \\
[2, F \rightarrow \cdot (E), +] \\
[2, F \rightarrow \cdot a, +] \\
[2, F \rightarrow \cdot (E), *] \\
[2, F \rightarrow \cdot a, *]
\end{array} \right\} *_1
\end{array}
\quad
\begin{array}{c}
I_3 \\
\left. \begin{array}{l}
[2, F \rightarrow (\cdot E), \$] \\
[2, F \rightarrow (\cdot E), +] \\
[2, F \rightarrow (\cdot E), *] \\
[3, E \rightarrow \cdot E + T, )] \\
[3, E \rightarrow \cdot T, )] \\
[3, E \rightarrow \cdot E + T, +] \\
[3, E \rightarrow \cdot T, +] \\
[3, T \rightarrow \cdot T * F, )] \\
[3, T \rightarrow \cdot F, )] \\
[3, T \rightarrow \cdot T * F, +] \\
[3, T \rightarrow \cdot F, +] \\
[3, T \rightarrow \cdot T * F, *] \\
[3, T \rightarrow \cdot F, *] \\
[3, F \rightarrow \cdot (E), )] \\
[3, F \rightarrow \cdot a, )] \\
[3, F \rightarrow \cdot (E), +] \\
[3, F \rightarrow \cdot a, +] \\
[3, F \rightarrow \cdot (E), *] \\
[3, F \rightarrow \cdot a, *]
\end{array} \right\} ( _1
\end{array}$$

Figure 8: continued

$$\begin{array}{cc}
\begin{array}{l}
I_4 \\
\left. \begin{array}{l} [3, F \rightarrow a \cdot, \$] \\ [3, F \rightarrow a \cdot, +] \\ [3, F \rightarrow a \cdot, *] \end{array} \right\} a_1 \\
\left. \begin{array}{l} [3, T \rightarrow F \cdot, \$] \\ [3, T \rightarrow F \cdot, +] \\ [3, T \rightarrow F \cdot, *] \end{array} \right\} F_1 \\
\left. \begin{array}{l} [3, E \rightarrow T \cdot, \$] \\ [3, E \rightarrow T \cdot, +] \\ [3, T \rightarrow T \cdot * F, \$] \\ [3, T \rightarrow T \cdot * F, +] \\ [3, T \rightarrow T \cdot * F, *] \end{array} \right\} T_1 \\
\left. \begin{array}{l} [2, F \rightarrow (E \cdot), \$] \\ [2, F \rightarrow (E \cdot), +] \\ [2, F \rightarrow (E \cdot), *] \end{array} \right\} E_1 \\
\left. \begin{array}{l} [3, E \rightarrow \cdot E + T, )] \\ [3, E \rightarrow \cdot E + T, +] \end{array} \right\}
\end{array}
\quad
\begin{array}{l}
I_5 \\
\left. \begin{array}{l} [3, E \rightarrow E + \cdot T, )] \\ [3, E \rightarrow E + \cdot T, +] \\ [5, T \rightarrow \cdot T * F, )] \\ [5, T \rightarrow \cdot F, )] \\ [5, T \rightarrow \cdot T * F, +] \\ [5, T \rightarrow \cdot F, +] \\ [5, T \rightarrow \cdot T * F, *] \\ [5, T \rightarrow \cdot F, *] \\ [5, F \rightarrow \cdot (E), )] \\ [5, F \rightarrow \cdot a, )] \\ [5, F \rightarrow \cdot (E), +] \\ [5, F \rightarrow \cdot a, +] \\ [5, F \rightarrow \cdot (E), *] \\ [5, F \rightarrow \cdot a, *] \end{array} \right\} +_2
\end{array}
\end{array}$$
  

$$\begin{array}{cc}
\begin{array}{l}
I_6 \\
\left. \begin{array}{l} [5, F \rightarrow a \cdot, )] \\ [5, F \rightarrow a \cdot, +] \\ [5, F \rightarrow a \cdot, *] \end{array} \right\} a_2 \\
\left. \begin{array}{l} [5, T \rightarrow F \cdot, )] \\ [5, T \rightarrow F \cdot, +] \\ [5, T \rightarrow F \cdot, *] \end{array} \right\} F_2 \\
\left. \begin{array}{l} [3, E \rightarrow E + T \cdot, )] \\ [3, E \rightarrow E + T \cdot, +] \\ [5, T \rightarrow T \cdot * F, )] \\ [5, T \rightarrow T \cdot * F, +] \\ [5, T \rightarrow T \cdot * F, *] \end{array} \right\} T_4
\end{array}
\quad
\begin{array}{l}
I_7 \\
\left. \begin{array}{l} [5, T \rightarrow T * \cdot F, )] \\ [5, T \rightarrow T * \cdot F, +] \\ [5, T \rightarrow T * \cdot F, *] \\ [7, F \rightarrow \cdot (E), )] \\ [7, F \rightarrow \cdot a, )] \\ [7, F \rightarrow \cdot (E), +] \\ [7, F \rightarrow \cdot a, +] \\ [7, F \rightarrow \cdot (E), *] \\ [7, F \rightarrow \cdot a, *] \end{array} \right\} *_2
\end{array}$$

Figure 8: continued

$$\begin{array}{cc}
\begin{array}{c} I_8 \\
\left. \begin{array}{l} [7, F \rightarrow a\cdot, )] \\
[7, F \rightarrow a\cdot, +] \\
[7, F \rightarrow a\cdot, *] \end{array} \right\} a_2 \\
\left. \begin{array}{l} [5, T \rightarrow T * F\cdot, )] \\
[5, T \rightarrow T * F\cdot, +] \\
[5, T \rightarrow T * F\cdot, *] \end{array} \right\} F_4 \\
\left. \begin{array}{l} [3, E \rightarrow E + T\cdot, )] \\
[3, E \rightarrow E + T\cdot, +] \\
[3, T \rightarrow T \cdot * F, )] \\
[3, T \rightarrow T \cdot * F, +] \\
[3, T \rightarrow T \cdot * F, *] \end{array} \right\} T_4 \\
\left. \begin{array}{l} [2, F \rightarrow (E\cdot), \$] \\
[2, F \rightarrow (E\cdot), +] \\
[2, F \rightarrow (E\cdot), *] \\
[2, E \rightarrow \cdot E + T, )] \\
[2, E \rightarrow \cdot E + T, +] \end{array} \right\} E_1
\end{array} &
\begin{array}{c} I_9 \\
\left. \begin{array}{l} [2, F \rightarrow (E)\cdot, \$] \\
[2, F \rightarrow (E)\cdot, +] \\
[2, F \rightarrow (E)\cdot, *] \end{array} \right\} )_1 \\
\left. \begin{array}{l} [0, T \rightarrow T * F\cdot, \$] \\
[0, T \rightarrow T * F\cdot, +] \\
[0, T \rightarrow T * F\cdot, *] \end{array} \right\} F_3 \\
\left. \begin{array}{l} [0, E \rightarrow T\cdot, \$] \\
[0, E \rightarrow T\cdot, +] \\
[0, T \rightarrow T \cdot * F, \$] \\
[0, T \rightarrow T \cdot * F, +] \\
[0, T \rightarrow T \cdot * F, *] \end{array} \right\} T_1 \\
\left. \begin{array}{l} [0, S' \rightarrow E \cdot \$, \$] \\
[0, E \rightarrow E \cdot + T, \$] \\
[0, E \rightarrow E \cdot + T, +] \end{array} \right\} E_3
\end{array}
\end{array}$$

$$\begin{array}{c} I_{10} \\
[0, S' \rightarrow E \$ \cdot, \$] \} f
\end{array}$$

Figure 8: continued

Another point of view, which aids in the development of the  $LR(k)$  Early's Algorithm, is that of simulating the nondeterministic  $LR(k)$  parser. A major concern in such a simulation is the maintenance of the parse stack for each copy of the  $LR(k)$  parser. Thus, if the  $LR(k)$  Early's Algorithm is going to be viewed as simulating the  $LR(k)$  parser, it must have a means of "linking" states together into a stack.

The result of these considerations is Algorithm 3.1.1, the  $LR(k)$  Early's Algorithm. This algorithm uses  $n + 2$  parse lists  $I_0, I_1, \dots, I_n$  and  $I_{n+1}$  just like Early's algorithm. However, the entries on the parse list are ordered triples  $[q, i, r]$  where  $q$  and  $r$  are states of the  $LR(k)$  parser and  $i$  is the number of a parse list. The entry  $[q, i, r]$  on a parse list  $I_j$  is interpreted as meaning that the simulated  $LR(k)$  parser is in a configuration  $(\alpha q r, w_{j+1:n+k+1})$  and that, in addition, the entry corresponding to the configuration in which  $q$  was on top of the stack is found on parse list  $I_i$ .

Algorithm 3.1.1 also uses a set  $T$  and  $n + 2$  *pending* lists  $H_0, H_1, \dots, H_n$  and  $H_{n+1}$  that do not correspond to any data structures in Early's algorithm. These data structures are required to handle the subtleties introduced by reductions of empty productions in the  $LR(k)$  parser.

Algorithm 3.1.1 operates by simulating the nondeterministic  $LR(k)$  parser. The simulation of shifts by step II.A.2 of Algorithm 3.1.1 is straightforward, but the simulation of reductions by step II.A.3 requires some explanation. When an entry  $[q, i, r]$  is on parse list  $I_j$  and reduce  $p \in f_r(w)$ , Algorithm 3.1.1 must simulate a reduction using the  $p^{\text{th}}$  production. When Algorithm 3.1.1 simulates a reduction, it must take into account the possibility that  $[q, i, r]$  is part of different stacks for several copies of the  $LR(k)$  parser and be careful to apply the reduction to all the copies. To do this, Algorithm 3.1.1 constructs the sets  $R_0, R_1, \dots, R_m$  where  $m = |\text{RHS}(p)|$ . These sets contain elements of the form  $([q, i, r], j)$  where  $[q, i, r]$  is an entry and  $j$  is the number of the parse list on which the entry is found. The sets are constructed so that  $R_l$  contains all the entries which correspond to states that are the  $l^{\text{th}}$  state below the top of the stack in some configuration when the reduction is applied. Thus, the entries in  $R_m$  correspond to the possible states on top of the stack after  $m$  states have been popped from the stack. Algorithm 3.1.1 simulates the effect of the reduction for each of the entries in  $R_m$  so that it simulates all the copies of the  $LR(k)$  parser.

Like Early's algorithm, Algorithm 3.1.1 avoids infinite looping by not adding duplicate entries to parse lists and not placing such entries on the pending lists. However, step II.A.3.iii of Algorithm 3.1.1 can not use this method since a reduction by an empty production initiates a sequence of reductions (possibly only

ALGORITHM 3.1.1 *LR(k) Early's Algorithm*

I. Set  $j = 0$ ; place  $[0, 0, 0]$  on parse list  $I_0$ ; place  $[0, 0, 0]$  on pending list  $H_0$ ; initialize  $T$  to contain the state pair  $(0, 0)$ ; and append  $\$^{k+1}$  to  $w$ .

II. While  $j \leq n$ , perform the following steps:

A. While pending list  $H_j$  is not empty, remove an entry  $[q, i, r]$  from the front of  $H_j$  and perform the following steps:

1. If  $i \neq j$  then remove all state pairs from  $T$ .
2. If  $\text{shift} \in f_r(w_{j+1:j+k})$  and there exists  $s \in g_r(w_{j+1:j+k})$  then if the entry  $[r, j, s]$  is not on parse list  $I_{j+1}$ , add  $[r, j, s]$  to  $I_{j+1}$  and to the rear of pending list  $H_{j+1}$ .
3. For each **reduce**  $p \in f_r(w_{j+1:j+k})$  perform the following steps:
  - i. Let  $m = |\text{RHS}(p)|$  and let the sets  $R_0, R_1, \dots, R_m$  be defined as follows:
    - $R_0 = \{([q, i, r], j)\}$  where  $[q, i, r]$  is the entry from  $H_j$
    - for  $0 < l \leq m$ ,  $R_l = \{([q, i, r], o) \mid [q, i, r] \in I_o \text{ and, for some } s \text{ and } n, ([r, o, s], n) \in R_{l-1}\}$
  - ii. For each  $s \in g_r(\text{LHS}(p))$  and  $([q, i, r], o) \in R_m$  for which  $i < j$ : if the entry  $[r, o, s]$  is not already on parse list  $I_j$  then add  $[r, o, s]$  to  $I_j$  and add  $[r, o, s]$  to the rear of pending list  $H_j$ .
  - iii. For each  $s \in g_r(\text{LHS}(p))$  and  $([q, i, r], o) \in R_m$  for which  $i = j$ : if the entry  $[r, o, s]$  is not already on parse list  $I_j$ , add  $[r, o, s]$  to  $I_j$ ; and, if the state pair  $(r, s)$  is not on  $T$ , add  $[r, o, s]$  to the front of pending list  $H_j$  and add  $(r, s)$  to  $T$ .

B. Let  $j = j + 1$ .

the reduction itself) that take place independent of the input and of the previous contents of the stack. Even if the resulting entries are already on the parse list, the sequence of reductions must be simulated since it may eventually pop part of the stack from before the sequence. Entries from such a reduction sequence are added to the front of  $H_j$  so that the reduction sequence is completely simulated at once. This allows the set  $T$  of state pairs to be used to guard against an infinite loop by keeping track of the states that have followed each other on the top of the

$I_0$ [0, 0, 0]	$I_1$ [0, 0, $a_1$ ] [0, 0, $F_1$ ] [0, 0, $T_1$ ]	$I_2$ [ $T_1$ , 1, $*_1$ ]	$I_3$ [ $*_1$ , 2, (1]
$I_4$ [(1, 3, $a_1$ ] [(1, 3, $F_1$ ] [(1, 3, $T_1$ ] [(1, 3, $E_1$ ]	$I_5$ [ $E_1$ , 4, +2]	$I_6$ [+2, 5, $a_2$ ] [+2, 5, $F_2$ ] [+2, 5, $T_4$ ]	$I_7$ [ $T_4$ , 6, $*_2$ ]
$I_8$ [ $*_2$ , 7, $a_2$ ] [ $*_2$ , 7, $F_4$ ] [+2, 5, $T_4$ ] [(1, 3, $E_1$ ]	$I_9$ [ $E_1$ , 8, ) $_1$ ] [ $*_1$ , 2, $F_3$ ] [0, 0, $T_1$ ] [0, 0, $E_3$ ]	$I_{10}$ [ $E_3$ , 9, $f$ ]	

Figure 9: Parse Lists for LR( $k$ ) Early's Algorithm

stack. When a pair of states is found to repeat itself, the loop can be avoided by not adding the entry to  $H_j$ .

The parse lists generated by Algorithm 3.1.1 for the grammar in Figure 1 and the input string

$$a * (a + a * a)$$

are given in Figure 9. As one would expect, for each parse list the states in the third component of the entries correspond to the states identified in Figure 8 on the parse lists for Early's algorithm. This correspondence is why the name LR( $k$ ) Early's Algorithm is given to Algorithm 3.1.1.

As presented, Algorithm 3.1.1 only constructs parse lists. However, it is shown in the next section that  $[?, ?, f]$  is on parse list  $I_{n+1}$  if and only if the input string is accepted by the LR( $k$ ) parser. Also, Algorithm 3.1.1 does not show how a right parse is obtained for the input string. In principle, a right parse can be extracted from the parse lists in the same way a right parse can be extracted from the parse lists for Early's algorithm, but the issue is not explored in this dissertation.



### 3.2 PROOFS OF CORRECTNESS AND COMPLETENESS

This section presents two theorems which, taken together, show that the  $LR(k)$  Early's Algorithm and the  $LR(k)$  parser accept the same strings when they use the same set of states,  $Q$ , and functions  $f_q$  and  $g_q$ . The first theorem shows that for every entry  $[q, i, r]$  the  $LR(k)$  Early's Algorithm places on a parse list  $I_j$ , there is a corresponding sequence of moves

$$(0, w_{1:n+k+1}) \vdash^* (\alpha \wr q, w_{i+1:n+k+1}) \vdash^+ (\alpha qr, w_{j+1:n+k+1})$$

that can be made by the  $LR(k)$  parser. This property is called *correctness* since it guarantees that any string accepted by Algorithm 3.1.1 is accepted by the  $LR(k)$  parser; which is to say that Algorithm 3.1.1 correctly simulates the  $LR(k)$  parser.

The second theorem shows that for every sequence of moves

$$(0, w_{1:n+k+1}) \vdash^* (\alpha \wr q, w_{i+1:n+k+1}) \vdash^+ (\alpha qr, w_{j+1:n+k+1})$$

that can be made by the  $LR(k)$  parser, Algorithm 3.1.1 places the entry  $[q, i, r]$  on the parse list  $I_j$ . This property is called *completeness* since it guarantees that any string accepted by the  $LR(k)$  parser is accepted by Algorithm 3.1.1; which is to say that the  $LR(k)$  Early's Algorithm completely simulates the  $LR(k)$  parser.

Algorithm 3.1.1 uses the pending lists,  $H_j$ , to hold entries waiting to be processed by the algorithm. The following lemma establishes that entries added to the pending list are eventually processed.

**LEMMA 3.2.1** (All Entries on  $H_j$  Are Processed) *Every entry added to a pending list  $H_j$  is eventually processed by step II.A of Algorithm 3.1.1.*

**Proof:** An entry can remain unprocessed only if an unbounded number of entries can be added to a pending list  $H_j$ . The lemma is proved by showing the number of entries which can be added to a pending list  $H_j$  is bounded. Note that the number of entries  $[q, i, r]$  on  $I_j$  is bounded since  $q$ ,  $i$ , and  $r$  are all bounded and no step of Algorithm 3.1.1 allows duplicates to be added to  $I_j$ .

Only four steps can add an entry to a pending list: steps I, II.A.2, II.A.4.i, and II.A.4.ii. Step I adds just one entry to a pending list. The number of entries added to a pending list by step II.A.2 is bounded because it only adds entries which can also be added to their parse lists.

Steps II.A.4.i and II.A.4.ii add a bounded number of entries to a pending list when they are invoked. Also, they are invoked a bounded number of times for entries  $[q, i, r]$  on  $I_j$  for which  $i < j$  because there are only a bounded number of

such entries that can be added to  $I_j$ . Since step II.A.4.i is invoked only for  $i < j$ , only a bounded number of entries are added to  $H_j$  by it.

Step II.A.4.ii only adds entries  $[q, i, r]$  for which  $i = j$  to  $H_j$ . Step A is organized so that only a bounded number of entries are added by step II.A.4.ii between the processing of entries for which  $i \neq j$ . Only steps II.A.2 and I.A.4.i can add entries for which  $i \neq j$  and the number of entries added by these steps is bounded. Therefore, the total number of entries added to  $H_j$  by step II.A.4.ii is bounded. ■

Then next lemma shows a technical property of the initial state for Algorithm 3.1.1.

**LEMMA 3.2.2** (0 is the Unique Initial State) *If  $[q, i, r]$  is on  $I_j$  and  $r = 0$  then  $q = 0$ ,  $i = 0$ , and  $j = 0$ .*

**Proof:** The definition of an  $LR(k)$  parser does not allow  $0 \in g_s(X)$  for any  $s$  or  $X$ . Thus, only step I of Algorithm 3.1.1 could add  $[q, i, 0]$  to a parse list. ■

The following definition facilitates the use of induction in the proofs of completeness and correctness for Algorithm 3.1.1

**Definition 3.2.1** (Ordered List of Entries) *An ordered list of entries is a sequence of entries and their parse lists*

$$\begin{array}{l} [q_1, i_1, r_1] \text{ on } I_{j_1} \\ [q_2, i_2, r_2] \text{ on } I_{j_2} \\ \vdots \\ [q_N, i_N, r_N] \text{ on } I_{j_N} \end{array}$$

given in an order in which they can be added to their parse lists by step II of Algorithm 3.1.1 during an execution of Algorithm 3.1.1.

The entry  $[0, 0, 0]$  on  $I_0$  is not on any ordered list of entries since it is not added to its parse list by step II of Algorithm 3.1.1. No particular ordering of entries is required other than an order in which the entries could be added to their parse lists by Algorithm 3.1.1.

**THEOREM 3.2.1** (Algorithm 3.1.1 Correctly Simulates the  $LR(k)$  Parser) *Given the same  $Q$ ,  $f_q$  and  $g_q$  for Algorithm 3.1.1 and the  $LR(k)$  parser, if an entry  $[r, i, s]$  is added to parse list  $I_j$  (except for  $[0, 0, 0]$  on  $I_0$ ) by Algorithm 3.1.1 then*

the  $LR(k)$  parser can make the sequence of moves

$$(0, w_{1:n+k+1}) \vdash^* (\alpha \wr r, w_{i+1:n+k+1}) \vdash^+ (\alpha r s, w_{j+1:n+k+1}).$$

Proof: The theorem is proved by induction on an ordered list of entries, using the theorem as the induction hypothesis. The induction proceeds in two steps:

- first, the theorem is proved for the first entry on an ordered list of entries; and
- second, the theorem is proved for the  $N^{\text{th}}$  entry on the ordered list of entries, assuming it holds for all entries that precede the  $N^{\text{th}}$  entry.

For the first induction step, the first entry on the ordered list of entries must be added by applying step II.A of Algorithm 3.1.1 to the entry  $[0, 0, 0]$  on  $I_0$ . Likewise, the first move of the  $LR(k)$  parser must be from the configuration  $(0, w_{1:n+k+1})$ . Thus, if step II.A.2 adds an entry  $[0, 0, s]$  to  $I_1$  then the shift  $(0, w_{1:n+k+1}) \vdash (0s, w_{2:n+k+1})$  can be made by the  $LR(k)$  parser; and if step II.A.3.ii or step II.A.3.iii adds an entry  $[0, 0, s]$  to  $I_0$  then the reduction

$$(0, w_{1:n+k+1}) \vdash (0s, w_{1:n+k+1})$$

can be made by the  $LR(k)$  parser.

For the second induction step, the theorem is assumed to hold for all entries on the ordered list of entries that precede the  $N^{\text{th}}$  entry. Let the  $N^{\text{th}}$  entry be  $[r, i, s]$  on  $I_j$ . Since the  $N^{\text{th}}$  entry is added by step II.A of Algorithm 3.1.1, this step must have been applied to an entry  $[q_{m-1}, l_{m-1}, q_m]$  from  $H_{l_m}$ , where  $m$  is a convenient index which will be specified later. Since  $[q_{m-1}, l_{m-1}, q_m]$  is on  $H_{l_m}$ , the entry  $[q_{m-1}, l_{m-1}, q_m]$  must be on  $I_{l_m}$  and must precede the entry  $[r, i, s]$  in the order. Therefore, the  $LR(k)$  parser can make the sequence of moves

$$(0, w_{1:n+k+1}) \vdash^* (\alpha q_m, w_{l_m+1:n+k+1}).$$

There are two cases to consider:

- step II.A.2 adds  $[r, i, s]$  to  $I_j$  and Algorithm 3.1.1 is simulating a shift by the  $LR(k)$  parser; or
- step II.A.3 adds  $[r, i, s]$  to  $I_j$  and Algorithm 3.1.1 is simulating a reduction by the  $LR(k)$  parser.

In the first case, step II.A.2 adds  $[r, i, s]$  to  $I_j$  and Algorithm 3.1.1 is simulating a shift by the  $\text{LR}(k)$  parser. Thus,  $l_m = i = j - 1$ ,  $q_m = r$ , and the shift  $(\alpha r, w_{i+1:n+k+1}) \vdash (\alpha r s, w_{j+1:n+k+1})$  can be made by the  $\text{LR}(k)$  parser. Therefore, the  $\text{LR}(k)$  parser can make the sequence of moves

$$(0, w_{1:n+k+1}) \vdash^* (\alpha \wr r, w_{i+1:n+k+1}) \vdash^+ (\alpha r s, w_{j+1:n+k+1}).$$

In the second case, step II.A.3 adds  $[r, i, s]$  to  $I_j$  and Algorithm 3.1.1 is simulating a reduction by the  $\text{LR}(k)$  parser. The entry  $[q_{m-1}, l_{m-1}, q_m]$ , where  $l_m = j$ , must call for a reduction of length  $m$ . Step II.A.3.i constructs the sets  $R_0, R_1, R_2, \dots, R_m$ . Each of these sets must have at least one member so let these members be

$$\begin{aligned} ([q_{m-1}, l_{m-1}, q_m], l_m) &\in R_0, \\ ([q_{m-2}, l_{m-2}, q_{m-1}], l_{m-1}) &\in R_1, \\ &\dots, \\ ([q_0, l_0, q_1], l_1) &\in R_{m-1}, \\ ([q_{-1}, l_{-1}, q_0], l_0) &\in R_m \end{aligned}$$

where  $q_0 = r$  and  $l_0 = i$ . The reduction must be one of three possible types, each of which must be considered separately:

- a reduction by an empty production (i.e.  $m = 0$ );
- a reduction by a non-empty production which does not cause the stack to underflow (i.e.  $m > 0$  and  $q_x \neq 0$  for  $0 < x \leq m$ ); or
- a reduction by a non-empty production which causes the stack to underflow (i.e.  $m > 0$  and  $q_x = 0$  for some  $x$  where  $0 < x \leq m$ ).

For the first type of reduction, an empty production is used so  $m = h = 0$ . Thus,  $l_m = l_0 = i = j$ ,  $q_m = q_0 = r$ , and the reduction

$$(\alpha r, w_{i+1:n+k+1}) \vdash (\alpha r s, w_{j+1:n+k+1})$$

can be made by the  $\text{LR}(k)$  parser. Therefore, the  $\text{LR}(k)$  parser can make the sequence of moves

$$(0, w_{1:n+k+1}) \vdash^* (\alpha \wr r, w_{i+1:n+k+1}) \vdash^+ (\alpha r s, w_{j+1:n+k+1}).$$

For the second type of reduction, a non-empty production is used and the stack does not underflow so  $m > 0$  and  $q_x \neq 0$  for  $0 < x \leq m$ . This implies the entries  $[q_{x+1}, l_{x+1}, q_x]$  on  $I_{l_x}$  for  $0 < x \leq m$  are entries on the ordered list of entries

and that these entries precede  $[r, i, s]$  on  $I_j$ . Applying the induction hypothesis to each of these entries, the  $\text{LR}(k)$  parser can make the sequences of moves:

$$\begin{aligned} (0, w_{1:n+k+1}) &\vdash^* (\alpha_0 \setminus r, w_{i+1:n+k+1}) \vdash^+ (\alpha_1 r q_1, w_{l_1:n+k+1}) \\ (0, w_{1:n+k+1}) &\vdash^* (\alpha_1 \setminus q_1, w_{l_1:n+k+1}) \vdash^+ (\alpha_1 q_1 q_2, w_{l_2:n+k+1}) \\ &\vdots \\ (0, w_{1:n+k+1}) &\vdash^* (\alpha_{m-1} \setminus q_{m-1}, w_{l_{m-1}:n+k+1}) \vdash^+ (\alpha_{m-1} q_{m-1} q_m, w_{j:n+k+1}). \end{aligned}$$

As a result, the  $\text{LR}(k)$  parser can make the sequence of moves

$$(0, w_{1:n+k+1}) \vdash^* (\alpha_0 \setminus r, w_{i+1:n+k+1}) \vdash^+ (\alpha_0 r q_1 \dots q_m, w_{j+1:n+k+1}).$$

Also, step II.A.3 calls for a reduction of length  $m$  so the reduction

$$(\alpha_0 r q_1 \dots q_m, w_{j+1:n+k+1}) \vdash (\alpha_0 r s, w_{j+1:n+k+1})$$

can be made by the  $\text{LR}(k)$  parser. Therefore, the  $\text{LR}(k)$  parser can make the sequence of moves

$$(0, w_{1:n+k+1}) \vdash^* (\alpha_0 \setminus r, w_{i+1:n+k+1}) \vdash^+ (\alpha_0 r s, w_{j+1:n+k+1}).$$

For the third type of reduction, a non-empty production is used and the stack underflows so  $m > 0$  and  $q_x = 0$  for some  $x$  where  $0 < x \leq m$ . Let  $e$  be the greatest such  $x$ . Since  $1 \leq e$ ,  $r = q_e = 0$ . Recursively applying Lemma 3.2.2 shows that, for  $0 \leq x \leq e$ ,  $([q_{x-1}, l_{x-1}, q_x], l_x) = ([0, 0, 0], 0)$ . If  $e = m$  then there is only the state 0 on the stack when the reduction is made and  $q_m = q_0 = 0$  and  $l_m = l_0 = i = j = 0$ . Therefore, the  $\text{LR}(k)$  parser can make the sequence of moves

$$(0, w_{1:n+k+1}) \vdash^* (0, w_{i+1:n+k+1}) \vdash^+ (0 s, w_{j+1:n+k+1}).$$

If  $e < m$  there are some states (but not enough) on the stack when the reduction is applied. For  $e < x \leq m$ ,  $[q_{x+1}, l_{x+1}, q_x]$  on  $I_{l_x}$  is an entry in the order that precedes  $[r, i, s]$  on  $I_j$ . Applying the induction hypothesis to each of these entries, the  $\text{LR}(k)$  parser can make the sequences of moves:

$$\begin{aligned} (0, w_{1:n+k+1}) &\vdash^* (\alpha_e \setminus q_e, w_{i+1:n+k+1}) \vdash^+ (\alpha_e q_e q_{e+1}, w_{l_{e+1}:n+k+1}) \\ (0, w_{1:n+k+1}) &\vdash^* (\alpha_{e+1} \setminus q_{e+1}, w_{l_{e+1}:n+k+1}) \vdash^+ (\alpha_{e+1} q_{e+1} q_{e+2}, w_{l_{e+2}:n+k+1}) \\ &\vdots \\ (0, w_{1:n+k+1}) &\vdash^* (\alpha_{m-1} \setminus q_{m-1}, w_{l_{m-1}:n+k+1}) \vdash^+ (\alpha_{m-1} q_{m-1} q_m, w_{j:n+k+1}). \end{aligned}$$

Since  $q_e = 0$  and 0 is never pushed onto the stack,  $\alpha_e = \epsilon$ . As a result, the  $LR(k)$  parser can make the sequence of moves

$$(0, w_{1:n+k+1}) \vdash^* (0, w_{i+1:n+k+1}) \vdash^+ (0, q_{e+1}, w_{l_{e+1}+1:n+k+1}) \vdash^+ (0, q_{e+1} \dots q_m, w_{j+1:n+k+1}).$$

Also, step II.A.3 calls for a reduction of length  $m$  so the reduction

$$(0, q_{e+1} \dots q_m, w_{j+1:n+k+1}) \vdash (0s, w_{j+1:n+k+1})$$

can be made by the  $LR(k)$  parser. Therefore, the  $LR(k)$  parser can make the sequence of moves

$$(0, w_{1:n+k+1}) \vdash^* (0, w_{i+1:n+k+1}) \vdash^+ (0s, w_{j+1:n+k+1}).$$

■

**THEOREM 3.2.2** (Algorithm 3.1.1 Completely Simulates the  $LR(k)$  Parser)  
*Given the same  $Q$ ,  $f_q$ , and  $g_q$  for the  $LR(k)$  parser and Algorithm 3.1.1, if the sequence of moves*

$$(0, w_{1:n+k+1}) \vdash^L (\alpha)r, w_{i+1:n+k+1}) \vdash^{M+1} (\alpha rs, w_{j+1:n+k+1})$$

*can be made by the  $LR(k)$  parser, where  $L \geq 0$  and  $M \geq 0$ , then Algorithm 3.1.1 will add the entry  $[r, i, s]$  to parse list  $I_j$ .*

**Proof:** The theorem is proved by induction on the sum of  $L$  and  $M$ . The induction proceeds in three steps:

- first, the theorem is proved for  $L + M = 0$ ;
- second, the theorem is proved for  $L = N$  and  $M = 0$ , assuming it holds whenever  $L + M < N$ ; and
- third, the theorem is proved for  $L + M = N$  when  $M > 0$ , assuming it holds whenever  $L + M < N$ .

If the parameter space formed by  $L$  and  $M$  is imagined as an infinite table with  $L$  as the row number and  $M$  as the column number, then the steps of the induction can be viewed as proving the theorem diagonal by diagonal, using the diagonals that run from the lower left to upper right sides of the table.

For the first induction step,  $L + M = 0$  and the theorem can be written as follows:

If  $(0, w_{1:n+k+1}) \vdash (0s, w_{j+1:n+k+1})$  then  $[0, 0, s]$  is added to  $I_j$ .

The move  $(0, w_{1:n+k+1}) \vdash (0s, w_{j+1:n+k+1})$  can be either a shift, in which case  $j = 1$ , or a reduction, in which case  $j = 0$ . When  $[0, 0, 0]$  from  $H_0$  is processed by step II.A of Algorithm 3.1.1, if the move is a shift then step II.A.2 of Algorithm 3.1.1 will add  $[0, 0, s]$  to  $I_1$ . Likewise, if the move is a reduction then step II.A.3 of Algorithm 3.1.1 will add  $[0, 0, s]$  to  $I_0$ .

For the second induction step,  $L = N$ ,  $M = 0$  and the theorem is assumed to be true for  $L + M < N$ . Since  $M = 0$ , the theorem can be written as follows:

If  $(0, w_{1:n+k+1}) \vdash^N (\alpha \wr r, w_{i+1:n+k+1}) \vdash (\alpha rs, w_{j+1:n+k+1})$ , where  $N \geq 0$ , then the entry  $[r, i, s]$  is added to parse list  $I_j$ .

If  $N = 0$  then this induction step degenerates to the first induction step. Therefore, only the case of  $N > 0$  needs to be considered. When  $N > 0$ ,  $|\alpha| \geq 1$  since no move of the  $LR(k)$  parser can push 0 onto the stack and no move of the  $LR(k)$  parser can pop 0 from the stack. Since  $|\alpha| \geq 1$ ,  $\alpha = \beta q$ . Furthermore, the sequence of moves

$$(0, w_{1:n+k+1}) \vdash^* (\beta \wr q, \dots) \vdash^+ (\beta qr, w_{i+1:n+k+1})$$

is less than  $N$  in length so  $[q, ?, r]$  must be on  $I_i$  and must also be on  $H_j$  at some point during the execution of Algorithm 3.1.1.

The move  $(\alpha r, w_{i+1:n+k+1}) \vdash (\alpha rs, w_{j+1:n+k+1})$  can be either a shift, in which case  $j = i + 1$ , or a reduction of an empty production, in which case  $j = i$ . When  $[q, ?, r]$  is processed by step II.A of Algorithm 3.1.1, if the move is a shift then step II.A.2 of Algorithm 3.1.1 will add  $[r, i, s]$  to  $I_j$ . Likewise, if the move is a reduction of an empty production then step II.A.3 of Algorithm 3.1.1 will add  $[r, i, s]$  to  $I_j$ .

For the third induction step,  $M > 0$ ,  $L + M = N$  and the theorem is assumed to hold for  $L + M < N$ . Since  $M > 0$ , the theorem can be written as follows:

If  $(0, w_{1:n+k+1}) \vdash^L (\alpha \wr r, w_{i+1:n+k+1}) \vdash^{M+1} (\alpha rs, w_{j+1:n+k+1})$ , where  $L \geq 0$  and  $M > 0$ , then the entry  $[q, i, r]$  is added to parse list  $I_j$ .

To show that the entry  $[?, ?, r]$  is on  $I_i$ , there are two cases to consider:

- $\alpha = \epsilon$ .
- $\alpha = \beta t$ .

In the first case,  $\alpha = \epsilon$  implies that  $r = 0$  and that, using Lemma 3.2.2  $i = 0$  and  $[?, ?, r] = [0, 0, 0]$ . In the second case,  $\alpha = \beta t$  and the induction hypothesis can be applied to the sequence of moves

$$(0, w_{1:n+k+1}) \stackrel{L-x}{\vdash} (\beta t, \dots) \stackrel{x}{\vdash} (\beta t r, w_{i+1:n+k+1}),$$

to show that  $[?, ?, r]$  is on  $I_i$ .

Since a move of the  $LR(k)$  parser can add at most one symbol to the stack, the sequence of moves

$$(\alpha \setminus r, w_{i+1:n+k+1}) \stackrel{M+1}{\vdash} (\alpha r s, w_{j+1:n+k+1})$$

can be written as

$$\begin{aligned} (\alpha \setminus r, w_{i+1:n+k+1}) &\stackrel{+}{\vdash} (\alpha r \setminus q_1, w_{l_1+1:n+k+1}) \\ &\stackrel{+}{\vdash} (\alpha r q_1 \setminus q_2, w_{l_2+1:n+k+1}) \\ &\stackrel{+}{\vdash} \dots \\ &\stackrel{+}{\vdash} (\alpha r q_1 q_2 \dots \setminus q_{m-1}, w_{l_{m-1}+1:n+k+1}) \\ &\stackrel{+}{\vdash} (\alpha r q_1 q_2 \dots q_m, w_{j+1:n+k+1}) \\ &\vdash (\alpha r s, w_{j+1:n+k+1}) \end{aligned}$$

where  $0 < m \leq M$ . The move  $(\alpha r q_1 q_2 \dots q_m, w_{j+1:n+k+1}) \vdash (\alpha r s, w_{j+1:n+k+1})$  is either a reduction that pops  $m$  states off the stack or, if  $\alpha = \epsilon$ , possibly a reduction which underflows the stack. Any proper subsequence of the sequence of moves

$$(\alpha \setminus r, w_{i+1:n+k+1}) \stackrel{+}{\vdash} (\alpha r s, w_{j+1:n+k+1})$$

has length less than  $N$  so  $[r, i, q_1]$  is on  $I_{l_1}$ ,  $[q_1, l_2, q_2]$  is on  $I_{l_2}$ , ..., and  $[q_{m-1}, l_{m-1}, q_m]$  is on  $I_j$ .

While  $[r, i, q_1]$ ,  $[q_1, l_2, q_2]$ , ..., and  $[q_{m-1}, l_{m-1}, q_m]$  are on their respective parse lists when Algorithm 3.1.1 terminates, they must also be on them when step II.A removes  $[q_{m-1}, l_{m-1}, q_m]$  from  $H_j$ . Since Algorithm 3.1.1 processes parse lists in increasing order, this will be the case whenever  $l_x < j$ .

To see that this is also the case when  $l_x = j$ , let  $o$  be the smallest index for which  $l_x = j$ . Since no step of Algorithm 3.1.1 adds an entry to a parse list which precedes the parse list being processed,  $l_x = j$  for  $o \leq x \leq m$ . Furthermore, all the moves in the sequence

$$(\alpha \setminus q_o, w_{j+1:n+k+1}) \stackrel{+}{\vdash} (\alpha q_o q_{o+1} \dots q_m, w_{j+1:n+k+1})$$



must be reductions. Let this sequence of reductions be

$$\begin{aligned} (\alpha'_0 q'_0, w_{j+1:n+k+1}) &\vdash (\alpha'_1 q'_0 q'_1, w_{j+1:n+k+1}) \\ &\vdash \dots \\ &\vdash (\alpha'_t q'_0 q'_1 \dots q'_t, w_{j+1:n+k+1}) \end{aligned}$$

where the number of reductions is  $t$  and  $q_o = q'_0$  and  $q_m = q'_t$ .

Examining step II.A shows that, when an entry  $[q'_{x-1}, j, q'_x]$  is removed from  $H_j$ ,  $[q'_x, j, q'_{x+1}]$  is added to  $H_j$  and  $I_j$ ; unless  $(q'_x, q'_{x+1})$  is on  $T$ , in which case  $[q'_x, j, q'_{x+1}]$  must already be on  $I_j$ . If  $(q'_x, q'_{x+1})$  is on  $T$  then the parser must be looping thru a sequence of moves

$$(\beta q'_x q'_{x+1}, w_{j+1:n+k+1}) \vdash^+ (\delta q'_x q'_{x+1}, w_{j+1:n+k+1}).$$

Algorithm 3.1.1 follows this loop only once because the loop does not terminate. If the  $LR(k)$  parser is nondeterministic, it may follow the loop an arbitrary number of times and then possibly exit the loop. However, Algorithm 3.1.1 does not need to follow the loop more than once since the first iteration of the loop will add  $[q'_x, j, q'_{x+1}]$  to  $H_j$  and when  $[q'_x, j, q'_{x+1}]$  is processed any configuration which leads to an exit from the loop will be found and followed by Algorithm 3.1.1. The following sequence of moves illustrates this case:

$$\begin{aligned} (\beta q'_x q'_{x+1}, w_{j+1:n+k+1}) &\vdash (\beta q'_x q'_{x+1} q'_x q'_{x+1}, w_{j+1:n+k+1}) \\ &\vdash (\beta q'_x q'_{x+1} q'_x q'_{x+1} q'_x q'_{x+1}, w_{j+1:n+k+1}) \\ &\vdash (\beta q'_x q'_{x+1} q'_x q'_{x+1} q'_x q'_{x+1} q'_x q'_{x+2}, w_{j+1:n+k+1}) \end{aligned}$$

In this case, step II.A of Algorithm 3.1.1 will skip all but the first iteration of the loop, add the entry  $[q'_x, j, q'_{x+2}]$  to  $H_j$  and continue beyond the loop.

From the preceding examination of step II.A, it is clear that  $[r, i, q_1]$ ,  $[q_1, l_2, q_2]$ ,  $\dots$ , and  $[q_{m-1}, l_{m-1}, q_m]$  are on their respective parses lists when the entry  $[q_{m-1}, l_{m-1}, q_m]$  is added to  $H_j$ . Therefore, when  $[q_{m-1}, l_{m-1}, q_m]$  is removed from  $H_j$  and processed by step II.A.3 of Algorithm 3.1.1,

$$\begin{aligned} ([q_{m-1}, l_{m-1}, q_m], j) &\in R_0, \\ ([q_{m-2}, l_{m-1}, q_{m-1}], l_{m-1}) &\in R_1, \\ &\vdots, \\ ([r, i, q_1], l_1) &\text{ to } R_{m-1} \text{ and} \\ ([?, ?, r], i) &\text{ to } R_m. \end{aligned}$$

Furthermore, if  $\alpha = \epsilon$ , which implies  $([?, ?, r], i) = ([0, 0, 0], 0)$ , then step II.A.3.i also adds  $([0, 0, 0], 0)$  to  $R_x$  for  $x > m$ . Thus, either step II.A.3.ii or step II.A.3.iii of Algorithm 3.1.1 will attempt to add  $[r, i, s]$  to the pending list  $H_j$ , which implies  $[r, i, s]$  is on  $I$ , when Algorithm 3.1.1 terminates. ■

### 3.3 RUN TIME ANALYSIS

In this section, the time and space complexities of Algorithm 3.1.1 are shown to be  $\mathcal{O}(n^4)$  and  $\mathcal{O}(n^2)$  respectively. During the analysis of the algorithm it becomes evident that, to achieve these bounds, the sets  $R_0, R_1, \dots, R_m$  in step II.A.2 of Algorithm 3.1.1 must be computed very carefully. In fact, the changes required to efficiently compute these sets are so great that two different algorithms for implementing step II.A.2 are presented in this section.

