MEASURING AND EXTENDING LR(1) PARSER GENERATION

A DISSERTATION SUBMITTED TO THE GRADUATE DIVISION OF THE
UNIVERSITY OF HAWAI‘I IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

COMPUTER SCIENCE

AUGUST 2009

By
Xin Chen

Dissertation Committee:

David Pager, Chairperson
YingFei Dong
David Chin
David Streveler
Scott Robertson

We certify that we have read this dissertation and that, in our opinion, it is satisfactory in scope and quality as a dissertation for the degree of Doctor of Philosophy in Computer Science.

DISSERTATION COMMITTEE

_____
Chairperson

_____

_____

_____

_____

To my family

Acknowledgements

I wish to thank my family for their love and support.

I thank my advisor, professor David Pager, for his guidance and supervision. Without him this work would not be possible. Every time when I was stuck at an issue, it was the discussion with him that could lead me through the difficulty.

I thank my PhD committee members for giving me feedback on my work, as well as occasional talks and suggestions. I would regret the unfortunate pass away of professor Art Lew and miss his passion and dedication on research.

I would like to thank many people in the compiler and parser generator field that I have communicated to in person, by email or via online discussion. From them I received invaluable suggestions and have learned a lot: Francois Pottier, Akim Demaille, Paul Hilfinger, Joel Denny, Paul Mann, Chris Clark, Hans Aberg, Hans-Peter Diettrich, Terence Parr, Sean O'Connor, Vladimir Makarov, Aho Alfred, Heng Yuan, Felipe Angriman, Pete Jinks and more.

I would like to thank my GA supervisor Shi-Jen He for allowing me more freedom to concentrate on research in the final phase.

Thanks also go to people not mentioned here but who have nevertheless supported my work in one way or another.

ABSTRACT

Commonly used parser generation algorithms such as LALR, LL and SLR all have their restrictions. The canonical LR(1) algorithm proposed by Knuth in 1965 is regarded as the most powerful parser generation algorithm for context-free languages, but is very expensive in time and space costs and has long been considered as impractical by the community. There have been LR(1) algorithms that can improve the time and space efficiency of the canonical LR(1) algorithm, but there had been no systematic study of them. Good LR(1) parser generator implementation is also rare. LR(k) parser generation is even more expensive and complicated than LR(1), but it can be an alternative to GLR in natural language processing and other applications.

To solve these problems, this work explored ways of improving data structure and algorithms, and implemented an efficient, practical and Yacc-compatible LR(0)/LALR(1)/LR(1)/LR(k) parser generator Hyacc, which has been released to the open source community. An empirical study was conducted comparing different LR(1) parser generation algorithms and with LALR(1) algorithms. The result shows that LR(1) parser generation based upon improved algorithms and carefully selected data structures can be sufficiently efficient to be of practical use with modern computing facilities. An extension was made to the unit production elimination algorithm to remove redundant states. A LALR(1) implementation based on the first phase of lane-tracing was done, which is another alternative of LALR(1) algorithm. The second phase of lane-tracing algorithm, which has not been discussed in detail before, was analyzed and implemented. A new LR(k) algorithm called the edge-pushing algorithm, which is based on recursively applying the lane-tracing process, was designed and implemented. Finally, a latex2gDPS compiler was created using Hyacc to demonstrate its usage.

Contents

Acknowledgements **v**

Abstract **vi**

List of Tables **xii**

List of Figures **xiv**

1 Introduction **1**

2 Background and Related Work **4**

2.1 Parsing Theory 4

2.1.1 History of Research on Parsing Algorithms 4

2.1.2 Classification of Parsing Algorithms 5

2.2 LR(1) Parsing Theory 8

2.2.1 The Canonical LR(k) Algorithm of Knuth (1965) 8

2.2.2 The Partitioning Algorithm of Korenjak (1969) 8

2.2.3 The Lane-tracing Algorithm of Pager (1977) 9

2.2.4 The Practical General Method of Pager (1977) 9

2.2.5 The Splitting Algorithm of Spector (1981, 1988) 10

2.2.6 The Honalee Algorithm of Tribble (2002) 10

2.2.7 Other LR(1) Algorithms 11

2.3 LR(1) Parser Generators 11

2.3.1 LR(1) Parser Generators Based on the Practical General Method 11
2.3.2 LR(1) Parser Generators Based on the Lane-Tracing Algorithm 12
2.3.3 LR(1) Parser Generators Based on Spector's Splitting Algorithm 13
2.3.4 Other LR(1) Parser Generators 13
2.4 The Need For Revisiting LR(1) Parser Generation 14
2.4.1 The Problems of Other Parsing Algorithms 14
2.4.2 The Obsolete Misconception of LR(1) versus LALR(1) 14
2.4.3 The Current Status of LR(1) Parser Generators 15
2.5 LR(k) Parsing 15
2.6 Conclusion 17

3 The Hyacc Parser Generator 18
3.1 Overview 18
3.2 Architecture of the Hyacc parser generator 21
3.3 Architecture of the LR(1) Parse Engine 23
3.3.1 Architecture 23
3.3.2 Storing the Parsing Table 25
3.3.3 Handling Precedence and Associativity 33
3.3.4 Error Handling 35
3.4 Data Structures 35

4 LR(1) Parser Generation 38
4.1 Overview 38
4.2 Knuth's Canonical Algorithm 39
4.2.1 The Algorithm 39
4.2.2 Implementation Issues 41
4.3 Pager's Practical General Method 50
4.3.1 The Algorithm 50
4.3.2 Implementation Issues 52

4.4 Pager's Unit Production Elimination Algorithm 56
4.4.1 The Algorithm 56
4.4.2 Implementation Issues 59
4.5 Extension To The Unit Production Elimination Algorithm 66
4.5.1 Introduction and the Algorithm 66
4.5.2 Implementation Issues 71
4.6 Pager's Lane-tracing Algorithm 72
4.6.1 The Algorithm 72
4.6.2 Lane-tracing Phase 1 73
4.6.3 Lane-tracing Phase 2 76
4.6.4 Lane-tracing Phase 2 First Step: Get Lanehead State List 76
4.6.5 Lane-tracing Phase 2 Based on PGM 79
4.6.6 Lane-tracing Phase 2 Based on A Lane-tracing Table 83
4.7 Framework of Reduced-Space LR(1) Parser Generation 94
4.8 Conclusion 95

5 Measurements and Evaluations of LR(1) Parser Generation 97
5.1 About the Measurement 97
5.1.1 The Environment and Metrics Collection 97
5.1.2 The Algorithms 98
5.1.3 The Grammars 99
5.2 LR(1), LALR(1) and LR(0) Algorithms 100
5.2.1 Parsing Table Size Comparison 100
5.2.2 Parsing Table Conflict Comparison 103
5.2.3 Running Time Comparison 105
5.2.4 Memory Usage Comparison 107
5.3 Extension Algorithm to Unit Production Elimination 109
5.3.1 Parsing Table Size Comparison 109

5.3.2 Parsing Table Conflict Comparison . . . 112
5.3.3 Running Time Comparison . . . 114
5.3.4 Memory Usage Comparison . . . 114
5.4 Comparison with Other Parser Generators . . . 118
5.4.1 Comparison to Dragon and Parsing . . . 118
5.4.2 Comparison to Menhir and MSTA . . . 119
5.5 Conclusion . . . 121
5.5.1 LR(1) and LALR(1) Algorithms . . . 121
5.5.2 The Unit Production Elimination Algorithm and Its Extension Algorithm . 122
5.5.3 Hyacc and Other Parser Generators . . . 122

6 LR(k) Parser Generation 124

6.1 LR(k) Algorithm . . . 125
6.1.1 LR(k) Parser Generation Based on Recursive Lane-tracing . . . 125
6.1.2 Edge-pushing Algorithm: A Conceptual Example . . . 128
6.1.3 The Edge-pushing Algorithm . . . 133
6.1.4 Edge-pushing Algorithm on Cycle Condition . . . 135
6.2 Computation of $theads(\alpha, k)$ . . . 137
6.2.1 The Problem . . . 137
6.2.2 Literature Review on $theads(\alpha, k)$ Calculation . . . 139
6.2.3 The $theads(\alpha, k)$ Algorithm Used in Hyacc . . . 141
6.3 Storage of LR(k) Parsing Table . . . 145
6.4 LR(k) Parse Engine . . . 148
6.5 Performance . . . 149
6.6 Examples . . . 150
6.7 Lane-tracing at Compile Time . . . 164
6.8 More Issues . . . 166

7 The Latex2gDPS compiler 171

7.1 Introduction . . . 171
7.2 Design of the Latex2gDPS Compiler . . . 172
7.2.1 Overall Design . . . 172
7.2.2 Data Structures . . . 173
7.2.3 Use of Special Declarations . . . 174
7.3 Current Status . . . 175
7.4 An Example . . . 177

8 Conclusion **179**

9 Future Work **181**
9.1 Study of More LR(1) Algorithms . . . 181
9.2 Issues in LR(k) Parser Generation . . . 181
9.3 Ambiguity . . . 182
9.4 More Work on Hyacc . . . 182

A Hyacc User Manual **183**

B Simple Grammars Used for Testing **203**

C Latex2gDPS Compiler Grammar **208**

Bibliography **212**

List of Tables

2.1 Classification of Parsing Algorithms 6

3.1 Storage tables for the parsing machine in Hyacc parse engine 27
3.2 Parsing table for grammar G3.1 28
3.3 Storage tables in y.tab.c for grammar G3.1 29