To analyze the complexity of Algorithm 3.1.1, the method of Aho and Ullman [3] is used. In this method, the time complexity of an algorithm is determined by counting the number of *primitive* operations performed by the algorithm as a function of the size of the algorithm's input. A primitive operation is any sequence of machine instructions that is performed in a constant amount of time by a random access computer on an appropriate data structure in its memory. Examples of primitive operations are:

- accessing an element in an array.
- inserting a item at the head (or tail) of a list.
- accessing the next item on a list.

Extending this definition, a primitive operation is also any sequence of machine instructions that is performed in an amount of time that is bounded by a bound that is independent of the size of the input. An example of such a primitive operation for Algorithm 3.1.1 is evaluating  $f_q$  or  $g_q$  for an argument, because the time to evaluate these functions depends only on the size of the grammar and is independent of the length of the input string. From the definition of a primitive operation, it follows that the time complexity of any primitive operation is  $\mathcal{O}(1)$ .

The space complexity of an algorithm is determined in the same manner as time complexity. The size of the data structures used by the algorithm are analyzed for their dependence on the size of the input. The space complexity of data structures with sizes that are independent of the size of the input is  $\mathcal{O}(1)$ .

The input to Algorithm 3.1.1 is the string  $w$  to be parsed. The size of this input is the string's length, which is denoted by  $n$ . Some data structures, such as the parse lists, depend directly on  $n$ . It is implicitly assumed that  $n$  is known when these data structures are allocated, so that a position in the input string can be used in  $\mathcal{O}(1)$  time as an index into these data structures. This assumption presents no real difficulties, since the value of  $n$  can always be determined in  $\mathcal{O}(n)$  time by scanning the input string before starting the algorithm.

The parse lists, pending lists and set  $T$  are the primary data structures used by Algorithm 3.1.1. The parse lists are organized as an array of parse lists and each parse list  $I_j$  is  $|Q| + 1$  (initially empty) linked lists of entries. One list is used for entries  $[q, i, r]$  for which  $i = j$ . The other  $|Q|$  lists are used for entries  $[q, i, r]$  for which  $i \neq j$ ; these lists are indexed by  $r$  and an entry  $[q, i, r]$  is stored on the  $r^{\text{th}}$  list. The pending lists are organized as an array of pending lists and each pending list is an (initially empty) linked list of entries. The set  $T$  is organized as a two-dimensional bit map and is indexed by state in each dimension.

The space complexity of Algorithm 3.1.1 is easy to determine. There are  $\mathcal{O}(n)$  parse lists and  $\mathcal{O}(n)$  pending lists. Each parse list can have at most  $\mathcal{O}(n)$  entries because duplicate entries are not allowed and the only component of an entry that depends on the length of the input is the parse list number and it is bounded by  $n + 1$ . Also, each pending list can have at most  $\mathcal{O}(n)$  entries. This can be seen by examining step II.A of Algorithm 3.1.1 and noting that an entry  $[q, i, r]$  can be added to  $H_j$  under only two conditions:

1.  $i \neq j$  and  $[q, i, r]$  is not already on  $I_j$ .
2.  $i = j$  and  $[q, i, r]$  has not been added to  $H_j$  since the last entry  $[s, l, t]$  for which  $l \neq j$  was removed from  $H_j$ .

The first condition limits the number of entries  $[q, i, r]$  added to  $H_j$  under it to  $\mathcal{O}(n)$ . This in turn also limits the number of entries added under the second condition to  $\mathcal{O}(n)$  also. Thus, the space complexity of Algorithm 3.1.1 is  $\mathcal{O}(n^2)$ .

To analyze the time complexity of Algorithm 3.1.1, the following general scheme for counting primitive operations is used:

- each primitive operation is charged to an entry, parse list, or some other object used by the algorithm.
- the primitive operations charged to objects are determined by examining Algorithm 3.1.1 step-by-step.
- the number of primitive operations for each class of objects is obtained by summing the primitive operations charged to the objects in the class.

- the number of primitive operations for the algorithm is obtained by summing the primitive operations charged to each class of objects.

Beginning the analysis of the time complexity of Algorithm 3.1.1, one should note that the first step of the algorithm implicitly assumes the parse lists, pending lists, and set  $T$  are initialized. Given the organization of parse lists, pending lists, and set  $T$ , they can be initialized in  $\mathcal{O}(n)$  primitive operations which are charged to the *algorithm object* (a convenient object used only for the purpose of charging primitive operations to the algorithm as a whole).

For step I of Algorithm 3.1.1, the  $\mathcal{O}(1)$  primitive operations for initializing  $j$  are charged to the algorithm object. The  $\mathcal{O}(1)$  primitive operations for creating the initial entry and adding it to the first parse list and pending list are charged to the entry.

For step II of Algorithm 3.1.1, the  $\mathcal{O}(n)$  primitive operations for checking  $j$  against  $n$  for each iteration of the loop are charged to the algorithm object. Also, for step II.B of Algorithm 3.1.1, the  $\mathcal{O}(n)$  operations for incrementing  $j$  for each iteration of the loop are charged to the algorithm object.

For step II.A of Algorithm 3.1.1, the  $\mathcal{O}(1)$  primitive operations for initializing the loop to process the entries on the pending list are charged to the pending list. For each iteration of the loop, the  $\mathcal{O}(1)$  primitive operations for obtaining an entry and subsequently advancing to the next entry are charged to the obtained entry. In general, an entry may be obtained  $\mathcal{O}(n)$  times since an entry  $[q, i, r]$  on  $I_j$ , for which  $i = j$  may be added to  $H$ , a total of  $\mathcal{O}(n)$  times.

For step II.A.1 of Algorithm 3.1.1, the  $\mathcal{O}(1)$  primitive operations for emptying the set  $T$  are charged to the entry being processed.

For step II.A.2 of Algorithm 3.1.1, the  $\mathcal{O}(1)$  primitive operations for evaluating  $f_r$  and  $g_r$  are charged to the entry being processed.

For steps II.A.2, II.A.3.ii, and II.A.3.iii, the  $\mathcal{O}(n)$  primitive operations for checking if an entry  $[q, i, r]$  to be added is already on parse list  $I_j$  are charged to the entry being processed. The check requires  $\mathcal{O}(n)$  primitive operations because the number of entries on the  $r^{\text{th}}$  list of a parse list is  $\mathcal{O}(n)$  when  $i \neq j$  and each entry must be checked. The  $\mathcal{O}(1)$  primitive operations for adding an entry to its parse list are charged to the entry itself.

For steps II.A.2, II.A.3.ii, and II.A.3.iii, the  $\mathcal{O}(1)$  primitive operations for adding an entry to its pending list are charged to the entry itself. In general, an entry may be added to its pending list  $\mathcal{O}(n)$  times since an entry  $[q, i, r]$  on  $I_j$  for which  $i = j$  may be added to  $H$ ,  $\mathcal{O}(n)$  times.

For step II.A.3 of Algorithm 3.1.1, the  $\mathcal{O}(1)$  primitive operations for evaluating  $f_r$  are charged to the entry being processed.

ALGORITHM 3.3.1 *Calculate the  $R_l$ 's*

Let  $[q, i, r]$  on  $I_j$  be the entry for which the  $R_l$ 's are to be calculated for the action **reduce**  $p$ .

I. Let  $l = 1$ ,  $m = |\text{RHS}(p)|$ , and  $R_0 = \{([q, i, r], j)\}$ .

II. While  $l \leq m$ , perform the following steps:

A. Initialize  $R_l$  and get the first element  $([t, h, ?], ?)$  from  $R_{l-1}$ .

B. While  $R_{l-1}$  is not exhausted, perform the following steps:

1. Get the first entry  $[s, o, v]$  from parse list  $I_h$ .

2. While  $I_h$  is not exhausted, perform the following step:

i. If  $v = t$  then add the element  $([s, o, v], h)$  to  $R_l$ , if the element is not already in  $R_l$ .

ii. Get the next entry  $[s, o, v]$  from parse list  $I_h$ .

3. Get the next element  $([t, h, ?], ?)$  from  $R_{l-1}$ .

For steps II.A.3.i, II.A.3.ii, and II.A.3.iii of Algorithm 3.1.1, the primitive operations must be carefully determined. Algorithm 3.3.1 is a straight forward implementation of step II.A.3.i in which each set  $R_l$  is represented by a linked list of elements. Algorithm 3.3.1 suffers from poor space and time complexities because the number of elements that can be in a set  $R_l$  is  $\mathcal{O}(n^2)$  and the algorithm has loops which access each element in each  $R_l$ . Fortunately, Algorithm 3.3.1 can be improved by using the sets  $B_{-1}, B_0, B_1, \dots, B_m$  which are defined as follows:

$$B_{-1} = \{(r, j)\}$$

$$B_0 = \{(q, i)\}$$

$$B_l = \{(s, o) \mid ([s, o, v], h) \in R_l\}$$

The  $B_l$ 's can be calculated recursively using the following formula, obtained by substitution from the formula for the  $R_l$ 's.

$$B_l = \{(s, o) \mid [s, o, v] \in I_h \text{ and } (v, h) \in B_{l-1}\}$$

The set  $R_m$  can be replaced in steps II.A.3.ii and II.A.3.iii of Algorithm 3.1.1 because of the following simple relationship

$$B_{l-1} = \{(v, h) \mid ([s, o, v], h) \in R_l\}.$$

ALGORITHM 3.3.2 *Calculate the  $B_l$ 's*

Let  $[q, i, r]$  on  $I_j$  be the entry for which the  $B_l$ 's are to be calculated for the action **reduce**  $p$ .

- I. Let  $l = 1$ ,  $m = |\text{RHS}(p)|$ ,  $B_{-1} = (r, j)$ , and  $B_0 = (q, i)$ .
- II. While  $l < m$ , perform the following steps:
  - A. Initialize  $B_l$  and get the first element  $(t, h)$  from  $B_{l-1}$ .
  - B. While  $B_{l-1}$  is not exhausted, perform the following steps:
    1. Get the first entry  $[s, o, v]$  on  $I_h$ .
    2. While  $I_h$  is not exhausted, perform the following steps:
      - i. If  $v = t$  then add the element  $(s, o)$  to  $B_l$ , if it is not already in  $B_l$ .
      - ii. Get the next entry  $[s, o, v]$  on  $I_h$ .
    3. Get the next element  $(t, h)$  from  $B_{l-1}$ .
  - C. Let  $l = l + 1$ .

This relationship follows immediately from the recursive formula for the  $R_l$ 's. Thus, step II.A.3.i of Algorithm 3.1.1 can compute the  $B_l$ 's and steps II.A.3.ii and II.A.3.iii can use  $B_{m-1}$  instead of  $R_m$ .

Algorithm 3.3.2 computes the sets  $B_{-1}, B_0, \dots, B_{m-1}$  and is very similar to Algorithm 3.3.1. However, the number of elements that can be in a set  $B_l$  is  $\mathcal{O}(n)$ , so Algorithm 3.3.2 has better time and space complexities than Algorithm 3.3.1.

For Algorithm 3.3.2, the sets  $B_0, B_1, \dots, B_m$  are organized as an array of sets and each set is an (initially empty) linked list of elements. Each set can have at most  $\mathcal{O}(n)$  elements because the elements are not duplicated and the only component of an element that is not bounded independently of  $n$  is the parse list number and it is bounded by  $n + 1$ . The number of sets is  $\mathcal{O}(1)$ , so the space complexity of Algorithm 3.3.2 is  $\mathcal{O}(n)$  and the space complexity of Algorithm 3.1.1 is not increased.

The time complexity of Algorithm 3.3.2 is not determined directly. Instead, the primitive operations used in the steps of Algorithm 3.3.2 are charged to the objects of Algorithm 3.1.1 since Algorithm 3.3.2 is step II.A.3.i of Algorithm 3.1.1.

For step I of Algorithm 3.3.2, its  $\mathcal{O}(1)$  primitive operations are charged to the

entry being processed.

The loop in step II of Algorithm 3.3.2 is executed at most  $\mathcal{O}(1)$  times since  $m$ , the length of a production, is independent of  $n$ . The  $\mathcal{O}(1)$  primitive operations for the loop termination test in step II are charged to the entry being processed by step II.A of Algorithm 3.1.1. Likewise, the  $\mathcal{O}(1)$  primitive operations for steps II.A and II.C are charged to the entry being processed by step II.A of Algorithm 3.1.1.

The loop in step II.B of Algorithm 3.3.2 is executed at most  $\mathcal{O}(n)$  times since  $B_l$  has  $\mathcal{O}(n)$  elements. Thus, the  $\mathcal{O}(n)$  primitive operations for the loop termination test in step II.B are charged to the entry being processed by step II.A of Algorithm 3.1.1. Likewise, the  $\mathcal{O}(n)$  primitive operations for steps II.B.1 and II.B.3 are charged to the entry being processed by step II.A of Algorithm 3.1.1.

The loop in step II.B.2 of Algorithm 3.3.2 is executed at most  $\mathcal{O}(n)$  times since  $I_h$  has  $\mathcal{O}(n)$  elements. Step II.B.2 is also nested inside the loop formed by step II.B. Thus, the  $\mathcal{O}(n^2)$  primitive operations for the loop termination test in step II.B.2 are charged to the entry being processed by step II.A of Algorithm 3.1.1. Likewise, the  $\mathcal{O}(n^2)$  primitive operations for step II.B.2.ii are charged to the entry being processed by step II.A of Algorithm 3.1.1.

Step II.B.2.i of Algorithm 3.3.2 takes  $\mathcal{O}(n)$  primitive operations since it must check all the elements in  $B_l$ . Step II.B.2.i is also nested within the loops formed by steps II.B and II.B.2. Therefore, the  $\mathcal{O}(n^3)$  primitive operations for step II.B.2.i are charged to the entry being processed by step II.A of Algorithm 3.1.1.

Returning to Algorithm 3.1.1 and examining step II.A.3.ii, there are  $\mathcal{O}(1)$  elements in  $g_r$  and  $\mathcal{O}(n)$  elements in  $B_{m-1}$  for which  $i \neq j$ . So the actions of this step are performed  $\mathcal{O}(n)$  times. Likewise, for step II.A.3.iii, there are  $\mathcal{O}(1)$  elements in  $g_r$  and  $\mathcal{O}(1)$  elements in  $B_{m-1}$  for which  $i = j$ . So the actions of this step are performed  $\mathcal{O}(1)$  times. Furthermore, for step II.A.3.iii, checking for  $(q, r)$  on  $T$  and adding  $(q, r)$  to  $T$  takes only  $\mathcal{O}(1)$  primitive operations.

All the primitive operations of Algorithm 3.1.1 have been charged to the objects used by the algorithm. Now the number of primitive operations for each class of objects can be calculated. Considering the entries first, each entry has the following number of primitive operations charged to it:

- $\mathcal{O}(1)$  when the entry is added to its parse list in Algorithm 3.1.1.
- $\mathcal{O}(n)$  when the entry is processed up to  $\mathcal{O}(n)$  times in step II.A of Algorithm 3.1.1.
- $\mathcal{O}(1)$  when the set  $T$  is emptied in step II.A.1 of Algorithm 3.1.1.
- $\mathcal{O}(1)$  when  $f_r$  and  $g_r$  are evaluated in step II.A.2 of Algorithm 3.1.1.

- $\mathcal{O}(n)$  when the check for a duplicate entry is made in step II.A.2 of Algorithm 3.1.1.
- $\mathcal{O}(n)$  when the entry is added up to  $\mathcal{O}(n)$  times to the pending list in steps II.A.2, A.3.ii, or II.A.3.iii of Algorithm 3.1.1.
- $\mathcal{O}(1)$  when  $f_r$  is evaluated in step II.A.3 of Algorithm 3.1.1.
- $\mathcal{O}(1)$  when the entry causes the execution of steps I, II, II.A, and II.C of Algorithm 3.3.2.
- $\mathcal{O}(n)$  when the entry causes the execution of steps II.B, II.B.1, and II.B.3 of Algorithm 3.3.2.
- $\mathcal{O}(n^2)$  when the entry causes the execution of steps II.B.2 and II.B.2.ii of Algorithm 3.3.2.
- $\mathcal{O}(n^3)$  when the entry causes the execution of step II.B.2.i of Algorithm 3.3.2.
- $\mathcal{O}(n^2)$  when the check for a duplicate entry is made  $\mathcal{O}(n)$  times in step A.3.ii or II.A.3.iii of Algorithm 3.1.1.
- $\mathcal{O}(n)$  when the check for a duplicate state pair is made  $\mathcal{O}(n)$  times in step II.A.3.iii of Algorithm 3.1.1.

Thus, the total number of primitive operations charged to an entry is  $\mathcal{O}(n^3)$ . The number of primitive operations for all the entries is  $\mathcal{O}(n^5)$  since the number of entries is  $\mathcal{O}(n^2)$ .

No primitive operations are charged to the parse lists. The number of primitive operations charged to a pending list are as follows:

- $\mathcal{O}(1)$  when the loop in step II.A of Algorithm 3.1.1 is initialized.
- $\mathcal{O}(1)$  when the loop in step II.A of Algorithm 3.1.1 is terminated.

Since the number of parse lists is  $\mathcal{O}(n)$ , the total number of primitive operations for all the parse lists is  $\mathcal{O}(n)$ .

Finally, the number of primitive operations charged to the algorithm object are as follows:

- $\mathcal{O}(n)$  for the initialization of Algorithm 3.1.1 in step I.
- $\mathcal{O}(n)$  for the loop test and increment in steps II and II.B of Algorithm 3.1.1.



Thus the total time charged to the algorithm object is  $\mathcal{O}(n)$ .

Summing the number of primitive operations for the entries, the parse lists, pending lists, and the algorithm object shows the time complexity of Algorithm 3.1.1 is  $\mathcal{O}(n^5)$ .

Using a trick from Early's algorithm, the time complexity of Algorithms 3.1.1 and 3.3.2 can be improved at the expense of increasing their best-case times. The trick is to store the  $B_i$ 's as  $n$  lists where the  $i^{\text{th}}$  list contains all the  $(q, i)$  elements. The time to initialize a set  $B_i$  is increased to  $\mathcal{O}(n)$  but the time to check for a duplicate element is decreased to  $\mathcal{O}(n)$ .

For Algorithm 3.3.2, this change means that step II.A takes  $\mathcal{O}(n)$  primitive operations to initialize  $B_i$  and step II.B.2.i takes only  $\mathcal{O}(1)$  primitive operations. Step II.B.2.i is nested within two loops that are each executed  $\mathcal{O}(n)$  times while step II.A is not. Thus, this change reduces the time complexity of Algorithm 3.3.2 to  $\mathcal{O}(n^2)$  and the time complexity of Algorithm 3.1.1 to  $\mathcal{O}(n^4)$ .

The draw back of this trick is that it forces the time complexity of Algorithm 3.3.2 to be  $\mathcal{O}(n^2)$  regardless of the grammar being used. This is not always desirable. The use of this trick by Early creates a similar situation for Early's algorithm.

The  $\mathcal{O}(n^4)$  time complexity of Algorithm 3.1.1 is greater than the  $\mathcal{O}(n^3)$  time complexity of Early's algorithm under the same conditions. Further analysis of Algorithm 3.1.1 might provide a lower upper bound on its time complexity. However, the time complexity of Algorithm 3.1.1 is not important to the goal of this dissertation since Algorithm 3.1.1 is just a step towards a more efficient syntax error recovery algorithm for  $\text{LR}(k)$  parsers. In the next chapter, a depth-first version of Algorithm 3.1.1 is developed and it is shown to have  $\mathcal{O}(n)$  time and  $\mathcal{O}(n)$  space complexities for  $\text{LR}(k)$  grammars.

## CHAPTER IV

THE DEPTH-FIRST LR( $K$ ) EARLY'S ALGORITHM

In this chapter, a depth-first version of the LR( $k$ ) Early's Algorithm is presented. This algorithm is similar to the LR( $k$ ) Early's Algorithm, but it does not process all the entries on a parse list before proceeding to the next parse list. Instead, entries on the parse lists may be processed in any order. The algorithm is depth first because it can allow an individual parse of the input string to be simulated completely before any other parse is simulated. Conceptually, this ability corresponds to similar abilities of depth-first search algorithms for graphs.

## 4.1 THE ALGORITHM

The Depth-First LR( $k$ ) Early's Algorithm, Algorithm 4.1.1, uses the same  $n + 2$  parse lists  $I_0, I_1, \dots, I_{n+2}$  as Algorithm 3.1.1. In addition, the format of the entries on the parse lists is the same. However, the pending lists  $H_0, H_1, \dots, H_{n+1}$  are combined into one pending list,  $H$ . An entry placed on  $H$  has its associated parse list number in an ordered pair  $([q, i, r], j)$ . Unlike Algorithm 3.1.1, entries are added to their parse list only after they are removed from  $H$ .

The use of the pending list,  $H$ , allows entries to be processed by Algorithm 4.1.1 in an order that is independent of the order of the parse lists. In fact, if the pending list,  $H$ , is treated as a stack then a depth-first (one complete parse at a time) simulation of the nondeterministic LR( $k$ ) parser is achieved.

Like Algorithm 3.1.1, Algorithm 4.1.1 operates by simulating the nondeterministic LR( $k$ ) parser. The simulation of shifts by step II.B.1.i of Algorithm 4.1.1 operates in a manner similar to step II.A.2 of Algorithm 3.1.1. For reductions, Algorithm 4.1.1, unlike Algorithm 3.1.1, applies a reduction to only some of the copies of the nondeterministic LR( $k$ ) parser that can reach a configuration which calls for the reduction. This is because Algorithm 4.1.1, unlike Algorithm 3.1.1, may not have simulated the moves for all copies of the nondeterministic LR( $k$ ) parser up to the configuration where the reduction occurs. Instead, Algorithm 4.1.1 must defer the application of the reduction to other copies of the nondeterministic LR( $k$ ) parser until the moves of those copies reach the configuration where the reduction

ALGORITHM 4.1.1 *Depth First LR(k) Early's Algorithm*

- I. Place  $([0, 0, 0], 0)$  on the list  $H$  and append  $\$^{k+1}$  to  $w$ .
- II. While  $H$  is not empty, perform the following steps:
  - A. Remove an entry  $([q, i, r], j)$  from  $H$ .
  - B. If the entry  $[q, i, r]$  is not already on the parse list  $I_j$ , add the entry to its parse list and perform the following steps:
    1. If there were not any entries of the form  $[?, ?, r]$  on the parse list when  $[q, i, r]$  was added then perform each of the following steps:
      - i. If **shift**  $\in f_r(w_{j+1:j+k})$  and there exists  $s \in g_r(w_{j+1:j+1})$  then add  $([r, j, s], j+1)$  to  $H$  if it is not already on  $H$ .
      - ii. If there exists **reduce**  $p \in f_r(w_{j+1:j+k})$  then let  $G_j(r) = G_j(r) \cup \{(|\text{RHS}(p)|, p, j) \mid \text{reduce } p \in f_r(w_{j+1:j+k})\}$ .
    2. If  $G_j(r)$  is not empty, perform the following steps:
      - i. Let  $m = \max(\{h \mid (h, p, o) \in G_j(r)\})$ .
      - ii. Use Algorithm 4.1.2 to compute  $B_{-1}, B_0, \dots, B_{m-1}$ .
      - iii. For each  $(h, p, l) \in G_j(r)$  and for each  $(s, o) \in B_{h-1}$  for which there exists  $t \in g_r(\text{LHS}(p))$ , add  $([s, o, t], l)$  to  $H$  if it is not already on  $H$ .

occurs.

Deferred application of reductions is complicated by the fact that duplicate entries for a parse list are processed. Elimination of duplicate entries guarantees that Algorithm 4.1.1 will terminate. However, two copies of the nondeterministic LR(k) parser can reach configurations with the same state on top of the stack at the same position in the input string. And "blind" elimination of duplicate entries would stop the simulation of one of the copies when the stacks for the two configurations are not the same. Subsequent moves for the stopped copy might include a reduction which causes the copy to enter a configuration with a different state on top of the stack than the corresponding configuration for the stopped copy. In this case, the simulation of the stopped copy should continue. The mechanism for applying deferred reductions addresses this issue.

The deferred application of reductions is handled by propagating information about a reduction backwards down the stack so that the effect of the reduction on the stack can be maintained. A copy of the nondeterministic LR(k) parser may

### ALGORITHM 4.1.2 *Calculate $B_l$ 's*

Let  $[q, i, r]$  on  $I_j$  be the entry for which the  $B_l$ 's are to be calculated and let  $m$  be as given in Algorithm 4.1.1.

- I. Let  $l = 1$ ,  $B_{-1} = \{(r, j)\}$ ,  $B_0 = \{(q, i)\}$ , and let  $G_i(q) = G_i(q) \cup \{(n, p, k) \mid (n+1, p, k) \in G_j(r) \text{ and } n > 0\}$ .
- II. While  $l < m$ , perform the following steps:
  - A. Initialize  $B_l$  and get an element  $(t, h)$  from  $B_{l-1}$ .
  - B. While  $B_{l-1}$  is not exhausted, perform the following steps:
    1. Get an entry  $[s, o, v]$  on  $I_h$ .
    2. While  $I_h$  is not exhausted, perform the following steps:
      - i. If  $v = t$  then add the element  $(s, o)$  to  $B_l$ , if it is not already in  $B_l$ , and let  $G_o(s) = G_o(s) \cup \{(n, p, k) \mid (n+1, p, k) \in G_h(t) \text{ and } n > 0\}$ .
      - ii. Get another entry  $[s, o, v]$  on  $I_h$ .
    3. Get another element  $(t, h)$  from  $B_{l-1}$ .
  - C. Let  $l = l + 1$ .

enter a configuration with the same state on top of the stack at the same position in the input string as a previous configuration that was affected by the reduction. When the entry corresponding to this configuration is added to its parse list the effect of the reduction can be applied.

The deferred reduction information is stored in a set  $G_j(r)$  for each possible state  $r$  on top of the stack at each input position  $j$ . The elements of  $G_j(r)$  are ordered triples  $(h, p, l)$ ; where  $p$  is the number of the production used in the reduction,  $h$  is an integer between 0 and  $|\text{RHS}(p)|$ , and  $l$  is the number of the parse list on which the entry resulting from the reduction is to be placed. The integer  $h$  is used to indicate how many more states must be popped from the stack before the reduction can be applied.

The deferred application of reductions is achieved by examining  $G_j(r)$  whenever an entry  $[q, i, r]$  is added to a parse list  $I_j$ . Also, when  $[q, i, r]$  corresponds to a configuration from which a reduction normally is made, step II.B.1.ii of Algorithm 4.1.1 adds deferred reduction information to  $G_j(r)$  so that the deferred re-

duction mechanism also triggers the simulation of normal reductions. When  $G_j(r)$  is not empty, reductions are simulated by step II.B.2 of Algorithm 4.1.1 which is similar to step II.A.3 of Algorithm 3.1.1. In particular, step II.B.2.ii executes Algorithm 4.1.2 which computes the sets  $B_{-1}, B_0, B_1, \dots, B_{m-1}$  and propagates deferred reduction information down the stack.

## 4.2 PROOF OF CORRECTNESS

This section shows that Algorithm 4.1.1 correctly simulates an  $LR(k)$  parser when the algorithm and the parser use the same sets of states,  $Q$ , and functions  $f_q$  and  $g_q$ . The simulation is correct, if for every entry  $[q, i, r]$  that Algorithm 4.1.1 places on a parse list  $I_j$ , there is a corresponding sequence of moves

$$(0, w_{1:n+k+1}) \vdash^* (\alpha \downarrow q, w_{i+1:n+k+1}) \vdash^+ (\alpha q r, w_{j+1:n+k+1})$$

that can be made by the  $LR(k)$  parser.

Algorithm 4.1.1 uses the pending list,  $H$ , to hold entries waiting to be processed by the algorithm. The following lemma establishes that entries added to the pending list eventually are processed by the algorithm. This guarantees that Algorithm 4.1.1 terminates. It also guarantees that if  $([q, i, r], j)$  is added to  $H$  then  $[q, i, r]$  is on  $I_j$  when Algorithm 4.1.1 terminates since step II.B of Algorithm 4.1.1 will add the entry to its parse list unless it is already on it.

**LEMMA 4.2.1** (Every Entry Added to the Pending List is Processed) *If  $([r, i, s], j)$  is added to the pending list,  $H$ , then  $([r, i, s], j)$  is eventually processed by step II.A of Algorithm 4.1.1.*

**Proof:** Since step II.A of Algorithm 4.1.1 removes entries from the pending list,  $H$ , until the list is exhausted, there only two things that can cause an entry on the list not to be processed:

- an infinite number of distinct entries are placed on  $H$ ; or
- a single entry is placed on  $H$  an infinite number of times.

An infinite number of distinct entries cannot be placed on  $H$  because the values of  $r, i, s$  and  $j$  are bounded. Also, a single entry cannot be on  $H$  more than once since, before an entry is added to  $H$ , the entry is checked to determine it is not already on the list. ■

The next lemma establishes a technical property of the initial state 0 for Algorithm 4.1.1.

LEMMA 4.2.2 (0 is the Unique Initial State) *If  $r = 0$  then  $[q, i, r]$  on  $I_j$  is  $[0, 0, 0]$  on  $I_0$ .*

Proof: The definition of an LR( $k$ ) parser does not allow  $0 \in g_s(X)$  for any  $s$  or  $X$ . Any entry  $([q, i, r], j)$  added to the pending list,  $H$ , by either step II.B.1.ii or II.B.3.ii of Algorithm 4.1.1 may have  $r \in g_s(X)$  for some  $s$  and  $X$ . Only step I, which adds  $([0, 0, 0], 0)$  to  $H$ , can add an entry  $([q, i, r], j)$  for which  $r = 0$ . Thus,  $r = 0$  implies  $[q, i, r]$  on  $I_j$  is  $[0, 0, 0]$  on  $I_0$ . ■

As in the proofs of correctness and completeness for Algorithm 3.1.1, the concept of a sequence in which entries can be added to their parse lists is important and leads to the following definition.

*Definition 4.2.1 (Ordered List of Entries)* An ordered list of entries is a complete list of entries and their parse lists

$$\begin{array}{l} [q_1, i_1, r_1] \text{ on } I_{j_1} \\ [q_2, i_2, r_2] \text{ on } I_{j_2} \\ \vdots \\ [q_N, i_N, r_N] \text{ on } I_{j_N} \end{array}$$

given in a sequence in which they can be added to their parse lists by step II.B of Algorithm 4.1.1 during an execution of Algorithm 4.1.1.

The entry  $[0, 0, 0]$  on  $I_0$  is not on any ordered list of entries since it is not added to its parse list by step II.B of Algorithm 4.1.1.

Closely related to the concept of ordering a list of entries is the notion that, regardless of any specific ordering, an entry or an entry from a set of entries must be added to its parse list before another entry can be added to a parse list.

*Definition 4.2.2 (Direct Precursor)* Given two entries  $[?, ?, r]$  on  $I_j$  and  $[r, j, s]$  on  $I_l$ , the entry  $[?, ?, r]$  on  $I_j$  is said to be a direct precursor of the entry  $[r, j, s]$  on  $I_l$ .

The following lemma shows that every entry has a precursor which is on a parse list when the entry is itself added to its parse list by Algorithm 4.1.1.

LEMMA 4.2.3 (Every Entry Has a Direct Precursor) *If the entry  $[q, i, r]$  is on  $I_j$  then there is an entry  $[?, ?, q]$  on a parse list  $I_l$ .*

Proof: The lemma is trivially true for the entry  $[0, 0, 0]$  on  $I_0$ . For all other entries, the lemma is proved by induction on an ordered list of entries, using the lemma as the induction hypothesis. The induction proceeds in two steps:

- first, the lemma is proved for the first entry on the ordered list of entries; and
- second, the lemma is proved for the  $N^{\text{th}}$  entry on the ordered list of entries, assuming it holds for all entries before the  $N^{\text{th}}$  entry on the ordered list of entries.

For the first induction step, let  $[q, i, r]$  on  $I_i$  be the first entry on the ordered list of entries. This entry is added to its parse list by step II.B of Algorithm 4.1.1 so  $([q, i, r], j)$  must have been on  $H$ . The entry  $([q, i, r], j)$  must be added to  $H$  while either step II.B.1.i or step II.B.2.iii of Algorithm 4.1.1 is processing  $[0, 0, 0]$  on  $I_0$ . The entry  $[0, 0, 0]$  on  $I_0$  is the only entry on any parse list when it is processed. Examining steps II.B.1.i and II.B.2 and Algorithm 4.1.2 shows that  $[q, i, r] = [0, 0, r]$  and  $[0, 0, 0]$  is a precursor for the entry.

For the second induction step, the lemma is assumed to hold for all entries before the  $N^{\text{th}}$  entry on the ordered list of entries. Let the  $N^{\text{th}}$  entry be  $[q, i, r]$  on  $I_j$ . This entry is added to its parse list by step II.B of Algorithm 4.1.1 so  $([q, i, r], j)$  must have been on  $H$ . Therefore, there are two cases to consider:

- step II.B.1.i adds  $([q, i, r], j)$  to  $H$  and Algorithm 4.1.1 is simulating a shift by the  $\text{LR}(k)$  parser; or
- step II.B.2.iii adds  $([q, i, r], j)$  to  $H$  and Algorithm 4.1.1 is simulating a reduction by the  $\text{LR}(k)$  parser.

In the first case, step II.B.1.i adds  $([q, i, r], j)$  to  $H$  and this step must be processing an entry  $[?, ?, q]$  on  $I_i$ .

In the second case, step II.B.2.iii adds  $([q, i, r], j)$  to  $H$ . This implies that  $(q, i) \in B_{m-1}$  for some  $m \geq 0$ . Examining Algorithm 4.1.2 shows that if  $(q, i) \in B_{m-1}$  then, when  $m = 0$ ,  $[?, ?, q]$  is on  $I_i$  and, when  $m > 0$ ,  $[q, i, ?]$  is on  $I_j$ . Since  $[q, i, ?]$  on  $I_j$  precedes  $[q, i, r]$  on  $I_j$  on the ordered list of entries, the induction hypothesis can be applied to show that there must be an entry  $[?, ?, q]$  on  $I_i$  when  $m > 0$ . Thus, regardless of the value of  $m$ , there is an entry  $[?, ?, q]$  on  $I_i$ . ■

The next two lemmas establish precursor relationships that can be inferred among entries on their parse lists at key points during the execution of Algorithm 4.1.1.

LEMMA 4.2.4 *If Algorithm 4.1.2 is applied to  $[q_1, l_1, q_0]$  on  $I_{l_0}$  and  $(q_m, l_m) \in B_{m-1}$  then there exists*

$$\begin{aligned} & [q_{m+1}, l_{m+1}, q_m] \text{ on } I_{l_m} \\ & [q_m, l_m, q_{m-1}] \text{ on } I_{l_{m-1}} \\ & \vdots \\ & [q_2, l_2, q_1] \text{ on } I_{l_1}. \end{aligned}$$

Proof: The lemma is proved by induction on  $m$ , using the lemma as the induction hypothesis. The induction proceeds in two steps:

- first, the lemma is proved for  $m = 0$  and  $m = 1$ ; and
- second, the lemma is proved for  $m = N$ , where  $N > 1$ , assuming it holds when  $m = N - 1$ .

For the first induction step,  $m = 0$  or  $m = 1$ . Examination of step I of Algorithm 4.1.2 shows that  $(q_0, l_0) \in B_{-1}$  or  $(q_1, l_1) \in B_0$  only if  $[q_1, l_1, q_0]$  is on  $I_{l_0}$ .

For the second induction step,  $m = N$ , where  $N > 1$ , and the lemma is assumed to hold for  $m = N - 1$ . If  $(q_N, l_N) \in B_{N-1}$  then step II.B.1.i of Algorithm 4.1.2 must have added  $(q_N, l_N)$  to  $B_{N-1}$  and there must be an entry  $[q_N, l_N, q_{N-1}]$  on  $I_{l_{N-1}}$  and an element  $(q_{N-1}, l_{N-1}) \in B_{N-2}$ . Since  $(q_{N-1}, l_{N-1}) \in B_{N-2}$ , the induction hypothesis can be applied to show that there exists

$$\begin{aligned} & [q_N, l_N, q_{N-1}] \text{ on } I_{l_{N-1}} \\ & [q_{N-1}, l_{N-1}, q_{N-2}] \text{ on } I_{l_{N-2}} \\ & \vdots \\ & [q_2, l_2, q_1] \text{ on } I_{l_1}. \end{aligned}$$

Applying Lemma 4.2.3 to  $[q_N, l_N, q_{N-1}]$  on  $I_{l_{N-1}}$  shows that  $[q_{N+1}, l_{N+1}, q_N]$  is on  $I_{l_N}$ . ■

LEMMA 4.2.5 *If  $(h, p, l_m) \in G_{l_h}(q_h)$ , there exists*

$$\begin{aligned} & [q_{h-1}, l_{h-1}, q_h] \text{ on } I_{l_h} \\ & [q_h, l_h, q_{h+1}] \text{ on } I_{l_{h+1}} \\ & \vdots \\ & [q_{m-1}, l_{m-1}, q_m] \text{ on } I_{l_m} \end{aligned}$$

where reduce  $p \in f_{q_m}(w_{l_{m+1} \cdot l_{m+k}})$  and  $m = |RHS(p)|$ .



Proof: The lemma is proved by induction on  $m - h$ , using the lemma as the induction hypothesis. The induction proceeds in two steps:

- first, the lemma is proved for  $m - h = 0$ ; and
- second, the lemma is proved for  $m - h = N$ , where  $N > 0$ , assuming it holds for  $m - h = N - 1$ .

For the first induction step,  $m - h = 0$ . Thus,  $h = m = |\text{RHS}(p)|$ . Examination of step II.B.1.ii of Algorithm 4.1.1 shows that it adds  $(h, p, l_h)$ , where  $h = m = |\text{RHS}(p)|$ , to  $G_{l_h}(q_h)$  only if  $[q_{m-1}, l_{m-1}, q_m]$  on  $I_{l_m}$  and **reduce**  $p \in f_{q_m}(w_{l_{m+1}:l_{m+k}})$ . No other step of either Algorithm 4.1.1 or Algorithm 4.1.2 can add an element  $(h, p, l_h)$  to  $G_{l_h}(q_h)$  for which  $h = |\text{RHS}(p)|$ , particularly step II.B.2.i of Algorithm 4.1.2.

For the second induction step,  $m - h = N$ , where  $N > 0$ , and the lemma is assumed to hold when  $m - h \leq N - 1$ . An element  $(h, p, l_m)$  for which  $m - h = N$  and  $N > 0$ , can be added by step I or step II.B.1.ii of Algorithm 4.1.2. If either step I or step II.B.2.i of Algorithm 4.1.2 adds  $(h, p, l_m)$  to  $G_{l_h}(q_h)$  then there is an entry  $[q_h, l_h, q_{h+1}]$  on  $I_{l_{h+1}}$  and  $(h + 1, p, l_m) \in G_{l_{h+1}}(q_{h+1})$ . Applying the induction hypothesis to  $(h + 1, p, l_m) \in G_{l_{h+1}}(q_{h+1})$ , for which  $m - h = N - 1$ , there must be

$$\begin{aligned} & [q_h, l_h, q_{h+1}] \text{ on } I_{l_{h+1}} \\ & [q_{h+1}, l_{h+1}, q_{h+2}] \text{ on } I_{l_{h+2}} \\ & \vdots \\ & [q_{m-1}, l_{m-1}, q_m] \text{ on } I_{l_m} \end{aligned}$$

where **reduce**  $p \in f_{q_m}(w_{l_{m+1}:l_{m+k}})$ . Also, Lemma 4.2.3 shows that there is an entry  $[q_{h-1}, l_{h-1}, q_h]$  on  $I_{l_h}$ . Thus, there exists

$$\begin{aligned} & [q_{h-1}, l_{h-1}, q_h] \text{ on } I_{l_h} \\ & [q_h, l_h, q_{h+1}] \text{ on } I_{l_{h+1}} \\ & \vdots \\ & [q_{m-1}, l_{m-1}, q_m] \text{ on } I_{l_m} \end{aligned}$$

where **reduce**  $p \in f_{q_m}(w_{l_{m+1}:l_{m+k}})$ . ■

#### THEOREM 4.2.1 (Algorithm 4.1.1 Correctly Simulates the LR( $k$ ) Parser)

Given the same  $Q$ ,  $f_q$ , and  $g_q$  for Algorithm 4.1.1 and the LR( $k$ ) parser, if an entry  $[r, i, s]$  is added to parse list  $I$ , (except for  $[0, 0, 0]$  on  $I_0$ ) by Algorithm 4.1.1 then the LR( $k$ ) parser can make the sequence of moves

$$(0, w_{1:n+k+1}) \vdash^* (\alpha(r, w_{i+1:n+k+1})) \vdash^+ (\alpha r s, w_{j+1:n+k+1}).$$

Proof: The theorem is proved by induction on any ordered list of entries, using the theorem as the induction hypothesis. The induction proceeds in two steps:

- first, the theorem is proved for the first entry on the ordered list of entries; and
- second, the theorem is proved for the  $N^{\text{th}}$  entry on the ordered list of entries, assuming it holds for all entries that precede the  $N^{\text{th}}$  entry.

For the first induction step, let the first entry on the ordered list of entries be  $[q, i, r]$  on  $I_j$ . Step II.B of Algorithm 4.1.1 adds  $[q, i, r]$  to  $I_j$  so  $([q, i, r], j)$  must have been on  $H$ . Either step II.B.1.i or step II.B.2.iii of Algorithm 4.1.1 must have added  $([q, i, r], j)$  to  $H$  while processing  $[0, 0, 0]$  on  $I_0$ . Likewise, the first move of the  $\text{LR}(k)$  parser must be from the configuration  $(0, w_{1:n+k+1})$ . Thus, if step II.B.1.i adds  $([q, i, r], j) = ([0, 0, r], 1)$  to  $H$  then the shift  $(0, w_{1:n+k+1}) \vdash (0r, w_{2:n+k+1})$  can be made by the  $\text{LR}(k)$  parser; and if step II.B.2.iii adds  $([q, i, r], j) = ([0, 0, r], 0)$  to  $H$  then the reduction  $(0, w_{1:n+k+1}) \vdash (0r, w_{1:n+k+1})$  can be made by the  $\text{LR}(k)$  parser.