4.1 The ratio $4 * (n/m)^2$ for some grammars 42
4.2 Grammar G4.4: lane table constructed in lane-tracing 86

5.1 Number of terminals, non-terminals and rules in the grammars 99
5.2 Parsing table size comparison 101
5.3 Parsing table conflict comparison 104
5.4 Time performance comparison (sec) 106
5.5 Memory performance comparison (MB) 108
5.6 Parsing table size comparison 110
5.7 Parsing table conflict comparison 113
5.8 Time performance comparison (sec) 115
5.9 Memory usage comparison (MB) 116
5.10 Memory increase percentage of UPE (and UPE Ext) v.s. PGM LR(1) 117
5.11 Percentage of state number change compared to PGM LR(1) 117
5.12 Comparison with other parser generators 118
5.13 Parsing table size comparison 120

5.14 Conflict comparison 120
5.15 Running time comparison 120

6.1 LR(1) storage tables in y.tab.c for grammar G6.2 155
6.2 LR(k) storage tables in y.tab.c for grammar G6.2 156

7.1 DPFE types and their full names 175
7.2 DPFE types, significance, sources and status 176

List of Figures

2.1 Hierarchy of Parsing Algorithms 7

3.1 Overall architecture of the Hyacc parser generator 21
3.2 Relationship of algorithms from the point of view of data flow 22
3.3 Relationship of algorithms from the point of view of implementation 23
3.4 Parsing machine of grammar G3.1 26
3.5 LALR(1) parsing machine of grammar G3.2 31
3.6 LR(1) parsing machine of grammar G3.2 32

4.1 State 0 of the parsing machine of grammar G3.1 42
4.2 Unit Production Elimination on the parsing machine of grammar G3.1 58
4.3 Applying Unit Production Elimination on the parsing table 60
4.4 Assume states T_a and T_b have the same action on token y 61
4.5 Remove same-action states after unit production elimination 69
4.6 Apply UPE and UPE Ext on Grammar G4.2 70
4.7 The Two Phases of Lane-Tracing Algorithm 72
4.8 LR(0) parsing machine for grammar G4.3 74
4.9 Lane tracing on conflict configurations 74
4.10 LALR(1) parsing machine for G4.3 generated by lane tracing 75
4.11 Grammar G4.4: states on the conflicting lanes 85
4.12 Grammar G4.4: conflicting lanes traced in lane-tracing 86
4.13 The approaches to LR(1) parsing machine 96

5.1 Parsing Table Size Comparison 102
5.2 Running Time Comparison 106
5.3 Memory Usage Comparison 108
5.4 Parsing Table Size Comparison 111
5.5 Parsing Table Size Change Percentage 111
5.6 Running Time Comparison 115
5.7 Memory Usage Comparison 116

6.1 LR(k) lane-tracing: joint and cycle conditions 136
6.2 The need of getting more context for increasing k in LR(k) lane-tracing 137
6.3 Parsing machine of grammar G6.2 151
6.4 Parsing machine of grammar G6.2 - the part relevant to lane-tracing 151
6.5 Parsing machine of grammar G6.3 - the part relevant to lane-tracing 157
6.6 Parsing machine of grammar G6.4 - the part relevant to lane-tracing 160
6.7 LR(2) part of the LR(1) parsing machine for grammar G6.7 164
6.8 Parsing machine of the LR(2) grammar for Yacc 168
6.9 The part of the Yacc grammar parsing machine related to shift/reduce conflicts . . . 169
6.10 The part of Chris Clark's grammar's parsing machine related to reduce/reduce conflicts 170

7.1 Architecture of the DP2PN2Solver 171
7.2 Adding the latex2gDPS compiler to the architecture of the DP2PN2Solver 172
7.3 DPFE Type API of the latex2DPS compiler 173

Chapter 1

Introduction

Compiler theory and practice are among the fundamental and core research topics of computer science. The entire industry of computer science and engineering is based upon the capability of translating from human-understandable high-level programming languages into low-level, machine-executable instructions. This process is made possible by compilers. The theory and practice of compilers are related closely to computational complexity, automata theory, software engineering, computer architecture and operating systems. The writing of compilers used to be considered one of the most daunting programming tasks. Fortunately, this has been much simplified by the use of compiler/parser generation tools.

For 40 years it has been believed that the original canonical LR(1) algorithm proposed by Knuth in 1965 [29], although the most powerful parser generation algorithm for context-free languages, was too expensive in time and space costs to be practical. Further research proposed SLR, LALR and LL algorithms that can handle subsets of the LR(k) grammar. The LALR(1) algorithm is considered powerful enough to cover most programming languages and efficient enough in practice. LALR(1) parser generators like Yacc and Bison have been widely embraced by the industry since 1970s. Subsequent research on reduced-space LR(1) algorithms, which reduce the state space and thus improve the performance of canonical LR(1) parser generation, were made mostly by Pager [47] [48] and Spector [57] [58]. Despite its popularity, LALR(1) parsing can't resolve reduce/reduce conflicts and thus causes lots of effort in grammar redesign, and the tweaked grammars may differ from those before modification. Since the 1990s LL parser generators like ANTLR and JavaCC have started to gain popularity, but they cannot handle left-recursive grammars and often require grammar modification as well. Besides, both LALR and LL grammars are just subsets of LR(1)

grammars. In essence, all of SLR, LALR and LL grammars are proper subsets of LR grammars. All languages recognizable by SLR(k), LALR(k), LL(k) and LR(k) grammars can be recognized by corresponding LR(1) grammars. For these reasons, the compiler industry is constantly looking for LR(1) parser generators. Current implementations of LR(1) parser generators are often inefficiently based on Knuth's method, or based on unproved ad hoc methods, or employe the algorithms of Pager and Spector but implemented in relatively unpopular languages, or are proprietary and thus have unknown implementation details, and none is as popular as the LALR(1) parser generators Yacc and Bison. In addition, with all the versions of LR parser construction algorithms that have been developed, no study has been made of their comparative merits.

LR(k) parser generation is even more expensive and complicated than LR(1) parser generation. Although widely studied on the theoretical side, very little practical work has been done due to the performance problem. LR(k) parser generation can serve as an alternative to the more expensive GLR algorithm, and be used in areas such as natural language processing. It would be of value to design and implement a LR(k) algorithm based on reduced-space LR(1) algorithms.

For the reasons given above, this work has developed Hyacc, an efficient, practical and Yacc-compatible open source LR(0)/LALR(1)/LR(1)/LR(k) parser generation tool in C, based on the canonical LR(1) algorithm of Knuth, the general practical LR(1) parsing algorithm, the lane-tracing algorithm, the unit production elimination algorithm of Pager and a new LR(k) algorithm called the edge-pushing algorithm. The following have been achieved:

1) Extended the unit production elimination algorithm of Pager [46] by eliminating redundant states and thus minimizing the parsing machine.

2) Investigated details in the existing LR(1) algorithms, especially the second phase of the lane-tracing algorithm which has not been discussed in much detail before.

3) Compared the performance of LR(1) algorithms (Knuth's canonical algorithm [29], Pager's lane-tracing algorithm [47] and Pager's practical general method [48]) as implemented in Hyacc with the LALR(1) algorithm as implemented in Yacc and Bison with regard to conflict resolution, and time and space requirements. The performance study was conducted on 13 programming languages, including Ada, ALGOL60, COBOL, Pascal, Delphi, C, C++, Java and more. The result demonstrated that a careful LR(1) implementation based upon improved algorithms and carefully selected data structures can be sufficiently efficient in time and space to be of practical use with modern computing facilities.

4) Explored the extension of LR(1) parser generation to LR(k) parser generation using the lane-tracing algorithm. A new LR(k) algorithm called the edge-pushing algorithm was designed and implemented.

5) Developed a latex2gDPS compiler (gDPS stands for general Dynamic Programming Specification, and is the language proposed by Holger [40] to represent dynamic programming problems) using Hyacc.

This dissertation is arranged as follows: Chapter 1 is an overview of the motivation and thesis contribution. Chapter 2 introduces the background and literature research result on LR(1) and LR(k) parser generation theory and practice. Chapter 3 introduces the Hyacc parser generator and its design. Chapter 4 concentrates on LR(1) parser generation, introduces varies LR(1) algorithms, discusses their implementation issues and strategies, possible extensions, and compared different LR(1) algorithms. Chapter 5 concentrates on LR(k) parser generation, introduces the design of the proposed edge-pushing algorithm, and then discusses some unsolved issues. Chapter 6 is on algorithm performance measurement and comparison study. Chapter 7 is on the design and implementation of the latex2gDPS compiler. Chapter 8 concludes the current work. Chapter 9 introduces future work to explore. Finally, the user manuals of the parser generator Hyacc is attached in the end.

The notations used in this dissertation follow from [46][47][48] unless otherwise explained. In short, greek and roman letters are used to specify grammars and algorithms in text and figures. Greek letters such as α, β, γ, ψ, ϕ, ω, ... represent a string of symbols. ϵ represents the empty string. In the context of specifying a grammar, roman letters such as A, B, C, ..., a, b, c, ... represent a single symbol, of these upper case letters represent non-terminal symbols, and lower case letters represent terminal symbols. In the context of specifying algorithms or mathematical formula, roman letters may have other meanings such as a string, a number, a state or a set, and are not limited to terminal or non-terminal symbols. The symbol $\dashv$ means the end of an input stream. $\oslash$ stands for the empty set. The concepts of *state* and *configuration* used in this discussion may be referred to as "item set" and "item" in some other literature. The notation $threads(\alpha, k)$ is equivalent to $FIRST_k(\alpha)$ in other places.

Chapter 2

Background and Related Work

2.1 Parsing Theory

2.1.1 History of Research on Parsing Algorithms

Early syntax parsing endeavors started on arithmetic expressions and employed no formal methods. The programs often embedded grammar rules inside code. It was hard to implement and modify. This was improved after 1952 by left-to-right sequential processing. The first Fortran compiler was developed in 1956. Stack-based algorithms for the parsing of arithmetic expressions and high-level languages in a bottom-up manner were proposed by 1959. In 1957, Chomsky defined the concept of context free grammars and languages, upon which the Backus Naur Form (BNF) was derived. These were later used to describe the design of languages, first for ALGOL60 in 1963. The notions of handle and bounded context grammar were raised in the analysis of bottom up grammars since the early 1960s. In 1965, Knuth generalized these ideas into the computational expensive canonical LR(k) algorithm [29]. A series of algorithms were invented to make LR parsing more practical. These include the divide-and-conquer algorithm of Korenjak in 1969 [31], the SLR and LALR grammars in DeRemer's PhD thesis in the same year [24], and the L(m)R(k) algorithm of Pager in 1970 [43] and 1972 [44]. Actual parser generators were soon implemented using SLR(1) and LALR(1) algorithms. The well-known parser generator Yacc, a LALR(1) parser generator, came to life in 1975 at the AT&T lab and soon was widely accepted by compiler developers. Research on finding general methods to parse LR(1) grammars continued. Such efforts include those of Pager in 1970 [42], Aho and Ullman in 1972 [14] and Joliat in 1973 [28]. The first real breakthrough

was Pager's lane-tracing algorithm [47] and practical general method [48], both were designed for LR(k) grammars and published in 1977. In 1981 Spector reported another splitting method [57], which he refined and published again in 1988 [58]. In 1985, Tomita established the foundation of GLR algorithm [59]. This algorithm was first implemented in the 1992 PhD thesis of Rekers [54]. LL parsing was first studied in 1968 [32][55], and LL parser generators were considered impractical then. This view was changed since the release of parser generator ANTLR in 1992, and today the popular Java parser generator JavaCC also uses the LL(k) algorithm. Most recently, Tribble came up with the Honalee LR(k) algorithm in 2002 [61].

Examples of most recent research on parsing theory include algorithms on non-LR(1) algorithms [27], on natural language parsing and processing, on parsing ambiguity, on combining with different computation models like neural networks [64] and Petri nets [62], and on the development of parser generators for new programming languages and in parallel, quantum [56][26] and DNA [33] computing environments.

2.1.2 Classification of Parsing Algorithms

Common parsing methods can be classified into these categories: recursive descent, operator precedence, SLR, LALR, LL, LR and GLR. Most of the theoretical foundations of these paradigms were laid from the 1960s to 1980s.

As shown in Table 2.1, parsing algorithms for context-free grammars can be classified into top-down and bottom-up techniques. Recursive descent and LL parsers are examples of top-down parsers. Operator precedence, SLR, LALR, LR and GLR parsers are examples of bottom-up parsers. A bottom-up parser is also called a shift-reduce parser because shift and reduce are the two basic actions involved.

Type	Parsing methods	Pros	Cons
Top-down (leftmost derivation)	Recursive descent	Simple	Cannot be used on left-recursive grammars, may need backtracking
	LL	No backtracking needed	Cannot handle left-recursive grammars. LL grammar is also hard to write
Bottom-up (rightmost derivation)	Operator precedence	simple	Covers only a small set of grammars (math expressions), cannot handle operators with more than one precedence
	SLR	Simple	Not powerful enough for most grammars
	LALR	Well balanced efficiency and power	Contains reduce/reduce conflicts
	LR	Most powerful for context-free grammars, is a super set of LL, SLR and LALR grammars, linear in parsing cost	Complex and computationally expensive in parser generation
	GLR	Handles non-deterministic and ambiguous grammars by branching, good for natural language processing	Even more complex and expensive in time and space, cubic in parsing cost on average

Table 2.1: Classification of Parsing Algorithms

Figure 2.1 shows the hierarchy of parsing algorithms (figure adapted from [16]). The set of LR grammars is the superset of all the SLR, LALR and LL grammars, and can cover all unambiguous context-free grammars. LR(k) grammars can actually be converted into LR(1) grammars. GLR is not shown because besides the fundamental characteristic of branching, it can apply different parsing methods in its parse engine, like LR(0), LALR(1) or LR(1), where LR(0) is the common choice when parsing natural languages, and LR(1) is rarely used because of the poor performance.

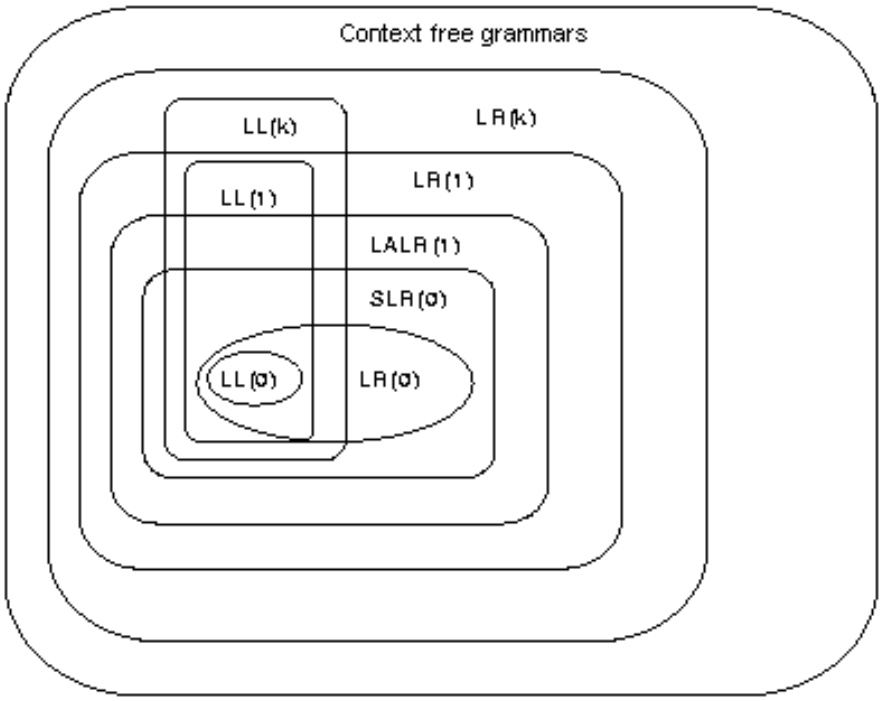


Figure 2.1: Hierarchy of Parsing Algorithms

2.2 LR(1) Parsing Theory

2.2.1 The Canonical LR(k) Algorithm of Knuth (1965)

The algorithm was first described in Knuth's 1965 paper [29]. It is known as the canonical LR(k) parser generation algorithm.

This algorithm was deemed as computationally expensive and not practical at the time when computer memories were small and processors were slow, since the worse case of complexity grows exponentially. Efficiency of LR(k) parser generation has remained a problem for many years, and today it is still a widely accepted conception by people in both industry and academia. Implementation attempts were usually limited to the case of k = 1, which is still quite difficult.

However, because of the theoretical attractiveness of the recognition power of the LR(k) algorithm, many researchers attempted to design algorithms that improve its performance. Many of the research results handle only a subset of the canonical LR(k) grammars. Several attempts, however, indeed decrease the state space of the canonical LR(k) algorithm without compromising its generality, as we soon will see in the discussion below.

2.2.2 The Partitioning Algorithm of Korenjak (1969)

Korenjak's method [30] is to partition a large grammar into small parts, check each part to see whether it is LR(1), generate Knuth canonical LR(1) parsing table, and combine these small tables into a large LR(1) parsing table for the original grammar.

The obvious problem faced by this method is how to partition the grammar. Relevant problems include how many times should partition be used, and what is the general heuristic used to do the partitioning, etc.

It would also be interesting to see if we can apply the reduced-space LR(1) algorithms discussed later in this dissertation on the partitioned parts instead of the Knuth canonical LR(1) algorithm. Korenjak designed the partitioning method before the appearance of such algorithms, so back then it was not possible for him to consider this.

2.2.3 The Lane-tracing Algorithm of Pager (1977)

This is the first practical general method [45] [47] used to create LR(k) parser generator. This algorithm first generates the LR(0) parsing machine, and then proceeds to split inadequate states (those that cause conflicts). The lane tracing algorithm was tried on a series of LR(1), LR(2) and LR(3) grammars without computational difficulty. As an example of application to complex grammars, the lane-tracing algorithm was applied to the ALGOL grammar on a 360/65 and finished the work in less than 0.8 second in 1977 [47].

The lane-tracing algorithm however is difficult to understand and implement. The literature research shows no known public implementation of it today.

2.2.4 The Practical General Method of Pager (1977)

Compared to the lane-tracing algorithm, the practical general method [48] solves the state-space problem of LR(k) parser generation from the other end of the spectrum. Instead of splitting inadequate states, it generates all the states, while merging compatible ones along the way. This keeps low the actual number of states, and thus the need for time and space. The size of the resulting parsing table is similar to that of an LALR parsing table. The merging is based on the concept of weak compatibility defined in the paper. Two states are combined if and only if they are weakly compatible. The concept of strong compatibility is also defined, which can combine more states and result in the most compact parsing machine, but at the cost of much more computation.

This method based on weak compatibility is simpler in concept and easier to implement than the lane-tracing algorithm. Both generate parsing machines of the same size.

There are several known implementations of this algorithm, including LR, LRSYS, LALR, GDT_PC, Menhir and the Parsing module, which we will discuss in section 2.3.

Besides the lane-tracing algorithm and the practical general method, Pager also proposed other supplemental optimization algorithms. Among these is the algorithm of unit production elimination.