For the second induction step, the theorem is assumed to hold for all entries on the ordered list of entries that precede the  $N^{\text{th}}$  entry. Let the  $N^{\text{th}}$  entry be  $[r, i, s]$  on  $I_j$ . Since  $[r, i, s]$  is added to  $I_j$  by step II.B of Algorithm 4.1.1,  $([r, i, s], j)$  must have been on  $H$ . There are two cases to consider:

- step II.B.1.i adds  $([r, i, s], j)$  to  $H$  and Algorithm 4.1.1 is simulating a shift by the  $\text{LR}(k)$  parser; or
- step II.B.2.iii adds  $([r, i, s], j)$  to  $H$  and Algorithm 4.1.1 is simulating a reduction by the  $\text{LR}(k)$  parser.

In either case, the step involved must have been applied to an entry  $[q_{h-1}, l_{h-1}, q_h]$  on  $I_{l_h}$  where  $h$  is a convenient index which will be specified later. Since  $[q_{h-1}, l_{h-1}, q_h]$  on  $I_{l_h}$  precedes  $[r, i, s]$  on  $I_j$  on the ordered list of entries, the  $\text{LR}(k)$  parser can make the sequence of moves

$$(0, w_{1:n+k+1}) \vdash^* (\alpha q_h, w_{l_h+1:n+k+1}).$$

In the first case, step II.B.1.i adds  $([r, i, s], j)$  to  $H$  and Algorithm 4.1.1 is simulating a shift by the  $\text{LR}(k)$  parser. Thus,  $l_h = i = j - 1$ ,  $q_h = r$ , and the shift

$(\alpha r, w_{j:n+k+1}) \vdash (\alpha r s, w_{j+1:n+k+1})$  can be made by the  $\text{LR}(k)$  parser. Therefore, the  $\text{LR}(k)$  parser can make the sequence of moves

$$(0, w_{1:n+k+1}) \vdash^* (\alpha r, w_{j:n+k+1}) \vdash (\alpha r s, w_{j+1:n+k+1}).$$

In the second case, step II.B.2.iii adds  $([r, i, s], j)$  to  $H$  and Algorithm 4.1.1 is simulating a reduction by the  $\text{LR}(k)$  parser. When  $[q_{h-1}, l_{h-1}, q_h]$  on  $I_{l_h}$  is processed there must be

$$(h, p, j) \in G_{l_h}(q_h)$$

and

$$(q_0, l_0) \in B_{h-1}$$

where  $q_0 = r$  and  $l_0 = i$ . Here,  $h$  has been chosen for convenience as the index into the  $B_i$ 's. Applying Lemma 4.2.4, there must be

$$\begin{aligned} & [q_0, l_0, q_1] \text{ on } I_{l_1} \\ & [q_1, l_1, q_2] \text{ on } I_{l_2} \\ & \vdots \\ & [q_{h-2}, l_{h-2}, q_{h-1}] \text{ on } I_{l_{h-1}}. \end{aligned}$$

Applying Lemma 4.2.5, there must also be

$$\begin{aligned} & [q_h, l_h, q_{h+1}] \text{ on } I_{l_{h+1}} \\ & [q_{h+1}, l_{h+1}, q_{h+2}] \text{ on } I_{l_{h+2}} \\ & \vdots \\ & [q_{m-2}, l_{m-2}, q_{m-1}] \text{ on } I_{l_{m-1}} \\ & [q_{m-1}, l_{m-1}, q_m] \text{ on } I_{l_m} \end{aligned}$$

where  $l_m = j$ ,  $m = |\text{RHS}(p)|$  and **reduce**  $p \in f_{q_m}(w_{l_m+1:l_m+k})$ . The reduction must be one of three possible types, each of which must be considered separately:

- a reduction by an empty production (i.e.  $m = 0$ );
- a reduction by a non-empty production which does not cause the stack to underflow (i.e.  $m > 0$  and  $q_x \neq 0$  for  $0 < x \leq m$ ); or
- a reduction by a non-empty production which causes the stack to underflow (i.e.  $m > 0$  and  $q_x = 0$  for some  $x$  where  $0 < x \leq m$ ).

For the first type of reduction, an empty production is used so  $m = h = 0$ . Thus,  $l_m = l_0 = i = j$ ,  $q_m = q_0 = r$ , and the reduction

$$(\alpha r, w_{j+1:n+k+1}) \vdash (\alpha r s, w_{j+1:n+k+1})$$

can be made by the  $\text{LR}(k)$  parser. Therefore, the  $\text{LR}(k)$  parser can make the sequence of moves

$$(0, w_{1:n+k+1}) \stackrel{*}{\vdash} (\alpha \wr r, w_{j+1:n+k+1}) \vdash (\alpha r s, w_{j+1:n+k+1}).$$

For the second type of reduction, a non-empty production is used and the stack does not underflow so  $m > 0$  and  $q_x \neq 0$  for  $0 < x \leq m$ . This implies that the entries  $[q_{x-1}, l_{x-1}, q_x]$  on  $I_{l_x}$ , for  $0 < x \leq m$ , are entries on the ordered list of entries and that these entries precede  $[r, i, s]$  on  $I_j$  on the ordered list. Applying the induction hypothesis to each of these entries, the  $\text{LR}(k)$  parser can make the sequences of moves:

$$\begin{aligned} (0, w_{1:n+k+1}) &\stackrel{*}{\vdash} (\alpha_0 \wr r, w_{i+1:n+k+1}) \stackrel{+}{\vdash} (\alpha_0 r q_1, w_{l_1+1:n+k+1}) \\ (0, w_{1:n+k+1}) &\stackrel{*}{\vdash} (\alpha_1 \wr q_1, w_{l_1+1:n+k+1}) \stackrel{+}{\vdash} (\alpha_1 q_1 q_2, w_{l_2+1:n+k+1}) \\ &\vdots \\ (0, w_{1:n+k+1}) &\stackrel{*}{\vdash} (\alpha_{m-1} \wr q_{m-1}, w_{l_{m-1}+1:n+k+1}) \stackrel{+}{\vdash} (\alpha_{m-1} q_{m-1} q_m, w_{j+1:n+k+1}) \end{aligned}$$

As a result, the  $\text{LR}(k)$  parser can make the sequence of moves

$$(0, w_{1:n+k+1}) \stackrel{*}{\vdash} (\alpha_0 \wr r, w_{i+1:n+k+1}) \stackrel{+}{\vdash} (\alpha_0 r q_1 \dots q_m, w_{j+1:n+k+1}).$$

Also, step II.B.2.iii calls for a reduction of length  $m$  so the reduction

$$(\alpha_0 r q_1 \dots q_m, w_{j+1:n+k+1}) \vdash (\alpha_0 r s, w_{j+1:n+k+1})$$

can be made by the  $\text{LR}(k)$  parser. Therefore the  $\text{LR}(k)$  parser can make the sequence of moves

$$(0, w_{1:n+k+1}) \stackrel{*}{\vdash} (\alpha_0 \wr r, w_{i+1:n+k+1}) \stackrel{+}{\vdash} (\alpha_0 r s, w_{j+1:n+k+1}).$$

For the third type of reduction, a reduction by non-empty production is used and the stack underflows so  $m > 0$  and  $q_x = 0$  for some  $x$  where  $0 < x \leq 0$ . Let  $e$  be the greatest such  $x$ . Since  $1 \leq e$ ,  $r = q_e = 0$ . Recursively applying Lemma 4.2.2

shows that, for  $0 < x \leq e$ ,  $([q_{x-1}, l_{x-1}, q_x], l_x) = ([0, 0, 0], 0)$ . If  $e = m$  then there is only the state 0 on the stack when the reduction is made and  $q_m = q_0 = 0$  and  $l_m = l_0 = i = j = 0$ . Therefore, the  $\text{LR}(k)$  parser can make the sequence of moves

$$(0, w_{1:n+k+1}) \vdash^* (0, w_{1:n+k+1}) \vdash^+ (0s, w_{1:n+k+1}).$$

If  $e < m$  there are some states (but not enough) on the stack when the reduction is applied. For  $e < x \leq m$ ,  $[q_{x-1}, l_{x-1}, q_x]$  on  $I_{l_x}$  is an entry that precedes  $[r, i, s]$  on  $I_j$  on the ordered list of entries. Applying the induction hypothesis to each of these entries, the  $\text{LR}(k)$  parser can make the sequences of moves:

$$\begin{aligned} (0, w_{1:n+k+1}) &\vdash^* (\alpha_e \wr q_e, w_{l_e+1:n+k+1}) \vdash^+ (\alpha_e q_e q_{e+1}, w_{l_{e+1}+1:n+k+1}) \\ (0, w_{1:n+k+1}) &\vdash^* (\alpha_{e+1} \wr q_{e+1}, w_{l_{e+1}+1:n+k+1}) \vdash^+ (\alpha_{e+1} q_{e+1} q_{e+2}, w_{l_{e+2}+1:n+k+1}) \\ &\vdots \\ (0, w_{1:n+k+1}) &\vdash^* (\alpha_{m-1} \wr q_{m-1}, w_{l_{m-1}+1:n+k+1}) \vdash^+ (\alpha_{m-1} q_{m-1} q_m, w_{j+1:n+k+1}). \end{aligned}$$

Since  $q_e = 0$  and 0 is never pushed onto the stack,  $\alpha_e = \epsilon$ . As a result, the  $\text{LR}(k)$  parser can make the sequence of moves

$$(0, w_{1:n+k+1}) \vdash^* (0, w_{1:n+k+1}) \vdash^+ (0q_{e+1}q_{e+2} \dots q_m, w_{j+1:n+k+1}).$$

Also, step II.B.2.iii calls for a reduction of length  $m$  so the reduction

$$(0q_{e+1}q_{e+2} \dots q_m, w_{j+1:n+k+1}) \vdash (0s, w_{j+1:n+k+1})$$

can be made by the  $\text{LR}(k)$  parser. Therefore the  $\text{LR}(k)$  parser can make the sequence of moves

$$(0, w_{1:n+k+1}) \vdash^* (0, w_{1:n+k+1}) \vdash^+ (0s, w_{j+1:n+k+1}).$$

■

### 4.3 PROOF OF COMPLETENESS

This section shows that Algorithm 4.1.1 completely simulates the  $\text{LR}(k)$  parser. The simulation is complete if for every sequence of moves

$$(0, w_{1:n+k+1}) \vdash^* (\alpha \wr q, w_{i+1:n+k+1}) \vdash^+ (\alpha q r, w_{j+1:n+k+1})$$

that can be made by the  $LR(k)$  parser, Algorithm 4.1.1 places the entry  $[q, i, r]$  on the parse list  $I_j$ .

The next two lemmas establish reduction information that can be inferred from the entries on their parse lists at key points during the execution of Algorithm 4.1.1.

**LEMMA 4.3.1** *If Algorithm 4.1.2 is applied to  $[q_1, l_1, q_0]$  on  $I_{l_0}$  and there exists*

$$\begin{aligned} & [q_{m+1}, l_{m+1}, q_m] \text{ on } I_{l_m} \\ & [q_m, l_m, q_{m-1}] \text{ on } I_{l_{m-1}} \\ & \vdots \\ & [q_2, l_2, q_1] \text{ on } I_{l_1} \end{aligned}$$

*then, for  $m \geq 0$ ,*

$$(q_m, l_m) \in B_{m-1}.$$

**Proof:** The lemma is proved by induction on  $m$ , using the lemma as the induction hypothesis. The induction proceeds in two steps:

- first, the lemma is proved for  $m = 0$  and  $m = 1$ ; and
- second, the lemma is proved for  $m = N$  when  $N > 1$ , assuming it holds when  $m = N - 1$ .

For the first induction step,  $m = 0$  or  $m = 1$ . Examination of step I of Algorithm 4.1.2 shows that  $(q_0, l_0) \in B_{-1}$  and  $(q_1, l_1) \in B_0$ .

For the second induction step,  $m = N$  where  $N > 1$  and the lemma is assumed to hold for  $m = N - 1$ . The induction hypothesis can be applied to

$$\begin{aligned} & [q_N, l_N, q_{N-1}] \text{ on } I_{l_{N-1}} \\ & [q_{N-1}, l_{N-1}, q_{N-2}] \text{ on } I_{l_{N-2}} \\ & \vdots \\ & [q_2, l_2, q_1] \text{ on } I_{l_1} \end{aligned}$$

to show  $(q_{N-1}, l_{N-1}) \in B_{N-2}$ . Since  $(q_{N-1}, l_{N-1}) \in B_{N-2}$  and  $[q_{N+1}, l_{N+1}, q_N]$  on  $I_{l_N}$ , step II.B.1.i of Algorithm 4.1.2 adds  $(q_N, l_N) \in B_{N-1}$ . ■

**LEMMA 4.3.2** *If  $m > 0$ ,  $h > 0$ ,  $m+h = |RHS(p)|$ , **reduce**  $p \in f_{q_m}(w_{l_m+1:l_m+k})$ , and there exists*

$$\begin{aligned} & [q_0, l_0, q_1] \text{ on } I_{l_1} \\ & [q_1, l_1, q_2] \text{ on } I_{l_2} \\ & \vdots \\ & [q_{m-1}, l_{m-1}, q_m] \text{ on } I_{l_m} \end{aligned}$$

then

$$(h, p, l_m) \in G_{l_0}(q_0).$$

Proof: The lemma is proved by induction on  $m$ , using the lemma as the induction hypothesis. The induction proceeds in two steps:

- first, the lemma is proved for  $m = 1$ ; and
- second, the lemma is proved for  $m = N$ , assuming it holds when  $m < N$ .

For the first induction step,  $m = 1$ . The entry  $[q_0, l_0, q_1]$  is on  $I_{l_1}$  so it must have been processed by step II.B.1.ii of Algorithm 4.1.1. Since **reduce**  $p \in f_{q_1}(w_{l_1+1:l_1+k})$ , step II.B.1.ii adds  $(h+1, p, l_1)$  to  $G_{l_1}(q_1)$ . Thus, step II.B.2.ii executes Algorithm 4.1.2 and step I of Algorithm 4.1.2 adds  $(h, p, l_1)$  to  $G_{l_0}(q_0)$  since  $h > 0$ .

For the second induction step,  $m = N$  and the lemma is assumed to hold for  $m < N$ . Since  $m = N$  there are  $N$  entries

$$\begin{aligned} & [q_0, l_0, q_1] \text{ on } I_{l_1} \\ & [q_1, l_1, q_2] \text{ on } I_{l_2} \\ & \vdots \\ & [q_{N-1}, l_{N-1}, q_N] \text{ on } I_{l_N}. \end{aligned}$$

Let  $k$  be the index of the entry  $[q_{k-1}, l_{k-1}, q_k]$  on  $I_{l_k}$  which is the last of these entries to be processed by step II.B of Algorithm 4.1.1. There must be  $(h+k, p, l_m) \in G_{l_k}(q_k)$  when  $[q_{k-1}, l_{k-1}, q_k]$  on  $I_{l_k}$  is processed by step II.B since the induction hypothesis can be applied to the  $N - k$  entries

$$\begin{aligned} & [q_k, l_k, q_{k+1}] \text{ on } I_{l_{k+1}} \\ & [q_{k+1}, l_{k+1}, q_{k+2}] \text{ on } I_{l_{k+2}} \\ & \vdots \\ & [q_{N-1}, l_{N-1}, q_N] \text{ on } I_{l_N}. \end{aligned}$$

Since  $(h+k, p, l_m) \in G_{l_k}(q_k)$  when  $[q_{k-1}, l_{k-1}, q_k]$  on  $I_{l_k}$  is processed by step II.B.2.ii of Algorithm 4.1.1, Algorithm 4.1.2 is executed. Step I of Algorithm 4.1.1 initializes  $B_{-1}$  to  $(q_k, l_k)$  and  $B_0$  to  $(q_{k-1}, l_{k-1})$  and adds  $(h+k-1, p, l_m)$  to  $G_{l_{k-1}}(q_{k-1})$ . The loop in step II of Algorithm 4.1.2 must iterate through at least the sequence  $o = 1, 2, \dots, k-1$  and it has the following property:

If  $(q_x, l_x) \in B_{o-1}$ ,  $(n+1, p, l_m) \in G_{l_x}(q_x)$ , and  $[q_{x-1}, l_{x-1}, q_x]$  is on  $I_{l_x}$  then  $(q_{x-1}, l_{x-1})$  is added to  $B_o$  and  $(n, p, l_m)$  is added to  $G_{l_{x-1}}(q_{x-1})$ .

Therefore, Algorithm 4.1.2 must add  $(h, p, l_m)$  to  $G_{l_0}(q_0)$ . ■

**THEOREM 4.3.1** (Algorithm 4.1.1 Completely Simulates the  $LR(k)$  Parser)  
*Given the same  $Q$ ,  $f_q$ , and  $g_q$  for the  $LR(k)$  parser and Algorithm 4.1.1, if the sequence of moves*

$$(0, w_{1:n+k+1}) \stackrel{L}{\vdash} (\alpha r, w_{i+1:n+k+1}) \stackrel{M+1}{\vdash} (\alpha r s, w_{j+1:n+k+1})$$

*can be made by the  $LR(k)$  parser, where  $L \geq 0$  and  $M \geq 0$ , then Algorithm 4.1.1 adds the entry  $[r, i, s]$  to parse list  $I_j$ .*

**Proof:** The theorem is proved by induction on the sum of  $L$  and  $M$ . The induction proceeds in three steps:

- first, the theorem is proved for  $L + M = 0$ ;
- second, the theorem is proved for  $L = N$  and  $M = 0$ , assuming it holds whenever  $L + M < N$ ; and
- third, the theorem is proved for  $L + M = N$  when  $M > 0$ , assuming it holds whenever  $L + M < N$ .

The parameter space formed by  $L$  and  $M$  is an infinite table with  $L$  as the row number and  $M$  as the column number, so the steps of the induction can be viewed as proving the theorem diagonal by diagonal, using the diagonals that run from the lower left to upper right sides of the table.

For the first induction step,  $L + M = 0$  and the theorem can be written as follows:

If  $(0, w_{1:n+k+1}) \vdash (0s, w_{j+1:n+k+1})$  then  $[0, 0, s]$  is added to  $I_j$ .

The move  $(0, w_{1:n+k+1}) \vdash (0s, w_{j+1:n+k+1})$  can be either a shift, in which case  $j = 1$ , or a reduction, in which case  $j = 0$ . When  $([0, 0, 0], 0)$  is processed by step II.B of Algorithm 4.1.1, if the move is a shift, then step II.B.1.i of Algorithm 4.1.1 adds  $([0, 0, s], 1)$  to  $H$ . Likewise, if the move is a reduction, then step II.B.2.iii of Algorithm 4.1.1 adds  $([0, 0, s], 0)$  to  $H$ .

For the second induction step,  $L = N$ ,  $M = 0$ , and the theorem is assumed to hold for  $L + M < N$ . Since  $M = 0$ , the theorem can be written as follows:

If  $(0, w_{1:n+k+1}) \stackrel{N}{\vdash} (\alpha r, w_{i+1:n+k+1}) \vdash (\alpha r s, w_{j+1:n+k+1})$ , where  $N \geq 0$ , then the entry  $[r, i, s]$  is added to parse list  $I_j$ .



If  $N = 0$  then this induction step degenerates to the first induction step. Therefore, only the case of  $N > 0$  needs to be considered. Since the length of the sequence of moves

$$(0, w_{1:n+k+1}) \vdash^{N-1} (\beta, w_{?:n+k+1}) \vdash (\alpha r, w_{i+1:n+k+1})$$

is less than  $N$ , the induction hypothesis can be applied to show that there must be an entry  $[?, ?, r]$  on  $I_i$ .

The move  $(\alpha r, w_{i+1:n+k+1}) \vdash (\alpha r s, w_{j+1:n+k+1})$  can be one of three types of moves:

- a shift (**shift**  $\in f_r(w_{i+1:i+k})$ );
- a reduction by an empty production (**reduce**  $p \in f_r(w_{i+1:i+k})$  and  $|\text{RHS}(p)| = 0$ ); or
- a reduction by a non-empty production which causes the stack to underflow (**reduce**  $p \in f_r(w_{i+1:i+k})$  and  $|\text{RHS}(p)| > 0$ ).

For any of the three types of moves,  $s \in g_r(w_{i+1:i+k})$ .

For a shift, **shift**  $\in f_r(w_{i+1:i+k})$  and  $j = i+1$ . When  $[?, ?, r]$  on  $I_i$  is processed, step II.B.1.i of Algorithm 4.1.1 adds  $([r, i, s], j)$  to  $H$ .

For a reduction by an empty production, **reduce**  $p \in f_r(w_{i+1:i+k})$ ,  $|\text{RHS}(p)| = 0$ , and  $i = j$ . When  $[?, ?, r]$  on  $I_i$  is processed: step II.B.1.ii adds  $(0, p, j)$  to  $G_i(r)$ ; step II.B.1.i of Algorithm 4.1.1 adds  $([r, i, s], j)$  to  $H$ ; step II.B.2.ii executes Algorithm 4.1.2 which adds  $(r, i)$  to  $B_{-1}$ ; and step II.B.2.iii adds  $([r, i, s], j)$  to  $H$ .

For a reduction by a non-empty production which underflows the stack, **reduce**  $p \in f_r(w_{i+1:i+k})$ ,  $|\text{RHS}(p)| > 0$  and  $i = j$ . This implies  $\alpha = \epsilon$ ,  $[?, ?, r] = [0, 0, 0]$ , and  $i = j = 0$  since no move of the  $\text{LR}(k)$  parser can add 0 to the stack. But this also implies  $N = 0$  so this type of move need not be considered any further.

For the third induction step,  $L + M = N$ ,  $M > 0$ , and the theorem is assumed to hold for  $L + M < N$ . Since  $M > 0$ , the theorem can be written as follows:

If  $(0, w_{1:n+k+1}) \vdash^L (\alpha \wr r, w_{i+1:n+k+1}) \vdash^{M+1} (\alpha r s, w_{j+1:n+k+1})$  where  $L \geq 0$  and  $M > 0$  then  $[r, i, s]$  is added to parse list  $I_j$ .

The sequence of moves

$$(\alpha \wr r, w_{i+1:n+k+1}) \vdash^{M+1} (\alpha r s, w_{j+1:n+k+1})$$

can be written as

$$\begin{aligned}
(\alpha \wr r, w_{i+1:n+k+1}) & \vdash^+ (\alpha r \wr q_1, w_{l_1+1:n+k+1}) \\
& \vdash^+ (\alpha r q_1 \wr q_2, w_{l_2+1:n+k+1}) \\
& \vdash^+ \dots \\
& \vdash^+ (\alpha r q_1 q_2 \dots \wr q_{m-1}, w_{l_{m-1}+1:n+k+1}) \\
& \vdash^+ (\alpha r q_1 q_2 \dots q_m, w_{l_m+1:n+k+1}) \\
& \vdash (\alpha r s, w_{j+1:n+k+1})
\end{aligned}$$

where  $1 \leq m \leq M$ , since a move of the  $\text{LR}(k)$  parser can add at most one symbol to the stack and  $r$  is not allowed to be popped from the stack by any of the  $M+1$  moves. The reduction  $(\alpha r q_1 q_2 \dots q_m, w_{l_m+1:n+k+1}) \vdash (\alpha r s, w_{j+1:n+k+1})$  must be a reduction that pops  $m$  states off the stack, or, if  $\alpha = \epsilon$  and  $r = 0$ , a reduction which possibly pops more than  $m$  states and underflows the stack. Also, the reduction implies  $l_m = j$  and that there exists  $s \in g_r(|\text{LHS}(p)|)$  and **reduce**  $p \in f_{q_m}(w_{l_m+1:n+k+1})$ .

Any proper subsequence of the sequence of moves

$$(\alpha \wr r, w_{i+1:n+k+1}) \vdash^+ (\alpha r s, w_{j+1:n+k+1})$$

has a length less than  $N$  so, applying the induction hypothesis, there must be

$$\begin{aligned}
& [r, i, q_1] \text{ on } I_{l_1} \\
& [q_1, l_1, q_2] \text{ on } I_{l_2} \\
& \vdots \\
& [q_{m-1}, l_{m-1}, q_m] \text{ on } I_{l_m}.
\end{aligned}$$

These entries may be processed by step II.B in any order so let  $[q_{h-1}, l_{h-1}, q_h]$  on  $I_{l_h}$  be the last entry processed by step II.B. If  $h = m$ , then step II.B.1.ii adds  $(|\text{RHS}(p)|, p, j)$  to  $G_{q_h}(l_h)$ . If  $h < m$ , applying Lemma 4.3.2 for

$$\begin{aligned}
& [q_h, l_h, q_{h+1}] \text{ on } I_{l_{h+1}} \\
& \vdots \\
& [q_{m-2}, l_{m-2}, q_{m-1}] \text{ on } I_{l_{m-1}} \\
& [q_{m-1}, l_{m-1}, q_m] \text{ on } I_{l_m}
\end{aligned}$$

shows that

$$(|\text{RHS}(p)| - (m - h), p, j) \in G_{q_h}(l_h)$$

when the entry is processed by step II.B.2.iii. Applying Lemma 4.2.3 to  $[r, i, q_1]$  on  $I_{l_1}$  shows there must be  $[?, ?, r]$  on  $I_i$  and applying Lemma 4.3.1 for

$$\begin{aligned} & [?, ?, r] \text{ on } I_i \\ & [r, i, q_1] \text{ on } I_{l_1} \\ & [q_1, l_1, q_2] \text{ on } I_{l_2} \\ & \vdots \\ & [q_{h-2}, l_{h-2}, q_{h-1}] \text{ on } I_{l_{h-1}} \end{aligned}$$

shows that

$$(r, i) \in B_{(|\text{RHS}(p)| - (m-h)) - 1}$$

when the entry is processed by step II.B.2.iii. Therefore, step II.B.2.iii of Algorithm 4.1.1 adds  $([r, i, s], j)$  to  $H$  and Lemma 4.2.1 guarantees that  $[r, i, s]$  is added to  $I_j$ . ■

## 4.4 RUN TIME ANALYSIS

This section is divided into two parts: the first part shows that Algorithm 4.1.1, unlike Algorithm 3.1.1, has  $\mathcal{O}(n^5)$  time complexity and  $\mathcal{O}(n^2)$  space complexity; the second part shows that for  $\text{LR}(k)$  grammars the time and space complexities are  $\mathcal{O}(n)$ . The increased time complexity of Algorithm 4.1.1 over Algorithm 3.1.1 is due to the operations on the sets  $G_j(r)$  which maintain the deferred reduction information. However, for  $\text{LR}(k)$  grammars the time and space complexities of Algorithm 4.1.1 are  $\mathcal{O}(n)$ .

### 4.4.1 $\mathcal{O}(n^5)$ Time and $\mathcal{O}(n^2)$ Space Complexities

The time and space complexities of Algorithm 4.1.1 are shown to be  $\mathcal{O}(n^5)$  and  $\mathcal{O}(n^2)$  respectively. The analysis is performed using the same techniques introduced in Chapter 3.

The input to Algorithm 4.1.1 is the string  $w$  to be parsed. The size of this input is the string's length, which is denoted by  $n$ . Some data structures, such as the parse lists, depend directly on  $n$ . It is implicitly assumed that  $n$  is known when these data structures are allocated so that a position in the input string can be used in  $\mathcal{O}(1)$  time as an index into these data structures. This assumption presents no difficulties, since the value of  $n$  can be determined in  $\mathcal{O}(n)$  time by scanning the input string before starting the algorithm.

The parse lists, the pending list, the sets  $B_i$ , and the sets  $G_i(q)$  are the primary data structures used by Algorithm 4.1.1. The parse lists are organized as an array of parse lists and each parse list  $I_r$  is  $|Q|$  (initially empty) linked lists of entries  $[q, i, r]$ . The  $|Q|$  lists are indexed by  $r$  and an entry  $[q, i, r]$  is stored on the  $r^{\text{th}}$  list. The pending list,  $H$ , is organized as an array of  $n + 2$  (initially empty) lists of ordered pairs,  $([q, i, r], j)$ , where the lists are indexed by  $j$ . The sets  $B_i$  are organized as an array of sets, where the sets are indexed by  $i$ . The sets  $G_i(q)$  are organized as a two-dimensional array of sets, where the sets are indexed by  $i$  and  $q$ .

The space complexity of Algorithm 4.1.1 is simple to determine. There are  $\mathcal{O}(n)$  parse lists. Each parse list can have at most  $\mathcal{O}(n)$  entries since duplicate entries are eliminated by step II.B and the parse list number of an entry is the only component of an entry that depends on the length of the input and it is bounded by  $n + 2$ . There is one pending list,  $H$ , and it can have at most  $\mathcal{O}(n^2)$  entries since duplicated ordered pairs are not added to the list. There are only  $\mathcal{O}(n)$  sets  $B_i$  and each set has at most  $\mathcal{O}(n)$  elements. Finally, there are  $\mathcal{O}(n)$  sets  $G_i(q)$  and the only component of the elements of these sets that is not bounded independently of  $n$  is the parse list number which is bounded by  $n + 2$ . Thus, the space complexity of Algorithm 4.1.1 is  $\mathcal{O}(n^2)$ .

The analysis of the time complexity of Algorithm 4.1.1 is determined by examining each step of the algorithm. Algorithm 4.1.1 implicitly assumes the parse lists, the pending list,  $H$ , and sets  $G_i(q)$  are initialized. Given the organization of parse lists, the pending list,  $H$ , and the sets  $G_i(q)$ , they can be initialized in  $\mathcal{O}(n)$  primitive operations which are charged to the algorithm object. (Recall that the algorithm object is simply a bookkeeping convention).

For step I of Algorithm 4.1.1, the  $\mathcal{O}(1)$  primitive operations for creating the initial entry and adding it to the pending list,  $H$ , are charged to the entry.

For step II of Algorithm 4.1.1, the  $\mathcal{O}(1)$  primitive operations for checking the pending list,  $H$ , to determine if it is empty are charged to the algorithm object, if  $H$  is empty, or to the entry obtained in step II.A, if the list is not empty.

For step II.A of Algorithm 4.1.1, the  $\mathcal{O}(1)$  primitive operations for obtaining an entry are charged to the corresponding entry on the parse list. In general, an entry may be obtained  $\mathcal{O}(n^2)$  times since an entry  $([q, i, r], j)$  may be added to the pending list,  $H$ , a total of  $\mathcal{O}(1)$  times for each entry processed by steps II.B.1.i or II.B.2.iii.

For step II.B of Algorithm 4.1.1, the  $\mathcal{O}(n)$  primitive operations for checking if an entry  $[q, i, r]$  is already on parse list  $I_r$  are charged to the corresponding entry on the parse list. The check requires  $\mathcal{O}(n)$  primitive operations because the number of entries on the  $r^{\text{th}}$  list of a parse list is  $\mathcal{O}(n)$ . The  $\mathcal{O}(1)$  primitive operations for

adding an entry to its parse list are charged to the entry itself.

For step II.B.1 of Algorithm 4.1.1, the  $\mathcal{O}(1)$  primitive operations for checking to see if the  $r^{\text{th}}$  list of a parse list is empty is charged to the entry being processed.

For step II.B.1.i of Algorithm 4.1.1, the  $\mathcal{O}(1)$  primitive operations for evaluating  $f_r$  and  $g_r$  are charged to the entry being processed. Also, the  $\mathcal{O}(n^2)$  primitive operations for adding an entry to the pending list,  $H$ , are charged to the entry being processed.

For step II.B.1.ii of Algorithm 4.1.1, the  $\mathcal{O}(1)$  primitive operations for evaluating  $f_r$  are charged to the entry being processed. The  $\mathcal{O}(n)$  primitive operations for adding  $\mathcal{O}(1)$  elements to the set  $G_j(r)$  are charged to the entry being processed.

For step II.B.2 of Algorithm 4.1.1, the  $\mathcal{O}(1)$  primitive operations for checking to see if  $G_j(r)$  is empty are charged to the entry being processed.

For step II.B.2.i of Algorithm 4.1.1, the  $\mathcal{O}(n)$  primitive operations for finding the maximum value of  $h$  for elements in  $G_i(q)$  are charged to the entry being processed.

For step II.B.2.ii of Algorithm 4.1.1, Algorithm 4.1.2 is executed. The time complexity of Algorithm 4.1.2 is determined indirectly. The primitive operations used in the steps of Algorithm 4.1.2 are charged to the objects of Algorithm 4.1.1.

The time complexity of Algorithm 4.1.2 is more easily analyzed if the union operations on the sets  $G_i(q)$  are explicitly performed. Algorithm 4.4.1 reflects this change. Also, Algorithm 4.4.1 optimizes the application of the inner most union operation for the sets  $G_i(q)$  by moving it out of the loop in step IV.C.3 of the algorithm.

For Algorithm 4.4.1, the sets  $B_0, B_1, \dots, B_m$  are organized as an array of sets and each set is an (initially empty) linked list of elements. Each set can have at most  $\mathcal{O}(n)$  elements because the elements are not duplicated and parse list number of an element is the only component of an element that is not bounded independently of  $n$  and it is bounded by  $n + 1$ . The number of sets is  $\mathcal{O}(1)$  and the space complexity of Algorithm 4.4.1 is  $\mathcal{O}(n)$  so the space complexity of Algorithm 4.1.1 is not increased.

For step I of Algorithm 4.4.1, its  $\mathcal{O}(1)$  primitive operations are charged to the entry being processed.

Step II and III.B of Algorithm 4.4.1 take  $\mathcal{O}(n)$  primitive operations to scan all the elements in  $G_i(q)$ . Thus  $\mathcal{O}(1)$  primitive operations for either step II or III.B are charged to each element scanned.

For step III of Algorithm 4.4.1, the  $\mathcal{O}(1)$  primitive operations for checking the set  $G_j(r)$  to see if it is exhausted are charged to the set, if  $G_j(r)$  is exhausted, or to the entry obtained in step II or step III.B, if the set is not exhausted. The loop in step III of Algorithm 4.4.1 is executed at most  $\mathcal{O}(n)$  times; once for each of the

$\mathcal{O}(n)$  elements in  $G_j(r)$ .

Step III.A of Algorithm 4.4.1 takes  $\mathcal{O}(1)$  primitive operations for each element in  $G_i(q)$  since each element must be checked to avoid adding a duplicate. Also, step III.A is nested within the loop formed by step III. Therefore, the  $\mathcal{O}(n)$  primitive operations are charged to each element of  $G_i(q)$ .

The loop in step IV of Algorithm 4.4.1 is executed at most  $\mathcal{O}(1)$  times since  $m$ , the length of a production, is independent of  $n$ . The  $\mathcal{O}(1)$  primitive operations for the loop termination test in step IV are charged to the entry being processed by step II.B of Algorithm 4.1.1. Likewise, the  $\mathcal{O}(1)$  primitive operations for step IV.D are charged to the entry being processed by step II.B of Algorithm 4.1.1.

Step IV.A of Algorithm 4.4.1 takes  $\mathcal{O}(1)$  primitive operations. Step IV.A is nested within the loop formed by step IV, which causes it to be repeated  $\mathcal{O}(1)$  times. Therefore, the  $\mathcal{O}(1)$  primitive operations for step IV.A are charged to the entry being processed by step II.B of Algorithm 4.1.1.

Step IV.B of Algorithm 4.4.1 takes  $\mathcal{O}(1)$  primitive operations. Step IV.B is nested within the loop formed by step IV, which causes it to be repeated  $\mathcal{O}(1)$  times. Therefore, the  $\mathcal{O}(1)$  primitive operations for step IV.B are charged to the entry being processed by step II.B of Algorithm 4.1.1.

The loop in step IV.C of Algorithm 4.4.1 is executed at most  $\mathcal{O}(n)$  times since  $B_i$  has  $\mathcal{O}(n)$  elements. Thus, the  $\mathcal{O}(n)$  primitive operations for the loop termination test in step IV.C are charged to the entry being processed by step II.B of Algorithm 4.1.1. Likewise, the  $\mathcal{O}(n)$  primitive operations for step IV.C.6 are charged to the entry being processed by step II.B of Algorithm 4.1.1.

Step IV.C.1 of Algorithm 4.4.1 takes  $\mathcal{O}(n)$  primitive operations. Step IV.C.1 is nested within the loops formed by steps IV and IV.C, which causes it to be repeated  $\mathcal{O}(n)$  times. Therefore, the  $\mathcal{O}(n^2)$  primitive operations for step IV.C.1 are charged to the entry being processed by step II.B of Algorithm 4.1.1.

Step IV.C.2 of Algorithm 4.4.1 takes  $\mathcal{O}(1)$  primitive operations. Step IV.C.2 is nested within the loops formed by steps IV and IV.C, which causes it to be repeated  $\mathcal{O}(n)$  times. Therefore, the  $\mathcal{O}(n)$  primitive operations for step IV.C.2 are charged to the entry being processed by step II.B of Algorithm 4.1.1.

The loop in step IV.C.3 of Algorithm 4.4.1 is executed at most  $\mathcal{O}(n)$  times since  $I_h$  has  $\mathcal{O}(n)$  elements. Also, step IV.C.3 is nested inside the loops formed by steps IV and IV.C, which causes it to be repeated  $\mathcal{O}(n)$  times. Therefore, the  $\mathcal{O}(n^2)$  primitive operations for the loop termination test in step IV.C.3 are charged to the entry being processed by step II.B of Algorithm 4.1.1. Likewise, the  $\mathcal{O}(n^2)$  primitive operations for step IV.C.3.iii are charged to the entry being processed by step II.B of Algorithm 4.1.1.

ALGORITHM 4.4.1 *Calculate  $B_l$ 's*

Let  $[q, i, r]$  on  $I_j$  be the entry for which the  $B_l$ 's are to be calculated and let  $m$  be as given in Algorithm 4.1.1.

- I. Let  $l = 1$ ,  $B_{-1} = \{(r, j)\}$  and  $B_0 = \{(q, i)\}$ .
- II. Get an element  $(n + 1, p, x)$  from  $G_j(r)$  where  $n > 0$ .
- III. While  $G_j(r)$  is not exhausted, perform the following steps:
  - A. Add  $(n, p, x)$  to  $G_i(q)$  if it is not already in  $G_i(q)$ .
  - B. Get another element  $(n + 1, p, x)$  from  $G_h(t)$  where  $n > 0$ .
- IV. While  $l < m$ , perform the following steps:
  - A. Initialize  $B_l$ .
  - B. Get an element  $(t, h)$  from  $B_{l-1}$ .
  - C. While  $B_{l-1}$  is not exhausted, perform the following steps:
    1. Empty the set  $T$ .
    2. Get an entry  $[s, o, v]$  on  $I_h$ .
    3. While  $I_h$  is not exhausted, perform the following steps:
      - i. If  $v = t$  then add  $(s, o)$  to  $B_l$  if it is not already in  $B_l$ .
      - ii. If  $v = t$  then add  $(s, o)$  to  $T$  if it is not already in  $T$ .
      - iii. Get another entry  $[s, o, v]$  on  $I_h$ .
    4. Get an element  $(n + 1, p, x)$  from  $G_h(t)$  where  $n > 0$ .
    5. While  $G_h(t)$  is not exhausted, perform the following steps:
      - i. Get an element  $(s, o)$  from  $T$ .
      - ii. While  $T$  is not exhausted, perform the following steps:
        - a. Add  $(n, p, x)$  to  $G_o(s)$  if it is not already in  $G_o(s)$ .
        - b. Get another element  $(s, o)$  from  $T$ .
      - iii. Get another element  $(n + 1, p, x)$  from  $G_h(t)$  where  $n > 0$ .
    6. Get another element  $(t, h)$  from  $B_{l-1}$ .
  - D. Let  $l = l + 1$ .

Step IV.C.3.i of Algorithm 4.4.1 takes  $\mathcal{O}(n)$  primitive operations since it must check all the elements in  $B_l$ . Also, step IV.C.3.i is nested within the loops formed by steps IV, IV.C and IV.C.3, which causes it to be repeated  $\mathcal{O}(n^2)$  times. Therefore, the  $\mathcal{O}(n^3)$  primitive operations for step IV.C.3.i are charged to the entry being processed by step II.B of Algorithm 4.1.1.

Step IV.C.3.ii of Algorithm 4.4.1 takes  $\mathcal{O}(n)$  primitive operations since it must check all the elements in  $T$ . Also, step IV.C.3.ii is nested within the loops formed by steps IV, IV.C and IV.C.3, which causes it to be repeated  $\mathcal{O}(n^2)$  times. Therefore, the  $\mathcal{O}(n^3)$  primitive operations for step IV.C.3.ii are charged to the entry being processed by step II.B of Algorithm 4.1.1.

Steps IV.C.4 and IV.C.5.iii of Algorithm 4.4.1 take  $\mathcal{O}(n)$  primitive operations to scan the set  $G_o(s)$ . Also, steps IV.C.4 and IV.C.5.iii are nested within the loops formed by steps IV and IV.C, which causes it to be repeated  $\mathcal{O}(n)$  times. Therefore,  $\mathcal{O}(n)$  primitive operations for scanning each element  $\mathcal{O}(n)$  times are charged to each element scanned.

For step IV.C.5 of Algorithm 4.4.1, checking to see if  $G_h(t)$  is exhausted takes  $\mathcal{O}(1)$  primitive operations. Step IV.C.5 is nested within the loops formed by steps IV and IV.C, which causes it to be repeated  $\mathcal{O}(n)$  times. Therefore, the  $\mathcal{O}(n)$  primitive operations for checking  $G_h(t)$  to see if it is exhausted are charged to  $G_h(t)$ , if it is exhausted, or to the element obtained in step IV.C.4 or step IV.C.5.iii, if the set is not exhausted. The loop in step IV.C.5 of Algorithm 4.4.1 is executed at most  $\mathcal{O}(n)$  times; once for each of the  $\mathcal{O}(n)$  elements in  $G_h(t)$ .

Step IV.C.5.i of Algorithm 4.4.1 takes  $\mathcal{O}(1)$  primitive operations. Step IV.C.5.i is nested within the loops formed by steps IV, IV.C and IV.C.5, which causes it to be repeated  $\mathcal{O}(n^2)$  times. Therefore, the  $\mathcal{O}(n^2)$  primitive operations for step IV.C.5.i are charged to the entry being processed by step II.B of Algorithm 4.1.1.

The loop in step IV.C.5.ii of Algorithm 4.4.1 is executed at most  $\mathcal{O}(n)$  times since  $T$  has  $\mathcal{O}(n)$  elements. Step IV.C.5.ii is nested inside the loops formed by steps IV.C and IV.C.5, which causes it to be repeated  $\mathcal{O}(n^2)$  times. Therefore, the  $\mathcal{O}(n^3)$  primitive operations for the loop termination test in step IV.C.5.ii are charged to the entry being processed by step II.B of Algorithm 4.1.1. Likewise, the  $\mathcal{O}(n^3)$  primitive operations for step IV.C.5.ii.b are charged to the entry being processed by step II.B of Algorithm 4.1.1.