2.2.5 The Splitting Algorithm of Spector (1981, 1988)

Spector first proposed his algorithm in 1981 [57], based on splitting the inadequate states of an LR(0) parsing machine. In that sense it is very similar to the Lane-tracing algorithm of Pager, and actually may constitute a part of the Lane-tracing algorithm. He further refined his algorithm in 1988 [58] and implemented his algorithm in a 2300-line C program as a demonstration. However the implementation just aimed as a demonstration, and did not implement other common optimization techniques. His implementation is not known to exist today. He did not have a formal proof of the validity of the algorithm, and only gave some examples in his publications to show how it worked. His papers also lack details on how to handle special situations such as loop conditions, so a real implementation has to rely on the implementer's creativity.

It is known that the Muskox parser generator implemented a version of Spector's algorithm. The author Boris Burshteyn of Muskox said that the 1988 paper of Spector lacked implementation details, so Muskox implemented the algorithm in a modified way according to his understanding of it. Performance-wise, "it takes 400Kbytes for each of the C, F77, and MUSKOX grammars", which seems fairly good. Boris also mentioned that "The only interesting pathological case I know about is a COBOL grammar from PCYACC (TM of ABRAXAS SOFTWARE). There, MUSKOX algorithm reached the limit of 90Mbytes and then stopped since there were no virtual memory left." It would be interesting to see if it would work out on today's computer with much larger memory.

It is interesting how Spector referred to Pager's practical general method as an example of an existing practical LR(k) algorithm in his papers of both 1981 [57] and 1988 [58]. In the 1981 paper he said Pager's algorithm is "almost as efficient as current LALR methods", but in the 1988 paper he commented "Unfortunately, his algorithm is extremely slow". This comment is believed more just as a justification for the proposal of his algorithm, but not the true reality for Pager's algorithm.

2.2.6 The Honalee Algorithm of Tribble (2002)

Tribble proposed his algorithm in 2002 and further refined it until 2006 [61]. It works identically to the practical general method of Pager in concept, merging similar states as new states are generated. Tribble independently derived his algorithm, which he first called the MLR (Merged LR(k)) algorithm, and then called the Honalee LR(k) algorithm.

Tribble later stated that his algorithm was not totally LR(1), just larger than LALR(1). The problem is that the Honalee algorithm avoids merging states if the merging causes immediate reduce/reduce conflict. However it is possible that even though an immediate conflict does not occur, conflicts can occur in successor states later. An example is given by Sylvain Schmitz in [6].

2.2.7 Other LR(1) Algorithms

Besides the above algorithms, there is a general conception of how LR(1) can be achieved by starting from a LR(0) parsing machine and splitting those states that cause conflicts. This in concept is very similar to the lane-tracing algorithm of Pager and the splitting algorithm of Spector.

2.3 LR(1) Parser Generators

2.3.1 LR(1) Parser Generators Based on the Practical General Method

A survey over the Internet shows that there are about 15 LR(1) parser generators. Of these that we are aware of, six are implementations of Pager's algorithm.

The six LR(1) parser generators that implemented Pager's practical general method are: LR, LRSYS, LALR, GDT_PC, Menhir and the Python Parsing module.

The LR program in ANSI standard Fortran 66 was developed in 1979 at the Lawrence Livermore National Laboratory [63]. It implemented Pager's algorithm and can accept all LR(1) grammars. It was ported to more than nine platforms, and was used for developing compilers and system utilities. However, it is rarely used today, and unfamiliar to most people. One reason may be because that it was implemented in a language specifically for science computation, and not in a general-purpose language like Pascal or C. Its rigid and weird input format also limited its popularity. Besides, LR is controlled by the government and is not open source. In addition, the use of LR is not free.

The LRSYS system in Pascal was developed around 1985, also at the Lawrence Livermore National Laboratory [5]. It was based on the LR parser generator. There were versions for CRAY1, DEC VAX 11 and IBM PC. Parser engines in Pascal, FORTRAN 77, and C were provided. The CRAY1 and DEC VAX11 versions also contain engines for LRLTRAN and CFT-FORTRAN 77. The LRSYS program was tested under MS-DOS 3.3 on an 80286, but no performance data is

available. The LRSYS, like LR, sank into the dusty corner of history and became unknown to most people.

Certain source stated that Pager's practical general method was also used in a parser generator named LALR in 1988, implemented in the language MACRO-11 on a RSX-11 machine. This parser generator again is unknown to most people today.

The same source 4 stated that Pager's algorithm was also used in GDT_PC (Grammar Debugging Tool and Parser Constructor) in about 1988. The implementation language is unknown.

The Menhir program in Objective Caml was developed around 2004 in France by academic researchers [53], and the source code is actively maintained. It implemented Pager's algorithm with slight modification. It has since been widely used in the Caml language community, quickly replacing the previous Caml parser generator ocamlyacc. The slight modification to Pager's algorithm is to merge a new state into an existing one if it is a subset of the latter.

The Python Parsing module was developed most recently at the beginning of 2007 [10]. Its author got the idea when developing a language grammar in his work and felt an LALR(1) parser generator could not meet his needs. A wide literature survey led him to Pager's practical general method. This parser generator also implemented the CFSM (Characteristic Finite State Machine) and GLR drivers to handle non-deterministic and ambiguous grammars. It was released as open source software on March 20, 2007 [10]. The author estimated the Python implementation to be about 100 times slower than a C counterpart.

Proprietary implementations of Pager's practical general method may exist. But "proprietary" means that their technical details are hidden from the public.

2.3.2 LR(1) Parser Generators Based on the Lane-Tracing Algorithm

The lane-tracing algorithm was implemented by Pager in the 1970s [47]. According to the description, the performance of the implementation was at the same level as Hyacc implemented in this work. Considering how the hardware was restricted back then, this is quite impressive. However the implementation was done in Assembly for OS 360. This is not portable to other platforms. There is also no known running instance of this algorithm on OS 360 today.

We did not find any other available lane-tracing algorithm implementations.

2.3.3 LR(1) Parser Generators Based on Spector's Splitting Algorithm

Spector himself implemented this in an experimental, incomplete parser generator as described in his 1988 paper [58]. Later, in 1994 the Muskox parser generator 1 implemented a version of Spector's algorithm [18]. The author Boris Burshteyn said that the 1988 paper of Spector lacked implementation details, so Muskox implemented the algorithm in a modified way according to his understanding of it.

We found the splitting algorithm of Spector very similar to the lane-tracing algorithm of Pager in concept. At the same time it lacks enough details and also lacks strict proof as to the validity of the algorithm. For these reasons we did not implement his algorithm.

Although the splitting algorithm of Spector and the lane-tracing algorithms are very close in concept, most people that took the splitting approach to LR(1) parser generation claimed they got the idea from Spector's paper. This possibly is because Spector's paper was published more recently and caught more attention when the advancement in hardware made it possible to seriously consider such an implementation.

2.3.4 Other LR(1) Parser Generators

More efforts were done in this direction. But most of these other approaches are not formally available in literature, are implemented in proprietary products, or sometimes are not fully working as our literature research shows.

LRGen (C) [34] is a long-standing, highly efficient parser generator. Its LR(1) algorithm (2007) seemed to have some minor defects according to the description on the website, and is between LALR(1) and LR(1).

Yacc/M (Java) [60] implemented the MLR algorithm designed by the author (2006). However it seems the algorithm also has defects, and is between LALR(1) and LR(1).

There are several implementations that claim to have efficient LR(1) parser generation.

Yacc++ (C) [12] is a commercial product. It started as a LALR(k) parser generator in 1986, then added LR(1) around 1990 using a splitting approach that loosely based on Spector's algorithm [22] [23].

Dr. Parse (C/C++) [8] is another commercial production that claimed to use LALR(1)/LR(1). But its implementation details are unknown.

MSTA (C/C++) [11], which is a part of the COCOM toolset, took the splitting approach.

2.4 The Need For Revisiting LR(1) Parser Generation

2.4.1 The Problems of Other Parsing Algorithms

These other parsing algorithms include SLR, LALR, LL and GLR. SLR is too restrictive in recognition power. GLR often uses LR(0) or LALR(1) in its engine. GLR branches into multiple stacks for different parse options, eventually disregards the rest and only keeps one, which is very inefficient and is mostly used on natural languages due to its capability in handling ambiguity. LL does not allow left recursion on the input grammar, and tweaking the grammar is often needed. LALR has the "mysterious reduce/reduce conflict" problem and tweaking the grammar is also needed. Despite this, people consider the LALR(1) algorithm the best tradeoff in efficiency and recognition power. Yacc and Bison are popular open source LALR(1) parser generators.

2.4.2 The Obsolete Misconception of LR(1) versus LALR(1)

LR(1) can cover all the SLR, LALR and LL grammars, and is equivalent to LR(k) in the sense that every LR(k) grammar can be converted into a corresponding LR(1) grammar (at the cost of much more complicated structure and much bigger size), so is the most general in recognition power. However, the belief of most people is that an LR(1) parser generation is too slow, takes too much memory, and the generated parsing table is too big, thus impractical performance-wise.

The typical viewpoints on the comparison of LR(1) and LALR(1) algorithms are:

1) Although a subset of LR(1), LALR(1) can cover most programming language grammars.

2) The size of the LALR(1) parsing machine is smaller than the LR(1) parsing machine.

3) Each shift/reduce conflict in a LALR(1) parsing machine also exists in the corresponding LR(1) parsing machine.

4) “mysterious” reduce/reduce conflicts exist in LALR(1) parsing machines but not in LR(1) parsing machines, and “presumably” this can be handled by rewriting the grammar.

However, the LR(1) parser generation algorithm is superior in that the set of LR(1) grammars is a superset of LALR(1) grammars, and the LR(1) algorithm can resolve the “mysterious reduce/reduce conflicts” that cannot be resolved using LALR(1) algorithm. Compiler developers may spend days after days modifying the grammar in order to remove reduce/reduce conflicts without guaranteed success, and the modified grammar may not be the same language as initially desired. Besides, despite the general claim that LR(1) parsing machines are much bigger than LALR(1) parsing machines, the actual fact is that a LR(1) parsing machine is of the same size as a LALR(1) parsing machine for LALR(1) grammars [57][58]. Only for LR(1) grammars that are not LALR(1), are LR(1) parsing machines much bigger. Further, there exist algorithms that can reduce the running time and parsing table size, such as those by Pager and Spector.

2.4.3 The Current Status of LR(1) Parser Generators

As we have seen, there is a scarcity of good LR(1) parser generators, especially with reduced-space algorithms. Many people even have no idea of the existence of such algorithms. It would be of value to provide a practical tool to bring the power of these algorithms to life.

2.5 LR(k) Parsing

Much early theoretical work based their discussions on LR(k). From a theoretical point of view, LR(k) has been widely studied. But such theoretical advantage does not translate into practical success due to the complexity involved, and the time and space costs. The cost of LR(k) parser generation comes from its exponential behavior based on two factors: 1) the number of states in the parsing machine, and 2) the number of context tuples for the configurations.

The 1965 paper of Knuth was about LR(k) parser generation for arbitrary k. After that, a lot of work was done with the aim of reducing the performance cost so as to make it practical.

The work of Pager in the 1970s was about LR(k) parser generation. There have been reports of LR(k) analysis on the grammars of real languages such as ALGOL for LR(2) and LR(3).

M. Ancona et. al. published some papers on LR(k) parser generation from 1980s to 1990s [36][38][39][37][35]. [39] proposed a method in which non-terminals are not expanded to terminals in contexts, and expansion is not done until absolutely needed to resolve inadequacy. This actually defers the calculation of $FIRST_k(\alpha)$ until absolutely necessary. They claim savings in both time and storage space by deploying this method when tried on several programming language grammars. They have worked on a LR(k) parser generator for their research, but no publicly available product was reported.

In 1993, Terence Parr's Phd thesis "Obtaining practical variants of LL(k) and LR(k) for k $>$ 1 by splitting the atomic k-tuple" [52] provided important theoretical implications for working on multiple lookaheads and claimed close-to-linear approximation to the exponential problem. The idea is to break the context k-tuples, which can be applied to both LL(k) and LR(k). Such concept is close to what was in Pager's paper on how to handle LR(k) grammars for k $>$ 2 using the lane-tracing algorithms. Terence's ANTLR LL(k) parser generator was a big success. LL(k) parser generation is considered easier to work with. Theoretically it is also less powerful than LR(k) in recognition power. His PhD thesis argues that adding semantic actions to a LR(k) grammar degrades its recognition power to that of a LL(k) grammar. Based on this assumption he worked on LL(k) parser generation only.

Josef Groelsch worked on a LR(1)/LR(k) parser generator in 1995. In case of LR(1) grammars, it was practical only for small to medium size grammars. LR(k) is certainly more expensive.

Bob Buckley worked on a LR(1) parser generator called Gofer in 1995. He said it was a long way to go from being a production software.

More recently in 2005, Karsten Nyblad claimed to have a plan for an LR(k) implementation. But there was no more news from him.

Chris Clark worked on the LALR(k)/LR(1)/LR(k) parser generator Yacc++. It's LR(k) implementation is loosely based on Spector's paper [22] [23]. But there was an infinite loop problem on the LR(k) of Yacc++. Thus they only used the LR(k) feature internally and did not make it public.

Ralph Boland worked on this, but report on his results was not found.

Paul Mann mentioned that Ron Newman's Dr. Parser works on LR(k) for k = 2 or maybe 3.

It was mentioned that Etienne Gagnon's SableCC parser generator implemented LALR(k) parser generation for k $>$ 1. However checking the SableCC website found that it only claims LALR(1).

Will Donahue and Adrian Johnstone also have worked on LR(k).

The only claimed successful efficient LR(k) parser generator is the MSTA parser generator in the COCOM tool set. The author Vladimir Makarov says it generates fast LALR(k) and LR(k) grammar parsers with "acceptable space requirements". The author was from Russia and his publications on this around 1990s were not available in our literature research.

To conclude, LR(k) parser generation is hard. Most attempts have not turned out well.

2.6 Conclusion

In summary, we can conclude about the state of the art that:

1) Parsing algorithms such as SLR, LALR, LL and GLR all have their limitations compared to LR(1). The major problem of LR(1) algorithm is in its time and space cost.

2) There are always people looking for a LR(1) parser generator. But most often they do not get what they want, either because it is not implemented in the language they desired, or does not use the input format or other features they need, or is proprietary and not everyone wants to pay the asked price, or simply because they cannot find one.

3) The pure canonical Knuth LR(1) parser generation still

4) Information on LR(1) parsing is scarce both in the literature and on the Internet.

5) LR(k) parser generation, although widely studied in theory, is even less practical from a pragmatic point of view. There have been very little work on this.

The LL(k) algorithm was considered impractical in the 1970s and 1980s, but the myth was debunked in the 1990s when LL(k) parser generators like ANTLR and JavaCC were created. Considering all the advantages that LR(1) parsing can provide, we feel it is beneficial to revisit the LR(1) parser generation problem and to provide a practical solution to break the long-held misconception on its impracticality. Better yet, we hope to try LR(k) by extending our LR(1) solution.

Chapter 3

The Hyacc Parser Generator

3.1 Overview

This work has developed Hyacc, an efficient, practical and Yacc/Bison-compatible open source LR(0)/LALR(1)/LR(1)/LR(k) parser generator in ANSI C from scratch.

Hyacc is pronounced as "HiYacc", means Hawaii Yacc.

Hyacc supports these algorithms:

1) The original Knuth LR(1) algorithm (Knuth LR(1))
2) The LR(1) practical general method (weak compatibility) (PGM LR(1))
3) The UPE (unit production elimination) algorithm (UPE)
4) Extension to the UPE algorithm (UPE Ext)
5) LR(0) algorithm
6) LALR(1) based on the first phase of the lane-tracing algorithm (LT LALR(1))
7) The LR(1) lane-tracing algorithm. It contains two phases: phase 1 and phase 2. There are two alternatives for phase 2, one is based on the practical general method (LT LR(1) w/ PGM), the other is based on lane-tracing table (LT LR(1) w/ LTT).
8) The edge-pushing LR(k) algorithm (EP).

Current features include:

1) Implements the original Knuth LR(1) algorithm [29].
2) Implements the practical general method (weak compatibility) [48]. It is a LR(1) algorithm.
3) Removes unit productions [46].
4) Removes repeated states after removing unit productions.
5) Implements the lane-tracing algorithm [45][47]. It is a LR(1) algorithm.
6) Supports LALR(1) based on the lane-tracing algorithm phase 1.
7) Supports LR(0).
8) Experimental LR(k) with the edge-pushing algorithm, which now can accept LR(k) grammars where lane-tracing on increasing k do not involve cycles.
9) Allows empty productions.
10) Allows mid-production actions.
11) Allows these directives: %token, %left, %right, %expect, %start, %prec.
12) In the case of ambiguous grammars, uses precedence and associativity to resolve conflicts. When unavoidable conflicts happen, in the case of shift/reduce conflicts the default action is to use shift, in the case of reduce/reduce conflicts the default is to use the production that appears first in a grammar.
13) Is compatible to yacc and bison in input file format, ambiguous grammar handling, error handling and output file format.
14) Works together with Lex. Or the users can provide the yylex() function themselves.
15) If specified, can generate a graphviz input file for the parsing machine.
16) If specified, the generated compiler can record the parsing steps in a file.
17) Is ANSI C compliant.
18) Rich information in its debug output.

What's not working so far and to be implemented:

1) Hyacc is not reentrant.

2) Hyacc does not support these Yacc directives: %nonassoc, %union, %type.

3) The optimization of removing unit productions can possibly lead to shift/shift conflicts in the case of grammars that are ambiguous or not LR(1), and thus should not be applied in such situation.

4) Full LR(k) where the cycle problem is solved.

Hyacc is ANSI C compliant, which makes it extremely easy to port to other platforms.

All the source files of Hyacc comes under the GPL license. The only exceptions are the LR(1) parse engine file hyaccpar and LR(k) parse engine file hyaccpark, which come under the BSD license. This should guarantee that Hyacc itself is protected by GPL, but the parser generators created by Hyacc can be used in both open source and proprietary software. This addresses the problem that Richard Stallman discussed in "Conditions for Using Bison" of his Bison 1.23 manual and Bison 1.24 manual.

Hyacc version 0.9 has been released to the open source community at sourceforge.net [20] in January 2008, and a notice posted to the comp.compiler news group [19]. So far there are over 400 downloads at sourceforge.net (average one download per day). The version 0.9 contains the Knuth LR(1), PGM LR(1), UPE and UPE Ext algorithms. When ready at a later time, we will release the newest version of Hyacc, which contains a bug fix to version 0.9, new interface features, and new algorithms including LR(0), LALR(1), lane-tracing LR(1) and LR(k).

3.2 Architecture of the Hyacc parser generator

The following are the steps that constitute the architecture of the Hyacc parser generator.

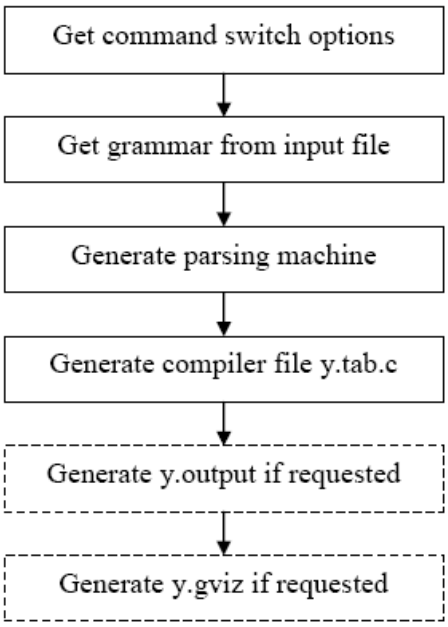


Figure 3.1: Overall architecture of the Hyacc parser generator

These steps are independent of each other.