Step IV.C.5.ii.a of Algorithm 4.4.1 takes  $\mathcal{O}(1)$  primitive operations for each element in  $G_o(s)$  since each element must be checked to avoid adding a duplicate. Step IV.C.5.ii.a is also nested within the loop formed by step IV.C, IV.C.5, and IV.C.5.ii, which causes it to be repeated  $\mathcal{O}(n^3)$  times. However,  $G_o(s)$  can



only be accessed  $\mathcal{O}(1)$  times within the loop formed by step IV.C.5.ii since  $T$  can contain only one element  $(s, o)$ . Therefore,  $\mathcal{O}(n^2)$  primitive operations are charged to each element of  $G_o(s)$ .

Returning to Algorithm 4.1.1 and examining step II.B.2.iii, there are  $\mathcal{O}(n)$  elements in  $G_j(r)$ ,  $\mathcal{O}(1)$  elements in  $g_r$ , and  $\mathcal{O}(n)$  elements in  $B_{h-1}$ . Therefore, the actions of this step are performed  $\mathcal{O}(n^2)$  times and this step can add  $\mathcal{O}(n^2)$  entries to the pending list,  $H$ . The  $\mathcal{O}(n^3)$  primitive operations for adding  $\mathcal{O}(n^2)$  entries to the pending list,  $H$ , are charged to the entry being processed.

Now that all the steps of Algorithm 4.1.1 have been examined, the number of primitive operations charged to each class of objects can be calculated. Considering the entries first, each entry has the following number of primitive operations charged to it:

- $\mathcal{O}(n^2)$  when the entry is added to the pending list,  $H$ ,  $\mathcal{O}(n^2)$  times in Algorithm 4.1.1.
- $\mathcal{O}(n^2)$  when the entry is removed from the pending list,  $H$ ,  $\mathcal{O}(n^2)$  times by step II.A of Algorithm 4.1.1.
- $\mathcal{O}(1)$  when the entry is added to its parse list by step II.B of Algorithm 4.1.1.
- $\mathcal{O}(n^3)$  when the check for a duplicate entry is made  $\mathcal{O}(n^2)$  times by step II.B of Algorithm 4.1.1.
- $\mathcal{O}(1)$  when the  $r^{\text{th}}$  list of a parse list is checked for entries in step II.B.1 of Algorithm 4.1.1.
- $\mathcal{O}(1)$  when  $f_r$  and  $g_r$  are evaluated in step II.B.1.i of Algorithm 4.1.1.
- $\mathcal{O}(n^2)$  when the check for a duplicate entry on the pending list,  $H$ , is made in step II.B.1.i of Algorithm 4.1.1.
- $\mathcal{O}(1)$  when  $f_r$  is evaluated in step II.B.1.ii of Algorithm 4.1.1.
- $\mathcal{O}(n)$  when  $\mathcal{O}(1)$  elements are added to  $G_j(r)$  in step II.B.1.ii of Algorithm 4.1.1.
- $\mathcal{O}(1)$  when  $G_j(r)$  is checked for elements in step II.B.2 of Algorithm 4.1.1.
- $\mathcal{O}(n)$  when the maximum value of  $h$  for the elements of  $G_i(q)$  is calculated in step II.B.2.i of Algorithm 4.1.1.

- $\mathcal{O}(1)$  when the entry causes the execution of step I of Algorithm 4.4.1.
- $\mathcal{O}(1)$  when the entry causes the execution of steps IV and IV.D of Algorithm 4.4.1.
- $\mathcal{O}(n)$  when the entry causes the execution of step IV.A of Algorithm 4.4.1.
- $\mathcal{O}(1)$  when the entry causes the execution of step IV.B of Algorithm 4.4.1.
- $\mathcal{O}(n)$  when the entry causes the execution of steps IV.C, IV.C.6, and II.B.3 of Algorithm 4.4.1.
- $\mathcal{O}(n^2)$  when the entry causes the execution of step IV.C.1 of Algorithm 4.4.1.
- $\mathcal{O}(n)$  when the entry causes the execution of step IV.C.2 of Algorithm 4.4.1.
- $\mathcal{O}(n^2)$  when the entry causes the execution of steps IV.C.3 and IV.C.3.iii of Algorithm 4.4.1.
- $\mathcal{O}(n^3)$  when the entry causes the execution of step II.C.3.i of Algorithm 4.4.1.
- $\mathcal{O}(n^3)$  when the entry causes the execution of step II.C.3.ii of Algorithm 4.4.1.
- $\mathcal{O}(n^2)$  when the entry causes the execution of step II.C.5.i of Algorithm 4.4.1.
- $\mathcal{O}(n^3)$  when the entry causes the execution of steps IV.C.5.ii and IV.C.5.ii.b of Algorithm 4.4.1.
- $\mathcal{O}(n^3)$  when the check for a duplicate entry on the pending list,  $H$ , is made  $\mathcal{O}(n^2)$  times in step II.B.2.iii of Algorithm 4.1.1.

Thus, the maximum number of primitive operations charged to an entry is  $\mathcal{O}(n^3)$ . The number of primitive operations for all the entries is  $\mathcal{O}(n^5)$  since the number of entries is  $\mathcal{O}(n^2)$ .

The number of primitive operations charged to the elements of a set  $G_i(q)$  are as follows:

- $\mathcal{O}(n^2)$  when the element is scanned by step II or III.B of Algorithm 4.4.1 for  $\mathcal{O}(n^2)$  entries.

- $\mathcal{O}(n^3)$  when the element is scanned by step IV.C.4 or IV.C.5.iii of Algorithm 4.4.1 for  $\mathcal{O}(n^2)$  entries.
- $\mathcal{O}(n^3)$  when the element is obtained in step IV.C.5 of Algorithm 4.4.1 for  $\mathcal{O}(n^2)$  entries.
- $\mathcal{O}(n^4)$  when the element is scanned by step IV.C.5.ii.a of Algorithm 4.4.1 for  $\mathcal{O}(n^2)$  entries.

Thus, the maximum number of primitive operations charged to elements of a set  $G_i(q)$  is  $\mathcal{O}(n^4)$ . The number of primitive operations for all elements of sets  $G_i(q)$  is  $\mathcal{O}(n^6)$  since the number of elements is  $\mathcal{O}(n^2)$ .

The number of primitive operations charged to the sets  $G_i(q)$  are as follows:

- $\mathcal{O}(n^2)$  for step III of Algorithm 4.4.1 when the set  $G_i(q)$  is empty.
- $\mathcal{O}(n^3)$  for step IV.C.5 of Algorithm 4.4.1 when the set  $G_h(t)$  is empty.

Thus, the maximum number of primitive operations charged to a set  $G_i(q)$  is  $\mathcal{O}(n^3)$ . The number of primitive operations for all the sets  $G_i(q)$  is  $\mathcal{O}(n^4)$  since the number of sets is  $\mathcal{O}(n)$ .

No primitive operations are charged to the parse lists, the pending list,  $H$ , or the sets  $B_i$ . Finally, the number of primitive operations charged to the algorithm object are as follows:

- $\mathcal{O}(n)$  for the initialization of Algorithm 4.1.1 in step I.
- $\mathcal{O}(1)$  when the pending list,  $H$ , is empty in step II of Algorithm 4.1.1.

Thus, the maximum time charged to the algorithm object is  $\mathcal{O}(n)$ .

Summing the number of primitive operations for the entries, the parse lists, pending list, the sets  $G_i(q)$  and their elements, and the algorithm object shows the time complexity of Algorithm 4.1.1 is  $\mathcal{O}(n^6)$ .

Using a trick similar to those in Chapter 3, the time complexity of Algorithms 4.1.1 can be improved at the expense of increasing its best-case times. The trick is to store the sets  $G_i(q)$  as  $n$  lists where the  $i^{\text{th}}$  list contains all the  $(h, p, i)$  elements. The time to initialize a set  $G_i(q)$  is increased to  $\mathcal{O}(n)$  but the time to perform step IV.5.ii.a is reduced to  $\mathcal{O}(n^3)$ . This trick reduces the time complexity of Algorithm 4.1.1 to  $\mathcal{O}(n^5)$ .

The drawback of this trick is that it forces the time complexity of Algorithm 4.1.1 to be  $\mathcal{O}(n^2)$  regardless of the grammar being used. This is not always desirable. For example, the next section shows that for LR( $k$ ) grammars the time complexity of Algorithm 4.1.2 is  $\mathcal{O}(n)$  without this trick.

#### 4.4.2 $\mathcal{O}(n)$ Time and Space for $LR(k)$ Grammars

Since the development of Algorithm 4.1.1 is motivated by the desire to find an efficient least-cost syntax error recovery scheme for  $LR(k)$  parsers, the time and space complexities of Algorithm 4.1.1, when the canonical states and functions  $f_q$  and  $g_q$  for an  $LR(k)$  grammar are used, are of particular interest.

To determine the time and space complexities of Algorithm 4.1.1 for an  $LR(k)$  grammar, additional analysis of the algorithm is required. Two properties of canonical  $LR(k)$  parsers for  $LR(k)$  grammars are crucial to this analysis:

- the canonical  $LR(k)$  parser is deterministic, and
- the canonical  $LR(k)$  parser does not enter an infinite loop for any input string.

However, the next lemma does not depend on any special properties of  $LR(k)$  grammars. It shows the relationship between the order in which entries are added to their parse lists and the order in which configurations can appear in a sequence of moves by the  $LR(k)$  parser.

**LEMMA 4.4.1** *Given the same  $Q$ ,  $f_q$ , and  $g_q$  for Algorithm 4.1.1 and for the canonical  $LR(k)$  parser for an  $LR(k)$  grammar, if  $[q, h, r]$  on  $I_l$  precedes  $[s, i, t]$  on  $I_j$  in an ordered list of entries then the canonical  $LR(k)$  parser can make a sequence of moves*

$$(0, w_{1:n+k+1}) \stackrel{*}{\vdash} (\alpha \downarrow q, w_{h+1:n+k+1}) \stackrel{+}{\vdash} (\alpha q r, w_{l+1:n+k+1})$$

*in which the subsequence*

$$(\beta \downarrow s, w_{i+1:n+k+1}) \stackrel{+}{\vdash} (\beta s t, w_{j+1:n+k+1})$$

*does not occur for any  $\beta$ .*

**Proof:** The theorem is proved by induction on an ordered list of entries, using the theorem as the induction hypothesis. The induction proceeds in two steps:

- first, the lemma is proved for the first entry on the ordered list of entries; and
- second, the lemma is proved for the  $N^{\text{th}}$  entry on the ordered list of entries, assuming it holds for all entries that precede the  $N^{\text{th}}$  entry.

For the first induction step, let the first entry on the ordered list of entries be  $[q, i, r]$  on  $I_j$ . Step II.B of Algorithm 4.1.1 adds  $[q, i, r]$  to  $I_j$  so  $([q, i, r], j)$  must have been on the pending list,  $H$ . Either step II.B.1.i or step II.B.2.iii of Algorithm 4.1.1 must have added  $([q, i, r], j)$  to  $H$  while processing  $[0, 0, 0]$  on  $I_0$ . Likewise, the first move of the  $LR(k)$  parser must be from the configuration  $(0, w_{1:n+k+1})$ . Thus, if step II.B.1.i adds  $([q, i, r], j) = ([0, 0, r], 1)$  to  $H$  then the shift  $(0, w_{1:n+k+1}) \vdash (0r, w_{2:n+k+1})$  can be made by the  $LR(k)$  parser; and if step II.B.2.iii adds  $([q, i, r], j) = ([0, 0, r], 0)$  to  $H$  then the reduction  $(0, w_{1:n+k+1}) \vdash (0r, w_{1:n+k+1})$  can be made by the  $LR(k)$  parser. For either move of the  $LR(k)$  parser, there are no proper subsequences so, trivially, the lemma holds.

For the second induction step, the lemma is assumed to hold for all entries on the ordered list of entries that precede the  $N^{\text{th}}$  entry. Let the  $N^{\text{th}}$  entry be  $[r, i, s]$  on  $I_j$ . Since  $[r, i, s]$  is added to  $I_j$  by step II.B of Algorithm 4.1.1,  $([r, i, s], j)$  must have been on the pending list,  $H$ . There are three cases to consider:

- step II.B.1.i adds  $([r, i, s], j)$  to  $H$ , simulating a shift;
- step II.B.2.iii adds  $([r, i, s], j)$  to  $H$ , simulating a reduction by an empty production; or
- step II.B.2.iii adds  $([r, i, s], j)$  to  $H$ , simulating a reduction by a non-empty production.

For any of the cases, the step involved must have been applied to an entry  $[q_{h-1}, l_{h-1}, q_h]$  on  $I_{l_h}$ , where  $h$  is a convenient index which will be specified later.

In the first case, step II.B.1.i adds  $([r, i, s], j)$  to  $H$  and Algorithm 4.1.1 is simulating a shift by the  $LR(k)$  parser. Thus,  $l_h = i = j - 1$ ,  $q_h = r$ , and, applying Theorem 4.2.1, the  $LR(k)$  parser can make the sequence of moves

$$(0, w_{1:n+k+1}) \vdash^* (\alpha r, w_{j:n+k+1}) \vdash (\alpha r s, w_{j+1:n+k+1}).$$

For this case, the lemma is proved by assuming it does not hold for the  $N^{\text{th}}$  entry  $[r, i, s]$  on  $I_j$  and showing that this assumption leads to a contradiction. If  $[r', i', s']$  on  $I_{j'}$  is an entry on the ordered list of entries which follows  $[r, i, s]$  on  $I_j$  and for which the lemma does not hold, the sequence of moves

$$(\beta r', w_{i'+1:n+k+1}) \vdash^+ (\beta r' s', w_{j'+1:n+k+1})$$

must be embedded in the sequence of moves

$$(0, w_{1:n+k+1}) \vdash^* (\alpha r, w_{j:n+k+1}) \vdash (\alpha r s, w_{j+1:n+k+1}).$$

The lemma holds for  $[q_{h-1}, l_{h-1}, r]$  on  $I_{j-1}$  since it precedes  $[r, i, s]$  on  $I_j$  on the ordered list of entries. Any entry on the ordered list of entries that follows  $[r, i, s]$  on  $I_j$  also follows  $[q_{h-1}, l_{h-1}, r]$  on  $I_{j-1}$ . Therefore, the sequence of moves

$$(\beta \lambda r', w_{i'+1:n+k+1}) \vdash^+ (\beta r' s', w_{j'+1:n+k+1})$$

can not be embedded in the sequence of moves

$$(0, w_{1:n+k+1}) \vdash^* (\delta \lambda q_{h-1}, w_{l_{h-1}:n+k+1}) \vdash^+ (\delta q_{h-1} r, w_{j:n+k+1})$$

and, letting  $\alpha = \delta q_{h-1}$ , it must end with the move

$$(\alpha \lambda r, w_{j:n+k+1}) \vdash (\alpha r s, w_{j+1:n+k+1}).$$

Thus,  $(\beta r' s', w_{j'+1:n+k+1}) = (\alpha r s, w_{j+1:n+k+1})$  which implies that  $\beta = \alpha$ ,  $r' = r$ ,  $s' = s$ ,  $j' = j$  and  $i' \neq i$  since  $[r, i, s]$  is distinct from  $[r, i', s]$ .

The entry  $[r, i', s]$  must be added to  $I_j$  by step II.B of Algorithm 4.1.1 so  $([r, i', s], j)$  must have been on the pending list,  $H$ , and  $([r, i', s], j)$  must have been added to  $H$  while step II.B was processing an entry  $[q'_{h'-1}, l'_{h'-1}, q'_{h'}]$  on  $I'_{h'}$ , where  $h'$  is a convenient index which will be specified later. There are three types of moves step II.B could be simulating while processing  $[q'_{h'-1}, l'_{h'-1}, q'_{h'}]$  on  $I'_{h'}$ :

- a shift,
- a reduction by an empty production, or
- a reduction by a non-empty production.

The first type of move is a shift so  $l'_{h'} = i' = j - 1$ ,  $q'_{h'} = r$ , and the sequence of moves corresponding to  $[r, i', s]$  on  $I_j$  is

$$(0, w_{1:n+k+1}) \vdash^* (\alpha \lambda r, w_{i':n+k+1}) \vdash (\alpha r s, w_{j+1:n+k+1}).$$

However,  $i' = j - 1$  implies  $i' = i$  and  $[r, i', s] = [r, i, s]$  which is a contradiction.

The second type of move is a reduction by an empty production so  $l'_{h'} = i' = j$ ,  $q'_{h'} = r$ , and the sequence of moves corresponding to  $[r, i', s]$  on  $I_j$  is

$$(\alpha \lambda r, w_{j+1:n+k+1}) \vdash (\alpha r s, w_{j+1:n+k+1}).$$

However, this sequence of moves does not end with the move

$$(\alpha \lambda r, w_{j:n+k+1}) \vdash (\alpha r s, w_{j+1:n+k+1})$$

which is a contradiction.

The third type of move is a reduction by a non-empty production so the sequence of moves corresponding to  $[r, i', s]$  on  $I_j$  is

$$(\alpha \backslash r, w_{i'+1:n+k+1}) \vdash^+ (\alpha r q'_1 \dots q'_{h'} \dots q'_{m'}, w_{j+1:n+k+1}) \vdash (\alpha r s, w_{j+1:n+k+1}).$$

However, this sequence of moves does not end with the move

$$(\alpha r, w_{j:n+k+1}) \vdash (\alpha r s, w_{j+1:n+k+1})$$

which is a contradiction.

In the second case, where step II.B.2.iii adds  $([r, i, s], j)$  to  $H$ , Algorithm 4.1.1 is simulating a reduction by an empty production for the  $LR(k)$  parser. Thus,  $l_h = i = j$ ,  $q_h = r$ , and, applying Theorem 4.2.1, the  $LR(k)$  parser can make the sequence of moves

$$(0, w_{1:n+k+1}) \vdash^* (\alpha \backslash r, w_{j+1:n+k+1}) \vdash (\alpha r s, w_{j+1:n+k+1}).$$

For this case, the lemma is proved by showing the assumption that the lemma does not hold for the  $N^{\text{th}}$  entry  $[r, i, s]$  on  $I_j$  leads to a contradiction. If  $[r', i', s']$  on  $I_{j'}$  is an entry on the ordered list of entries for which the lemma does not hold then it must follow  $[r, i, s]$  on  $I_j$  and the sequence of moves

$$(\beta \backslash r', w_{i'+1:n+k+1}) \vdash^+ (\beta r' s', w_{j'+1:n+k+1})$$

must be embedded in the sequence of moves

$$(0, w_{1:n+k+1}) \vdash^* (\alpha \backslash r, w_{j+1:n+k+1}) \vdash (\alpha r s, w_{j+1:n+k+1}).$$

The lemma holds for  $[q_{h-1}, l_{h-1}, r]$  on  $I_j$  since it precedes  $[r, i, s]$  on  $I_j$  on the ordered list of entries. Any entry on the ordered list of entries that follows  $[r, i, s]$  on  $I_j$  also follows  $[q_{h-1}, l_{h-1}, r]$  on  $I_j$ . Therefore, the sequence of moves

$$(\beta \backslash r', w_{i'+1:n+k+1}) \vdash^+ (\beta r' s', w_{j'+1:n+k+1})$$

can not be embedded in the sequence of moves

$$(0, w_{1:n+k+1}) \vdash^* (\delta \backslash q_{h-1}, w_{l_{h-1}:n+k+1}) \vdash^+ (\delta q_{h-1} r, w_{j+1:n+k+1})$$

and, letting  $\alpha = \delta q_{h-1}$ , it must end with the move

$$(\alpha q_h, w_{l_h:n+k+1}) \vdash (\alpha r s, w_{j+1:n+k+1}).$$

Thus,  $(\beta r' s', w_{j'+1:n+k+1}) = (\alpha r s, w_{j+1:n+k+1})$  which implies that  $\beta = \alpha$ ,  $r' = r$ ,  $s' = s$ ,  $j' = j$  and  $i' \neq i$  since  $[r, i, s]$  is distinct from  $[r, i', s]$ .

The entry  $[r, i', s]$  must be added to  $I_j$  by step II.B of Algorithm 4.1.1 so  $([r, i', s], j)$  must have been on the pending list,  $H$ , and  $([r, i', s], j)$  must have been added to  $H$  while step II.B was processing an entry  $[q'_{h'-1}, l'_{h'-1}, q'_{h'}]$  on  $I_{l'_{h'}}$ , where  $h'$  is a convenient index which will be specified later. There are three types of moves step II.B could be simulating while processing  $[q'_{h'-1}, l'_{h'-1}, q'_{h'}]$  on  $I_{l'_{h'}}$ :

- a shift,
- a reduction by an empty production, or
- a reduction by a non-empty production.

The first type of move is a shift so  $l'_{h'} = i' = j - 1$ ,  $q'_{h'} = r$ , and, the sequence of moves corresponding to  $[r, i', s]$  on  $I_j$  is

$$(\alpha \mid r, w_{j:n+k+1}) \vdash (\alpha r s, w_{j+1:n+k+1}).$$

However, this sequence does not end with the move

$$(\alpha \mid r, w_{j+1:n+k+1}) \vdash (\alpha r s, w_{j+1:n+k+1})$$

which is a contradiction.

The second type of move is a reduction by an empty production so  $l'_{h'} = i' = j$ ,  $q'_{h'} = r$ , and the sequence of moves corresponding to  $[r, i', s]$  on  $I_j$  is

$$(\alpha \mid r, w_{j+1:n+k+1}) \vdash (\alpha r s, w_{j+1:n+k+1}).$$

However,  $i' = j$  implies  $i' = i$  and  $[r, i', s] = [r, i, s]$  which is a contradiction.

The third type of move is a reduction by a non-empty production so the sequence of moves corresponding to  $[r, i', s]$  on  $I_j$  is

$$(\alpha \mid r, w_{i'+1:n+k+1}) \stackrel{+}{\vdash} (\alpha r q'_1 \dots q'_{h'} \dots q'_{m'}, w_{j+1:n+k+1}) \vdash (\alpha r s, w_{j+1:n+k+1}).$$

However, this sequence of moves does not end with the move

$$(\alpha r, w_{j+1:n+k+1}) \vdash (\alpha r s, w_{j+1:n+k+1})$$

which is a contradiction.

In the third case, where step II.B.2.iii adds  $([r, i, s], j)$  to  $H$ , Algorithm 4.1.1 is simulating a reduction by a non-empty production for the LR( $k$ ) parser. Thus,



$(h, p, j) \in G_{l_h}(q_h)$ ,  $(r, i) \in B_{h-1}$ , and, applying Theorem 4.2.1, the LR( $k$ ) parser can make the sequence of moves

$$\begin{aligned} (0, w_{1:n+k+1}) &\stackrel{*}{\vdash} (\alpha \setminus r, w_{i+1:n+k+1}) \\ &\stackrel{+}{\vdash} (\alpha r q_1 q_2 \dots q_m, w_{j+1:n+k+1}) \\ &\vdash (\alpha r s, w_{j+1:n+k+1}) \end{aligned}$$

where  $m = |\text{RHS}(p)|$  and **reduce**  $p \in f_{q_m}(w_{j+1:j+k})$ .

For this case, the lemma is proved by assuming it does not hold for the  $N^{\text{th}}$  entry  $[r, i, s]$  on  $I_j$  and showing that this assumption leads to a contradiction. If  $[r', i', s']$  on  $I_{j'}$  is an entry on the ordered list of entries which follows  $[r, i, s]$  on  $I_j$  and for which the lemma does not hold, the sequence of moves

$$(\beta \setminus r', w_{i'+1:n+k+1}) \stackrel{+}{\vdash} (\beta r' s', w_{j'+1:n+k+1})$$

must be embedded in the sequence of moves

$$(0, w_{1:n+k+1}) \stackrel{*}{\vdash} (\alpha \setminus r, w_{i+1:n+k+1}) \stackrel{+}{\vdash} (\alpha r s, w_{j+1:n+k+1}).$$

Applying Lemma 4.2.4 and Lemma 4.2.5, there must be entries

$$\begin{aligned} &[r, i, q_1] \text{ on } I_{l_1} \\ &[q_1, l_1, q_2] \text{ on } I_{l_2} \\ &\vdots \\ &[q_{m-2}, l_{m-2}, q_{m-1}] \text{ on } I_{l_{m-1}} \\ &[q_{m-1}, l_{m-1}, q_m] \text{ on } I_j \end{aligned}$$

which precede  $[r, i, s]$  on  $I_j$  in the ordered list of entries. The lemma holds the entries  $[q_{x-1}, l_{x-1}, q_x]$  on  $I_{l_x}$  for  $1 \leq x \leq m$  since they precede  $[r, i, s]$  on  $I_j$  on the ordered list of entries. Any entry on the ordered list of entries that follows  $[r, i, s]$  on  $I_j$  also follows  $[q_{x-1}, l_{x-1}, q_x]$  on  $I_{l_x}$  for  $1 \leq x \leq m$ . Therefore, the sequence of moves

$$(\beta \setminus r', w_{i'+1:n+k+1}) \stackrel{+}{\vdash} (\beta r' s', w_{j'+1:n+k+1})$$

can not be embedded in the sequence of moves

$$\begin{aligned} (0, w_{1:n+k+1}) &\stackrel{*}{\vdash} (\delta \setminus r, w_{i+1:n+k+1}) \\ &\stackrel{+}{\vdash} (\delta r \setminus q_1, w_{l_1+1:n+k+1}) \\ &\stackrel{+}{\vdash} (\delta r q_1 \setminus q_2, w_{l_2+1:n+k+1}) \\ &\vdots \\ &\stackrel{+}{\vdash} (\delta r q_1 q_2 \dots q_m, w_{j+1:n+k+1}) \end{aligned}$$

and, letting  $\alpha = \delta$ , it must end with the move

$$(\alpha r q_1 q_2 \dots q_m, w_{l_m+1:n+k+1}) \vdash (\alpha r s, w_{j+1:n+k+1}).$$

Thus,  $(\beta r' s', w_{j'+1:n+k+1}) = (\alpha r s, w_{j+1:n+k+1})$  which implies that  $\beta = \alpha$ ,  $r' = r$ ,  $s' = s$ ,  $j' = j$  and  $i' \neq i$  since  $[r, i, s]$  is distinct from  $[r, i', s]$ .

The entry  $[r, i', s]$  must be added to  $I_j$  by step II.B of Algorithm 4.1.1 so  $([r, i', s], j)$  must have been on the pending list,  $H$ , and  $([r, i', s], j)$  must have been added to  $H$  while step II.B was processing an entry  $[q'_{h'-1}, l'_{h'-1}, q'_{h'}]$  on  $I_{l'_{h'}}$ , where  $h'$  is a convenient index which will be specified later. There are three types of moves step II.B could be simulating while processing  $[q'_{h'-1}, l'_{h'-1}, q'_{h'}]$  on  $I_{l'_{h'}}$ :

- a shift,
- a reduction by an empty production, or
- a reduction by a non-empty production.

The first type of move is a shift so  $l'_{h'} = i' = j - 1$ ,  $q'_{h'} = r$ , and, the sequence of moves corresponding to  $[r, i', s]$  on  $I_j$  is

$$(\alpha \lambda r, w_{j:n+k+1}) \vdash (\alpha r s, w_{j+1:n+k+1}).$$

However, this sequence of moves does not end with the move

$$(\alpha r q_1 q_2 \dots q_m, w_{j+1:n+k+1}) \vdash (\alpha r s, w_{j+1:n+k+1})$$

which is a contradiction.

The second type of move is a reduction by an empty production so  $l'_{h'} = i' = j$ ,  $q'_{h'} = r$ , and the sequence of moves corresponding to  $[r, i', s]$  on  $I_j$  is

$$(\alpha \lambda r, w_{j+1:n+k+1}) \vdash (\alpha r s, w_{j+1:n+k+1}).$$

However, this sequence of moves does not end with the move

$$(\alpha r q_1 q_2 \dots q_m, w_{l_m+1:n+k+1}) \vdash (\alpha r s, w_{j+1:n+k+1})$$

which is a contradiction.

The third type of move is a reduction by a non-empty production so the sequence of moves corresponding to  $[r, i', s]$  on  $I_j$  is

$$(\alpha \lambda r, w_{i'+1:n+k+1}) \vdash^+ (\alpha r q'_1 \dots q'_{h'} \dots q'_{m'}, w_{j+1:n+k+1}) \vdash (\alpha r s, w_{j+1:n+k+1}).$$

This sequence can end with the move

$$(\alpha r q_1 q_2 \dots q_m, w_{l_{m+1}:n+k+1}) \vdash (\alpha r s, w_{j+1:n+k+1})$$

only if  $m' = m$  and  $r q'_1 \dots q'_{h'} \dots q'_{m'} = r q_1 q_2 \dots q_m$ . However, this implies that either

$$(\alpha \wr r, w_{i'+1:n+k+1}) \vdash^* (\alpha \wr r, w_{i+1:n+k+1})$$

or

$$(\alpha \wr r, w_{i+1:n+k+1}) \vdash^* (\alpha \wr r, w_{i'+1:n+k+1}).$$

Either of these two sequences of moves is possible only if  $i' = i$  which is a contradiction. ■

The next two lemmas show that, for  $LR(k)$  grammars,  $|B_i|$  and  $|G_i(q)|$  are always less than or equal to one.

**LEMMA 4.4.2** ( $|B_i| \leq 1$ ) *Given the same  $Q$ ,  $f_q$ , and  $g_q$  for Algorithm 4.1.1 and for the canonical  $LR(k)$  parser for an  $LR(k)$  grammar, if step II.B.2.ii of Algorithm 4.1.1 computes  $B_{-1}, B_0, \dots, B_{m-1}$  then, for  $-1 \leq i \leq m-1$ ,  $|B_i| \leq 1$ .*

*Proof:* When step II.B.2.ii of Algorithm 4.1.1 computes  $B_{-1}, B_0, \dots, B_{m-1}$ , step II.B must be processing an entry  $[q_{h-1}, l_{h-1}, q_h]$  on  $I_{l_h}$  for which

$$(h, p, j) \in G_{l_h}(q_h).$$

Here,  $h$  has been chosen as a convenient index. Since  $[q_{h-1}, l_{h-1}, q_h]$  is on  $I_{l_h}$ , Lemma 4.2.3 can be applied repeatedly to show that there must be

$$\begin{aligned} & [q_0, l_0, q_1] \text{ on } I_{l_1} \\ & [q_1, l_1, q_2] \text{ on } I_{l_2} \\ & \vdots \\ & [q_{h-2}, l_{h-2}, q_{h-1}] \text{ on } I_{l_{h-1}} \end{aligned}$$

which precede  $[q_{h-1}, l_{h-1}, q_h]$  on  $I_{l_h}$  in an ordered list of entries. Furthermore, Lemma 4.3.1 shows that there must be

$$\begin{aligned} & (q_h, l_h) \in B_{-1} \\ & (q_{h-1}, l_{h-1}) \in B_0 \\ & \vdots \\ & (q_0, l_0) \in B_{h-1}. \end{aligned}$$

Applying Lemma 4.2.5, there must also be

$$\begin{array}{l}
 [q_h, l_h, q_{h+1}] \text{ on } I_{l_{h+1}} \\
 [q_{h+1}, l_{h+1}, q_{h+2}] \text{ on } I_{l_{h+2}} \\
 \vdots \\
 [q_{m-2}, l_{m-2}, q_{m-1}] \text{ on } I_{l_{m-1}} \\
 [q_{m-1}, l_{m-1}, q_m] \text{ on } I_{l_m}
 \end{array}$$

where  $l_m = j$ ,  $m = |\text{RHS}(p)|$  and **reduce**  $p \in f_{q_m}(w_{l_m+1:l_m+k})$ . Applying Theorem 4.2.1 shows that the canonical LR( $k$ ) parser can make the sequence of moves

$$\begin{array}{l}
 (0, w_{1:n+k+1}) \vdash^* (\alpha \wr q_0, w_{l_0+1:n+k+1}) \\
 \vdash^+ (\alpha q_0 \wr q_1, w_{l_1+1:n+k+1}) \\
 \vdash^+ \dots \\
 \vdash^+ (\alpha q_0 q_1 \dots \wr q_{h-(i+1)}, w_{l_{h-(i+1)}+1:n+k+1}) \\
 \vdash^+ (\alpha q_0 q_1 \dots q_{h-(i+1)} \wr q_{h-i}, w_{l_{h-i}+1:n+k+1}) \\
 \vdash^+ \dots \\
 \vdash^+ (\alpha q_0 q_1 \dots q_{h-(i+1)} q_{h-i} \dots \wr q_{h-1}, w_{l_{h-1}+1:n+k+1}) \\
 \vdash^+ (\alpha q_0 q_1 \dots q_{h-(i+1)} q_{h-i} \dots q_{h-1} \wr q_h, w_{l_h+1:n+k+1}) \\
 \vdash^+ \dots \\
 \vdash^+ (\alpha q_0 q_1 \dots q_{h-(i+1)} q_{h-i} \dots q_{h-1} q_h \dots \wr q_m, w_{j+1:n+k+1}) \\
 \vdash (\alpha q_0 s, w_{j+1:n+k+1}).
 \end{array}$$

The lemma is proved by assuming there are two distinct entries  $(q_{h-(i+1)}, l_{h-(i+1)})$  and  $(q'_{h-(i+1)}, l'_{h-(i+1)})$  in  $B_i$ , and showing that this assumption leads to a contradiction. Note that  $i$  must be greater than zero since examining Algorithm 4.1.2 shows that  $B_{-1} = \{(q_h, l_h)\}$  and  $B_0 = \{(q_{h-1}, l_{h-1})\}$ . Lemma 4.2.3 can be applied to  $(q'_{h-(i+1)}, l'_{h-(i+1)})$  in  $B_i$  to show that there must be entries

$$\begin{array}{l}
 [q'_0, l'_0, q'_1] \text{ on } I'_{l'_1} \\
 [q'_1, l'_1, q'_2] \text{ on } I'_{l'_2} \\
 \vdots \\
 [q'_{h-(i+2)}, l'_{h-(i+2)}, q'_{h-(i+1)}] \text{ on } I'_{l'_{h-(i+1)}}
 \end{array}$$

which precede  $[q_{h-1}, l_{h-1}, q_h]$  on  $I_{l_h}$  in the ordered list of entries. Applying Theorem 4.2.1 shows that the canonical  $\text{LR}(k)$  parser can make the sequence of moves

$$\begin{aligned}
 (0, w_{1:n+k+1}) &\stackrel{*}{\vdash} (\beta \wr q'_0, w_{l'_0+1:n+k+1}) \\
 &\stackrel{+}{\vdash} (\beta q'_0 \wr q'_1, w_{l'_1+1:n+k+1}) \\
 &\stackrel{+}{\vdash} \dots \\
 &\stackrel{+}{\vdash} (\beta q'_0 q'_1 \dots \wr q'_{h-(i+1)}, w_{l_{h-(i+1)}+1:n+k+1}) \\
 &\stackrel{+}{\vdash} (\beta q'_0 q'_1 \dots q'_{h-(i+1)} \wr q_{h-i}, w_{l_{h-i}+1:n+k+1}) \\
 &\stackrel{+}{\vdash} \dots \\
 &\stackrel{+}{\vdash} (\beta q'_0 q'_1 \dots q'_{h-(i+1)} q_{h-i} \dots \wr q_{h-1}, w_{l_{h-1}+1:n+k+1}) \\
 &\stackrel{+}{\vdash} (\beta q'_0 q'_1 \dots q'_{h-(i+1)} q_{h-i} \dots q_{h-1} \wr q_h, w_{l_{h-1}+1:n+k+1}) \\
 &\stackrel{+}{\vdash} \dots \\
 &\stackrel{+}{\vdash} (\beta q'_0 q'_1 \dots q'_{h-(i+1)} q_{h-i} \dots q_{h-1} q_h \dots \wr q_m, w_{j+1:n+k+1}) \\
 &\vdash (\beta q'_0 s, w_{j+1:n+k+1}).
 \end{aligned}$$

Since the canonical  $\text{LR}(k)$  parser for  $\text{LR}(k)$  grammars is deterministic, the two sequences of moves

$$(0, w_{1:n+k+1}) \stackrel{*}{\vdash} (\alpha \wr q_0, w_{l_0+1:n+k+1}) \stackrel{+}{\vdash} (\alpha q_0 s, w_{j+1:n+k+1})$$

and

$$(0, w_{1:n+k+1}) \stackrel{*}{\vdash} (\beta \wr q'_0, w_{l'_0+1:n+k+1}) \stackrel{+}{\vdash} (\beta q'_0 s, w_{j+1:n+k+1})$$

must be subsequences of one larger sequence of moves. However,  $[q_{h-(i+2)}, l_{h-(i+2)}, q_{h-(i+1)}]$  on  $I_{l_{h-(i+1)}}$  and  $[q'_{h-(i+2)}, l'_{h-(i+2)}, q'_{h-(i+1)}]$  on  $I'_{l'_{h-(i+1)}}$  precede  $[q_{h-1}, l_{h-1}, q_h]$  on  $I_{l_h}$  on the ordered list of entries. Therefore, applying Lemma 4.4.1, the sequences of moves

$$(0, w_{1:n+k+1}) \stackrel{*}{\vdash} (\alpha \wr q_{h-(i+2)}, w_{l_{h-(i+2)}+1:n+k+1}) \stackrel{+}{\vdash} (\alpha q_{h-(i+2)} q_{h-(i+1)}, w_{l_{h-(i+1)}+1:n+k+1})$$

and the sequence of moves

$$(0, w_{1:n+k+1}) \stackrel{*}{\vdash} (\beta \wr q'_{h-(i+2)}, w_{l'_{h-(i+2)}+1:n+k+1}) \stackrel{+}{\vdash} (\beta q'_{h-(i+2)} q'_{l'_{h-(i+1)}}, w_{l'_{h-(i+1)}+1:n+k+1})$$

have the property that the sequence of moves

$$(\delta \setminus q_{h-1}, w_{l_{h-1}+1:n+k+1}) \vdash^+ (\delta q_{h-1} q_h, w_{l_h+1:n+k+1})$$

is not contained in either sequence. But the sequence of moves

$$(0, w_{1:n+k+1}) \vdash^* (\alpha \setminus q_0, w_{l_0+1:n+k+1}) \vdash^+ (\alpha q_0 s, w_{j+1:n+k+1})$$

and the sequence of moves

$$(0, w_{1:n+k+1}) \vdash^* (\beta \setminus q'_0, w_{l'_0+1:n+k+1}) \vdash^+ (\beta q'_0 s, w_{j+1:n+k+1})$$

cannot be subsequences of a larger sequence of moves without this property being violated which is a contradiction. ■

LEMMA 4.4.3 ( $|G_i(q)| \leq 1$ ) *Given the same  $Q$ ,  $f_q$ , and  $g_q$  for Algorithm 4.1.1 and the canonical  $LR(k)$  parser for an  $LR(k)$  grammar,*

$$|G_i(q)| = 1$$

for all  $0 \leq i \leq n$  and  $q \in Q$ .