The y.tab.c file is the parser generator file with the parsing machine stored in arrays.

The y.output file contains all kinds of information needed by the compiler developer to understand the parser generation process and the parsing machine.

The y.gviz file can be used as the input file to the Graphviz software to generate a graph of the parsing machine.

Of these steps, "Generate parsing machine" is the key step. It creates the parsing machine according to different algorithms as specified in the command line switches by the user. The details of these algorithms will be discussed in the next chapter. These algorithms may be independent or have a layered structure on each other. Figure 3.2 and Figure 3.3 show the relationship of these algorithms. The acronyms used in these figures are defined at the beginning of this chapter on page 18.

Figure 3.2 shows the relationship of these algorithms from the point of view of data flow. The data here is the input grammar. The input grammar can take the left side path, first be processed by the Knuth LR(1) algorithm, then end here or be processed by PGM LR(1) algorithm. Next it can either end, or be further processed by UPE and/or UPE Ext algorithms. The right side path is similar.

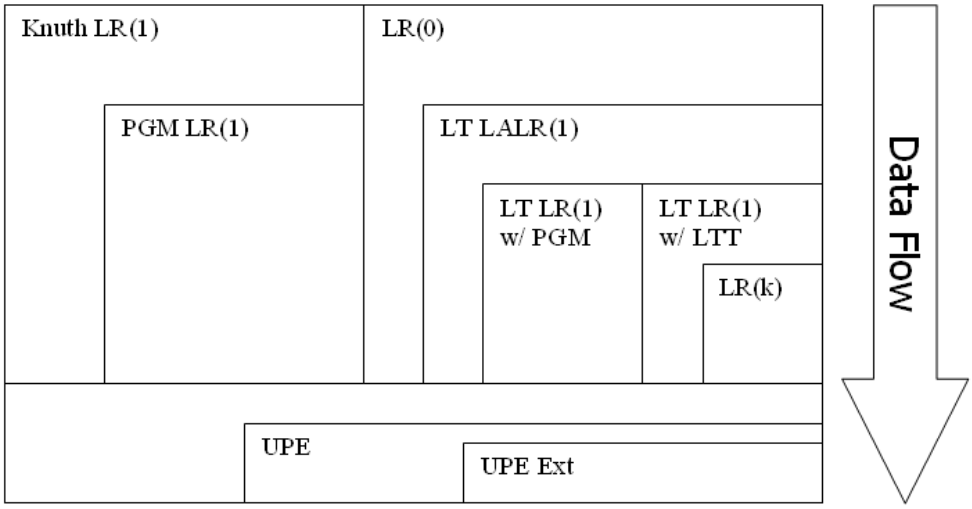


Figure 3.2: Relationship of algorithms from the point of view of data flow

This naturally shows the two basic approaches of LR(1) implemented in Hyacc: the approach of merging states as on the left side of Figure 3.2, and the approach of splitting state as on the right side of Figure 3.2.

On the merging side, the Knuth canonical LR(1) is the backbone algorithm. The PGM LR(1) algorithm adds in one step to decide whether to merge two states if they are "compatible" to each other.

On the splitting side, it always generates the LR(0) parsing machine first. If requested, it can generate the LALR(1) parsing machine based on the first phase of the lane-tracing algorithm. Then if specified, it can go on with the second phase of lane-tracing to generate LR(1) parsing machine. There are two methods for the second phase of lane-tracing. The first is based on the PGM method, the second is based on a lane-tracing table. Then if further specified, it can generate a LR(k) parsing machine for LR(k) grammars.

The generated parsing machine may contain unit productions that can be eliminated. The UPE algorithm achieves this task. The UPE Ext algorithm can be used to further remove redundant states after the UPE step.

Figure 3.3 shows the relationship of the algorithms from the point of view of implementation, i.e., how one algorithm is based on the other. UPE and UPE Ext are independent from these and are not shown.

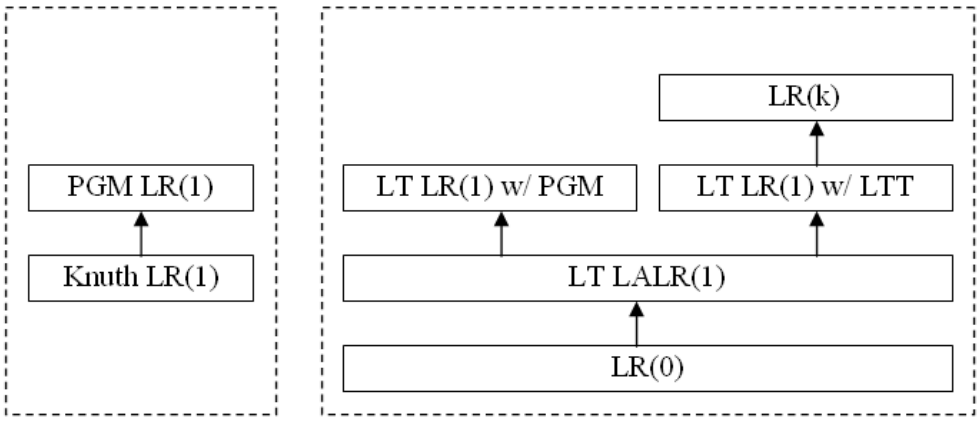


Figure 3.3: Relationship of algorithms from the point of view of implementation

3.3 Architecture of the LR(1) Parse Engine

3.3.1 Architecture

Similar to yaccpar of Yacc, hyaccpar is the parse engine of Hyacc. The parser generation process use hyaccpar as the parse engine, insert the parsing table, then use the parse engine to drive the table. The hyaccpar parse engine is similar to what is described in many compiler construction books. Specifically for Hyacc, the parse engine is shown as Algorithm 3.1. The LR(k) parse engine considers k lookaheads and is different. It is described in Chapter 6 (on page 148).

In the parse engine Algorithm 3.1, a state_stack is used to keep track of the current status of traversing the state machine. The parameter 'S' or current state is the state on the top of the state_stack. The parameter 'L' or lookahead is the symbol used to decide the next action from

the current state. The parameter 'A' or action is the action to take, and is found by looking at the parsing table entry (S, L).

Algorithm 3.1: Hyacc LR(1) Parse Engine Algorithm

```
1  Initialization:
2   push state 0 onto state_stack;
3  while next token is not EOF do
4      S ← current state;
5      L ← next token/lookahead;
6      A ← action for (S, L) in parsing table;
7      if A is shift then
8          push target state on state_stack;
9          pop lookahead symbol;
10         update S and L;
11     else if A is reduce then
12         output code associated with this reduction if any;
13         r1 ← left hand side symbol of reduction A;
14         r2 ← right hand side symbol count of reduction A;
15         pop r2 states from state_stack;
16         update current state S;
17         A_tmp ← action for (S, r1);
18         push target goto state A_tmp to state_stack;
19     else if A is accept then
20         if next token is EOF, then is valid accept. exit;
21         else, is error. do error recovery or exit;
22     else
23         is error, do error recovery;
```

3.3.2 Storing the Parsing Table

Storage tables

The following describes the tables that are used in hyaccpar to store the parsing table.

Let the parsing table have n rows (states) and m columns (number of terminals + non-terminals). Assume there are r rules, the number of nonempty entries in the parsing table is p.

Table 3.1 in next page lists all the storage tables and explanations to their use. These accurately define all information in the parsing table.

Complexity analysis

Suppose at state i there is a token j, we can find if an action exists by looking through the yytbltok table from yytbltok[yyrowoffset[i]] to yytbltok[yyrowoffset[i+1]-1]:

1) if yytbltok[k] == j, then yytblact[k] is the associated action;

2) if yytblact[k] $>$ 0, this is a shift/goto action;

3) if yytblact[k] $<$ 0, is a reduction, then use yyr1 and yyr2 to find number of states to pop and the next state to goto;

4) if yytblact[k] == 0 then it's an accept action, which is valid when j is the end of an input string.

The space used by the storage is: n + 2p + 2r. In most cases the parsing table is a sparse matrix, n*m $>$ 2p. It can be safe to say that usually n + 2p + 2r $<$ n*m.

For the time used by the search, the main factor is when looking through the yytbltok table from yytbltok[yyrowoffset[i]] to yytbltok[yyrowoffset[i+1]-1]. This can be made faster by binary search, which is possible if non-terminals and terminals are sorted alphabetically (or in this case, maybe numerically since they will be stored numerically here). Or instead of a binary search alone, use the combination of binary search (e.g., range $>$ 4) and linear search (range $\leq$ 4). Time complexity is O(ln(n)).

It can be made such that time complexity is O(1), by using the double displacement method which stores the entire row of each state. That requires more space though.

Examples

An example is given to demonstrate how to use these tables to represent the parsing table.

Example. Given grammar G3.1:

$E \rightarrow E + T \mid T$

$T \rightarrow T * a \mid a$

This is a LALR(1) grammar, so its LALR(1) parsing machine and LR(1) parsing machine are the same. The parsing machine is:

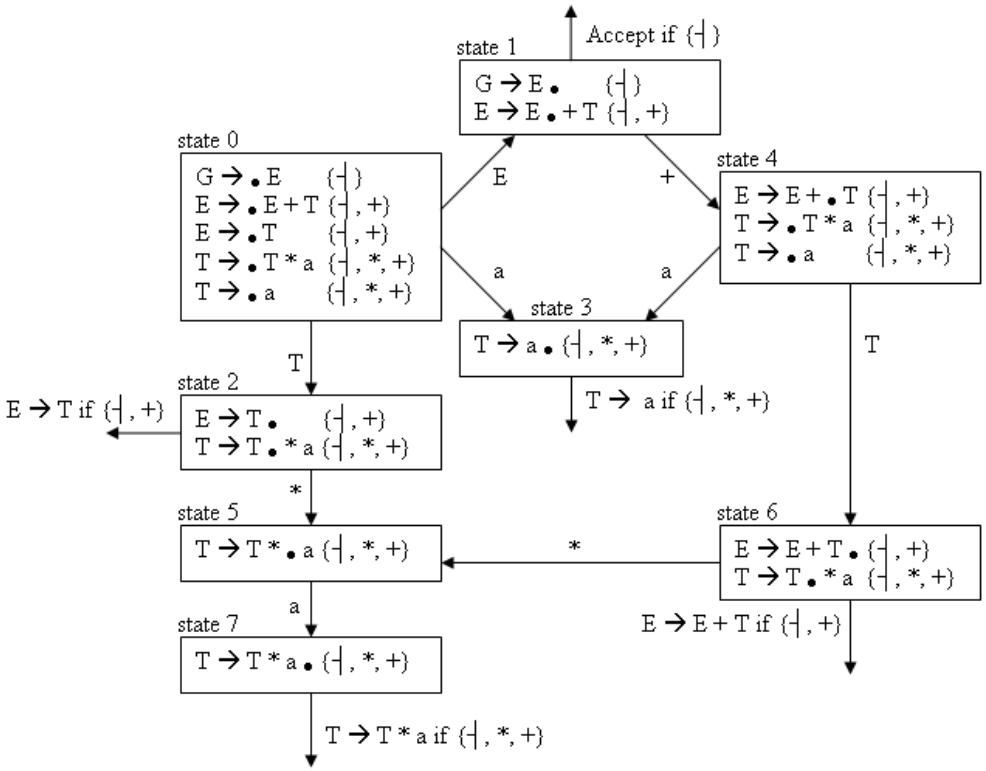


Figure 3.4: Parsing machine of grammar G3.1

Array Name	Explanation
yyfs[]	List the default reduction for each state. If a state does not have default reduction, its entry is 0. Array size = n.
yyrowoffset[]	The offset of parsing table rows in arrays yytblact[] and yytbltok[]. Array size = n.
yyptblact[]	Destination state of an action (shift goto reduce accept). if yytblact[i] $>$ 0, the action is shift/goto, if yytblact[i] $<$ 0, the action is reduce, if yytblact[i] is 0, the action is accept. -10000000 labels the end of the array. Array size = number of non-empty entries in the parsing table.
yyptbltok[]	The token for an action. if yytbltok[i] is positive, the token is a terminal, if yytbltok[i] is negative, the token is a nonterminal. -10000001 is just a place holder for a row. -10000000 labels the end of the array. Array size = number of non-empty entries in the parsing table.
yyr1[]	If the LHS symbol of rule i is a nonterminal, and its index among nonterminals (in the order of appearance in the grammar rules) is x, yyr1[i] = -x. If the LHS symbol of rule i is a terminal (only in case of unit production removing is used, in such case step 5 of the algorithm changes the LHS nonterminal of a rule into the corresponding leaf symbol (can be terminal or nonterminal) in the multi-rooted tree) and its token value is t, then yyr1[i] = t. Note this is different from yyr1[] of AT&T yacc or Bison, which only have nonterminals on the LHS of its rules, so the LHS symbol is always a nonterminal, and yyr1[i] = x, where x is defined the same as above. Array size = number of rules. (Including the augmented rule)
yyr2[]	Same as AT&T yacc yyr2[]. Let x[i] be the number of RHS symbols of rule i, then yyr2[i] = x[i] $\ll$ 1 + y[i], where y[i] = 1 if production i has associated code, y[i] = 0 otherwise. Array size = number of rules (including the augmented rule $accept = ...). This array is used for generating semantic actions.
yynts[]	List of non-terminals. Actually used only in DEBUG mode. Array size is the number of non-terminals plus 1.
yytoks[]	List of tokens (terminals). Actually used only in DEBUG mode. Array size is the number of terminals plus 1.
yyreds[]	List of the reductions. Note this does not include the augmented rule. Actually used only in DEBUG mode. Array size is the number of rules (including the augmented rule).

Table 3.1: Storage tables for the parsing machine in Hyacc parse engine

The parsing table is:

state	$	+	*	a	E	T
0	0	0	0	s3	g1	g2
1	a0	s4	0	0	0	0
2	r2	r2	s5	0	0	0
3	r4	r4	r4	0	0	0
4	0	0	0	s3	0	g6
5	0	0	0	s7	0	0
6	r1	r1	s5	0	0	0
7	r3	r3	r3	0	0	0

Table 3.2: Parsing table for grammar G3.1

Here the parsing table has n = 8 rows, and m = 6 columns. There r = 3 rules including the augmented rule.

The storage tables in y.tab.c are shown in Table 3.3.

Array yyfs[] lists the default reduction for each state: state 3 has default reduction on rule 4, and state 7 has default reduction on rule 3.

Array yyrowoffset[] defines the offset of parsing table rows in arrays yytblact[] and yytbltok[]. E.g., row 1 starts at offset 0, row 2 starts at offset 3.

Array yytblact[] is the destination state of an action. The first entry is 97, which can be seen in the yytoks[] array. The second entry is 1, which stands for non-terminal E. And as we see in the parsing table, entry (0, a) has action s3, entry (0, E) has action g1, thus in yytblact[] we see correspondingly the first entry is 3, and the second entry is 1. Entry 10000000 in both yytblact[] and yytbltok[] labels the end of the array. Entry 0 is yytblact[] labels the accept action. Entry 0 in yytbltok[] stands for the token end marker $. Entry -10000001 in yytbltok[] labels that this state (row in parsing table) has no other actions but the default reduction. 10000001 is actually just a dummy value that is never used, and servers as a place holder so yyrowoffset[] can have a corresponding value for this row. It may be possible to remove this place holder and let the yyrowoffset[] value for this row be the same as the next row, but this has not been tried so far.

Entries of array yyr1[] are defined as the index of the LHS token among non-terminals in the order of the appearance in the grammar rules. So first entry 0 stands for $accept, the second and the third entries 1 stands for E, the fourth and fifth entries 2 stands for T.

```
#define YYCONST const
typedef int yytabelem;

static YYCONST yytabelem yyfs[] = { 0, 0, 0, -4, 0, 0, 0, -3};

static YYCONST yytabelem yyptbltok[] = {
97, -1, -2, 0, 43, 0, 43, 42, -10000001, 97,
-2, 97, 0, 43, 42, -10000001, -10000000};

static YYCONST yytabelem yyptblact[] = {
3, 1, 2, 0, 4, -2, -2, 5, -4, 3,
6, 7, -1, -1, 5, -3, -10000000};

static YYCONST yytabelem yyrowoffset[] = { 0, 3, 5, 8, 9, 11, 12, 15, 16};

static YYCONST yytabelem yyr1[] = { 0, -1, -1, -2, -2};
static YYCONST yytabelem yyr2[] = { 0, 6, 2, 6, 2};

#ifdef YYDEBUG

typedef struct char *t_name; int t_val; yytoktype;

yytoktype yynts[] = {
    "E", -1,
    "T", -2,
    "-unknown-", 1 /* ends search */
};
yytoktype yytoks[] = {
    "a", 97,
    "+", 43,
    "*", 42,
    "-unknown-", -1 /* ends search */
};
char * yyreds[] = {
    "-no such reduction-"
    "E : 'E' '+' 'T'",
    "E : 'T'",
    "T : 'T' '*' 'a'",
    "T : 'a'",
};
#endif /* YYDEBUG */
```

Table 3.3: Storage tables in y.tab.c for grammar G3.1

Array yyr2[] is defined as described in Table 3.1, and it is easy to see the correspondence of the values. For example, the first entry 0 is derived as this:

$$yyr2[0] = x[0] << 1 + y[0] = 1 << 1 + 0 = 0$$

where x[0] is 1 because the RHS symbol count of rule 0 (this first rule) is 1, and y[0] = 0 because this rule has no associated semantic action. The second entry 6 is derived as this:

$$yyr2[1] = x[1] << 1 + y[1] = 3 << 1 + 0 = 6$$

where x[0] is 1 because the RHS symbol count of rule 0 (this first rule) is 1, and y[0] = 0 because this rule has no associated semantic action.

Example. Given grammar G3.2:

S → c X t | c Y n | r Y t | r X n
X → a
Y → a

G3.2 is a LR(1) grammar, so the LR(1) parsing machine is bigger. This example shows how a reduce/reduce conflict in the LALR(1) parsing machine is resolved in the LR(1) parsing machine.

Figure 3.5 shows the LALR(1) parsing machine of G3.2. State 6 contains a reduce/reduce conflict, because both reductions 5 and 6 can be applied upon context symbols 'n' and 't'. LALR(1) parser generators such as Yacc and Bison by default use the rule that appears first in the grammar specification to solve the conflict, in this case it is reduction 5 'X → a'. However a LR(1) parsing machine can divide state 6 into two states to avoid such a reduce/reduce conflict. This is shown in Figure 3.6: state 13 is separated out of state 6, and now both do not have reduce/reduce conflicts.