Proof: The lemma is proved by assuming there are two distinct entries  $(h, p, j)$  and  $(h', p', j')$  in  $G_{l_h}(q_h)$ , and showing that this assumption leads to a contradiction. Since the proof is the same if the roles of  $h$  and  $h'$  are reversed,  $h$  is assumed to be greater than  $h'$ . Applying Lemma 4.2.5 to  $(h, p, j)$  in  $G_{l_h}(q_h)$ , there must be

$$\begin{aligned} & [q_h, l_h, q_{h+1}] \text{ on } I_{l_{h+1}} \\ & [q_{h+1}, l_{h+1}, q_{h+2}] \text{ on } I_{l_{h+2}} \\ & \vdots \\ & [q_{m-2}, l_{m-2}, q_{m-1}] \text{ on } I_{l_{m-1}} \\ & [q_{m-1}, l_{m-1}, q_m] \text{ on } I_{l_m} \end{aligned}$$

where  $l_m = j$ ,  $m = |\text{RHS}(p)|$  and **reduce**  $p \in f_{q_m}(w_{l_m+1:l_m+k})$ . Also, applying Lemma 4.2.5 to  $(h', p', j')$  in  $G_{l_h}(q_h)$  and letting  $q_{h'} = q_h$  and  $l_{h'} = l_h$ , there must be

$$\begin{aligned} & [q_{h'}, l_{h'}, q'_{h'+1}] \text{ on } I'_{l'_{h'+1}} \\ & [q'_{h'+1}, l'_{h'+1}, q'_{h'+2}] \text{ on } I'_{l'_{h'+2}} \\ & \vdots \\ & [q'_{m'-2}, l'_{m'-2}, q'_{m'-1}] \text{ on } I'_{l'_{m'-1}} \\ & [q'_{m'-1}, l'_{m'-1}, q'_{m'}] \text{ on } I'_{l'_{m'}} \end{aligned}$$

where  $l'_{m'} = j'$ ,  $m' = |\text{RHS}(p')|$  and reduce  $p' \in f_{q'_{m'}}(w_{l'_{m'}+1:l'_{m'}+k})$ . Since  $[q_h, l_h, q_{h+1}]$  is on  $I_{l_{h+1}}$ , Lemma 4.2.3 can be applied repeatedly to show that there must be entries

$$\begin{aligned} & [q_0, l_0, q_1] \text{ on } I_{l_1} \\ & [q_1, l_1, q_2] \text{ on } I_{l_2} \\ & \vdots \\ & [q_{h-(h'+1)}, l_{h-(h'+1)}, q_{h-h'}] \text{ on } I_{l_{h-h'}} \\ & \vdots \\ & [q_{h-2}, l_{h-2}, q_{h-1}] \text{ on } I_{l_{h-1}} \\ & [q_{h-1}, l_{h-1}, q_h] \text{ on } I_{l_h} \end{aligned}$$

which precede  $[q_h, l_h, q_{h+1}]$  on  $I_{l_{h+1}}$  in an ordered list of entries. Applying Theorem 4.2.1 shows that the canonical  $\text{LR}(k)$  parser can make the sequence of moves

$$\begin{aligned} (0, w_{1:n+k+1}) & \stackrel{*}{\vdash} (\alpha \setminus q_0, w_{l_0+1:n+k+1}) \\ & \vdash (\alpha q_0 \setminus q_1, w_{l_1+1:n+k+1}) \\ & \vdash \dots \\ & \vdash (\alpha q_0 q_1 \dots \setminus q_{h-h'}, w_{l_{h-h'}+1:n+k+1}) \\ & \vdash \dots \\ & \vdash (\alpha q_0 q_1 \dots q_{h-h'} \dots \setminus q_m, w_{j+1:n+k+1}) \\ & \vdash (\alpha q_0 s, w_{j+1:n+k+1}). \end{aligned}$$

Also, applying Theorem 4.2.1 shows that the canonical  $\text{LR}(k)$  parser can make the sequence of moves

$$\begin{aligned} (0, w_{1:n+k+1}) & \stackrel{*}{\vdash} (\alpha \setminus q_0, w_{l_0+1:n+k+1}) \\ & \vdash (\alpha q_0 \setminus q_1, w_{l_1+1:n+k+1}) \\ & \vdash \dots \\ & \vdash (\alpha q_0 q_1 \dots \setminus q_{h-h'}, w_{l_{h-h'}+1:n+k+1}) \\ & \vdash \dots \\ & \vdash (\alpha q_0 q_1 \dots q_{h-h'} \dots \setminus q_{h'}, w_{l_{h'}+1:n+k+1}) \\ & \vdash (\alpha q_0 q_1 \dots q_{h-h'} \dots q_{h'} \setminus q'_{h'+1}, w_{l_{h'+1}+1:n+k+1}) \\ & \vdash \dots \end{aligned}$$

$$\begin{array}{l}
\vdash^+ (\alpha q_0 q_1 \dots q_{h-h'} \dots q_{h'} q'_{h'} \dots \lambda q'_{m'}, w_{j'+1:n+k+1}) \\
\vdash (\alpha q_0 q_1 \dots q_{h-h'} s', w_{j'+1:n+k+1}).
\end{array}$$

Since the canonical  $\text{LR}(k)$  parser for  $\text{LR}(k)$  grammars is deterministic, the two sequences of moves must be subsequences of one larger sequence of moves. Furthermore, this larger sequence of moves must have the sequence of moves

$$(\alpha q_0 q_1 \dots \lambda q_{h-h'} w_{l_{h-h'}+1:n+k+1}) \vdash^+ (\alpha q_0 q_1 \dots q_{h-h'} s', w_{j'+1:n+k+1})$$

embedded in the sequence of moves

$$(\alpha \lambda q_0, w_{l_0+1:n+k+1}) \vdash^+ (\alpha q_0 s, w_{j+1:n+k+1}).$$

This must be the case since, if the configuration  $(\alpha q_0 q_1 \dots q_x, w_{l_x+1:n+k+1})$  appears more than once in the sequence of moves, the canonical  $\text{LR}(k)$  parser will enter an infinite loop when  $w$  is its input. Thus, the only possible sequence of moves is

$$\begin{array}{l}
(0, w_{1:n+k+1}) \vdash^* (\alpha \lambda q_0, w_{l_0+1:n+k+1}) \\
\vdash^+ (\alpha q_0 \lambda q_1, w_{l_1+1:n+k+1}) \\
\vdash^+ \dots \\
\vdash^+ (\alpha q_0 q_1 \dots \lambda q_{h-h'}, w_{l_{h-h'}+1:n+k+1}) \\
\vdash^+ \dots \\
\vdash^+ (\alpha q_0 q_1 \dots q_{h-h'} \dots \lambda q_{h'}, w_{l_{h'}+1:n+k+1}) \\
\vdash^+ (\alpha q_0 q_1 \dots q_{h-h'} \dots q_{h'} \lambda q'_{h'+1}, w_{l_{h'+1}+1:n+k+1}) \\
\vdash^+ \dots \\
\vdash^+ (\alpha q_0 q_1 \dots q_{h-h'} \dots q_{h'} q'_{h'} \dots \lambda q'_{m'}, w_{j'+1:n+k+1}) \\
\vdash (\alpha q_0 q_1 \dots \lambda q_{h-h'} s', w_{j'+1:n+k+1}) \\
\vdash^* (\alpha q_0 q_1 \dots q_{h-h'} \lambda q_{h-(h'-1)}, w_{l_{h-(h'-1)}+1:n+k+1}) \\
\vdash^+ \dots \\
\vdash^+ (\alpha q_0 q_1 \dots \lambda q_h, w_{l_h+1:n+k+1}) \\
\vdash^+ \dots
\end{array}$$



$$\begin{array}{l}
+ \\
\vdash (\alpha q_0 q_1 \dots q_m, w_{j+1:n+k+1}) \\
\vdash (\alpha q_0 s, w_{j+1:n+k+1}).
\end{array}$$

However, the configuration  $(\alpha q_0 q_1 \dots q_h, w_{l_h+1:n+k+1})$  appears twice in this sequence of moves which implies the canonical  $\text{LR}(k)$  parser will enter an infinite loop when  $w$  is its input. This situation can be avoided only if  $h = h' = 0$  and  $m = m' = 0$ , but this implies  $j = j' = l_h$  and that  $f_{q_m}(w_{l_m+1:l_m+k})$  contains two different reductions. This is a contradiction since the canonical  $\text{LR}(k)$  parser for an  $\text{LR}(k)$  grammar is deterministic. ■

Now, showing  $\mathcal{O}(n)$  space complexity for  $\text{LR}(k)$  grammars is straightforward. It is well known that the canonical  $\text{LR}(k)$  parser for an  $\text{LR}(k)$  grammar makes only  $\mathcal{O}(n)$  moves when either accepting or rejecting an input string. Since the addition of an entry to the pending list,  $H$ , in Algorithm 4.1.1 corresponds to a move by the canonical  $\text{LR}(k)$  parser, there are only  $\mathcal{O}(n)$  entries on the pending list,  $H$ , and on the parse lists. Furthermore, Lemma 4.4.2 shows that the sets  $B_i$  only require  $\mathcal{O}(1)$  space and Lemma 4.4.3 shows that the sets  $G_i(q)$  only require  $\mathcal{O}(n)$  space. Therefore, the space complexity of Algorithm 4.1.1 is  $\mathcal{O}(n)$  for  $\text{LR}(k)$  grammars.

Showing  $\mathcal{O}(n)$  time complexity for  $\text{LR}(k)$  grammars is also straightforward. Together, Lemma 4.4.2 and Lemma 4.4.3 guarantee that step II.B.2.iii adds only one entry to the pending list,  $H$ . Also, duplicate entries will not occur because the canonical  $\text{LR}(k)$  parser for an  $\text{LR}(k)$  grammar does not loop. This also allows checks for duplicate entries to be eliminated from the algorithm. Finally, Lemma 4.4.2 and Lemma 4.4.3 imply that Algorithm 4.4.1 has an  $\mathcal{O}(1)$  time complexity. This can be seen by noting that all the loops depend upon the sizes of these sets, except for the loop in step II.B.2 of the algorithm which scans  $I_h$ . But each parse list  $I_h$  is organized as  $|Q|$  parse lists where entries of the form  $[q, i, r]$  are stored on the  $r^{\text{th}}$  list. Thus, the loop in step II.B.2 only needs to examine the  $t^{\text{th}}$  list and this list can contain at most one entry since each entry on it implies an element is in  $B_{l-1}$ . Combining the preceding points shows that Algorithm 4.1.1 has  $\mathcal{O}(n)$  time complexity for  $\text{LR}(k)$  grammars when the canonical  $\text{LR}(k)$  parser is used.

## CHAPTER V

THE LEAST-COST LR( $k$ ) EARLY'S ALGORITHM

In this chapter, the Least-Cost LR( $k$ ) Early's Algorithm is developed, proved correct, and analyzed to determine its time and space complexities. The Least-Cost LR( $k$ ) Early's Algorithm is a modification of the Depth-First LR( $k$ ) Early's Algorithm which finds a least-cost edit of a string such that the resulting string is in the language accepted by the underlying LR( $k$ ) parser.

## 5.1 THE ALGORITHM

The Least Cost LR( $k$ ) Early's Algorithm, Algorithm 5.1.1, can be viewed as being derived from the Depth-First LR( $k$ ) Early's Algorithm. From this point of view, there are three basic differences between the two algorithms. First, Algorithm 5.1.1 simulates an explicitly-advancing LR( $k$ ) parser instead of a standard LR( $k$ ) parser. Second, Algorithm 5.1.1 edits the input string as it is scanned and simulates multiple parsers, each of which uses a different edit as its input string. Third, Algorithm 5.1.1 computes the cost of the edits required to produce an entry on a parse list.

These differences are reflected by the change in the format of an entry from an ordered triple to an ordered septuple  $[q, u, i, c, d, r, v]$ , where  $q$  and  $r$  are states of the LR( $k$ ) parser,  $u$  and  $v$  are lookahead strings,  $i$  is the number of a parse list, and  $c$  and  $d$  are non-negative integers, called the cost components. The lookahead strings,  $u$  and  $v$  must be present in each entry because a lookahead string is an explicit part of any configuration for an explicitly-advancing LR( $k$ ) parser. The first cost component,  $c$ , is the cost of the edits applied up to the point at which an explicitly-advancing LR( $k$ ) parser is in a configuration  $(\alpha q, u, z_{i+1:n+k+1})$ . The second cost component,  $d$ , is the cost of the edits applied during the moves from  $(\alpha q, u, z_{i+1:n+k+1})$  to  $(\alpha qr, v, z_{j+1:n+k+1})$ . Together, the two cost components represent the total cost of reaching the configuration  $(\alpha qr, w, z_{j+1:n+k+1})$ .

Because Algorithm 5.1.1 simulates explicitly-advancing LR( $k$ ) parsers, it uses  $n + k + 2$  parse lists,  $I_0, I_1, \dots, I_{n+k+1}$ , instead of the  $n + 2$  parse lists used by Algorithm 4.1.1. This change arises because; unlike Algorithm 4.1.1, where an

ALGORITHM 5.1.1 *Least-Cost LR(k) Early's Algorithm*

I. Place  $[0, e, 0, 0, 0, e]$  on  $I_0$ ; place  $([0, e, 0, 0, 0, e], 0)$  on the pending list,  $H$ ; and append  $\$^{k+1}$  to  $z$ .

II. While  $H$  is not empty, perform the following steps:

A. Remove an entry  $([q, v, i, c, d, r, w], j)$  with the least  $c + d$  from  $H$ .

B. If  $|w| < k$  then perform the following steps:

1. Add  $([q, v, i, c, d, r, wz_{j+1:j+1}], j + 1)$  to  $H$ .
2. If  $z_{j+1:j+1} \neq \$$  then let  $b = W((z_{j+1:j+1} \mapsto \epsilon))$  and add  $([q, v, i, c, d + b, r, w], j + 1)$  to  $H$ .
3. If  $z_{j,j} \neq \$$  then for each  $a \in (\Sigma - \{z_{j+1:j+1}\})$ , let  $b = W((\epsilon \mapsto a))$  and add  $([q, v, i, c, d + b, r, wa], j)$  to  $H$ .
4. If  $z_{j+1:j+1} \neq \$$  then for each  $a \in (\Sigma - \{z_{j+1:j+1}\})$ , let  $b = W((z_{j+1:j+1} \mapsto a))$  and add  $([q, v, i, c, d + b, r, wa], j + 1)$  to  $H$ .

C. If  $|w| = k$  then, if there is not an entry  $[q, v, i, ?, ?, r, w]$  on  $I_j$ , add  $[q, v, i, c, d, r, w]$  to  $I_j$  and perform the following steps:

1. If there was not an entry  $[?, ?, ?, ?, ?, r, w]$  on  $I_j$  before  $[q, v, i, c, d, r, w]$  was added then perform the following steps:
  - i. If  $\text{shift} \in f_r(w)$  and  $s \in g_r(w_{1:1})$  then add  $([r, w, j, c + d, 0, s, w_{2:k}], j)$  to  $H$ .
  - ii. For each **reduce**  $p \in f_r(w)$ , let  $G_j(r, w) = G_j(r, w) \cup \{(|\text{RHS}(p)|, 0, p, w, j) \mid \text{reduce } p \in f_r(w)\}$ .
2. If  $G_j(r, w)$  is not empty then perform the following steps:
  - i. Let  $m = \max(\{h \mid (h, b, p, w, o) \in G_j(r, w)\})$ .
  - ii. Use Algorithm 5.1.2 to compute  $B_{-1}, B_0, \dots, B_{m-1}$ .
  - iii. For each  $(h, b', p, u, l) \in G_j(r, w)$ ,  $(s, t, o, a, b) \in B_{h-1}$  and  $x \in g_r(\text{LHS}(p))$ , add  $([s, t, o, a, b + b', x, u], l)$  to  $H$ .

entry  $[q, i, r]$  on  $I_j$  uses the lookahead string  $z_{j+1:j+k+1}$ ; Algorithm 5.1.1 uses the lookahead string  $v$  for an entry  $[q, u, i, c, d, r, v]$  on  $I_j$  and none of the symbols in  $z_{j+1:n+k+1}$  have been scanned. The extra parse lists,  $I_{n+2}, I_{n+3}, \dots, I_{n+k+1}$  are needed so the  $k$  end markers that terminate the string can be scanned.

### ALGORITHM 5.1.2 *Calculate $B_l$ 's*

Let  $[q, v, i, c', d', r, w]$  on  $I_j$  be the entry for which the  $B_l$ 's are to be calculated and let  $m$  be as given in Algorithm 5.1.1.

- I. Let  $l = 1$ ,  $B_{-1} = \{(r, w, j, c' + d', 0)\}$ ,  $B_0 = \{(q, v, j, c', d')\}$ , and let  $G_i(q, v) = G_{i,v}(q) \cup \{(n, b + d', p, w, k) \mid (n + 1, b, p, w, k) \in G_j(r, w) \text{ and } n > 0\}$ .
- II. While  $l < m$  perform the following steps:
  - A. Initialize  $B_l$  and get an element  $(s', u', h', a', b')$  from  $B_{l-1}$ .
  - B. While  $B_{l-1}$  is not exhausted perform the following steps:
    1. Get an entry  $[s_l, u_l, h_l, c_{l-1}, d_{l-1}, s_{l-1}, u_{l-1}]$  on  $I_{h'}$ .
    2. While  $I_{h'}$  is not exhausted perform the following steps:
      - i. If  $s' = s_{l-1}$  and  $u' = u_{l-1}$  then add the element  $(s_l, u_l, h_l, c_{l-1}, b' + d_{l-1})$  to  $B_l$ , if it is not already in  $B_l$ , and let  $G_{h'}(s_l, u_l) = G_{h'}(s_l, u_l) \cup \{(n, b + d_{l-1}, p, w, k) \mid (n + 1, b, p, w, k) \in G_{h_{l-1}}(s_{l-1}, u_{l-1}) \text{ and } n > 0\}$ .
      - ii. Get another entry  $[s_l, u_l, h_l, c_{l-1}, d_{l-1}, s_{l-1}, u_{l-1}]$  on  $I_{h'}$ .
    3. Get another element  $(s', u', h', a', b')$  from  $B_{l-1}$ .
  - C. Let  $l = l + 1$ .

The impact of lookahead strings on Algorithm 5.1.1 is pervasive. This is because the same configuration may be reached using different edits of the input string and the states must be associated with their lookaheads so that the proper stack linkage is maintained when step II.C.2 simulates a reduction. In general, where ever a state occurs in Alogirthm 4.1.1, a paired state and lookahead string occurs in Algorithm 5.1.1. For example,  $G_j(r, w)$  has a lookahead string as an additional argument and the elements of the  $B_l$ 's contain a lookahead string paired with a state.

Once the impact of lookahead strings and cost components are taken into account, Algorithm 5.1.1 is almost identical to Algorithm 4.1.1. The major change is the additon of step II.B which simulates advances using each of the possible edits of the next input symbol. Specifically, step II.B.1 simulates the advance action of the explicitly-advancing  $LR(k)$  parser. Steps II.B.2 thru II.B.4 simulate the effect of editing the input string and then advancing using the result of the

edit; where step II.B.2 simulates deleting  $z_{j+1:j+1}$ , step II.B.3 simulates inserting  $a$  before  $z_{j+1:j+1}$ , and step II.B.4 simulates replacing  $z_{j+1:j+1}$  with another symbol  $a$ . It is important to note that steps II.B.2 thru II.B.4 do not edit  $\$^{k+1}$  since it is the end marker and is not actually part of the input string.

In actual practice, Algorithm 5.1.1 would not execute until the pending list is exhausted. Instead, it would execute until an entry of the form  $[q, u, i, c, d, f, \$]$  is added to  $I_{n+k+1}$ . This allows the algorithm to terminate as soon as the least-cost parse of the edited input string is found.

## 5.2 EXAMPLE EXECUTIONS

The advantages of Algorithm 5.1.1 can be seen using two examples. Both of these examples assume that Algorithm 5.1.1 terminates when an entry of the form  $[q, u, i, c, d, f, \$]$  is added to  $I_{n+k+1}$ . In the first example, the parse lists generated by Algorithm 5.1.1, using the grammar in Figure 1 and the input string

$$a * (a + a * a)$$

are given in Figure 10. Comparing Figure 10 to Figure 9 on page 49 shows that, if the input string is syntactically correct, the Least-Cost LR( $k$ ) Early's Algorithm generates parse tables that have the same number of entries as the parse tables generated by the LR( $k$ ) Early's Algorithm, when the differences between the LR( $k$ ) parser and explicitly-advancing LR( $k$ ) parser are taken into account.

In the second example, the Least-Cost LR( $k$ ) Early's Algorithm is compared to Lyon's algorithm, assuming the cost of all single token edits is one. The parse lists generated by Algorithm 5.1.1, using the grammar in Figure 1 and the input string

$$a * a +$$

are given in Figure 11. For this example, step II.B of Algorithm 5.1.1 is assumed to have been changed so that only tokens that are valid lookaheads can be edited into the lookahead string for an entry. This change makes the algorithm comparable to Lyon's algorithm. The change is not used in this dissertation because it needlessly complicates the proofs of correctness and completeness for the algorithm.

Comparing Figure 11 to Figure 7 on page 40 shows that the Least-Cost LR( $k$ ) Early's Algorithm can avoid much of the useless work performed by Lyon's algorithm. The Least-Cost Early's Algorithm only examines the possible single token edits since a single token edit can repair the input string. Lyon's algorithm examines many multiple token edits even though it is given the syntactically correct

$I_0$ [0, $\epsilon$ , 0, 0, 0, 0, $\epsilon$ ]	$I_1$ [0, $\epsilon$ , 0, 0, 0, 0, $a$ ]	$I_2$ [0, $a$ , 0, 0, 0, $a_1$ , *] [0, $a$ , 0, 0, 0, $F_1$ , *] [0, $a$ , 0, 0, 0, $T_1$ , *]
$I_3$ [ $T_1$ , *, 2, 0, 0, $*_1$ , (]	$I_4$ [* <sub>1</sub> , (, 3, 0, 0, ( <sub>1</sub> , $a$ ]	$I_5$ [( <sub>1</sub> , $a$ , 4, 0, 0, $a_1$ , +] [( <sub>1</sub> , $a$ , 4, 0, 0, $F_1$ , +] [( <sub>1</sub> , $a$ , 4, 0, 0, $T_1$ , +] [( <sub>1</sub> , $a$ , 4, 0, 0, $E_1$ , +]
$I_6$ [ $E_1$ , +, 5, 0, 0, + <sub>2</sub> , $a$ ]	$I_7$ [+ <sub>2</sub> , $a$ , 6, 0, 0, $a_2$ , *] [+ <sub>2</sub> , $a$ , 6, 0, 0, $F_2$ , *] [+ <sub>2</sub> , $a$ , 6, 0, 0, $T_4$ , *]	$I_8$ [ $T_4$ , *, 7, 0, 0, $*_2$ , $a$ ]
$I_9$ [* <sub>2</sub> , $a$ , 8, 0, 0, $a_2$ , )] [* <sub>2</sub> , $a$ , 8, 0, 0, $F_4$ , )] [+ <sub>2</sub> , $a$ , 6, 0, 0, $T_4$ , )] [+ <sub>2</sub> , $a$ , 6, 0, 0, $E_1$ , )]	$I_{10}$ [ $E_1$ , ), 9, 0, 0, ) <sub>1</sub> , \$] [* <sub>1</sub> , (, 3, 0, 0, $F_3$ , \$] [0, $a$ , 1, 0, 0, $T_1$ , \$] [0, $a$ , 1, 0, 0, $E_3$ , \$]	$I_{11}$ [ $E_3$ , \$, 10, 0, 0, $f$ , \$]

Figure 10: Parse Lists for First Example

input string

$a * a.$

For the syntax error used in the example, the Least-Cost LR( $k$ ) Early's Algorithm would do even better if entries with the same cost were assumed to be removed from the pending list in last-in-first-out order.

These two examples demonstrate the inherent advantages of the Least-Cost LR( $k$ ) Early's Algorithm over other globally least-cost error recovery schemes:

- no extra work is done for correct input; and
- only edits which cost the same or less than the least-cost repair are examined.

$I_0$	$I_1$	$I_2$
$[0, \epsilon, 0, 0, 0, 0, \epsilon]$	$[0, \epsilon, 0, 0, 0, 0, a]$	$[0, a, 0, 0, 0, a_1, *]$
$[0, \epsilon, 0, 0, 1, 0, (]$	$[0, \epsilon, 0, 0, 1, 0, (]$	$[0, a, 0, 0, 0, F_1, *]$
	$[0, (, 0, 1, 0, (, a]$	$[0, a, 0, 0, 0, T_1, *]$
	$[0, a, 1, 0, 1, a_1, +]$	$[0, (, 1, 1, 0, (, *]$
	$[0, a, 1, 1, 0, F_1, +]$	$[(1, a, 1, 1, 0, a_2, *]$
	$[0, a, 1, 1, 0, T_1, +]$	$[(1, a, 1, 1, 0, F_2, *]$
	$[0, a, 1, 1, 0, E_3, +]$	$[(1, a, 1, 1, 0, T_2, *]$
		$[E_3, +, 1, 1, 0, +_1, *]$
		$[0, a, 0, 0, 1, a_1, +]$
		$[0, a, 0, 1, 0, F_1, +]$
		$[0, a, 0, 1, 0, T_1, +]$
		$[0, a, 0, 1, 0, E_3, +]$
		$[T_1, *, 2, 0, 1, *_1, (]$

Figure 11: Parse Lists for Second Example

### 5.3 PROOF OF CORRECTNESS

This section shows that Algorithm 5.1.1 correctly simulates an explicitly-advancing LR( $k$ ) parser for edits of the input string, when the algorithm and the parser use the same sets of states,  $Q$ , and functions  $f_q$  and  $g_q$ . The simulation is correct if for every entry  $[q, u, i, c, d, r, v]$  that Algorithm 5.1.1 places on a parse list  $I_j$ , there is a corresponding sequence of moves

$$(0, \epsilon, yy') \vdash^* (\alpha)q, u, y') \vdash^+ (\alpha)qr, v, \epsilon)$$

that can be made by the explicitly-advancing LR( $k$ ) parser and there exist edit sequences  $S$  and  $S'$  such that  $z_{1:n} \xrightarrow{S} y$ ,  $z_{i+1:j} \xrightarrow{S'} y'$ ,  $W(S) = c$  and  $W(S') = d$ .

Algorithm 5.1.1 uses the pending list,  $H$ , to hold entries waiting to be processed by the algorithm. The following lemma establishes that entries added to the pending list are eventually processed.

**LEMMA 5.3.1** (Every Entry Added to the Pending List is Processed) *If  $([q, u, i, c, d, r, v], j)$  is added to the pending list,  $H$ , then it will eventually be processed by step II.C of Algorithm 5.1.1.*

$I_3$	$I_4$	$I_5$
$[T_1, *, 2, 0, 0, *_1, a]$	$[*_1, a, 3, 0, 0, a_1, +]$	$[E_3, +, 4, 0, 0, +_1, \$]$
$[T_2, *, 2, 1, 0, *_2, a]$	$[*_1, a, 3, 0, 0, F_3, +]$	$[E_1, +, 4, 1, 0, +_1, \$]$
$[0, a, 0, 1, 0, a_1, a]$	$[0, a, 1, 0, 0, T_1, +]$	$[T_1, *, 4, 1, 0, *_1, \$]$
$[E_3, +, 2, 1, 0, +_1, a]$	$[0, a, 1, 0, 0, E_3, +]$	$[*_1, a, 3, 0, 1, a_1, \$]$
$[T_1, *, 2, 0, 1, *_1, (]$	$[*_2, a, 3, 1, 0, a_2, +]$	$[*_1, a, 3, 1, 0, F_3, \$]$
$[*_1, (, 2, 1, 0, (, a]$	$[*_2, a, 3, 1, 0, F_4, +]$	$[0, a, 1, 1, 0, T_1, \$]$
$[*_1, a, 3, 0, 1, a_1, *]$	$[(, a, 1, 1, 0, T_2, +]$	$[0, a, 1, 1, 0, E_3, \$]$
$[*_1, a, 3, 1, 0, F_3, *]$	$[(, a, 1, 1, 0, E_1, +]$	$[+_1, a, 4, 0, 1, a_1, \$]$
$[0, a, 1, 1, 0, T_1, *]$	$[+_1, a, 3, 1, 0, a_1, +]$	$[+_1, a, 4, 1, 0, F_1, \$]$
	$[+_1, a, 3, 1, 0, F_1, +]$	$[+_1, a, 4, 1, 0, T_3, \$]$
	$[+_1, a, 3, 1, 0, T_3, +]$	
	$[0, a, 0, 1, 0, E_3, +]$	
	$[(, a, 3, 1, 0, a_2, +]$	
	$[(, a, 3, 1, 0, F_2, +]$	
	$[(, a, 3, 1, 0, T_2, +]$	
	$[(, a, 3, 1, 0, E_1, +]$	
	$[*_1, (, 3, 1, 0, (, +]$	
	$[*_1, a, 3, 0, 1, a_1, *]$	
	$[*_1, a, 3, 1, 0, F_3, *]$	
	$[0, a, 1, 1, 0, T_1, *]$	
	$[T_1, *, 3, 1, 0, *_1, +]$	
	$[E_3, +, 4, 0, 1, +_1, a]$	
$I_6$		
$[E_3, \$, 5, 1, 0, f, \$]$		

Figure 11: continued

Proof: A entry can remain unprocessed only if Algorithm 5.1.1 adds an unbounded number of entries to the pending list,  $H$ . The lemma is proved by showing that the number of entries added to  $H$  is bounded. There are four steps that add entries to the pending list: step I; step II.B; step II.C.1.i; and step II.C.2.iii.

Before analyzing the steps that add entries to  $H$ , the number of entries on a parse list  $I_j$  must be shown to be bounded. Step II.C does not add duplicate entries of the form  $[q, u, i, ?, ?, r, v]$  to  $I_j$ , where  $|v| = k$ . Therefore, the number of



entries on  $I_j$  is bounded since  $q, u, i, r, v$  and  $j$  are all bounded.

Proceeding with the analysis of the steps that add entries to  $H$ , note that step I only adds one entry. Step II.C.1.i adds a bounded number of entries,  $([q, u, i, c, d, r, v], j)$  where  $|v| < k$ , since it adds at most one such entry for each entry of the form  $[q, u, i, ?, ?, r, v]$  on  $I_j$  where  $|v| = k$ .

Examining step II.B shows that it adds a bounded number of entries for each entry it processes. Step II.B only processes entries  $([q, u, i, c, d, r, v], j)$  for which  $|v| < k$ . The only other steps which can add these entries are step I and step II.C.1.i and it has already been shown that the number of entries added by these steps is bounded. Furthermore, when step II.B processes an entry  $([q, u, i, c, d, r, v], j)$ , either  $|v|$  or  $j$  is increased in the resulting entry. Both  $|v|$  and  $j$  are bounded so the total number of entries added by step II.B is bounded.

The number of entries added by step II.C.2.iii is bounded by

$$|G_j(r, v)| \cdot \max_{i=1}^m (|B_i|).$$

The elements of  $|G_j(r, v)|$  are of the form  $(h, b', p, u, l)$ . The values of  $h, p, u$  and  $l$  are bounded. Furthermore, the possible values of  $b'$  are bounded since  $b'$  is the sum of a bounded number of  $d$ 's from entries of the form  $[q, u, i, c, d, r, v]$  where  $|v| = k$ . Therefore,  $|G_j(r, v)|$  is bounded.

The elements of  $|B_i|$  are of the form  $(s, t, o, a, b)$ . The values of  $s, t$  and  $o$  are bounded. The possible values of  $a$  are bounded since  $a$  must be from an entry  $[q, u, i, a, d, r, v]$  where  $|v| = k$ . The possible values of  $b$  are bounded since  $b$  is the sum of a bounded number of  $d$ 's from entries of the form  $[q, u, i, c, d, r, v]$  where  $|v| = k$ . Therefore,  $|B_i|$  is bounded.

Since  $|G_j(r, v)|$  and  $|B_i|$  are bounded, the number of entries added by step II.C.2.iii is bounded. Each of the steps – step I, step II.B, step II.C.1.i, and step II.C.2.iii – which add entries to the pending list,  $H$ , has been shown to add a bounded number of entries for each entry it processes. Therefore, the number of entries added to  $H$  is bounded. ■

As in the proofs of correctness and completeness for Algorithm 4.1.1, the concept of a sequence in which entries can be added to their parse lists is important and leads to the following definition.

*Definition 5.3.1 (Ordered List of Entries)* An ordered list of entries is a list of

entries and their parse lists

$$\begin{aligned} & [q_1, u_1, i_1, c_1, d_1, r_1, v_1] \text{ on } I_{j_1} \\ & [q_2, u_2, i_2, c_2, d_2, r_2, v_2] \text{ on } I_{j_2} \\ & \vdots \\ & [q_N, u_N, i_N, c_N, d_N, r_N, v_N] \text{ on } I_{j_N} \end{aligned}$$

where  $r_x \neq 0$  for  $1 \leq x \leq N$  and the entries are given in a sequence in which they can be added to their parse lists by step II.C of Algorithm 5.1.1 during an execution of Algorithm 5.1.1.

Entries of the form  $[0, \epsilon, 0, 0, d, 0, u]$  on  $I_j$  are not on any ordered list of entries. They are excluded because  $[0, \epsilon, 0, 0, 0, 0, \epsilon]$  is not added to its parse list by step II.C of Algorithm 5.1.1.

Closely related to the concept of ordering a list of entries is the notion that, regardless of any specific ordering, an entry or an entry from a set of entries must be added to its parse list before another entry can be added to some other parse list. This notion is captured in the following definition.

*Definition 5.3.2 (Direct Precursor)* Given two entries  $[?, ?, ?, ?, ?, q, u]$  on  $I_i$  and  $[q, u, i, c, d, r, v]$  on  $I_j$ , the entry  $[?, ?, ?, ?, ?, q, u]$  on  $I_i$  is said to be a direct precursor of the entry  $[q, u, i, c, d, r, v]$  on  $I_j$ .

For Algorithm 5.1.1, the exact order in which entries are added to their parse lists or to the pending list,  $H$ , is critical to proving properties of the algorithm. The exact order is important because it affects the calculation of the cost components of the entries. The following definition defines a predecessor/successor relationship between entries on  $H$ . This relationship captures the idea of one entry being added to  $H$  due to the processing of another entry.

*Definition 5.3.3 (Direct Predecessor)* If step II.B or II.C processes an entry  $([q, u, h, a, b, r, v], l)$  and adds  $([s, w, i, c, d, t, y], j)$  to  $H$  then  $([q, u, h, a, b, r, v], l)$  is called the direct predecessor of  $([s, w, i, c, d, t, y], j)$  and  $([s, w, i, c, d, t, y], j)$  is called the direct successor of  $([q, u, h, a, b, r, v], l)$ .

The direct predecessor/successor relationship is denoted by  $\models$ , the transitive closure of the relationship is denoted by  $\models^+$ , and the reflexive and transitive closure of the relationship is denoted by  $\models^*$ . By convention,  $([0, \epsilon, 0, 0, 0, 0, \epsilon], 0)$  is a direct predecessor of itself. The predecessor/successor relationship is extended to entries on their parse lists with the following definition.

*Definition 5.3.4 (Direct Predecessor of an Entry on a Parse List)*

If  $[q, u, h, a, b, r, v]$  is on  $I_l$ ,  $[s, w, i, c, d_n, t, y_n]$  is added to  $I_{j_n}$  and

$$\begin{aligned} ([q, u, h, a, b, r, v], l) &\models ([s, w, i, c, d_1, t, y_1], j_1) \\ &\models ([s, w, i, c, d_2, t, y_2], j_2) \\ &\models \dots \\ &\models ([s, w, i, c, d_n, t, y_n], j_n) \end{aligned}$$

where  $|y_x| < k$  for  $1 \leq x < n$  then  $[q, u, h, a, b, r, v]$  on  $I_l$  is called the direct predecessor of  $[s, w, i, c, d_n, t, y_n]$  on  $I_{j_n}$  and  $[s, w, i, c, d_n, t, y_n]$  on  $I_{j_n}$  is called the direct successor of  $[q, u, h, a, b, r, v]$  on  $I_l$ .

For entries on their parse lists, the predecessor/successor relationship is denoted as

$$([q, u, h, a, b, r, v] \text{ on } I_l) \models ([s, w, i, c, d, t, y] \text{ on } I_j).$$

By convention,  $[0, \epsilon, 0, 0, 0, 0, \epsilon]$  on  $I_0$  is a direct predecessor of itself. Note that an entry has a unique predecessor and no entry is a predecessor of itself, except for  $[0, \epsilon, 0, 0, 0, 0, \epsilon]$  on  $I_0$ .

The following lemma establishes that step II.B of Algorithm 5.1.1 simulates the advance moves of an explicitly-advancing  $LR(k)$  parser for some edit of the input string.

**LEMMA 5.3.2 (Step II.B Simulates Advances)** *If  $([q, u, i, c, d, r, vw], j)$ , where  $|vw| = k$ , is added by step II.B of Algorithm 5.1.1 to the pending list,  $H$ , then either step I or step II.C.1.i of Algorithm 5.1.1 adds an entry  $([q, u, i, c, 0, r, v], i)$  to  $H$  where  $|v| < k$ ,*

$$([q, u, i, c, 0, r, v], i) \models^+ ([q, u, i, c, d, r, vw], j),$$

and the explicitly-advancing  $LR(k)$  parser can make the sequence of moves

$$(\alpha r, v, w) \vdash^{a^*} (\alpha r, vw, \epsilon)$$

for some edit sequence  $S$  such that  $d = W(S)$  and  $z_{i+1:j} \xrightarrow{S} w$ .

**Proof:** When step II.B processes an entry  $([q, u, i, c, d, r, v], i)$ , it adds an entry  $([q, u, i, c, d + b, r, va], j)$  to the pending list,  $H$ , for which  $(z_{i+1:j} \mapsto a)$ ,  $b = W((z_{i+1:j} \mapsto a))$ , and  $|va| > |v|$  or  $j > i$ . Thus, if step II.B adds an entry

$$([q, u, i, c, b' + \sum_{x=1}^l b_x, r, a_1 a_2 \dots a_l], j_l)$$

to  $H$ , it must be processing an entry

$$([q, u, i, c, b' + \sum_{x=1}^{l-1} b_x, r, a_1 a_2 \dots a_{l-1}], j_{l-1})$$

for which  $j_{l-1} \leq j_l$ ,  $(z_{j_{l-1}+1:j_l} \mapsto a_l)$  and  $b_l = W((z_{j_{l-1}+1:j_l} \mapsto a_l))$ .

Let  $([q, u, i, c, d, r, vw], j)$  be  $([q, u, i, c, b' + \sum_{x=1}^l b_x, r, va_1 a_2 \dots a_l], j_l)$  where  $w = a_1 \dots a_l$ . In general, there is a sequence of entries

$$\begin{aligned} ([q, u, i, c, b', r, v], j_0) & \models ([q, u, i, c, b' + b_1, r, va_1], j_1) \\ & \models \dots \\ & \models ([q, u, i, c, b' + \sum_{x=1}^{l-1} b_x, r, va_1 \dots a_{l-1}], j_{l-1}) \\ & \models ([q, u, i, c, b' + \sum_{x=1}^l b_x, r, va_1 \dots a_l], j_l) \end{aligned}$$

where  $([q, u, i, c, b', r, v], j_0)$  is the only entry in the sequence not added to  $H$  by step II.B. For  $1 \leq x \leq l$ , it must be that  $j_{x-1} \leq j_x$ ,  $(z_{j_{x-1}+1:j_x} \mapsto a_x)$ , and  $b_x = W((z_{j_{x-1}+1:j_x} \mapsto a_x))$ . Here,  $a_x$  may be  $\epsilon$  and  $|va_1 \dots a_l| = k$  even though  $l$  may be greater than  $k$ .

Since  $([q, u, i, c, b', r, v], j_0)$  is processed by step II.B,  $|v| < k$ , the only other steps that can add entries of the form  $([q, u, i, c, b', r, v], j_0)$ , where  $|v| < k$ , are step I and step II.C.i. Entries added by these steps have the form  $([q, u, i, c, 0, r, v], j_0)$ . Therefore,  $b' = 0$  and there exists an edit sequence

$$S = (z_{j_0+1:j_1} \mapsto a_1)(z_{j_1+1:j_2} \mapsto a_2) \dots (z_{j_{l-1}+1:j_l} \mapsto a_l)$$

for which  $W(S) = d = \sum_{x=1}^l b_x$ . Finally, inspection of the explicitly-advancing  $LR(k)$  parser shows that for any state  $r$  the parser can always make the sequence of moves

$$(\alpha r, v, w) \stackrel{a^*}{\vdash} (\alpha r, vw, \epsilon)$$

where  $|v| < k$  and  $|vw| = k$ . ■

The next lemma establishes a technical property of the initial state for Algorithm 5.1.1.

**LEMMA 5.3.3 (0 is the Unique Initial State)** *If  $[q, u, i, c, d, r, v]$  is on  $I_j$  and  $r = 0$  then  $q = 0$ ,  $u = \epsilon$ ,  $i = 0$ ,  $c = 0$ ,*

$$([0, \epsilon, 0, 0, 0, 0, \epsilon] \text{ on } I_0) \models ([0, \epsilon, 0, 0, d, 0, v] \text{ on } I_j),$$

and the explicitly-advancing  $LR(k)$  parser can make the sequence of moves

$$(0, \epsilon, v) \stackrel{a^*}{\vdash} (0, v, \epsilon)$$

for some edit sequence  $S$  such that  $d = W(S)$  and  $z_{1:j} \xrightarrow{S} v$ .

Proof: The definition of an  $LR(k)$  parser does not allow  $0 \in g_s(X)$  for any  $s$  or  $X$ . Thus, only step I or step II.B of Algorithm 5.1.1 can add  $([q, u, i, c, d, 0, vw], j)$  to  $H$ . If step I adds the entry, the lemma must be true since the only entry added by step I is  $([0, \epsilon, 0, 0, 0, 0, \epsilon], 0)$ .

If step II.B adds  $([q, u, i, c, d, 0, vw], j)$  to  $H$  then, applying Lemma 5.3.2, either step I or step II.C.1.i must add  $([q, u, i, c, 0, 0, v], l)$  to  $H$ , where

$$([q, u, i, c, 0, 0, v], l) \stackrel{+}{\models} ([q, u, i, c, d, 0, vw], j),$$

and the explicitly-advancing  $LR(k)$  parser must be able to make the sequence of moves

$$(\alpha r, v, y) \stackrel{a^*}{\vdash} (\alpha r, vw, \epsilon)$$

for some edit sequence  $S$  such that  $d = W(S)$  and  $z_{l+1:j} \xrightarrow{S} w$ . Again, the definition of an  $LR(k)$  parser does not allow  $0 \in g_s(X)$  for any  $s$  or  $X$  so  $([q, u, i, c, 0, 0, v], l)$  can not be added by step II.C.1.i of Algorithm 5.1.1. The only entry added by step I is  $([0, \epsilon, 0, 0, 0, 0, \epsilon], 0)$ . Therefore,  $q = 0, u = \epsilon, i = 0, c = 0, v = \epsilon, l = 0, \alpha = \epsilon$ , and

$$([0, \epsilon, 0, 0, 0, 0, \epsilon], 0) \stackrel{+}{\models} ([0, \epsilon, 0, 0, d, 0, w], j).$$

■

The following lemma shows that every entry has a direct precursor which is on a parse list when the entry is itself added to its parse list by Algorithm 5.1.1.

**LEMMA 5.3.4 (Every Entry has a Direct Precursor)** *If there is an entry  $[q, u, i, c, d, r, v]$  on a parse list  $I_j$  then there is an entry  $[?, ?, ?, ?, ?, q, u]$  on parse list  $I_i$ .*

Proof: The lemma is trivially true for entries of the form  $[0, \epsilon, 0, 0, d, 0, v]$  on  $I_0$ . For all other entries, the lemma is proved by induction on an ordered list of entries, using the lemma as the induction hypothesis. The induction proceeds in two steps:

- first, the lemma is proved for the first entry on the ordered list of entries; and

- second, the lemma is proved for the  $N^{\text{th}}$  entry on the ordered list of entries, assuming it holds for all entries before the  $N^{\text{th}}$  entry on the ordered list of entries.

For the first induction step, let  $[r, v, i, c, d, s, w]$  on  $I_j$  be the first entry on the ordered list of entries. Since the first entry must be added to its parse list by step II.C of Algorithm 5.1.1,  $([r, v, i, c, d, s, w], j)$  must have been on  $H$ . Step I or step II.C.1.i did not add  $([r, v, i, c, d, s, w], j)$  to  $H$  since if one of these steps adds  $([r, v, i, c, d, s, w], j)$  then  $|v| < k$ . Therefore, there are only two cases to consider:

- step II.B adds  $([r, v, i, c, d, s, w], j)$  to  $H$ , or
- step II.C.2.iii adds  $([r, v, i, c, d, s, w], j)$  to  $H$ .

In the first case, step II.B of Algorithm 5.1.1 adds  $([r, v, i, c, d, s, w], j)$  to  $H$  so, applying Lemma 5.3.2, either step I or II.C.1.i of Algorithm 5.1.1 must add  $([r, v, i, c, 0, s, w_{1:t}], i)$  to  $H$  where  $t < k$ . Step I could not add  $([r, v, i, c, 0, s, w_{1:t}], i)$  since that implies  $s = 0$  and entries for which  $s = 0$  are not on an ordered list of entries. Therefore, step II.C.1.i must add  $([r, v, i, c, 0, s, w_{1:t}], i)$  to  $H$ . Examining step II.C.1.i shows this step must be processing an entry  $[q, u, l, a, b, r, v]$  on  $I_i$  which is a direct precursor of  $[r, v, i, c, d, s, w]$  on  $I_j$ .

In the second case, step II.C.2.iii adds  $([r, v, i, c, d, s, w], j)$  to  $H$ . Step II.C.2.iii must be processing an entry  $[0, \epsilon, 0, 0, b, 0, u]$  on  $I_l$  since this is the only kind of entry that is not on an ordered list of entries. Examining steps II.C.1.ii and II.C.2 of Algorithm 5.1.1 shows that  $[r, v, i, c, d, s, w] = [0, u, l, c, d, s, v]$  and  $[0, \epsilon, 0, 0, b, 0, u]$  on  $I_l$  is a precursor for the entry.

For the second induction step, the lemma is assumed to hold for all entries before the  $N^{\text{th}}$  entry on the ordered list of entries. Let the  $N^{\text{th}}$  entry be  $[r, v, i, c, d, s, w]$  on  $I_j$ . Since the  $N^{\text{th}}$  entry is added to its parse list by step II.C of Algorithm 5.1.1,  $([r, v, i, c, d, s, w], j)$  must have been on  $H$  and  $|w| = k$ . Step I and step II.C.1.i did not add  $([r, v, i, c, d, s, w], j)$  to  $H$  since if one of these steps adds  $([r, v, i, c, d, s, w], j)$  then  $|w| < k$ . Therefore, there are only two cases to consider:

- step II.B adds  $([r, v, i, c, d, s, w], j)$  to  $H$ , or
- step II.C.2.iii adds  $([r, v, i, c, d, s, w], j)$  to  $H$ .

In the first case, step II.B of Algorithm 5.1.1 adds  $([r, v, i, c, d, s, w], j)$  to  $H$ . Applying Lemma 5.3.2, either step I or step II.C.1.i of Algorithm 5.1.1 adds  $([r, v, i, c, 0, s, w_{1:t}], i)$  to  $H$  where  $t < k$ . Step I could not add  $([r, v, i, c, 0, s, w_{1:t}], i)$  since that implies  $s = 0$  and that  $[r, v, i, c, d, s, w]$  on  $I_j$  is not on the ordered list

of entries. Therefore, step II.C.1.i adds  $([r, v, i, c, 0, s, w_{1:t}], i)$  to  $H$ . Examining step II.C.1.i shows this step must be processing an entry  $[q, u, l, a, b, r, v]$  on  $I_i$  which is a direct precursor of  $[r, v, i, c, d, s, w]$  on  $I_j$ .

In the second case, step II.C.2.iii of Algorithm 5.1.1 adds  $([r, v, i, c, d, s, w], j)$  to  $H$ . This implies that  $(r, v, i, c, b) \in B_{m-1}$  for some  $m \geq 0$ . Examining Algorithm 5.1.2 shows that if  $(r, v, i, c, b) \in B_{m-1}$  then, when  $m = 0$ ,  $[?, ?, ?, ?, ?, r, v]$  is on  $I_i$  and, when  $m > 0$ ,  $[r, v, i, ?, ?, ?, ?]$  is on  $I_i$ . Since  $[r, v, i, ?, ?, ?, ?]$  on  $I_i$  precedes  $[r, v, i, c, d, s, w]$  on  $I_j$  on the ordered list of entries, the induction hypothesis can be applied to show that there must be an entry  $[?, ?, ?, ?, ?, r, v]$  on  $I_i$  when  $m > 0$ . Thus, regardless of the value of  $m$  there is an entry  $[?, ?, ?, ?, ?, r, v]$  on  $I_i$ . ■

The next two lemmas establish precursor relationships that can be inferred among entries on their parse lists at key points during the execution of Algorithm 5.1.1.

**LEMMA 5.3.5** *If Algorithm 5.1.2 is applied to  $[q_1, u_1, l_1, c_0, d_0, q_0, u_0]$  on  $I_{l_0}$  and*

$$\left( q_m, u_m, l_m, \begin{cases} c_0 + d_0 & \text{if } m = 0 \\ c_{m-1} & \text{if } m > 0 \end{cases}, \sum_{i=0}^{m-1} d_i \right) \in B_{m-1}$$

*then there exist*

$$\begin{aligned} & [q_{m+1}, u_{m+1}, l_{m+1}, c_m, d_m, q_m, u_m] \text{ on } I_{l_m} \\ & [q_m, u_m, l_m, c_{m-1}, d_{m-1}, q_{m-1}, u_{m-1}] \text{ on } I_{l_{m-1}} \\ & \vdots \\ & [q_2, u_2, l_2, b_1, c_1, q_1, u_1] \text{ on } I_{l_1}. \end{aligned}$$

**Proof:** The lemma is proved by induction on  $m$ , using the lemma as the induction hypothesis. The induction proceeds in two steps:

- first, the lemma is proved for  $m = 0$  and  $m = 1$ ; and
- second, the lemma is proved for  $m = N$ , where  $N > 1$ , assuming it holds when  $m = N - 1$ .

For the first induction step,  $m = 0$  or  $m = 1$ . Examination of step I of Algorithm 5.1.2 shows that  $\{(q_0, u_0, l_0, c_0 + d_0, 0)\} \in B_{-1}$  or  $\{(q_1, u_1, l_1, c_0, d_0)\} \in B_0$  only if  $[q_1, u_1, l_1, c_0, d_0, q_0, u_0]$  is on  $I_{l_0}$ .

For the second induction step,  $m = N$ , where  $N > 1$ , and the lemma is assumed to hold when  $m = N - 1$ . If  $(q_N, u_N, l_N, c', d') \in B_{N-1}$  then step II.B.2.i of Algorithm 5.1.2 added  $(q_N, u_N, l_N, c', d')$  to  $B_{N-1}$  and there was an entry

$[q_N, u_N, l_N, c', d_{N-1}, q_{N-1}, u_{N-1}]$  on  $I_{N-1}$ , and an element  $(q_{N-1}, u_{N-1}, l_{N-1}, c_{N-1}, d' - d_{N-1}) \in B_{N-2}$  where  $c' = c_{N-1}$ . Since  $(q_{N-1}, u_{N-1}, l_{N-1}, c_{N-1}, d' - d_{N-1}) \in B_{N-2}$ , the induction hypothesis can be applied to show that there exist

$$\begin{aligned} & [q_N, u_N, l_N, c_{N-1}, d_{N-1}, q_{N-1}, u_{N-1}] \text{ on } I_{l_{N-1}} \\ & [q_{N-1}, u_{N-1}, l_{N-1}, c_{N-2}, d_{N-2}, q_{N-2}, u_{N-2}] \text{ on } I_{l_{N-2}} \\ & \vdots \\ & [q_2, u_2, l_2, b_1, c_1, q_1, u_1] \text{ on } I_{l_1} \end{aligned}$$

where  $d' - d_{N-1} = \sum_{i=0}^{N-2} d_i$ . Therefore,  $d' = \sum_{i=0}^{N-1} d_i$ . Finally, applying Lemma 5.3.4 to  $[q_N, u_N, l_N, c_{N-1}, d_{N-1}, q_{N-1}, u_{N-1}]$  on  $I_{N-1}$  shows that  $[q_{N+1}, u_{N+1}, l_{N+1}, c_N, d_N, q_N, u_N]$  is on  $I_{l_N}$ . ■

**LEMMA 5.3.6** *If  $(h, \sum_{x=h+1}^m d_x, p, u_m, l_m) \in G_{l_h}(q_h, u_h)$  then there exist*

$$\begin{aligned} & [q_h, u_h, l_h, c_{h+1}, d_{h+1}, q_{h+1}, u_{h+1}] \text{ on } I_{l_{h+1}} \\ & [q_{h+1}, u_{h+1}, l_{h+1}, c_{h+2}, d_{h+2}, q_{h+2}, u_{h+2}] \text{ on } I_{l_{h+2}} \\ & \vdots \\ & [q_{m-1}, u_{m-1}, l_{m-1}, c_m, d_m, q_m, u_m] \text{ on } I_{l_m} \end{aligned}$$

where  $\text{reduce } p \in f_{q_m}(u_m)$  and  $m = |\text{RHS}(p)|$ .

**Proof:** The lemma is proved by induction on  $m - h$ , using the lemma as the induction hypothesis. The induction proceeds in two steps:

- first, the lemma is proved for  $m - h = 0$ ; and
- second, the lemma is proved for  $m - h = N$ , where  $N > 0$ , assuming it holds for  $m - h = N - 1$ .

For the first induction step,  $m - h = 0$ . Thus,  $h = m = |\text{RHS}(p)|$ . Examination of step II.C.1.ii of Algorithm 5.1.1 shows that it adds  $(|\text{RHS}(p)|, 0, p, u_m, i)$  to  $G_{l_m}(q_m, u_m)$  only if  $[q_{m-1}, u_{m-1}, l_{m-1}, c_m, d_m, q_m, u_m]$  is on  $I_{l_m}$  and  $\text{reduce } p \in f_{q_m}(u_m)$ . No other step of either Algorithm 5.1.1 or Algorithm 5.1.2, particularly step II.B.2.i of Algorithm 5.1.2, can add an element  $(h, 0, p, u_m, l_m)$  to  $G_{l_m}(q_m, u_m)$  for which  $h = |\text{RHS}(p)|$ .



For the second induction step,  $m - h = N$ , where  $N > 0$ , and the lemma is assumed to hold for  $m - h < N - 1$ . An element  $(h, d', p, u_m, l_m)$  for which  $m - h = N$  and  $N > 0$  can only be added by step I or step II.B.2.i of Algorithm 5.1.2. Step I or step II.B.2.i adds  $(h, d', p, u_m, l_m)$  to  $G_{l_h}(q_h, u_h)$  only if there is an entry  $[q_h, u_h, l_h, c_{h+1}, d_{h+1}, q_{h+1}, u_{h+1}]$  on  $I_{l_{h+1}}$  and  $(h+1, d' - d_{h+1}, p, u_m, l_m) \in G_{l_{h+1}}(q_{h+1}, u_{h+1})$ . Applying the induction hypothesis to  $(h+1, d' - d_{h+1}, p, u_m, l_m)$ , for which  $m - h = N - 1$ , there must be

$$\begin{aligned} & [q_h, u_h, l_h, c_{h+1}, d_{h+1}, q_{h+1}, u_{h+1}] \text{ on } I_{l_{h+1}} \\ & [q_{h+1}, u_{h+1}, l_{h+1}, c_{h+2}, d_{h+2}, q_{h+2}, u_{h+2}] \text{ on } I_{l_{h+2}} \\ & \vdots \\ & [q_{m-1}, u_{m-1}, l_{m-1}, c_m, d_m, q_m, u_m] \text{ on } I_{l_m} \end{aligned}$$

where **reduce**  $p \in f_{q_m}(u_m)$  and  $d' - d_{h+1} = \sum_{x=h+2}^m d_x$ . Thus,  $d' = \sum_{x=h+1}^m d_x$ . Furthermore, applying Lemma 5.3.4 to  $[q_h, u_h, l_h, c_{h+1}, d_{h+1}, q_{h+1}, u_{h+1}]$  on  $I_{l_{h+1}}$  shows that there exists  $[q_{h-1}, u_{h-1}, l_{h-1}, c_h, d_h, q_h, u_h]$  on  $I_{l_h}$ . ■

**THEOREM 5.3.1** *Given the same  $Q$ ,  $f_q$ , and  $g_q$  for Algorithm 5.1.1 and the explicitly-advancing  $LR(k)$  parser, if an entry  $[r, v, i, c, d, s, w]$  is added to a parse list  $I_j$ , where  $s \neq 0$ , then the explicitly-advancing  $LR(k)$  parser can make the sequence of moves*

$$(0, \epsilon, yy') \vdash^* (\alpha \setminus r, v, y') \vdash^+ (\alpha rs, ?, \dots) \vdash^{a^*} (\alpha rs, w, \epsilon)$$

for which there exist edit sequences  $S$  and  $S'$  such that  $z_{1,i} \xrightarrow{S} y$ ,  $z_{i+1,j} \xrightarrow{S'} y'$ ,  $W(S) = c$  and  $W(S') = d$ .

**Proof:** The theorem is proved by induction on any ordered list of entries, using the theorem as the induction hypothesis. The induction proceeds in two steps:

- first, the theorem is proved for the first entry on an ordered list of entries; and
- second, the theorem is proved for the  $N^{\text{th}}$  entry on an ordered list of entries, assuming it holds for all entries before the  $N^{\text{th}}$  entry.

For the first induction step, let  $[r, v, i, c, d, s, w]$  on  $I_j$  be the first entry on the ordered list of entries. Since the first entry must be added to its parse list by

step II.C.2 of Algorithm 5.1.1,  $([r, v, i, c, d, s, w], j)$  must have been on  $H$ . Step I and step II.C.1.i did not add  $([r, v, i, c, d, s, w], j)$  to  $H$  since if one of these steps adds  $([r, v, i, c, d, s, w], j)$  then  $|w| < k$ . Therefore, there are only two cases to consider:

- step II.B adds  $([r, v, i, c, d, s, w], j)$  to  $H$ , or
- step II.C.2.iii adds  $([r, v, i, c, d, s, w], j)$  to  $H$ .

In the first case, step II.B of Algorithm 5.1.1 adds  $([r, v, i, c, d, s, w], j)$  to  $H$  so, applying Lemma 5.3.2, either step I or II.C.1.i of Algorithm 5.1.1 adds  $([r, v, i, c, 0, s, w_{1:t}], i)$  to  $H$ . Furthermore, the explicitly-advancing  $\text{LR}(k)$  parser can make the move

$$(\alpha s, w_{1:t}, w_{t+1:k}) \vdash^{a^*} (\alpha s, w, \epsilon)$$

and there exists an edit sequence  $S'$  such that  $z_{1:j} \xrightarrow{S'} w_{t+1:k}$  and  $d = W(S')$ . Step I could not add  $([r, v, i, c, 0, s, w_{1:t}], i)$  since that implies  $s = 0$  and entries for which  $s = 0$  are not on an ordered list of entries. Therefore, step II.C.1.i must add  $([r, v, i, c, 0, s, w_{1:t}], i)$  to  $H$ . Examining step II.C.1.i shows this step must also be processing an entry  $[q, u, l, a, b, r, v]$  on  $I_i$  for which  $\text{shift} \in f_r(v)$  and  $s \in g_r(v)$ . Thus,  $w_{1:t} = v_{2:k}$ ,  $c = a + b$  and the  $\text{LR}(k)$  parser is able to make the move

$$(\beta \wr r, v, \epsilon) \vdash (\beta r s, w_{1:t}, \epsilon).$$

Furthermore,  $|v| = k$  and  $r$  must be 0 since  $[q, u, l, a, b, r, v]$  is not on the ordered list of entries. Applying Lemma 5.3.3,  $[q, u, l, a, b, r, v] = [0, \epsilon, 0, 0, b, 0, v]$  and the explicitly-advancing  $\text{LR}(k)$  parser can make the sequence of moves

$$(0, \epsilon, v) \vdash^{a^*} (0, v, \epsilon)$$

for which there exists an edit sequence  $S$  such that  $z_{1:n} \xrightarrow{S} v$  and  $b = W(S)$ . Therefore, the explicitly-advancing  $\text{LR}(k)$  parser can make the sequence of moves

$$(0, \epsilon, v w_{t+1:k}) \vdash^* (0, v, w_{t+1:k}) \vdash (0 s, w_{1:t}, w_{t+1:k}) \vdash^{a^*} (0 s, w, \epsilon)$$

where  $z_{1:n} \xrightarrow{S} v$ ,  $z_{1:j} \xrightarrow{S'} w_{t+1:k}$ ,  $c = W(S)$ , and  $d = W(S')$ .

In the second case, step II.C.2.iii of Algorithm 5.1.1 adds  $([r, v, i, c, d, s, w], j)$  to  $H$ . This step must be processing an entry  $[q_{h-1}, u_{h-1}, l_{h-1}, c_h, d_h, q_h, u_h]$  on  $I_{l_h}$  where  $h$  is an arbitrary index chosen for convenience. Furthermore, examining step II.C.2.iii shows that  $s \in g_r(\text{LHS}(p))$ . Also, when the entry

$[q_{h-1}, u_{h-1}, l_{h-1}, c_h, d_h, q_h, u_h]$  was processed, there must have been an

$$(h, b, p, w, j) \in G_{l_h}(q_h, u_h)$$

and an

$$(q_0, u_0, l_0, c', d') \in B_{h-1}$$

where  $c = c'$ ,  $d = b + d'$ ,  $q_0 = r$ ,  $u_0 = v$ , and  $l_0 = i$ . Applying Lemma 5.3.5, there must be

$$\begin{aligned} & [q_0, u_0, l_0, c_1, d_1, q_1, u_1] \text{ on } I_{l_1} \\ & [q_1, u_1, l_1, c_2, d_2, q_2, u_2] \text{ on } I_{l_2} \\ & \vdots \\ & [q_{h-2}, u_{h-2}, l_{h-2}, c_{h-1}, d_{h-1}, q_{h-1}, u_{h-1}] \text{ on } I_{l_{h-1}} \end{aligned}$$

where  $c' = \begin{cases} c_0 + d_0 & \text{if } h = 0 \\ c_1 & \text{if } h > 0 \end{cases}$  and  $d' = \sum_{x=1}^h d_x$ . Applying Lemma 5.3.6, there must also be

$$\begin{aligned} & [q_h, u_h, l_h, c_{h+1}, d_{h+1}, q_{h+1}, u_{h+1}] \text{ on } I_{l_{h+1}} \\ & [q_{h+1}, u_{h+1}, l_{h+1}, c_{h+2}, d_{h+2}, q_{h+2}, u_{h+2}] \text{ on } I_{l_{h+2}} \\ & \vdots \\ & [q_{m-2}, u_{m-2}, l_{m-2}, c_{m-1}, d_{m-1}, q_{m-1}, u_{m-1}] \text{ on } I_{l_{m-1}} \\ & [q_{m-1}, u_{m-1}, l_{m-1}, c_m, d_m, q_m, u_m] \text{ on } I_{l_m} \end{aligned}$$

where  $u_m = w$ ,  $l_m = j$ ,  $m = |\text{RHS}(p)|$  reduce  $p \in f_{q_m}(w)$ , and  $b = \sum_{x=h+1}^m d_x$ .

Note  $q_m$  must be 0 since if  $q_m \neq 0$  then  $[q_{m-1}, u_{m-1}, l_{m-1}, c_m, d_m, q_m, u_m]$  would precede  $([r, v, i, c, d, s, w], j)$  in the order which contradicts the assumption that  $([r, v, i, c, d, s, w], j)$  is the first entry in the order. Applying Lemma 5.3.3 recursively to  $[q_{m-1}, u_{m-1}, l_{m-1}, c_m, d_m, q_m, u_m]$ , shows that, for  $0 \leq x < m$ ,  $q_x = 0$ ,  $u_x = \epsilon$ ,  $c_x = 0$ , and  $d_x = 0$ . Furthermore,  $c_m = 0$  and the explicitly-advancing  $\text{LR}(k)$  parser can make the sequence of moves

$$(0, \epsilon, w) \stackrel{a^*}{\vdash} (0, w, \epsilon)$$

for which there exists an edit sequence  $S$  such that  $z_1 \cdot_j \stackrel{S}{\rightsquigarrow} u_m$  and  $d_m = W(S)$ .

Since, for  $0 \leq x < m$ ,  $|u_h| = k$  and  $u_x = \epsilon$ ,  $h$  must equal  $m$ . Also, the explicitly-advancing  $\text{LR}(k)$  parser can make the move

$$(0, w, \epsilon) \vdash (0s, w, \epsilon)$$

since **reduce**  $p \in f_{q_m}(w)$  and  $s \in g_r(\text{LHS}(p))$ . Therefore, the explicitly-advancing  $\text{LR}(k)$  parser can make the moves

$$(0, \epsilon, w) \stackrel{a^*}{\vdash} (0, w, \epsilon) \vdash (0s, w, \epsilon)$$

where  $z_{1:j} \xrightarrow{s} w$ ,  $c = 0$ , and  $d = W(S)$ .

For the second induction step, the theorem is assumed to be true for all entries on the ordered list of entries that precede the  $N^{\text{th}}$  entry. Let the  $N^{\text{th}}$  entry be  $[r, v, i, c, d, s, w]$  on  $I_j$ . Since the  $N^{\text{th}}$  entry is added to its parse list by step II.C of Algorithm 5.1.1,  $([r, v, i, c, d, s, w], j)$  must have been on  $H$  and  $|w| = k$ . Step I and step II.C.1.i did not add  $([r, v, i, c, d, s, w], j)$  to  $H$  since if one of these steps adds  $([r, v, i, c, d, s, w], j)$  then  $|w| < k$ . Therefore, there are only two cases to consider:

- step II.B adds  $([r, v, i, c, d, s, w], j)$  to  $H$ , or
- step II.C.2.iii adds  $([r, v, i, c, d, s, w], j)$  to  $H$ .

In the first case, step II.B of Algorithm 5.1.1 adds  $([r, v, i, c, d, s, w], j)$  to  $H$  and Algorithm 5.1.1 simulates an advance. Applying Lemma 5.3.2, either step I or step II.C.1.i of Algorithm 5.1.1 adds  $([r, v, i, c, 0, s, w_{1:t}], i)$  to  $H$ . Furthermore, the explicitly-advancing  $\text{LR}(k)$  parser can make the move

$$(\alpha s, w_{1:t}, w_{t+1:k}) \stackrel{a^*}{\vdash} (\alpha s, w, \epsilon)$$

and there exists an edit sequence  $S'$  such that  $z_{1:j} \xrightarrow{S'} w_{t+1:k}$  and  $d = W(S')$ . Step I could not add  $([r, v, i, c, 0, s, w_{1:t}], i)$  since that implies  $s = 0$  and  $[r, v, i, c, d, s, w]$  on  $I_j$  is not an entry in the order. Therefore, step II.C.1.i adds  $([r, v, i, c, 0, s, w_{1:t}], i)$  to  $H$  and Algorithm 5.1.1 simulates a shift. Examining step II.C.1.i shows this step must be processing an entry  $[q, u, l, a, b, r, v]$  on  $I_i$  for which **shift**  $\in f_r(v)$  and  $s \in g_r(v)$ . Thus,  $w_{1:t} = v_{2:k}$ ,  $c = a + b$  and the explicitly advancing  $\text{LR}(k)$  parser is able to make the move

$$(\beta)r, v, \epsilon \vdash (\beta)rs, w_{1:t}, \epsilon.$$

If  $r = 0$ , Lemma 5.3.3 shows that  $[q, u, l, a, b, r, v] = [0, \epsilon, 0, 0, b, 0, v]$  and the explicitly-advancing  $\text{LR}(k)$  parser can make the moves

$$(0, \epsilon, v) \stackrel{a^*}{\vdash} (0, v, \epsilon)$$

for which there exists an edit sequence  $S$  such that  $z_{1::} \xrightarrow{S} v$  and  $b = W(S)$ . Therefore, the explicitly-advancing  $\text{LR}(k)$  parser can make the moves

$$(0, \epsilon, v w_{t+1:k}) \stackrel{*}{\vdash} (0, v, w_{t+1:k}) \vdash (0s, w_{1:t}, w_{t+1:k}) \stackrel{a^*}{\vdash} (0s, w, \epsilon)$$

where  $z_{1:t} \xrightarrow{S} v$ ,  $z_{t+1:j} \xrightarrow{S'} w_{t+1:k}$ ,  $c = b = W(S)$ , and  $d = W(S')$ .

If  $r \neq 0$  then  $[q, u, l, a, b, r, v]$  is an entry on the ordered list of entries that precedes  $([r, v, i, c, d, s, w], j)$ . Applying the induction hypothesis to  $[q, u, l, a, b, r, v]$ , the explicitly-advancing LR( $k$ ) parser can make the moves

$$(0, \epsilon, y) \vdash^* (\delta r, v, \epsilon)$$

and there exists an edit sequence  $S$  such that  $z_{1:t} \xrightarrow{S} y$  and  $a+b = W(S)$ . Therefore, the explicitly-advancing LR( $k$ ) parser can make the moves

$$(0, \epsilon, yw_{t+1:k}) \vdash^* (\delta(r, v, w_{t+1:k}) \vdash (\delta r s, w_{1:t}, w_{t+1:k}) \vdash^* (\delta r s, w, \epsilon)$$

where  $z_{1:t} \xrightarrow{S} y$ ,  $z_{t+1:j} \xrightarrow{S'} w_{t+1:k}$ ,  $c = a + b = W(S)$ , and  $d = W(S')$ .

In the second case of the second induction step, step II.C.2.iii of Algorithm 5.1.1 adds  $([r, v, i, c, d, s, w], j)$  to  $H$  and Algorithm 5.1.1 simulates a reduction. Step II.C.2.iii must be processing an entry  $[q_{h-1}, u_{h-1}, l_{h-1}, c_h, d_h, q_h, u_h]$  on  $I_{l_h}$  where  $h$  is an arbitrary index chosen for convenience. Furthermore, examining step II.C.2.iii shows  $s \in g_r(\text{LHS}(p))$ . Also, when  $[q_{h-1}, u_{h-1}, l_{h-1}, c_h, d_h, q_h, u_h]$  is processed there must be

$$(h, b, p, w, j) \in G_{l_h}(q_h, u_h)$$

and

$$(q_0, u_0, l_0, c', d') \in B_{h-1}$$

where  $c = c'$ ,  $d = b + d'$ ,  $q_0 = r$ ,  $u_0 = v$ , and  $l_0 = i$ . Applying Lemma 5.3.5, there must be

$$\begin{aligned} & [q_0, u_0, l_0, c_1, d_1, q_1, u_1] \text{ on } I_{l_1} \\ & [q_1, u_1, l_1, c_2, d_2, q_2, u_2] \text{ on } I_{l_2} \\ & \vdots \\ & [q_{h-2}, u_{h-2}, l_{h-2}, c_{h-1}, d_{h-1}, q_{h-1}, u_{h-1}] \text{ on } I_{l_{h-1}} \end{aligned}$$

where  $c' = \begin{cases} c_0 + d_0 & \text{if } h = 0 \\ c_1 & \text{if } h > 0 \end{cases}$  and  $d' = \sum_{x=1}^h d_x$ . Applying Lemma 5.3.1, there must also be

$$\begin{aligned} & [q_h, u_h, l_h, c_{h+1}, d_{h+1}, q_{h+1}, u_{h+1}] \text{ on } I_{l_{h+1}} \\ & [q_{h+1}, u_{h+1}, l_{h+1}, c_{h+2}, d_{h+2}, q_{h+2}, u_{h+2}] \text{ on } I_{l_{h+2}} \\ & \vdots \\ & [q_{m-2}, u_{m-2}, l_{m-2}, c_{m-1}, d_{m-1}, q_{m-1}, u_{m-1}] \text{ on } I_{l_{m-1}} \\ & [q_{m-1}, u_{m-1}, l_{m-1}, c_m, d_m, q_m, u_m] \text{ on } I_{l_m} \end{aligned}$$

where  $u_m = w$ ,  $l_m = j$ ,  $m = |\text{RHS}(p)|$ , reduce  $p \in f_{q_m}(w)$ , and  $b = \sum_{x=h+1}^m d_x$ . The reduction must be one of three possible types of reductions, each of which must be considered separately:

- a reduction by an empty production ( $m = 0$ );
- a reduction by a non-empty production which does not cause the stack to underflow ( $m > 0$  and  $q_x \neq 0$  for  $0 < x \leq m$ ); or
- a reduction by a non-empty production which causes the stack to underflow ( $m > 0$  and  $q_x = 0$  for some  $x$  where  $0 < x \leq m$ ).

For the first type of reduction, an empty production is used so  $m = h = 0$ . Thus  $l_m = l_j = i = j$ ,  $q_m = q_0 = r$ ,  $u_m = u_0 = w = v$ , and the reduction

$$(\alpha r, v, \epsilon) \vdash (\alpha r s, w, \epsilon)$$

can be made by the explicitly-advancing  $\text{LR}(k)$  parser. If  $r = 0$ , Lemma 5.3.3 shows that  $[q_{h-1}, u_{h-1}, l_{h-1}, c_h, d_h, r, v] = [0, \epsilon, 0, 0, b, 0, v]$  and the explicitly-advancing  $\text{LR}(k)$  parser can make the sequence of moves

$$(0, \epsilon, v) \vdash^{a^*} (0, v, \epsilon)$$

for which there exists an edit sequence  $S$  such that  $z_{1..} \xrightarrow{S} v$  and  $b = W(S)$ . Therefore, the explicitly-advancing  $\text{LR}(k)$  parser can make the sequence of moves

$$(0, \epsilon, v) \vdash^* (0, v, \epsilon) \vdash^{a^*} (0s, v, \epsilon)$$

for which there exists an edit sequence  $S$  such that  $z_{1..} \xrightarrow{S} v$ ,  $c = b = W(S)$  and  $d = 0$ .

If  $r \neq 0$  then  $[q_{h-1}, u_{h-1}, l_{h-1}, c_h, d_h, r, v]$  precedes  $([r, v, i, c, d, s, w], j)$  on the ordered list of entries. Applying the induction hypothesis to  $[q_{h-1}, u_{h-1}, l_{h-1}, c_h, d_h, r, v]$ , the explicitly-advancing  $\text{LR}(k)$  parser can make the moves

$$(0, \epsilon, y) \vdash^* (\delta r, v, \epsilon) \vdash^{a^*} (\delta r, v, \epsilon)$$

and there exists an edit sequence  $S$  such that  $z_{1..} \xrightarrow{S} y$  and  $c_h + d_h = W(S)$ . Therefore, the explicitly-advancing  $\text{LR}(k)$  parser can make the moves

$$(0, \epsilon, yw) \vdash^* (\delta r, w, \epsilon) \vdash^{a^*} (\delta r s, w, \epsilon) \vdash^{a^*} (\delta r s, w, \epsilon)$$

where  $z_{1:j} \xrightarrow{S} yw$ ,  $c = c_h + d_h = W(S)$ , and  $d = 0$ .

For the second type of reduction, a non-empty production is used and the stack does not underflow so  $m > 0$  and  $q_x \neq 0$  for  $0 < x \leq m$ . This implies, for  $0 < x \leq m$ , that  $[q_{x-1}, u_{x-1}, l_{x-1}, c_x, d_x, q_x, u_x]$  on  $I_{l_x}$  is an entry on the ordered list of entries and that these entries precede  $[r, v, i, c, d, s, w]$  on  $I_j$ . Applying the induction hypothesis to each entry, the explicitly-advancing LR( $k$ ) can make the sequences of moves

$$\begin{aligned} (0, \epsilon, y_1 y'_1) &\vdash^* (\alpha_0 \setminus q_0, u_0, y'_1) \vdash^+ (\alpha_0 q_0 q_1, u_1, \epsilon) \\ (0, \epsilon, y_2 y'_2) &\vdash^* (\alpha_1 \setminus q_1, u_1, y'_2) \vdash^+ (\alpha_1 q_1 q_2, u_2, \epsilon) \\ &\vdots \\ (0, \epsilon, y_m y'_m) &\vdash^* (\alpha_{m-1} \setminus q_{m-1}, u_{m-1}, y'_m) \vdash^+ (\alpha_{m-1} q_{m-1} q_m, u_m, \epsilon). \end{aligned}$$

Also, there exist edit sequences  $S_x$  and  $S'_x$ , for  $1 \leq x \leq m$ , such that  $z_{1:l_{x-1}} \xrightarrow{S_x} y_x$ ,  $z_{l_{x-1}+1:l_x} \xrightarrow{S'_x} y'_x$ ,  $W(S_x) = c_x$ , and  $W(S'_x) = d_x$ . As a result, the explicitly-advancing LR( $k$ ) parser can make the moves

$$\begin{aligned} (0, \epsilon, y_1 y'_1 y'_2 y'_3 \dots y'_m) &\vdash^* (\alpha_0 \setminus q_0, u_0, y'_1 y'_2 y'_3 \dots y'_m) \\ &\vdash^+ (\alpha_0 q_0 \setminus q_1, u_1, y'_2 y'_3 \dots y'_m) \\ &\vdash^+ (\alpha_0 q_0 q_1 \setminus q_2, u_2, y'_3 \dots y'_m) \\ &\vdash^+ \dots \\ &\vdash^+ (\alpha_0 q_0 q_1 \dots q_{m-1} \setminus q_m, u_m, \epsilon). \end{aligned}$$

Let  $S = S_1$ ,  $S' = S'_1 S'_2 \dots S'_m$ , and  $y' = y'_1 y'_2 \dots y'_m$ . Then,  $z_{1:n} \xrightarrow{S} y_1$ ,  $z_{i+1:j} \xrightarrow{S'} y'$ ,  $c_1 = W(S)$ , and  $b + d' = W(S')$ . Since  $\text{reduce } p \in f_{q_m}(u_m)$ ,  $q_0 = r$ ,  $u_0 = v$ ,  $l_0 = i$ ,  $u_m = w$ ,  $l_m = j$ , and the explicitly-advancing LR( $k$ ) parser can make the moves

$$(\alpha_0 r q_1 q_2 \dots q_m, w, \epsilon) \vdash (\alpha_0 r s, w, \epsilon).$$

Therefore, the explicitly-advancing LR( $k$ ) parser can make the moves

$$(0, \epsilon, y_1 y') \vdash^* (\alpha_0 \setminus r, v, y') \vdash^+ (\alpha_0 r s, w, \epsilon)$$

where there exist edit sequences  $S$  and  $S'$  such that  $z_{1:n} \xrightarrow{S} y_1$ ,  $z_{i+1:j} \xrightarrow{S'} y'$ ,  $c = W(S)$ , and  $d = W(S')$ .

For the third type of reduction, an non-empty production is used and the stack underflows so  $m > 0$  and  $q_x \neq 0$  for some  $x$  where  $0 < x \leq m$ . Let  $e$  be the greatest such  $x$ . Since  $1 \leq e$ ,  $r = q_0 = 0$ . Recursively applying Lemma 5.3.1 shows that, for  $x < e$ ,  $q_x = 0$ ,  $u_x = \epsilon$ ,  $l_x = 0$  and  $c_x = 0$ . Also, for  $x < e$ ,  $[q_{x-1}, u_{x-1}, l_{x-1}, c_x, d_x, q_x, u_x]$  on  $I_x$  is not processed by step II.C of Algorithm 5.1.1 since  $u_x = \epsilon$ . Therefore,  $h \geq e$ . Furthermore,  $d_x = 0$  for  $x < 0$  since  $[0, \epsilon, 0, 0, 0, \epsilon]$  on  $I_0$  is the only entry on a parse list that is not processed by step II.C.

Since  $q_e = 0$ , Lemma 5.3.1 shows there exists an edit sequence  $S$  such that  $z_{1..j} \xrightarrow{S} v$ ,  $d_e = W(S)$ , and the explicitly-advancing  $LR(k)$  parser can make the moves

$$(0, \epsilon, v) \vdash^{a^*} (0, v, \epsilon).$$

If  $e = m$  then there is only the state 0 on the stack when the reduction is made and  $h = m$ ,  $q_0 = q_m = r = 0$ ,  $u_m = v = w$ ,  $c_m = 0$ , and  $d_e = d_m$ . Also, since **reduce**  $p \in f_{q_m}(u_m)$  and  $s \in g_r(\text{LHS}(p))$ , the explicitly-advancing  $LR(k)$  parser can make the moves

$$(0, w, \epsilon) \vdash (0s, w, \epsilon).$$

Therefore, if  $m = 0$ , the explicitly-advancing  $LR(k)$  parser can make the moves

$$(0, \epsilon, y) \vdash^* (0, \epsilon, y) \vdash^+ (0s, w, \epsilon) \vdash^{a^*} (0s, w, \epsilon)$$

where there exists an edit sequence  $S$  such that  $z_{1..j} \xrightarrow{S} w$ ,  $c = d_e = W(S)$ , and  $d = 0$ .

If  $e < m$ , there are some states (but not enough) on the stack when the reduction is applied. For  $0 < x \leq m$ ,  $[q_{x-1}, u_{x-1}, l_{x-1}, c_x, d_x, q_x, u_x]$  on  $I_{l_x}$  is an entry that precedes  $[r, v, i, c, d, s, w]$  on  $I_j$  on the ordered list of entries. Applying the induction hypothesis to each of these entries, the explicitly-advancing  $LR(k)$  parser can make the sequences of moves

$$\begin{aligned} (0, \epsilon, y_{e+1}y'_{e+1}) &\vdash^* (\alpha_e \setminus q_e, u_e, y'_{e+1}) \vdash^+ (\alpha_e q_e q_{e+1}, u_{e+1}, \epsilon) \\ (0, \epsilon, y_{e+2}y'_{e+2}) &\vdash^* (\alpha_{e+1} \setminus q_{e+1}, u_{e+1}, y'_{e+2}) \vdash^+ (\alpha_{e+1} q_{e+1} q_{e+2}, u_{e+2}, \epsilon) \\ &\vdots \\ (0, \epsilon, y_m y'_m) &\vdash^* (\alpha_{m-1} \setminus q_{m-1}, u_{m-1}, y'_m) \vdash^+ (\alpha_{m-1} q_{m-1} q_m, u_m, \epsilon). \end{aligned}$$

Also, there exist edit sequences  $S'_x$  for  $e < x \leq m$  such that  $z_{l_{x-1}+1..l_x} \xrightarrow{S'_x} y'_x$  and  $W(S'_x) = d_x$ . Furthermore, since  $q_e = 0$  and 0 is never pushed onto the stack,



$\alpha_e = \epsilon$ . As a result, the explicitly-advancing  $\text{LR}(k)$  parser can make the sequence of moves

$$\begin{aligned} (0, u_e, y'_{e+1} y'_{e+2} y'_{e+3} \dots y'_m) &\vdash^+ (0 \mid q_{e+1}, u_{e+1}, y'_{e+2} y'_{e+3} \dots y'_m) \\ &\vdash^+ (0 q_{e+1} \mid q_{e+2}, u_{e+2}, y'_{e+3} \dots y'_m) \\ &\vdash^+ \dots \\ &\vdash^+ (0 q_{e+1} \dots q_m, u_m, \epsilon). \end{aligned}$$

Let  $S' = S'_{e+1} S'_{e+2} \dots S'_m$ , and  $y' = y'_{e+1} y'_{e+2} \dots y'_m$ . Then,  $z_{i+1:j} \xrightarrow{S'} y'$ . Since **reduce**  $p \in f_{q_m}(u_m)$ ,  $s \in g_r(\text{LHS}(p))$ ,  $q_0 = r$ ,  $u_0 = e$ ,  $l_0 = i$ ,  $u_m = w$ , and  $l_m = j$ , the explicitly-advancing  $\text{LR}(k)$  parser can make the move

$$(0 q_{e+1} q_{e+2} \dots q_m, u_m, \epsilon) \vdash (0s, w, \epsilon).$$

Therefore, the explicitly-advancing  $\text{LR}(k)$  parser can make the moves

$$(0, \epsilon, vy') \vdash^* (0, \epsilon, vy') \vdash^+ (0s, w, \epsilon) \vdash^{a^*} (0s, w, \epsilon)$$

where there exists an edit sequence  $SS'$  such that  $z_{1:j} \xrightarrow{SS'} vy'$ ,  $c = 0$ , and  $d = b + d' = W(SS')$ . ■

## 5.4 PROOF OF COMPLETENESS

This section shows that Algorithm 5.1.1 completely simulates the explicitly-advancing  $\text{LR}(k)$  parser. Since Algorithm 5.1.1 is a least-cost algorithm, the definition of completeness for it differs from the definition used for the earlier algorithms in this dissertation. The simulation is complete for an input  $z$  if for every sequence of moves

$$(0, \epsilon, yy') \vdash^* (\alpha \mid r, v, y') \vdash^+ (\alpha rs, w, \epsilon)$$

that can be made by the  $\text{LR}(k)$  parser for which there exist edit sequences  $S$  and  $S'$  such that  $z_{1:n} \xrightarrow{S} y$ ,  $z_{i+1:j} \xrightarrow{S'} y'$ ,  $W(S) = c$ ,  $W(S') = d$ , and

$$c + d = \min \left( \left\{ W(T) \mid z_{1:j} \xrightarrow{T} t \text{ such that } (0, \epsilon, t) \vdash^+ (\alpha rs, w, \epsilon) \right\} \right),$$

Algorithm 5.1.1 adds the entry  $[r, v, i, c', d', s, w]$  to parse list  $I$ , where  $c' + d' = c + d$ . This property is considered a completeness result since it shows that there is an entry on a parse list for each least-cost edit of a prefix of the input string, if the edit generates a prefix of some string in the language accepted by the explicitly-advancing  $\text{LR}(k)$  parser.

### 5.4.1 Consistent Ordered Lists of Entries

To show that Algorithm 5.1.1 is complete, it must first be shown that if an entry  $[r, v, i, c, d, s, w]$  is on  $I_j$  then there is an entry  $[q, u, l, a, b, r, v]$  on  $I_i$  for which  $c = a + b$ . Ordered lists of entries with this property are said to be consistent. The term consistent is used because Algorithm 5.1.1 is designed to have this property and the lack of it would be an inconsistency.

To show that any ordered list of entries is consistent, it is convenient to first prove the property for a subset of an ordered list of entries and then expand the property to the entire list. For this purpose, the following definition introduces the concept of a prefix of an ordered list of entries.

*Definition 5.4.1 (Prefix of an Ordered List of Entries)* Given an ordered list of entries

$$\begin{aligned} & [r_1, i_1, v_1, c_1, d_1, s_1, w_1] \text{ on } I_{j_1} \\ & [r_2, i_2, v_2, c_2, d_2, s_2, w_2] \text{ on } I_{j_2} \\ & \vdots \\ & [r_N, i_N, v_N, c_N, d_N, s_N, w_N] \text{ on } I_{j_N} \end{aligned}$$

a prefix of the ordered list of entries is any list of entries

$$\begin{aligned} & [r_1, i_1, v_1, c_1, d_1, s_1, w_1] \text{ on } I_{j_1} \\ & [r_2, i_2, v_2, c_2, d_2, s_2, w_2] \text{ on } I_{j_2} \\ & \vdots \\ & [r_n, i_n, v_n, c_n, d_n, s_n, w_n] \text{ on } I_{j_n} \end{aligned}$$

where  $n \leq N$ .

The following two definitions formally define the concept of a consistent ordered list of entries.

*Definition 5.4.2 (A Consistent Prefix of an Ordered List of Entries)* A prefix of an ordered list of entries

$$\begin{aligned} & [r_1, v_1, i_1, c_1, d_1, s_1, w_1] \text{ on } I_{j_1} \\ & [r_2, v_2, i_2, c_2, d_2, s_2, w_2] \text{ on } I_{j_2} \\ & \vdots \\ & [r_n, v_n, i_n, c_n, d_n, s_n, w_n] \text{ on } I_{j_n} \end{aligned}$$

is said to be consistent if for each entry  $[r_l, v_l, i_l, c_l, d_l, s_l, w_l]$  on  $I_{j_l}$  there exists an entry  $[q, u, l, a, b, r_l, v_l]$  on  $I_{i_l}$  for which  $c_l = a + b$  and

$$([q, u, l, a, b, r_l, v_l] \text{ on } I_{i_l}) \stackrel{+}{=} ([r_l, v_l, i_l, c_l, d_l, s_l, w_l] \text{ on } I_{j_l}).$$

*Definition 5.4.3 (A Consistent Ordered List of Entries)* An ordered list of entries

$$\begin{aligned} & [r_1, i_1, v_1, c_1, d_1, s_1, w_1] \text{ on } I_{j_1} \\ & [r_2, i_2, v_2, c_2, d_2, s_2, w_2] \text{ on } I_{j_2} \\ & \vdots \\ & [r_N, i_N, v_N, c_N, d_N, s_N, w_N] \text{ on } I_{j_N} \end{aligned}$$

is consistent if it is a consistent prefix of itself.

The following lemma shows that a consistent sequence of predecessor entries exists when step II.C.2 of Algorithm 5.1.1 is executed for an entry from a consistent prefix of an ordered list of entries.

**LEMMA 5.4.1** *For a consistent prefix of an ordered list of entries, if Algorithm 5.1.2 is applied to an entry  $[q_1, u_1, l_1, c_0, d_0, q_0, u_0]$  on  $I_{l_0}$  in the prefix and*

$$\left( q_m, u_m, l_m, c_m + d_m, \sum_{i=0}^{m-1} d_i \right) \in B_{m-1}$$

*then there exists*

$$\begin{aligned} & [q_{m+1}, u_{m+1}, l_{m+1}, c_m, d_m, q_m, u_m] \text{ on } I_{l_m} \\ & [q_m, u_m, l_m, c_{m-1}, d_{m-1}, q_{m-1}, u_{m-1}] \text{ on } I_{l_{m-1}} \\ & \vdots \\ & [q_2, u_2, l_2, b_1, c_1, q_1, u_1] \text{ on } I_{l_1} \end{aligned}$$

*such that, when  $m > 0$ ,  $c_x = c_{x+1} + d_{x+1}$  for  $0 \leq x < m$  and*

$$\begin{aligned} & ([q_{m+1}, u_{m+1}, l_{m+1}, c_m, d_m, q_m, u_m] \text{ on } I_{l_m}) \\ & \quad \stackrel{+}{=} ([q_m, u_m, l_m, c_{m-1}, d_{m-1}, q_{m-1}, u_{m-1}] \text{ on } I_{l_{m-1}}) \\ & \quad \stackrel{+}{=} \dots \\ & \quad \stackrel{+}{=} ([q_2, u_2, l_2, c_1, d_1, q_1, u_1] \text{ on } I_{l_1}) \\ & \quad \stackrel{+}{=} ([q_1, u_1, l_1, c_0, d_0, q_0, u_0] \text{ on } I_{l_0}). \end{aligned}$$

**Proof:** The lemma is proved by induction on the  $m$ , using the lemma as the induction hypothesis. The induction proceeds in two steps:

- first, the lemma is proved for  $m = 0$  and  $m = 1$ ; and

- second, the lemma is proved for  $m = N$ , where  $N > 1$ , assuming it holds for  $m = N - 1$ .

For the first induction step,  $m = 0$  or  $m = 1$ . Examination of step I of Algorithm 5.1.2 shows that  $\{(q_0, u_0, l_0, c_0 + d_0, 0)\} \in B_{-1}$  and  $\{(q_1, u_1, l_1, c_0, d_0, 0)\} \in B_0$ . Since the prefix of the ordered list of entries is consistent, there exists an entry  $[q_2, u_2, l_2, c_1, d_1, q_1, u_1]$  on  $I_{l_1}$  for which  $c_0 = c_1 + d_1$  and

$$([q_2, u_2, l_2, c_1, d_1, q_1, u_1] \text{ on } I_{l_1}) \stackrel{+}{=} ([q_1, u_1, l_1, c_0, d_0, q_0, u_0] \text{ on } I_{l_0}).$$

For the second induction step,  $m = N$ , where  $N > 1$ , and the lemma is assumed to hold for  $m = N - 1$ . If  $(q_m, u_m, l_m, c_{m-1}, d')$   $\in B_{N-1}$  then step II.B.1.i of Algorithm 5.1.2 added  $(q_m, u_m, l_m, c_{m-1}, d')$  to  $B_{N-1}$  and there is an entry  $[q_N, u_N, l_N, c_{N-1}, d_{N-1}, q_{N-1}, u_{N-1}]$  on  $I_{N-1}$  and an element  $(q_{N-1}, u_{N-1}, l_{N-1}, c', d' - d_{N-1}) \in B_{N-2}$ . Since  $(q_{N-1}, u_{N-1}, l_{N-1}, c', d' - d_{N-1}) \in B_{N-2}$ , the induction hypothesis can be applied to show that there exist

$$\begin{aligned} & [q_N, u_N, l_N, c_{N-1}, d_{N-1}, q_{N-1}, u_{N-1}] \text{ on } I_{l_{N-1}} \\ & [q_{N-1}, u_{N-1}, l_{N-1}, c_{N-2}, d_{N-2}, q_{N-2}, u_{N-2}] \text{ on } I_{l_{N-2}} \\ & \vdots \\ & [q_2, u_2, l_2, b_1, c_1, q_1, u_1] \text{ on } I_{l_1} \end{aligned}$$

such that, when  $N - 1 > 0$ ,  $c_x = c_{x-1} + d_{x-1}$  for  $0 \leq x < N - 1$  and

$$\begin{aligned} & ([q_{N-2}, u_{N-2}, l_{N-2}, c_{N-1}, d_{N-1}, q_{N-1}, u_{N-1}] \text{ on } I_{l_{N-1}}) \\ & \stackrel{+}{=} ([q_{N-1}, u_{N-1}, l_{N-1}, c_{N-2}, d_{N-2}, q_{N-2}, u_{N-2}] \text{ on } I_{l_{N-2}}) \\ & \stackrel{+}{=} \dots \\ & \stackrel{+}{=} ([q_2, u_2, l_2, b_1, c_1, q_1, u_1] \text{ on } I_{l_1}) \\ & \stackrel{+}{=} ([q_1, u_1, l_1, c_0, d_0, q_0, u_0] \text{ on } I_{l_0}) \end{aligned}$$

where  $c' = c_{N-1} + d_{N-1}$  and  $d' - d_{N-1} = \sum_{i=0}^{N-2} d_i$ . Therefore,  $d' = \sum_{i=0}^{N-1} d_i$ . Furthermore, since the prefix is consistent, there must also be an entry  $[q_{N+1}, u_{N+1}, l_{N+1}, c_N, d_N, q_N, u_N]$  on  $I_{l_N}$  for which  $c_{N-1} = c_N + d_N$  and

$$\begin{aligned} & ([q_{N+1}, u_{N+1}, l_{N+1}, c_N, d_N, q_N, u_N] \text{ on } I_{l_N}) \\ & \stackrel{+}{=} ([q_N, u_N, l_N, c_{N-1}, d_{N-1}, q_{N-1}, u_{N-1}] \text{ on } I_{l_{N-1}}). \end{aligned}$$

Using the previous lemma, all ordered lists of entries can be shown to be consistent.

**LEMMA 5.4.2 (All Ordered Lists of Entries are Consistent)** *For any ordered list of entries, if  $[r, v, i, c, d, s, w]$  is on  $I_j$  then there exists an entry  $[q, u, l, a, b, r, v]$  on  $I_i$  for which  $c = a + b$  and*

$$([q, u, l, a, b, r, v] \text{ on } I_i) \models^+ ([r, v, i, c, d, s, w] \text{ on } I_j).$$

**Proof:** If  $s = 0$ , the lemma is proved by applying Lemma 5.3.3 which shows that  $[r, v, i, c, d, s, w] = [0, \epsilon, 0, 0, d, 0, w]$  and

$$[0, \epsilon, 0, 0, 0, 0, \epsilon] \models [0, \epsilon, 0, 0, d, 0, w].$$

If  $s \neq 0$ , the lemma is proved by induction on an ordered list of entries, using the lemma as the induction hypothesis. The induction proceeds in two steps:

- first, the lemma is proved for the first entry on the ordered list of entries; and
- second, the lemma is proved for the  $N^{\text{th}}$  entry on the ordered list of entries, assuming it holds for all entries before the  $N^{\text{th}}$  entry (which says that the entries before the  $N^{\text{th}}$  entry form a consistent prefix of the ordered list of entries).

For the first induction step, let  $[r, i, v, c, d, s, w]$  on  $I_j$  be the first entry on the ordered list of entries. Since the first entry must be added to its parse list by step II.C of Algorithm 5.1.1,  $([r, v, i, c, d, s, w], j)$  must have been on  $H$ . Step I and step II.C.1.i did not add  $([r, v, i, c, d, s, w], j)$  to  $H$  since if one of these steps added  $([r, v, i, c, d, s, w], j)$  then  $|w| < k$ . Therefore, there are only two cases to consider:

- step II.B adds  $([r, v, i, c, d, s, w], j)$  to  $H$ , or
- step II.C.2.iii adds  $([r, v, i, c, d, s, w], j)$  to  $H$ .

In the first case, step II.B of Algorithm 5.1.1 adds  $([r, v, i, c, d, s, w], j)$  to  $H$  so, applying Lemma 5.3.2, either step I or II.C.1.i of Algorithm 5.1.1 adds  $([r, v, i, c, 0, s, w_1 \dots i], i)$  to  $H$ , where

$$([r, v, i, c, 0, s, w_1 \dots i], i) \models^+ ([r, v, i, c, d, s, w], j).$$

Step I can not add  $([r, v, i, c, 0, s, w_{1:t}], i)$  since that implies that  $s = 0$  and entries for which  $s = 0$  are not on the ordered list of entries. Therefore, step II.C.1.i must add  $([r, v, i, c, 0, s, w_{1:t}], i)$  to  $H$ . Examining step II.C.1.i shows this step is processing an entry  $[q, u, l, a, b, r, v]$  on  $I_i$  for which  $c = a + b$ . Furthermore,

$$([q, u, l, a, b, r, v] \text{ on } I_i) \models ([r, v, i, c, d, s, w] \text{ on } I_j).$$

In the second case, step II.C.2.iii of Algorithm 5.1.1 adds  $([r, v, i, c, d, s, w], j)$  to  $H$ . Step II.C.2.iii of Algorithm 5.1.1 must be processing an entry of the form  $[0, \epsilon, 0, 0, b, 0, u]$  on  $I_i$  since this is the only form of entry that is not on the ordered list of entries. When  $[0, \epsilon, 0, 0, b, 0, u]$  is processed, step II.C.2.iii must be simulating one of two types of reductions:

- a reduction by an empty production, or
- a reduction by a non-empty production.

For a reduction by an empty production,  $u = v$ ,  $c = b$ ,  $d = 0$  and

$$([0, \epsilon, 0, 0, c, 0, v] \text{ on } I_i) \models ([0, v, i, c, 0, s, w] \text{ on } I_j).$$

For a reduction by a non-empty production,  $u = v = \epsilon$ ,  $c = 0$ ,  $d = b$  and

$$([0, \epsilon, 0, 0, d, 0, v] \text{ on } I_i) \models ([0, \epsilon, 0, 0, d, s, w] \text{ on } I_j).$$

For the second induction step, the lemma is assumed to hold for all entries before the  $N^{\text{th}}$  entry on the ordered list of entries. Let the  $N^{\text{th}}$  entry be  $[r, v, i, c, d, s, w]$  on  $I_j$ . Since the  $N^{\text{th}}$  entry is added to its parse list by step II.C of Algorithm 5.1.1,  $([r, v, i, c, d, s, w], j)$  must have been on  $H$  and  $|w| = k$ . Step I and step II.C.1.i did not add  $([r, v, i, c, d, s, w], j)$  to  $H$  since if one of these steps adds  $([r, v, i, c, d, s, w], j)$  then  $|w| < k$ . Therefore, there are only two cases to consider:

- step II.B adds  $([r, v, i, c, d, s, w], j)$  to  $H$ , or
- step II.C.2.iii adds  $([r, v, i, c, d, s, w], j)$  to  $H$ .

In the first case, step II.B of Algorithm 5.1.1 adds  $([r, v, i, c, d, s, w], j)$  to  $H$  and Algorithm 5.1.1 simulates an advance. Applying Lemma 5.3.2, either step I or step II.C.1.i of Algorithm 5.1.1 must add  $([r, v, i, c, 0, s, w_{1:t}], i)$  to  $H$  where  $t < k$  and

$$([r, v, i, c, 0, s, w_{1:t}], i) \models^+ ([r, v, i, c, d, s, w], j).$$

Step I could not add  $([r, v, i, c, 0, s, w_{1:t}], i)$  since that implies  $s = 0$  and that  $[r, v, i, c, d, s, w]$  on  $I_j$  is not on the ordered list of entries. Therefore, step II.C.1.i adds  $([r, v, i, c, 0, s, w_{1:t}], i)$  to  $H$ , simulating a shift. Examining step II.C.1.i shows this step must be processing an entry  $[q, u, l, a, b, r, v]$  on  $I_i$  for which  $c = a + b$ . Furthermore,

$$([q, u, l, a, b, r, v] \text{ on } I_i) \models ([r, v, i, c, d, s, w] \text{ on } I_i).$$

In the second case, step II.C.2.iii of Algorithm 5.1.1 adds  $([r, v, i, c, d, s, w], j)$  to  $H$ . This step must be processing an entry  $[q_{h-1}, u_{h-1}, l_{h-1}, a_h, b_h, q_h, u_h]$  on  $I_{l_h}$  where  $h$  is an arbitrary index chosen for convenience. Therefore

$$([q_{h-1}, u_{h-1}, l_{h-1}, a_h, b_h, q_h, u_h] \text{ on } I_{l_h}) \models ([r, i, v, c, d, s, w] \text{ on } I_j).$$

Furthermore, when  $[q_{h-1}, u_{h-1}, l_{h-1}, a_h, b_h, q_h, u_h]$  is processed, there must be  $s \in g_r(\text{LHS}(p))$  and

$$(h, b'', p, w, j) \in G_{l_h}(q_h, u_h)$$

and

$$(q_0, u_0, l_0, a', b') \in B_{h-1}$$

where  $m = |\text{RHS}(p)|$ ,  $c = a'$ ,  $d = b' + b''$ ,  $q_0 = r$ ,  $u_0 = v$ , and  $l_0 = i$ . Applying Lemma 5.4.1, there are

$$\begin{aligned} & [q_{-1}, u_{-1}, l_{-1}, a_0, b_0, q_0, u_0] \text{ on } I_{l_0} \\ & [q_0, u_0, l_0, a_1, b_1, q_1, u_1] \text{ on } I_{l_1} \\ & \vdots \\ & [q_{h-2}, u_{h-2}, l_{h-2}, a_{h-1}, b_{h-1}, q_{h-1}, u_{h-1}] \text{ on } I_{l_{h-1}} \end{aligned}$$

where

$$\begin{aligned} & ([q_{-1}, u_{-1}, l_{-1}, a_0, b_0, q_0, u_0] \text{ on } I_{l_1}) \\ & \quad \models^+ ([q_0, u_0, l_0, a_1, b_1, q_1, u_1] \text{ on } I_{l_1}) \\ & \quad \models^+ \dots \\ & \quad \models^+ ([q_{h-1}, u_{h-1}, l_{h-1}, a_h, b_h, q_h, u_h] \text{ on } I_{l_{h-1}}) \end{aligned}$$

$a' = a_0 + b_0$  and  $b' = \sum_{x=1}^h b_x$ . Therefore,  $c = a_0 + b_0$  and

$$([q_{-1}, u_{-1}, l_{-1}, a_0, b_0, r, v] \text{ on } I_i) \models^+ ([r, i, v, c, d, s, w] \text{ on } I_j).$$

■

### 5.4.2 Monotonically Increasing Ordered Lists of Entries

Besides completeness, another important property of ordered lists of entries must be demonstrated before Algorithm 5.1.1 can be shown to be complete. This property is that all entries are added to their parse lists in order of increasing cost. The following two definitions define the idea of a monotonically increasing ordered list of entries.

*Definition 5.4.4 (Monotonically Increasing Prefix of Ordered List of Entries)*

A prefix of an ordered list of entries

$$\begin{aligned} & [r_1, v_1, i_1, c_1, d_1, s_1, w_1] \text{ on } I_{j_1} \\ & [r_2, v_2, i_2, c_2, d_2, s_2, w_2] \text{ on } I_{j_2} \\ & \vdots \\ & [r_n, v_n, i_n, c_n, d_n, s_n, w_n] \text{ on } I_{j_n} \end{aligned}$$

is monotonically increasing if  $c_l + d_l \leq c_{l+1} + d_{l+1}$  for  $1 \leq l < n$ .

*Definition 5.4.5 (Monotonically Increasing Ordered List of Entries)*

An ordered list of entries

$$\begin{aligned} & [r_1, v_1, i_1, c_1, d_1, s_1, w_1] \text{ on } I_{j_1} \\ & [r_2, v_2, i_2, c_2, d_2, s_2, w_2] \text{ on } I_{j_2} \\ & \vdots \\ & [r_N, v_N, i_N, c_N, d_N, s_N, w_N] \text{ on } I_{j_N} \end{aligned}$$

is monotonically increasing if it is a monotonically increasing prefix of itself.

The next two lemmas show that the monotonically increasing property is equivalent to the property that the sum of the cost components of any entry is greater than or equal to the sum of the cost components of any of its predecessors.

**LEMMA 5.4.3** *For a monotonically increasing prefix of an ordered list of entries, if*

$$([q, u, h, a, b, r, v] \text{ on } I_i) \stackrel{+}{=} ([s, w, j, c, d, t, y] \text{ on } I_l)$$

*then*  $a + b \leq c + d$ .

**Proof:** If  $[q, u, h, a, b, r, v] = [s, w, j, c, d, t, y]$  and  $i = l$  then the lemma is trivially proved. Otherwise,  $[q, u, h, a, b, r, v]$  on  $I_i$  must occur before  $[s, w, j, c, d, t, y]$  on  $I_l$  on the ordered list of entries. This implies  $a + b \leq c + d$ . ■



LEMMA 5.4.4 *Given a prefix of an ordered list of entries,*

$$\begin{aligned} & [r_1, v_1, i_1, c_1, d_1, s_1, w_1] \text{ on } I_{j_1} \\ & [r_2, v_2, i_2, c_2, d_2, s_2, w_2] \text{ on } I_{j_2} \\ & \vdots \\ & [r_n, v_n, i_n, c_n, d_n, s_n, w_n] \text{ on } I_{j_n} \end{aligned}$$

*if*  $c_l + d_l \leq c_m + d_m$  *whenever*

$$([r_l, v_l, i_l, c_l, d_l, s_l, w_l] \text{ on } I_{j_l}) \stackrel{+}{=} ([r_m, v_m, i_m, c_m, d_m, s_m, w_m] \text{ on } I_{j_m})$$

*then the prefix is monotonically increasing.*

Proof: Let  $[r_l, v_l, i_l, c_l, d_l, s_l, w_l]$  on  $I_{j_l}$  and  $[r_m, v_m, i_m, c_m, d_m, s_m, w_m]$  on  $I_{j_m}$  be any two entries on the prefix of the ordered list of entries for which  $l < m$ . When  $[r_l, v_l, i_l, c_l, d_l, s_l, w_l]$  on  $I_{j_l}$  is removed from  $H$  in step II.A of Algorithm 5.1.1, either  $([r_m, v_m, i_m, c_m, d_m, s_m, w_m], j_m)$  or a predecessor of  $([r_m, v_m, i_m, c_m, d_m, s_m, w_m], j_m)$  is on  $H$  since  $[r_l, v_l, i_l, c_l, d_l, s_l, w_l]$  on  $I_{j_l}$  precedes  $[r_m, v_m, i_m, c_m, d_m, s_m, w_m]$  on  $I_{j_m}$  on the ordered list of entries. Therefore,  $c_l + d_l \leq c_m + d_m$ . ■

The importance of monotonically increasing ordered lists of entries is demonstrated in the following lemma which shows that for monotonically increasing orders the duplicate entry elimination performed by step II.C of Algorithm 5.1.1 only discards higher cost entries.

LEMMA 5.4.5 *For a monotonically increasing ordered list of entries, if  $([r, v, i, c, d, s, w], j)$  is on  $H$ ,  $|w| = k$ ,  $c + d = \min(\{W(T) \mid z_{1:j} \xrightarrow{T} y\})$  and the explicitly-advancing LR( $k$ ) parser can make the sequence of moves*

$$(0, \epsilon, y) \stackrel{*}{\vdash} (\alpha \mid r, v, \dots) \stackrel{+}{\vdash} (\alpha r s, ?, \dots) \stackrel{a^*}{\vdash} (\alpha r s, w, \epsilon)$$

*then  $[r, v, i, c, d, s, w]$  is added to  $I_j$  or there is already an entry  $[r, v, i, c', d', s, w]$  on  $I_j$  where  $c' + d' = c + d$ .*

Proof: Lemma 5.3.1 guarantees that  $([r, v, i, c, d, s, w], j)$  will eventually be processed by step II.C of Algorithm 5.1.1. When  $([r, v, i, c, d, s, w], j)$  is processed, if there is no entry  $[r, v, i, c', d', s, w]$  on  $I_j$ , then the lemma is proved. If there is an entry  $[r, v, i, c', d', s, w]$  on  $I_j$ , then there are three cases to consider:

- first,  $c' + d' = c + d$ ;

- second,  $c' + d' < c + d$ ; or
- third,  $c' + d' > c + d$ .

For the first case,  $c' + d' = c + d$  and the lemma is trivially proved. The other two cases can not occur since assuming that they do occur leads to contradictions.

For the second case,  $c' + d' < c + d$ . Applying Theorem 5.3.1, there must exist an edit sequence  $T'$  such that  $c' + d' = W(T')$ ,  $z_{1:j} \xrightarrow{T'} y'$  and the explicitly-advancing  $LR(k)$  parser can make the sequence of moves

$$(0, \epsilon, y') \vdash^* (\alpha \backslash r, v, \dots) \vdash^+ (\alpha r s, ?, \dots) \vdash^{a^*} (\alpha r s, w, \epsilon).$$

However, this implies that  $c + d \neq \min(\{W(T) \mid z_{1:j} \xrightarrow{T} y\})$  which is a contradiction.

For the third case,  $c' + d' > c + d$ . When  $([r, v, i, c', d', s, w], j)$  is removed from  $H$  in step II.C of Algorithm 5.1.1,  $([r, v, i, c, d, s, w], j)$  or a predecessor of  $([r, v, i, c, d, s, w], j)$  is on  $H$ . Therefore,  $c' + d' < c + d$ , which is a contradiction. ■

The following lemma shows that all orders are monotonically increasing.

**LEMMA 5.4.6 (An Ordered List of Entries is Monotonically Increasing)** *If  $([q, u, h, a, b, r, v]$  on  $I_i$ )  $\models$   $([s, w, j, c, d, t, x]$  on  $I_l$ ) then  $a + b \leq c + d$ .*

**Proof:** When  $s = 0$ , the lemma is proved by applying Lemma 5.3.2. When  $s \neq 0$ , the lemma is proved by examining the ways in which an entry  $([r, v, i, c, d, s, w], j)$  can be added to the list of entries to be parsed. Step I and step II.C.1.i can not add  $([r, v, i, c, d, s, w], j)$  to  $H$  since if one of these steps adds  $([r, v, i, c, d, s, w], j)$  then  $|w| < k$ . Therefore, there are only two cases to consider:

- step II.B adds  $([r, v, i, c, d, s, w], j)$  to  $H$ , or
- step II.C.2.iii adds  $([r, v, i, c, d, s, w], j)$  to  $H$ .

In the first case, step II.B of Algorithm 5.1.1 adds  $([r, v, i, c, d, s, w], j)$  to  $H$  so, applying Lemma 5.3.2, either step I or II.C.1.i of Algorithm 5.1.1 must add  $([r, v, i, c, 0, s, w_{1:t}], j')$  to  $H$  and

$$([r, v, i, c, 0, s, w_{1:t}], j') \vdash^+ ([r, v, i, c, d, s, w], j).$$

Step I could not add  $([r, v, i, c, 0, s, w_{1:t}], j')$  to  $H$  since that implies  $s = 0$ . Therefore, step II.C.1.i must have added  $([r, v, i, c, 0, s, w_{1:t}], j')$  to  $H$ . Examining step II.C.1.i

shows this step must be processing an entry  $[q, u, h, a, b, r, v]$  on  $I_i$  for which  $c = a + b$  and

$$([q, u, h, a, b, r, v] \text{ on } I_i) \models ([r, v, i, c, d, s, w] \text{ on } I_j).$$

Since  $d \geq 0$ ,  $a + b \leq c + d$ .

In the second case, step II.C.2.iii of Algorithm 5.1.1 adds  $([r, v, i, c, d, s, w], j)$  to  $H$ . Step II.C.2.iii must have been processing an entry  $[q_{h-1}, u_{h-1}, l_{h-1}, a_h, b_h, q_h, u_h]$  on  $I_{l_h}$  where

$$([q_{h-1}, u_{h-1}, l_{h-1}, a_h, b_h, q_h, u_h] \text{ on } I_{l_h}) \models ([r, v, i, c, d, s, w] \text{ on } I_j).$$

Furthermore, examining step II.C.2.iii shows that when  $[q_{h-1}, u_{h-1}, l_{h-1}, a_h, b_h, q_h, u_h]$  was processed, there must have been  $(h, b'', p, w, j) \in G_{l_h}(q_h, u_h)$  and an  $(q_0, u_0, l_0, a', b') \in B_{h-1}$ , where  $m = |\text{RHS}(p)|$ ,  $c = a'$ ,  $d = b' + b''$ ,  $q_0 = r$ ,  $u_0 = v$ , and  $l_0 = i$ . It is important to note that  $h$  has been arbitrarily chosen as a convenient index for the entry  $[q_{h-1}, u_{h-1}, l_{h-1}, a_h, b_h, q_h, u_h]$  on  $I_{l_h}$ . Applying Lemma 5.4.1, there are

$$\begin{aligned} & [q_{-1}, u_{-1}, l_{-1}, a_0, b_0, q_0, u_0] \text{ on } I_{l_1} \\ & [q_0, u_0, l_0, a_1, b_1, q_1, u_1] \text{ on } I_{l_1} \\ & \vdots \\ & [q_{h-2}, u_{h-2}, l_{h-2}, a_{h-1}, b_{h-1}, q_{h-1}, u_{h-1}] \text{ on } I_{l_{h-1}} \end{aligned}$$

where  $a' = a_0 + b_0$ ,  $b' = \sum_{x=1}^h b_x$ , and  $a_h + b_h = a' + b'$ . Since  $c = a'$ ,  $b = b' + b''$  and  $b'' \geq 0$ ,  $a_h + b_h \leq c + d$  and the lemma is proved for this case. ■

### 5.4.3 Completeness

Since it has been established that all ordered lists of entries are consistent and monotonically increasing, the argument that Algorithm 5.1.1 is complete can proceed. The next two lemmas establish that step II.B completely simulates sequences of advances for the explicitly-advancing LR( $k$ ) parser.

**LEMMA 5.4.7 (Step II.B Simulates Edit Operations)** *If  $([q, u, i, c, d, r, v], l)$ , where  $|v| < k$ , is on the pending list,  $H$ , and there exists an edit operation  $(z_{l+1:j+1} \mapsto a)$ , where  $l - 1 \leq j \leq l$ ,  $b = W((z_{l+1:j+1} \mapsto a))$ , and  $|va| \leq k$ , then  $([q, u, i, c, d + b, r, va], j + 1)$  is added to  $H$ .*

**Proof:** Applying Lemma 5.3.1,  $([q, u, i, c, d, r, v], l)$  is processed by step II.B of Algorithm 5.1.1. There are three types of edit operations to consider:

- an insertion ( $j = l - 1$  and  $|a| = 1$ ),
- a deletion ( $j = l$  and  $a = \epsilon$ ), and
- a replacement ( $j = l$  and  $|a| = 1$ ).

For an insertion,  $j = l - 1$  and  $|a| = 1$ . When  $([q, u, i, c, d, r, v], l)$  is processed, step II.B.3 of Algorithm 5.1.1 adds  $([q, u, i, c, d + b, r, va], j + 1)$  to  $H$ . Note that step II.B.3 is optimized to exclude the insertion of  $a$  before another  $a$  since this can never result in a least-cost edit.

For a deletion,  $j = l$  and  $a = \epsilon$ . When  $([q, u, i, c, d, r, v], l)$  is processed, step II.B.2 of Algorithm 5.1.1 adds  $([q, u, i, c, d + b, r, va], j + 1)$  to  $H$ .

For a replacement,  $j = l$  and  $|a| = 1$ . When  $([q, u, i, c, d, r, v], l)$  is processed, step II.B.1 or step II.B.4 of Algorithm 5.1.1 adds  $([q, u, i, c, d + b, r, va], j + 1)$  to  $H$ . ■

**LEMMA 5.4.8 (Simulation of Advances is Complete)** *If  $([q, u, i, c, d, r, v], h)$ , where  $|v| < k$ , is on the pending list,  $H$ , and there exists an edit sequence  $S$  such that  $z_{h+1:j} \xrightarrow{S} y$ ,  $d' = W(S)$ , and  $|vy| = k$  then  $([q, u, i, c, d + d', r, vy], j + 1)$  is added to  $H$ .*

**Proof:** Let

$$S = (a_1 \mapsto b_1)(a_2 \mapsto b_2) \dots (a_n \mapsto b_n)$$

where  $z_{h+1:j} = a_1 a_2 \dots a_n$  and  $y = b_1 b_2 \dots b_n$ . Also, let  $l_0 = h$ ,  $l_n = j$ , and let  $l_0, l_1, \dots, l_n$  be a sequence of integers where, for  $1 \leq x \leq n$ ,  $z_{l_{x-1}+1:l_x} = a_x$ ,  $h \leq l_x \leq j$  and  $l_x - 1 \leq l_{x-1} \leq l_x$ . Finally, let  $d_x = W((a_x \mapsto b_x))$  which implies that  $\sum_{x=1}^n d_x = W(S)$ .

Applying Lemma 5.3.1 and Lemma 5.4.7 to  $([q, u, i, c, d, r, v], h)$  shows that  $([q, u, i, c, d + d_1, r, vb_1], l_1)$  is added to  $H$ . Repeated application of Lemma 5.3.1 and Lemma 5.4.7 show that the entries

$$\begin{aligned} & ([q, u, i, c, d + d_1, r, vb_1], l_1) \\ & ([q, u, i, c, d + d_1, r, vb_1 b_2], l_2) \\ & \vdots \\ & ([q, u, i, c, d + \sum_{x=1}^n d_x, r, vb_1 b_2 \dots b_n], l_n) \end{aligned}$$

are added to  $H$ . ■

The following lemma establishes an important property for least-cost edits of prefixes of the input string.

LEMMA 5.4.9 *If there exist edit sequences  $S$  and  $S'$  such that  $z_{1:n} \xrightarrow{S} y$ ,  $z_{i+1:j} \xrightarrow{S'} y'$ ,  $W(S) = c$ ,  $W(S') = d$ ,*

$$c + d = \min \left( \left\{ W(T) \mid z_{1:j} \xrightarrow{T} t \text{ such that } (0, \epsilon, t) \vdash^* (\alpha r s, w, \epsilon) \right\} \right)$$

*and the explicitly-advancing  $LR(k)$  parser can make the sequence of moves*

$$(0, \epsilon, yy') \vdash^* (\alpha \wr r, v, y') \vdash^* (\alpha r s, w, \epsilon)$$

*then*

$$c = \min \left( \left\{ W(T) \mid z_{1:n} \xrightarrow{T} t \text{ such that } (0, \epsilon, t) \vdash^* (\alpha r, v, \epsilon) \right\} \right).$$

Proof: The lemma is proved by contradiction. Assume there exists an edit sequence  $S''$  such that  $z_{1:n} \xrightarrow{S''} y''$  and  $W(S'') = b$  and for which  $b < c$  and

$$(0, \epsilon, y'') \vdash^* (\alpha r, v, \epsilon).$$

Then the explicitly-advancing  $LR(k)$  parser can make the sequence of moves

$$(0, \epsilon, y''y') \vdash^* (\alpha \wr r, v, y') \vdash^* (\alpha r s, w, \epsilon)$$

and  $b + d < c + d$ , which is a contradiction. ■

The next two lemmas establish reduction information that can be inferred from the entries on their parse lists at key points during the execution of Algorithm 5.1.1.

LEMMA 5.4.10 *If Algorithm 5.1.2 is applied to  $[q_1, u_1, l_1, c_0, d_0, q_0, u_0]$  on  $I_{l_0}$  and there exist*

$$\begin{aligned} & [q_{m+1}, u_{m+1}, l_{m+1}, c_m, d_m, q_m, u_m] \text{ on } I_{l_m} \\ & [q_m, u_m, l_m, c_{m-1}, d_{m-1}, q_{m-1}, u_{m-1}] \text{ on } I_{l_{m-1}} \\ & \vdots \\ & [q_2, u_2, l_2, b_1, c_1, q_1, u_1] \text{ on } I_{l_1} \end{aligned}$$

*then, for  $m > 0$ ,*

$$\left( q_m, u_m, l_m, c_{m-1}, \sum_{i=0}^{m-1} d_i \right) \in B_{m-1}.$$

Proof: The lemma is proved by induction on  $m$ , using the lemma as the induction hypothesis. The induction proceeds in two steps:

- first, the lemma is proved for  $m = 1$ ; and
- second, the lemma is proved for  $m = N$  when  $N > 1$ , assuming it holds when  $m = N - 1$ .

For the first induction step,  $m = 1$ . Examination of step I of Algorithm 5.1.2 shows that  $(q_0, u_0, l_0, c_0, d_0)$  is added to  $B_0$ .

For the second induction step,  $m = N$  where  $N > 1$  and the lemma is assumed to hold for  $m = N - 1$ . The induction hypothesis can be applied to

$$\begin{aligned} & [q_N, u_N, l_N, c_{N-1}, d_{N-1}, q_{N-1}, u_{N-1}] \text{ on } I_{l_{N-1}} \\ & [q_{N-1}, u_{N-1}, l_{N-1}, c_{N-2}, d_{N-2}, q_{N-2}, u_{N-2}] \text{ on } I_{l_{N-2}} \\ & \vdots \\ & [q_2, u_2, l_2, b_1, c_1, q_1, u_1] \text{ on } I_{l_1} \end{aligned}$$

to show that  $(q_{N-1}, u_{N-1}, l_{N-1}, c_{N-2}, \sum_{i=0}^{N-2} d_i) \in B_{N-2}$ . Since

$$(q_{N-1}, u_{N-1}, l_{N-1}, c_{N-2}, \sum_{i=0}^{N-2} d_i) \in B_{N-2}$$

and  $[q_{N+1}, u_{N+1}, l_{N+1}, c_N, d_N, q_N, u_N]$  on  $I_{l_N}$ , examining step II.B.1.i of Algorithm 5.1.2 shows that

$$(q_N, u_N, l_N, c_{N-1}, \sum_{i=0}^{N-1} d_i) \in B_{N-1}.$$

■

**LEMMA 5.4.11** *If  $m > 0$ ,  $h > 0$ ,  $m + h = |RHS(p)|$ , reduce  $p \in f_{q_m}(u_m)$ , and there exist*

$$\begin{aligned} & [q_0, u_0, l_0, c_1, d_1, q_1, u_1] \text{ on } I_{l_1} \\ & [q_1, u_1, l_1, c_2, d_2, q_2, u_2] \text{ on } I_{l_2} \\ & \vdots \\ & [q_{m-1}, u_{m-1}, l_{m-1}, c_m, d_m, q_m, u_m] \text{ on } I_{l_m} \end{aligned}$$

*then*

$$(h, \sum_{x=1}^m d_x, p, u_m, l_m) \in G_{l_0}(q_0, u_0).$$

Proof: The lemma is proved by induction on  $m$ , using the lemma as the induction hypothesis. The induction proceeds in two steps:

- first, the lemma is proved for  $m = 1$ ; and
- second, the lemma is proved for  $m = N$ , assuming it holds when  $m < N$ .

For the first induction step,  $m = 1$ . The entry  $[q_0, u_0, l_0, c_1, d_1, q_1, u_1]$  is on  $I_{l_1}$  so it must have been processed by step II.C.1.ii of Algorithm 5.1.1. Since **reduce**  $p \in f_{q_1}(u_1)$ , step II.C.1.ii adds  $(h + 1, 0, p, u_1, l_1)$  to  $G_{l_1}(q_1, u_1)$ . Thus, step II.C.2.ii executes Algorithm 5.1.2 and step I of Algorithm 5.1.2 adds  $(h, d_1, p, u_1, l_1)$  to  $G_{l_0}(q_0, u_0)$  since  $h > 0$ .

For the second induction step,  $m = N$  and the lemma is assumed to hold for  $m < N$ . Since  $m = N$ , there are  $N$  entries

$$\begin{aligned} & [q_0, u_0, l_0, c_1, d_1, q_1, u_1] \text{ on } I_{l_1} \\ & [q_1, u_1, l_1, c_2, d_2, q_2, u_2] \text{ on } I_{l_2} \\ & \vdots \\ & [q_{N-1}, u_{N-1}, l_{N-1}, c_N, d_N, q_N, u_N] \text{ on } I_{l_N}. \end{aligned}$$

Let  $k$  be the index of the entry  $[q_{k-1}, u_{k-1}, l_{k-1}, c_k, d_k, q_k, u_k]$  on  $I_{l_k}$  which is the last of these entries to be processed by step II.B of Algorithm 5.1.1. There must be

$$(h + k, \sum_{x=k+1}^N d_x, p, u_m, l_m) \in G_{l_k}(q_k, u_k)$$

when  $[q_{k-1}, u_{k-1}, l_{k-1}, c_k, d_k, q_k, u_k]$  on  $I_{l_k}$  is processed by step II.B since the induction hypothesis can be applied to the  $N - k$  entries

$$\begin{aligned} & [q_k, u_k, l_k, c_{k+1}, d_{k+1}, q_{k+1}, u_{k+1}] \text{ on } I_{l_{k+1}} \\ & [q_{k+1}, u_{k+1}, l_{k+1}, c_{k+2}, d_{k+2}, q_{k+2}, u_{k+2}] \text{ on } I_{l_{k+2}} \\ & \vdots \\ & [q_{N-1}, u_{N-1}, l_{N-1}, c_N, d_N, q_N, u_N] \text{ on } I_{l_N}. \end{aligned}$$

Since  $(h + k, \sum_{x=k+1}^N d_x, p, u_m, l_m) \in G_{l_k}(q_k, u_k)$  when  $[q_{k-1}, u_{k-1}, l_{k-1}, c_k, d_k, q_k, u_k]$  on  $I_{l_k}$  is processed by step II.B.2.ii of Algorithm 5.1.1, step I of Algorithm 5.1.1 initializes  $B_{-1}$  to  $(q_k, u_k, l_k, c_k + d_k, 0)$  and  $B_0$  to  $(q_{k-1}, u_{k-1}, l_{k-1}, c_k, d_k)$  and adds  $(h+k-1, \sum_{x=k}^N d_x, p, u_m, l_m)$  to  $G_{l_{k-1}}(q_{k-1}, u_{k-1})$ . The loop in step II of Algorithm 5.1.2 must iterate through at least the sequence  $o = 1, 2, \dots, k-1$  and it has the following property:

If  $(q_x, u_x, l_x, a, b) \in B_{o-1}$ ,  $(n+1, \sum_{i=x+1}^N d_i, p, u_m, l_m) \in G_{l_x}(q_x)$ , and  $[q_{x-1}, u_{x-1}, l_{x-1}, c_x, d_x, q_x, u_x]$  is on  $I_{l_x}$  then  $(q_{x-1}, u_{x-1}, l_{x-1}, c_x, b + d_x)$  is added to  $B_o$  and  $(n, \sum_{i=x}^N d_i, p, u_m, l_m)$  is added to  $G_{l_{x-1}}(q_{x-1}, u_{x-1})$ .

Therefore, Algorithm 5.1.2 must add  $(h, \sum_{x=1}^N d_x, p, u_m, l_m)$  to  $G_{l_0}(q_0)$ . ■

**THEOREM 5.4.1** *Given the same  $Q$ ,  $f_q$ , and  $g_q$  for the explicitly-advancing  $LR(k)$  parser and Algorithm 5.1.1, if the sequence of moves*

$$(0, \epsilon, yy') \vdash^{a^*} (0, ?, \dots) \vdash^L (\alpha(r, v, y')) \vdash^{M+1} (\alpha rs, ?, \dots) \vdash^{a^*} (\alpha rs, w, \epsilon)$$

*can be made by the explicitly-advancing  $LR(k)$  parser, where  $L \geq 0$ ,  $M \geq 0$ ,  $|v| = k$  or  $|v| = 0$ ,  $|w| = k$ , and there exist edit sequences  $S$  and  $S'$  such that  $z_{1..} \xrightarrow{S} y$ ,  $z_{i+1..j} \xrightarrow{S'} y'$ ,  $W(S) = c$ ,  $W(S') = d$ , and*

$$c + d = \min \left( \left\{ W(T) \mid z_{1..j} \xrightarrow{T} t \text{ such that } (0, \epsilon, t) \vdash^* (\alpha rs, w, \epsilon) \right\} \right)$$

*then Algorithm 5.1.1 adds the entry  $[r, v, i, c', d', s, w]$  to parse list  $I_j$  where  $c' + d' = c + d$ .*

**Proof:** The theorem is proved by induction on the sum of  $L$  and  $M$ . The induction proceeds in three steps:

- first, the theorem is proved for  $L + M = 0$ ;
- second, the theorem is proved for for  $L = N$  and  $M = 0$ , assuming it holds whenever  $L + M < N$ ; and
- third, the theorem is proved for for  $L + M = N$ , assuming it holds whenever  $L + M < N$ .

If the parameter space formed by  $L$  and  $M$  is seen as an infinite table with  $L$  as the row number and  $M$  as the column number, the steps of the induction can be viewed as proving the theorem diagonal by diagonal, using the diagonals that run from the lower left to upper right sides of the table.

For the first step of the induction,  $L + M = 0$  and the theorem can be written as follows:



If the explicitly-advancing  $\text{LR}(k)$  parser can make the sequence of moves

$$(0, \epsilon, vy') \stackrel{a^*}{\vdash} (0, v, y') \vdash (0s, ?, \dots) \stackrel{a^*}{\vdash} (0s, w, \epsilon)$$

where  $|v| = k$ ,  $|y'| = k$  and there exist edit sequences  $S$  and  $S'$  such that  $z_{1:j} \xrightarrow{S} v$ ,  $z_{i+1:j} \xrightarrow{S'} y'$ ,  $W(S) = c$ ,  $W(S') = d$  and

$$c + d = \min \left( \left\{ W(T) \mid z_{1:j} \xrightarrow{T} t \text{ such that } (0, \epsilon, t) \stackrel{*}{\vdash} (0s, w, \epsilon) \right\} \right)$$

then  $[0, v, i, c', d', s, w]$  is added to  $I_j$  where  $c' + d' = c + d$ .

Since the explicitly-advancing  $\text{LR}(k)$  parser can make the sequence of advances

$$(0, \epsilon, vy') \stackrel{a^*}{\vdash} (0, v, y')$$

and step I of Algorithm 5.1.1 places  $([0, \epsilon, 0, 0, 0, \epsilon], 0)$  on  $H$ , applying Lemma 5.4.7 shows that  $([0, \epsilon, 0, 0, c, 0, v], i)$  is added to  $H$ . Lemma 5.4.9 shows that

$$c = \min \left( \left\{ W(T) \mid z_{1:j} \xrightarrow{T} t \text{ such that } (0, \epsilon, t) \stackrel{*}{\vdash} (0, v, \epsilon) \right\} \right)$$

so Lemma 5.3.1, Lemma 5.3.3 and Lemma 5.4.5 show that  $[0, \epsilon, 0, 0, c, 0, v]$  is on  $I_i$ .

When  $([0, \epsilon, 0, 0, c, 0, v], i)$  is processed by step II.C, Algorithm 5.1.1 should simulate the move

$$(0, v, y') \vdash (0s, ?, \dots)$$

which can be made by the explicitly-advancing  $\text{LR}(k)$  parser. This move can be one of three types of moves:

- a shift (**shift**  $\in f_0(v)$ ),
- a reduction by an empty production (**reduce**  $p \in f_0(v)$  and  $|\text{RHS}(p)| = 0$ ),  
or
- a reduction by a non-empty production (**reduce**  $p \in f_0(v)$  and  $|\text{RHS}(p)| > 0$ ).

For each of the three types of moves,  $s \in g_0(v)$ .

For a shift, **shift**  $\in f_0(v)$ . Therefore,  $w = v_{2:k}y'$  and  $|y'| = 1$ . Examining step II.C.1.i shows that  $([0, v, i, c, 0, s, v_{2:k}], i)$  is added to  $H$ . Applying Lemma 5.4.7 shows that  $([0, v, i, c, d, s, w], j)$  is added to  $H$ . Lemma 5.3.1 and Lemma 5.4.5 show that  $[0, v, i, c', d', s, w]$  is added to  $I_j$  where  $c' + d' = c + d$ .

For a reduction by an empty production, **reduce**  $p \in f_0(v)$ ,  $|\text{RHS}(p)| = 0$  and  $j = i$ . Therefore,  $w = v$ ,  $y' = \epsilon$  and  $d = 0$ . Examining step II.C.2 shows that  $([0, v, i, c, 0, s, v], j)$  is added to  $H$ . Lemma 5.3.1 and Lemma 5.4.5 show that  $[0, v, i, c', d', s, w]$  is added to  $I_j$  where  $c' + d' = c$ .

For a reduction by a non-empty production, **reduce**  $p \in f_0(v)$ ,  $|\text{RHS}(p)| > 0$ ,  $v = \epsilon$ , and  $i = 0$ . Therefore,  $y' = \epsilon$  and  $d = 0$ . Examining step II.C.2 shows that  $([0, \epsilon, 0, 0, c, s, v], j)$  is added to  $H$ . Lemma 5.3.1 and Lemma 5.4.5 show that  $[0, v, i, c', d', s, w]$  is added to  $I_j$  where  $c' + d' = c$ .

For the second induction step,  $M = 0$ ,  $L = N$ , and the theorem is assumed to hold for  $L + M < N$ . Since  $M = 0$ , the theorem can be written as follows:

If the explicitly-advancing  $\text{LR}(k)$  parser can make the sequence of moves

$$(0, \epsilon, yy') \vdash^{a^*} (0, ?, \dots) \vdash^N (\alpha \backslash r, v, y') \vdash (\alpha rs, ?, \dots) \vdash^{a^*} (\alpha rs, w, \epsilon)$$

where  $|v| = k$ ,  $|y'| = k$  and there exist edit sequences  $S$  and  $S'$  such that  $z_{1..} \xrightarrow{S} y$ ,  $z_{i+1..j} \xrightarrow{S'} y'$ ,  $W(S) = c$ ,  $W(S') = d$  and

$$c + d = \min \left( \left\{ W(T) \mid z_{1..j} \xrightarrow{T} t \text{ and } (0, \epsilon, t) \vdash^* (\alpha rs, w, \epsilon) \right\} \right)$$

then  $[r, v, i, c', d', s, w]$  is added to  $I_j$  where  $c' + d' = c + d$ .

If  $N = 0$  then this induction step degenerates to the first induction step. Therefore, only the case of  $N > 0$  needs to be considered. Since the length of the sequence of moves

$$(0, \epsilon, yy') \vdash^{a^*} (0, ?, \dots) \vdash^{N-1} (\alpha, ?, \dots) \vdash^{a^*} (\alpha, ?, \dots) \vdash (\alpha r, ?, \dots) \vdash^{a^*} (\alpha r, v, y')$$

is less than  $N$ , the induction hypothesis and Lemma 5.4.9 can be applied to show that there must be an entry  $[?, ?, ?, a, b, r, v]$  on  $I_i$  where  $c = a + b = W(S)$ .

When  $([?, ?, ?, a, b, r, v], i)$  is processed by step II.C, Algorithm 5.1.1 should simulate the move

$$(\alpha \backslash r, v, y') \vdash (\alpha rs, ?, \dots)$$

which can be made by the explicitly-advancing  $\text{LR}(k)$  parser. This move can be one of two types of moves:

- a shift (**shift**  $\in f_r(v)$ ), or
- a reduction by an empty production (**reduce**  $p \in f_r(v)$  and  $|\text{RHS}(p)| = 0$ ).

For either of the two types of moves,  $s \in g_r(v)$ .

For a shift,  $\text{shift} \in f_r(v)$ . Therefore,  $w = v_{2:k}y'$  and  $|y'| = 1$ . Examining step II.C.2.i shows that  $([r, v, i, c, 0, s, v_{2:k}], i)$  is added to  $H$ . Applying Lemma 5.4.7 shows that  $([r, v, i, c, d, s, w], j)$  is added to  $H$ . Lemma 5.3.1 and Lemma 5.4.5 show that  $[r, v, i, c', d', s, w]$  is added to  $I_j$  where  $c' + d' = c + d$ .

For a reduction by an empty production,  $\text{reduce } p \in f_r(v)$ ,  $|\text{RHS}(p)| = 0$  and  $j = i$ . Therefore,  $w = v$ ,  $y' = \epsilon$  and  $d = 0$ . Examining step II.C.1.ii and step II.C.2 shows that  $([r, v, i, c, 0, s, w], j)$  is added to  $H$ . Lemma 5.3.1 and Lemma 5.4.5 show that  $[r, v, i, c', d', s, w]$  is added to  $I_j$  where  $c' + d' = c$ .

For the third induction step,  $M > 0$ ,  $M + L = N$ , and the theorem is assumed to hold for  $M + L < N$ . Since  $M > 0$ , the theorem can be written as follows:

If the explicitly-advancing  $\text{LR}(k)$  parser can make the sequence of moves

$$(0, \epsilon, yy') \vdash^{a^*} (0, ?, \dots) \vdash^L (\alpha \setminus r, v, y') \vdash^{M+1} (\alpha rs, w, \epsilon)$$

where  $L \geq 0$ ,  $M > 0$ ,  $|v| = k$ ,  $|w| = k$ , and there exist edit sequences

$S$  and  $S'$  such that  $z_{1:n} \xrightarrow{S} y$ ,  $z_{i+1:j} \xrightarrow{S'} y'$ ,  $W(S) = c$ ,  $W(S') = d$ , and

$$c + d = \min \left( \left\{ W(T) \mid z_{1:j} \xrightarrow{T} t \text{ and } (0, \epsilon, t) \vdash^* (\alpha rs, w, \epsilon) \right\} \right)$$

then  $[r, v, i, c', d', s, w]$  is added to  $I_j$  where  $c' + d' = c + d$ .

Since a move of the explicitly-advancing  $\text{LR}(k)$  parser can add at most one symbol to the stack and  $r$  is not allowed to be popped from the stack by any of the  $M + 1$  moves in the sequence of moves

$$(\alpha \setminus r, v, y') \vdash^{M+1} (\alpha rs, w, \epsilon),$$

this sequence of moves can be written as

$$\begin{aligned} (\alpha \setminus r, v, y'_1 y'_2 \dots y'_m) &\vdash^+ (\alpha r \setminus q_1, u_1, y'_2 y'_3 \dots y'_m) \\ &\vdash^+ (\alpha r q_1 \setminus q_2, u_2, y'_3 y'_4 \dots y'_m) \\ &\vdash^+ \vdots \\ &\vdash^+ (\alpha r q_1 q_2 \dots q_{m-2} \setminus q_{m-1}, u_{m-1}, y'_m) \\ &\vdash^+ (\alpha r q_1 q_2 \dots q_{m-1} \setminus q_m, u_m, \epsilon) \\ &\vdash^+ (\alpha rs, w, \epsilon) \end{aligned}$$

where  $1 \leq m \leq M$  and  $|u_x| = k$  for  $1 \leq x \leq m$ . This implies that  $u_m = w$ ,

$y' = y'_1 y'_2 \dots y'_m$  and  $S' = S'_1 S'_2 \dots S'_m$  such that, for  $1 \leq x \leq m$ ,  $S'_x$  is an edit sequence for which  $z_{l_{x-1}+1:l_x} \xrightarrow{S'_x} y'_x$  where  $l_0 = i$ ,  $l_m = j$  and  $l_{x-1} < l_x$  for  $1 \leq x \leq m$ .

Let  $b_x = W(S'_x)$ ,  $a_1 = c$  and  $a_{x+1} = W(S'_1 S'_2 \dots S'_x)$ . Then

$$a_x = c + \sum_{i=1}^{x-1} b_i$$

and

$$\sum_{i=1}^m b_i = d.$$

Applying Lemma 5.4.9,

$$a_x + b_x = \min \left( \left\{ W(T) \mid z_{1:l_x} \xrightarrow{T} t \text{ and } (0, \epsilon, t) \vdash^* (\alpha r q_1 q_2 \dots q_x, u_x, \epsilon) \right\} \right)$$

The reduction

$$(\alpha r q_1 q_2 \dots q_m, w, \epsilon) \vdash (\alpha r s, w, \epsilon)$$

must be a reduction that pops  $m$  states off the stack or, if  $\alpha = \epsilon$  and  $r = 0$ , a reduction which possibly pops more than  $m$  states and underflows the stack. Also, the reduction implies that there exists  $s \in g_r(|\text{LHS}(p)|)$  and **reduce**  $p \in f_{q_m}(w)$ .

Any proper subsequence of the sequence of moves  $(\alpha r, v, y') \vdash^+ (\alpha r s, w, \epsilon)$  has length less than  $N$  so, applying the induction hypothesis, there must be

$$\begin{aligned} & [r, v, i, a'_1, b'_1, q_1, u_1] \text{ on } I_{l_1} \\ & [q_1, u_1, l_1, a'_2, b'_2, q_2, u_2] \text{ on } I_{l_2} \\ & \vdots \\ & [q_{m-1}, u_{m-1}, l_{m-1}, a'_m, b'_m, q_m, u_m] \text{ on } I_{l_m} \end{aligned}$$

where  $a'_x + b'_x = a_x + b_x$  for  $1 \leq x \leq m$ . Applying Theorem 5.3.1,  $b'_x = W(S''_x)$  where  $S''_x$  is an edit sequence for which  $z_{l_{x-1}+1:l_x} \xrightarrow{S''_x} y''_x$  and

$$(\beta r q_1 q_2 \dots q_{x-1}, u_{x-1}, y''_x) \vdash^+ (\beta r q_1 q_2 \dots q_x, u_x, \epsilon).$$

Furthermore,  $a'_x = W(T'_x)$  where  $T'_x$  is an edit sequence for which  $z_{1:l_{x-1}} \xrightarrow{T'_x} t'_x$  and

$$(0, \epsilon, t'_x) \vdash^* (\beta r q_1 q_2 \dots q_{x-1}, u_{x-1}, \epsilon).$$

Therefore,  $b'_x = b_x$  and  $a_x = a'_x$  since if  $b'_x \neq b_x$  then either  $a_x + b'_x$  or  $a'_x + b_x$  is less than  $a_x + b_x$ , which is not possible.

The entries  $[q_{x-1}, u_{x-1}, l_{x-1}, a_x, b_x, q_x, u_x]$  on  $I_{l_x}$  may be processed by step II.C in any order so let  $[q_{h-1}, u_{h-1}, l_{h-1}, a'_h, b'_h, q_h, u_h]$  on  $I_{l_h}$  be the last entry processed by step II.C. If  $h = m$ , examining step II.C.1.ii shows that

$$(|\text{RHS}(p)| - (m - h), 0, p, w, j) \in G_{l_h}(q_h, u_h)$$

when the entry  $[q_{h-1}, u_{h-1}, l_{h-1}, a_h, b_h, q_h, u_h]$  on  $I_{l_h}$  is processed by step II.C.2.iii. If  $h < m$ , applying Lemma 5.4.10 for the entries

$$\begin{aligned} & [q_h, u_h, l_h, a_{h+1}, b_{h+1}, q_{h+1}, u_{h+1}] \text{ on } I_{l_{h+1}} \\ & [q_{h+1}, u_{h+1}, l_{h+1}, a_{h+2}, b_{h+2}, q_{h+2}, u_{h+2}] \text{ on } I_{l_{h+2}} \\ & \vdots \\ & [q_{m-1}, u_{m-1}, l_{m-1}, a_m, b_m, q_m, u_m] \text{ on } I_{l_m} \end{aligned}$$

shows that

$$(|\text{RHS}(p)| - (m - h), \sum_{x=h+1}^m b_x, p, w, j) \in G_{l_h}(q_h, u_h).$$

when the entry  $[q_{h-1}, u_{h-1}, l_{h-1}, a_h, b_h, q_h, u_h]$  on  $I_{l_h}$  is processed by step II.C.2.iii. Applying Lemma 5.4.2 to  $[r, v, i, a_1, b_1, q_1, u_1]$  on  $I_{l_1}$  shows that there is an entry  $[?, ?, ?, a_0, b_0, r, v]$  on  $I_i$  for which  $a_1 = a_0 + b_0$ . Furthermore, if  $m > |\text{RHS}(p)|$  then  $[?, ?, ?, a_0, b_0, r, v] = [0, \epsilon, 0, 0, b_0, 0, v]$ . Applying Lemma 5.4.11 to

$$\begin{aligned} & [?, ?, ?, a_0, b_0, r, v] \text{ on } I_i \\ & [r, v, i, a_1, b_1, q_1, u_1] \text{ on } I_{l_1} \\ & [q_1, u_1, l_1, a_2, b_2, q_2, u_2] \text{ on } I_{l_2} \\ & \vdots \\ & [q_{h-1}, u_{h-1}, l_{h-1}, a_h, b_h, q_h, u_h] \text{ on } I_{l_h} \end{aligned}$$

shows that if  $m = |\text{RHS}(p)|$  then

$$(r, v, i, a_1, \sum_{x=1}^h b_x) \in B_{(|\text{RHS}(p)| - (m - h)) - 1}$$

and that if  $m > |\text{RHS}(p)|$  then

$$(0, \epsilon, 0, 0, \sum_{x=0}^h b_x) \in B_{(|\text{RHS}(p)| - (m - h)) - 1}.$$

In either case, step II.C.2.iii of Algorithm 5.1.1 adds  $([r, v, i, c, d, s, w], j)$  to  $H$  and Lemma 5.3.1 and Lemma 5.4.5 show that  $[r, v, i, c', d', s, w]$  is added to  $I_j$  where  $c' + d' = c + d$ . ■

## 5.5 RUN TIME ANALYSIS

The run time analysis in this section is simplified by the fact that Algorithm 5.1.1 is a modified version of Algorithm 4.1.1. The analysis of Algorithm 5.1.1 time and space complexities is performed by comparing the two algorithms and drawing on the results for Algorithm 4.1.1.

Two major results are presented in this section. The first result shows that Algorithm 5.1.1 has  $\mathcal{O}(n^5)$  time complexity and  $\mathcal{O}(n^4)$  space complexity and the second result shows that for LR( $k$ ) grammars and a bounded stack size the time and space complexity is  $\mathcal{O}(n)$ .

### 5.5.1 $\mathcal{O}(n^5)$ Time and $\mathcal{O}(n^4)$ Space Complexities

Since Algorithm 5.1.1 is a modified version of Algorithm 4.1.1, its time and space complexities can be determined from a comparison of the two algorithms. Entries in Algorithm 5.1.1 have two cost components and two lookahead strings that are not present in entries for Algorithm 4.1.1. However, none of the operations affected by this change have their time or space complexity increased over those in Algorithm 4.1.1. This is because the maximum length of the lookahead strings is  $\mathcal{O}(1)$  and nothing is done with the cost components other than adding them in conjunction with other operations on the entries.

Step II.B of Algorithm 5.1.1 is a completely new step but it does not increase the algorithm's time complexity over Algorithm 4.1.1. This is because the function of step II.B is to fill the lookahead string for an entry and there are at most  $\mathcal{O}(1)$  different possible lookaheads. Thus the costs of executing step II.B can be charged to the corresponding entry which was processed by step I or step II.C.1.i.

The space complexity of Algorithm 5.1.1 is increased over Algorithm 4.1.1 by the elimination of duplicate checking for the pending list,  $H$ , in steps II.C.1.i and II.C.2.iii. The pending list may contain as many as  $\mathcal{O}(n^4)$  entries since step II.C.2.iii may add  $\mathcal{O}(n^2)$  entries for each entry it processes. The time complexity is not affected because any particular entry can still only be added  $\mathcal{O}(1)$  times for each entry processed by step II.C.1.i or step II.C.2.iii.

Step II.A of Algorithm 5.1.1 has been changed from Algorithm 4.1.1 so that it removes a least-cost entry. This change could potentially increase the time complexity of the removal or insertion operations but a simple reorganization of the pending list can be used to avoid an increase. The pending list is organized as

an array of  $m + 1$  lists where  $m = \max(\{W((a \mapsto b)) \mid (a \mapsto b) \in \Delta\})$ . When an entry  $([q, u, i, c, d, r, v], j)$  is added to the pending list, it is added to the list indexed by  $(c + d) \bmod (m + 1)$ . Thus, the number of primitive operations for adding an entry to the pending list is still  $\mathcal{O}(1)$ .

For removing entries from the pending list, an auxiliary variable,  $l$ , is used to index the pending list. The index  $l$  is initialized to 0 in step I of the algorithm. When a least-cost entry is to be removed from the pending list, the first entry on the list indexed by  $l$  is removed. If the list indexed by  $l$  is empty then  $l$  is incremented by  $1 \bmod (m + 1)$  and the next list is checked. This process continues until an entry is found or the pending list is determined to be empty. Examination of Algorithm 5.1.1 and the proofs of its correctness show that the entries on the list indexed by  $l$  will always be least-cost entries since  $m = \max(\{W((a \mapsto b)) \mid (a \mapsto b) \in \Delta\})$ . Thus, the number of primitive operations for removing an entry from the pending list is  $\mathcal{O}(1)$ .

The comparison of Algorithm 5.1.1 and Algorithm 4.1.1 shows that the only impact of the changes is to increase the space complexity because the pending list may grow to  $\mathcal{O}(n^4)$  entries.

### 5.5.2 $\mathcal{O}(n)$ Time and Space for $\text{LR}(k)$ Grammars

Since Algorithm 4.1.1 and Algorithm 5.1.1 are both simulating an (explicitly-advancing)  $\text{LR}(k)$  parser, The results for Algorithm 4.1.1 when an  $\text{LR}(k)$  grammar is used carry directly over to Algorithm 5.1.1 when there are no errors in the input string. However, for Algorithm 5.1.1 it is desirable to consider the effect of leaving the test for duplicate entries in step II.C.

Checking for duplicate entries only increases the time complexity if there are  $\mathcal{O}(n)$  entries on a parse list  $I_j$ . An entry  $[q, u, i, c, d, r, v]$  on  $I_j$  implies that the  $\text{LR}(k)$  parser can make the sequence of moves

$$(\alpha l q, w_{i+1, n+k+1}) \vdash^* (\alpha q r, w_{j+1, n+k+1}).$$

Since the  $\text{LR}(k)$  parser is deterministic for an  $\text{LR}(k)$  grammar,  $\mathcal{O}(n)$  entries on a parse list implies that there must be  $\mathcal{O}(n)$  states on the parse stack for the  $\text{LR}(k)$  parser at some point during the parse of the input string.

In actual practice,  $LR(k)$  parsers are mostly used with stacks of bounded size, even when the grammar contains right-recursive productions. Given the constraint of a bounded stack, Algorithm 5.1.1 has  $\mathcal{O}(n)$  time complexity for input strings with no errors even though duplicate checking is left in step II.C.

## 5.6 REGIONALLY LEAST-COST ERROR RECOVERY

This section shows how Algorithm 5.1.1 can be used for regionally least-cost error recovery in  $LR(k)$  parsers. The practical advantage of Algorithm 5.1.1's depth-first search over breadth-first searches is also discussed. The discussion in this section is limited to  $LR(k)$  parsers, but these results can be extended to  $LALR(k)$  or  $SLR(k)$  parsers.

In practice, an  $LR(k)$  parser operates normally until it encounters an error in its input string. The parser then invokes its syntax-error recovery algorithm. Once the parser recovers from the error, it continues parsing from the configuration generated by the syntax-error recovery algorithm. This style of parser operation is the one assumed when Algorithm 5.1.1 is used as a syntax-error recovery algorithm.

For regionally least-cost error recovery, one issue is how far beyond the parser-defined error location to extend the region. One approach is to always use a fixed number of tokens. Another approach is to use Mauney and Fischer's [21] MPLU symbols as markers for the end of the error recovery regions. Algorithm 5.1.1 is flexible enough to be used with either of these approaches.

Another issue that any syntax-error recovery algorithm must address is that the location in the input string at which the parser detects an error may be beyond the actual error location. There are several ways to deal with this issue during error recovery:

- ignore it and only allow changes to the unparsed input;
- retain the previously scanned input and allow it to be changed; and
- allow the stack (as a representation of parsed input) to be changed.

Once again, Algorithm 5.1.1 is flexible enough to be used with any of these approaches.



Assuming for the moment that repairs can only be made to the unparsed input, Algorithm 5.1.1 can be straightforwardly applied to any configuration in which an  $LR(k)$  parser detects an error. Let the configuration be

$$(0q_1q_2 \dots q_m, w_{j+1:j+k}, w_{j+k+1:n+k+1})$$

and let  $l$ , where  $l \geq j + 1$ , be the location of the end of the error recovery region. Then  $m + l - j + 1$  parse lists  $I_0, I_1, \dots, I_{m+l-j}$  are required and the input string for Algorithm 5.1.1 is  $z_{m+1:m+l-j} = w_{j+1:l}$ .

The first  $m + 1$  parse lists are used to record the stack so that reductions can be performed. Each parse list  $I_i$ , for  $0 < i < m$ , is initialized with an entry  $[q_{i-1}, \epsilon, i - 1, 0, 0, q_i, \epsilon]$ . Note that by convention  $q_0 = 0$ . Parse list  $I_0$  is initialized with the entry  $[0, \epsilon, 0, 0, 0, 0, \epsilon]$ . The pending list,  $H$ , is initialized with the entry  $[(q_{m-1}, \epsilon, m - 1, 0, 0, q_m, \epsilon), m]$ .

When Algorithm 5.1.1 is used for syntax-error recovery, step II.B needs to be changed so that tokens are not added to the lookahead string for an entry unless the resulting lookahead string is valid for the entry's state. Entries with invalid lookaheads are troublesome during error recovery and their generation can easily be avoided. This change does not affect the correctness of Algorithm 5.1.1.

Once initialized with the starting configuration, Algorithm 5.1.1 is started and allowed to execute normally. Note that it is assumed that the pending list,  $H$ , is processed in first-in last-out order for entries with the same costs. This gives Algorithm 5.1.1 its depth-first search capability.

Step II of Algorithm 5.1.1 needs to be changed to allow for a new termination condition:

If the last entry processed was  $([q, u, h, c, d, r, v], i)$  and  $i = m + l - j$  then halt; otherwise perform the following steps:

This termination condition uses the parse list  $I_{m+l-j}$  as a sentinel to detect the completion of a least-cost edit of the region. If an entry is added to  $I_{m+l-j}$  then the entry corresponds to an edit of the region. Since entries are added to their parse lists in order of increasing cost, the first entry added to  $I_{m+l-j}$  must correspond to a least-cost edit of the region.

If an entry  $[q, u, h, c, d, r, v]$  is added to  $I_{m+l-j}$  and  $v_{|v|:|v|} \neq w_{l:l}$  or  $|v| < k$  then the selection of  $w_{l:l}$  as the end of the region is questionable. However, this presents no difficulties for Algorithm 5.1.1 because additional parse lists can be added to expand the region and the execution of the algorithm can be continued.

Once Algorithm 5.1.1 halts, the edit of the region must be recovered and the parser placed in its new configuration. The recovery of the edit of the region requires that Algorithm 5.1.1 be modified to maintain a pointer in each entry. This pointer is used to point to the predecessor of the entry. Step II.C.1.i and step II.C.2.iii must be modified to properly initialize this pointer. Step II.B must be modified to copy this pointer to all successors of the entries it processes.

Using the entry  $[q, u, h, c, d, r, v]$  on  $I_{m+l-j}$ , the predecessor pointers are followed back to the entry  $[q_{m-2}, \epsilon, m-2, 0, 0, q_{m-1}, \epsilon]$  on  $I_{m-1}$ , reversing the pointers along the way so that they point to each entry's successor. Then the successor pointers are followed forward, simulating the moves of the parser to create the new configuration. The edits can be reconstructed during the simulation by comparing the lookaheads for each entry with the actual input string.

The advantages of the depth-first capabilities of Algorithm 5.1.1 for simple, single token errors can be seen easily from the above description. Suppose the error is a single token error which occurs at the parser defined error location. Algorithm 5.1.1 will attempt single token repairs at the error location. The first single token repair that parses to the end of the region will terminate the algorithm and no time will be wasted on the remaining repairs. If a single token repair can only be parsed part way through the region then Algorithm 5.1.1 will attempt to repair the "second" error as long as the cost of the two repairs does not exceed the cost of trying an alternate single token repair at the original error location. Any time a repair enables Algorithm 5.1.1 to correctly parse the rest of the region, Algorithm 5.1.1 will effectively stop trying alternatives. Thus, Algorithm 5.1.1 attempts to quickly find a least-cost repair for the region and only backtracks to consider alternatives when continuing forward leads to higher-cost repairs.

For more powerful syntax-error recovery, Algorithm 5.1.1 can also be used to alter contents of the stack when repairing the region. The full details for such a use of the algorithm are not presented here, but the general approach is outlined. This approach takes advantage of the fact that each state on the stack corresponds to a symbol in a viable prefix for the grammar and that the stack may be changed as long as it remains a viable prefix. The symbol corresponding to a state  $q$  on the stack is denoted  $\bar{q}$ .

Altering the stack requires the  $LR(k)$  parser to compute and record the following information for each stack position,  $i$ , during normal parsing:

- the number of tokens derived from  $\bar{q}_i$ , denoted  $C(i)$ ;
- the first  $k$  tokens derived from  $\bar{q}_i$ , denoted  $T(i)$ ; and
- the cost to delete the tokens that are derived from  $\bar{q}_i$ , denoted  $D(i)$ .

This information can be easily collected by the parser and should not greatly affect its efficiency.

As before, the starting configuration is

$$(0q_1q_2 \dots q_m, w_{j+1:j+k}, w_{j+k+1:n+k+1}).$$

However, the input string,  $z$ , for Algorithm 5.1.1 is allowed to contain the non-terminal symbols  $\bar{q}_i$ . Actually, each nonterminal  $\bar{q}_i$  which derives  $k$  or less tokens is expanded and replaced in  $z$  by its tokens. The following definitions aid in the initialization of the parse lists for Algorithm 5.1.1:

$$E(i) = \begin{cases} T(i) & \text{if } C(i) \leq k \\ \bar{q}_i & \text{if } C(i) > k \end{cases}$$

$$L(i) = \sum_{x=0}^{i-1} |E(x)|$$

$$M(i) = \sum_{x=0}^{i-1} |T(x)|$$

$$z = E(1)E(2) \dots E(m)$$

$$y = T(1)T(2) \dots T(m)$$

$$h_i = \begin{cases} L(i+1) & \text{if } C(i+1) \leq k \\ L(i) & \text{if } C(i+1) > k \end{cases}$$

$$u_i = y_{M(i)+1:M(i)+k}.$$

Finally, let  $m' = L(m)$  and  $z_{m'+1:m'+l-j} = w_{j+1:l}$ .

Using the input string  $z$ ,  $m' + l - j + 1$  parse lists  $I_0, I_1, \dots, I_{m'+l-j}$  are required. The first  $m' + 1$  parse lists record the stack and allow edits to the stack to be performed. Parse list  $I_{h_0}$  is initialized with the entry  $[0, \epsilon, 0, 0, 0, 0, u_0]$ . Each parse list  $I_{h_i}$  for  $0 < i < m$  is initialized with an entry  $[q_{i-1}u_{i-1}, h_{i-1}, 0, 0, q_i, u_i]$ . All of these entries have their predecessor pointer initialized appropriately since the predecessor pointers will now be followed back to an entry  $[0, \epsilon, 0, 0, ?, 0, ?]$  when Algorithm 5.1.1 halts. As before, this initialization of the parse lists allows the stack to be used for reductions as the unparsed input is processed.

The pending list,  $H$ , is initialized with the entry  $[q_0, \epsilon, 0, 0, 0, q_0, \epsilon]$  along with, for  $0 < i < m + 1$ , the entries  $([q_{i-1}, u_{i-1}, h_{i-1}, 0, 0, q_i, \epsilon], L(i))$ . Again, all of these entries have their predecessor pointer initialized. These entries allow edits to be made at any point in the stack.

Algorithm 5.1.1 must be modified to deal with the nonterminal symbols present in  $z$ . This also forces the algorithm to incorporate nonterminal symbols into its lookahead strings. In general, whenever a lookahead string contains a nonterminal symbol,  $\bar{q}_i$ , the  $k$  symbols in  $T(i)$  are substituted for the nonterminal symbol when the lookahead is used. If  $\bar{q}_i$  is not the first symbol in a lookahead string, the extra symbols added by  $T(i)$  are ignored. Note that the treatment of nonterminals in lookahead strings has the effect of only allowing one nonterminal in a lookahead string and this nonterminal is always the last symbol in the lookahead string.

Steps II.B.1 thru II.B.4 of Algorithm 5.1.1 must be changed to make sure that they are only applied to terminal symbols when the lookahead string does not contain nonterminal symbols. Two new steps need to be added to step B — one step for deleting nonterminals; and one step for “scanning” a nonterminal. Note that replacing nonterminals is not supported and that inserting before a nonterminal is the same operation as inserting before a terminal.

The new step for deleting a nonterminal is step II.B.5 and is written as follows, assuming that an entry  $([q, u, h, c, d, r, v], l)$  is being processed:

5. If  $z_{j+1:j+1} = \bar{q}_i$ , for any  $i$  such that  $0 < i \leq m$ , and  $v \in \Sigma^*$  then add  $([q, u, h, c, d + D(i), r, v], l + 1)$  to  $H$ .

This step simulates the deletion of a nonterminal, if the lookahead string does not contain any nonterminals. The requirement that the lookahead string not contain any nonterminals prevents the nonterminal from entering the lookahead string before it is deleted.

The new step for scanning a nonterminal is step II.B.6 and is written as follows, assuming that an entry  $([q, u, h, c, d, r, v], l)$  is being processed:

6. If  $z_{j+1:j+1} = \bar{q}_i$ , for  $0 < i \leq m$ , then add  $([q, u, h, c, d, r, v\bar{q}_i], l)$  to  $H$ .

This step simulates the scanning of a nonterminal symbol. Unlike step B.1 which scans a terminal symbol, the entry is not advanced to the next parse list.

Also, step C.1.i of Algorithm 5.1.1 must be modified to correctly “shift” nonterminal symbols. The step is rewritten as follows, assuming that an entry  $([q, u, h, c, d, r, v], l)$  is being processed:

1. If **shift**  $\in f_r(v)$  then perform one of the following steps:
  - a. If  $v_{1:1} \in \Sigma$  and  $s \in g_r(v_{1:1})$  then add  $([r, v, l, c + d, 0, s, v_{2:k}], l)$  to  $H$ .
  - b. If  $v_{1:1} \in N$  and  $s \in g_r(v_{1:1})$  then add  $([r, v, l, c + d, 0, s, \epsilon], l + 1)$  to  $H$ .

With all of these modifications, Algorithm 5.1.1 can edit both the stack and the unparsed input in the region. The termination condition for Algorithm 5.1.1 remains the same. Finally, the reconstruction of the edits and the final configuration follow the same procedure as outlined previously.

## CHAPTER VI

## CONCLUSION

## 6.1 SUMMARY

This dissertation develops a regionally least-cost error recovery scheme with a worst-case time complexity comparable to other error recovery schemes, such as Burke and Fisher's [6], and Pennello and DeRemer's [24], which are limited to less powerful repairs. This regionally least-cost error recovery scheme improves upon Mauney's [20] scheme through its ability to perform a depth-first search of the error's region and thus offers a greatly increased efficiency for simple single token repairs.

During the course of the development of this improved regionally least-cost error recovery scheme, three major algorithms were developed. The first is the  $LR(k)$  Early's Algorithm. This algorithm is essentially Early's algorithm changed to use the  $LR(k)$  states instead of  $LR(k)$  items. The  $LR(k)$  Early's Algorithm is important in its own right since in practice use of the  $LR(k)$  states reduces the number of entries that must be allocated and manipulated. However, the worst-case time complexity of the algorithm is  $\mathcal{O}(n^4)$  as opposed to  $\mathcal{O}(n^3)$  for Early's algorithm.

The second algorithm developed is the Depth-First  $LR(k)$  Early's Algorithm. This algorithm also uses the  $LR(k)$  states but it allows entries to be added to their parse lists in any order (including depth-first) consistent with the parses of the input string. In general, the worst-case time complexity of the algorithm is  $\mathcal{O}(n^5)$ , but it is only  $\mathcal{O}(n)$  for  $LR(k)$  grammars and a bounded stack. The Depth-First  $LR(k)$  Early's Algorithm is the only general parsing method known to the author that can perform in a depth-first manner with a worst-case time complexity less than  $\mathcal{O}(c^n)$ .

The third algorithm is the Least-Cost LR( $k$ ) Early's Algorithm. The algorithm extends the Depth-First LR( $k$ ) Early's Algorithm so that it performs the least-cost edit of its input string. This extension maintains the same worst-case time complexity. The Least-Cost LR( $k$ ) Early's Algorithm is the only globally least-cost algorithm known to the author which can be shown to have a time complexity of  $\mathcal{O}(n)$  for a syntactically correct input string for an LR( $k$ ) grammar.

## 6.2 FUTURE WORK

The development of the Least-Cost LR( $k$ ) Early's Algorithm has opened the door to many additional inquiries. In addition to the theoretical time complexity results presented here, the algorithm's performance in actual practice should be measured against a suitable benchmark. The actual quality of error recovery should also be studied.

More theoretical results for the worst-case time complexity of the algorithm would be useful. For example, it is desirable to know the worst-case time complexities for erroneous strings when the grammar used is left-linear or right-linear, regular, or LR( $k$ ).

Finally, the algorithms developed in this dissertation of syntax-error recovery may offer improvements in other areas where Early's algorithm has been used, such as parsing extensible languages and pattern matching.

## BIBLIOGRAPHY

- [1] Aho, A.V. and Ullman, J.D. *The Theory of Parsing, Translation, and Compiling Volume I: Parsing*. Prentice-Hall, Englewood Cliffs, New Jersey (1972).
- [2] Aho, A.V. and Peterson, T.G. A Minimum Distance Error-Correcting Parser for Context-Free Languages. *SIAM J. Comput.* 1, 4 (Dec 1972), 305-312.
- [3] Aho, A.V., Hopcroft J.E., and Ullman J.D. *Data Structures and Algorithms*. Addison-Wesley, Reading, Massachusetts (1983).
- [4] Aho, A.V., Sethi, R., and Ullman, J.D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts (1986).
- [5] Burke M.G. and Fisher G.A. A Practical Method for LR and LL Syntactic Error Diagnosis. *ACM Transactions on Programming Languages and Systems* 9, 2 (Apr. 1987), 164-197.
- [6] Burke M.G. *A Practical Method For LR and LL Syntactic Error Diagnosis and Recovery*. Ph.D. Thesis, New York University (1983).
- [7] Ciesinger, J. A Bibliography of Error-Handling. *ACM SIGPLAN Notices* 14, 1 (Jan. 1979), 16-26.
- [8] Dehnert, J.C. *The Analysis of Errors in Context Free Languages*. Ph.D. Thesis, University of California - Berkeley (1983).
- [9] Dion, B.A. *Locally Least-Cost Error Correctors for Context-Free and Context-Sensitive Parsers*. UMI Research Press, Ann Arbor, Michigan (1982).
- [10] Druseikis, F.C. and Ripley, D.G. Error Recovery for Simple LR(*k*) Parsers. *Proceedings of the Annual Conference of the ACM*, (1976), 369-400.
- [11] Earley, J. An Efficient Context-Free Parsing Algorithm. *Communications of the ACM* 13, 2 (Feb. 1970), 94-102.
- [12] Feyock, S. and Lazarus, P. Syntax-directed Correction of Syntax Errors. *Software-Practice and Experience* 6, 2 (1976), 207-219.



- [13] Graham, S.L. and Rhodes S.P. Practical Syntactic Error Recovery. *Communications of the ACM* 18, 11 (Nov. 1975), 639-650.
- [14] Graham, S.L., Haley, C.B. and Joy W.N., Practical LR Error Recovery. *ACM SIGPLAN Notices*, 14, 8 (Aug. 1979), 168-175.
- [15] Graham, S.L., Harrison, M.A., and Ruzzo W.L. An Improved Context-Free Recognizer. *ACM Transactions on Programming Languages and Systems* 2, 3 (July 1980), 415-462.
- [16] Hammond, K. and Rayward-Smith, V.J. A Survey on Syntactic Error Recovery and Repair. *Computer Languages* 9, 1 (1984), 51-67.
- [17] Johnson, S.C. *YACC — Yet Another Compiler Compiler*. Bell Laboratories, Murray Hill, 1977.
- [18] Leinius, R.P. *Error Detection and Recovery for Syntax Directed Compiler Systems*. Ph.D. Thesis, The University of Wisconsin (1970).
- [19] Lyon G. Syntax-Directed Least-Errors Analysis for Context-Free Languages: A Practical Approach. *Communications of the ACM* 17, 1 (Jan. 1974), 3-14.
- [20] Mauney, J. *Least-Cost Syntactic Error Repair Using Extended Right Context*. Ph.D. Thesis, University of Wisconsin - Madison (1983).
- [21] Mauney, J. and Fischer C. N. Determining the Extent of Lookahead in Syntactic Error Repair. *Transactions on Programming Languages and Systems* 10, 3 (Jul. 1988), 456-469.
- [22] Michunas, M.D. and Modry, J.A. Automatic Error Recovery for LR Parsers. *Communications of the ACM* 21, 6 (June 1978), 459-465.
- [23] Pai, A.B. and Kieburtz, R.B. Global Context Recovery: A New Strategy for Syntactic Error Recovery by Table-Driven Parsers. *ACM Transactions on Programming Languages Systems* 2, 1 (Jan. 1980), 18-41.
- [24] Pennello, T.J. and DeRemer F. A Forward Move Algorithm for LR Error Recovery. *5th ACM Symposium on Principles of Programming Languages* (1978), 241-255.
- [25] Peterson, T.G. *Syntax Error Detection, Correction and Recovery in Parsers*. Ph.D. Thesis, Stevens Institute of Technology (1972).

- [26] Ripley, G.D. and Druseikis, F.C. A Statistical Analysis of Syntax Errors. *Journal of Computer Languages* 3, 4 (1978), 227-240.
- [27] Sippu, S. and Soisalon-Soininen, E. A Scheme for LR( $k$ ) Parsing with Error Recovery. Part I: LR( $k$ ) Parsing. *Intern. J. Computer Math.* 8, (1980) 27-42.
- [28] Sippu, S. and Soisalon-Soininen, E. A Syntax-Error-Handling Technique and Its Experimental Analysis. *ACM Transactions on Programming Languages and Systems* 5, 4 (Oct. 1983), 656-679.
- [29] Tai, K.C. Syntactic Error Correction in Programming Languages. *IEEE Transactions on Software Engineering* 4, 5 (Sep. 1978), 414-425.
- [30] Wagner, R.A. and Seiferas, J.I. Correcting Counter-Automaton-Recognizable Languages. *SIAM Journal of Computing* 7, 3 (Aug. 1978), 357-375.
- [31] Wirth, N. *Algorithms + Data Structures = Programs*. Prentice-Hall, Englewood Cliffs, New Jersey (1976).

2  
VITA

Martin L. Traviolia

Candidate for the Degree of

Doctor of Philosophy

Thesis: IMPROVED REGIONALLY LEAST-COST SYNTAX ERROR  
RECOVERY

Major Field: Computer Science

Biographical:

Personal Data: Born in Torrance, California, January 28, 1958, the son of  
Marion R. and A. Rosemary Traviolia.

Education: Graduated from Charles C. Mason High School, Tulsa, Oklahoma, in June 1976; received Bachelor of Science degree in Mathematics and Computer Science from Oral Roberts University in May, 1980; received Master of Science degree in Mathematics from the University of Tulsa in December 1985; completed requirements for the Doctor of Philosophy degree at Oklahoma State University in December, 1991.

Professional Experience: Systems Engineer for The Williams Companies since June, 1986.