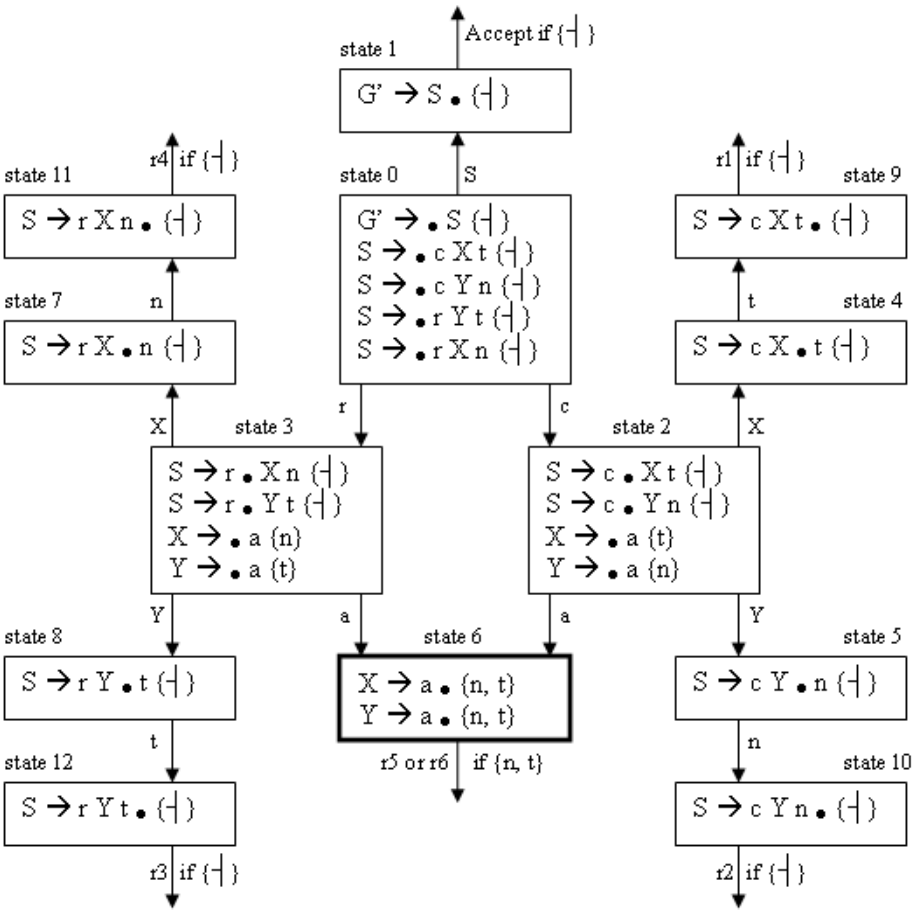


Figure 3.5: LALR(1) parsing machine of grammar G3.2

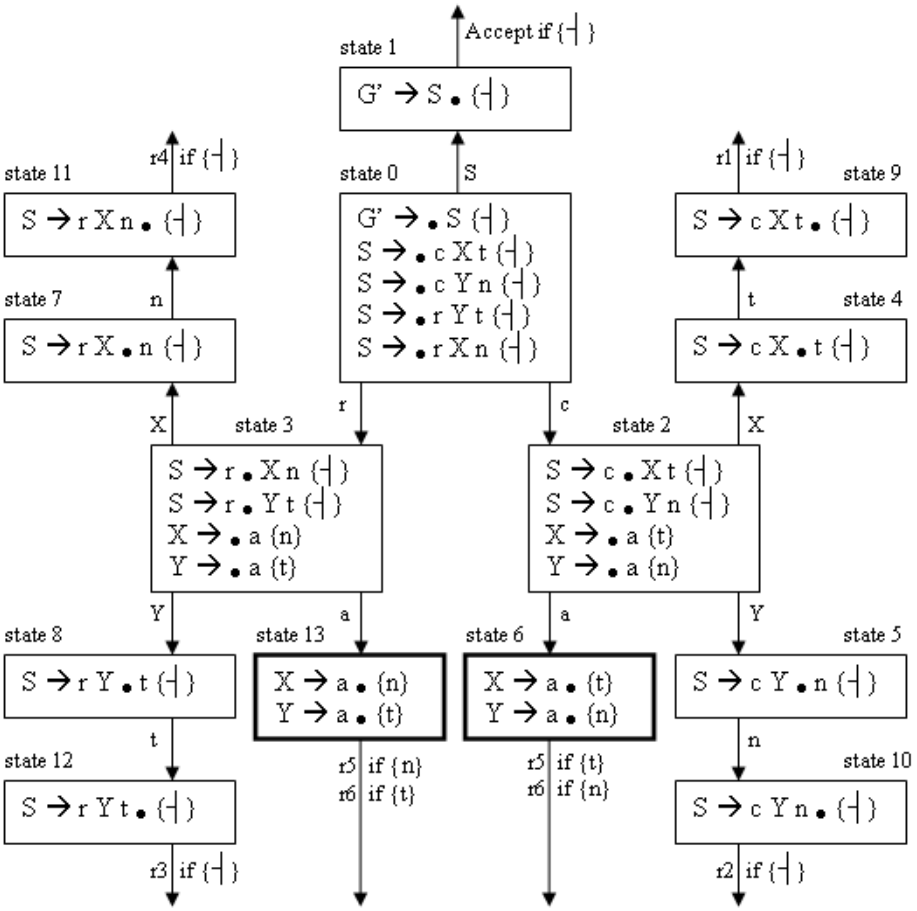


Figure 3.6: LR(1) parsing machine of grammar G3.2

3.3.3 Handling Precedence and Associativity

The way that Hyacc handles precedence and associativity is the same as Yacc and Bison. By default, in a shift/reduce conflict, shift is chosen; in a reduce/reduce conflict, the reduction whose rule appears first in the grammar is chosen. But this may not be what the user wants. So %left, %right and %nonassoc are used to declare tokens and specify precedence and associativity to solve this issue. Actually there is nothing new in this. But information on this is hard to find, so I summarize my findings below.

Define associativity and precedence

Associativity is defined by three directives: 1) %left: left associativity, 2) %right: right associativity, 3) %nonassoc: no associativity - find this symbol (often an operator) twice in a row is an error. In practice, shift is right associative, reduce is left associative.

Precedence is defined for both tokens and rules.

1) For tokens (terminals)

   (a) Two tokens declared in the same precedence declaration have the same precedence.

   (b) If declared in different precedence declarations, the one declared later has higher precedence.

   (c) If a token is declared by %token, then it has no associativity, and its precedence level is 0 (means no precedence).

   (d) If a token is declared by %left or %right, then with each declaration, the precedence is increased by 1.

2) For rules

   (a) A rule gets its precedence level from its last (right-most) terminal token.

   (b) Context-dependent precedence: defined using %prec TERMINAL_TOKEN, where the TERMINAL_TOKEN is declared using %left or %right earlier.

How conflicts are resolved using precedence and associativity

A conflict means that, for the same state, the same context symbol (lookahead), there is more than 1 possible action to take.

1) Resolve shift/reduce conflict

   E.g. a state containing the following two configurations has shift/reduce conflict over '+', because it can reduce using rule 1 or shift using rule 2 upon lookahead +:

   $E \rightarrow E + E \bullet \{;, +\}$ rule 1

   $E \rightarrow E \bullet + E \{;, +\}$ rule 2

   We define the precedence of a rule to be that of its right-most terminal.

   According to the Dragon book (page 263), to choose between shift (over token a) or reduce (by rule i), reduce if:

   (a) Precedence of rule i is greater than the precedence of token a, or

   (b) Token a and rule i have equal precedence, but the associativity of rule i is left. Otherwise, use shift.

   Two supplemental rules are:

   (c) If either the rule or the lookahead token has no precedence, then shift by default.

   (d) By default, to break ties, we chooses shift over reduce. It's like comparing action types, where s is favored over r.

2) Resolve reduce/reduce conflict.

   The following example has r/r conflict. On ; and + this state can reduce by either rule 1 or rule 2:

   $E \rightarrow E + E \bullet \{;, +\}$ rule 1

   $E \rightarrow E \bullet \quad \{;, +\}$ rule 2

   However, in principle all reduce/reduce conflicts should be studied carefully and better removed.

   By default, bison/yacc chooses the rule that appears first in the grammar.

3) There can be no shift/shift conflicts. Two such rules are just two core configurations of the successor state.

Implementation issues

1) Get input information.

   When parsing input grammar file, a) get precedence and associativity for each terminal, store information in symbol table. b) next get precedence and associativity for each grammar rule (that of its right-most terminal token), and store this information with each rule.

2) Solve conflicts.

   Do the following when a) doing transition operation when constructing the LR(1) parsing machine, b) combine compatible states and c) propagate context change:

```
For each final configuration {
    Compare its context symbols to the scanned symbol of non-final
        configurations for S/R conflict;
    Compare its context symbols with the context symbols of another
        final configuration for R/R conflict;
}
```

   All the conflicts are resolved at the time of constructing the parsing table.

3.3.4 Error Handling

Error handling is the same as in Yacc. There are a lot of complaints about the error recovery scheme of Yacc. But here we are concentrating on studying LR(1) algorithms, better error recovery is not the goal of our work. Also we want to keep compatible with Yacc and Bison. For these reasons we keep the way that Yacc handles errors.

3.4 Data Structures

The data structures should reflect the nature of the objects, and also take time and space performance into consideration. Good data structures can make it easy for algorithms implementation, enhance both efficiency and robustness.

These major data structures are defined for constructing the Knuth LR(1) parsing machine: Grammar, State_collection, State, Configuration, Production, Context, SymbolTblNode, HashTblNode, SymbolNode, Queue, Conflict, State_array.

A symbol table is implemented using hash table, and uses open hash to store elements. Symbols with the same hash value are stored at the same array entry of the hash table, one by one as a linked list.

This symbol table is used to achieve O(1) or close to O(1) performance for many operations. All the symbols (terminals and non-terminals) used in the grammar file are stored in this symbol table, and no second copy of each string is stored. All the string references are made to symbol table nodes in the symbol table that contain the symbol strings, and string comparisons are converted to pointers comparisons of the symbol table nodes. This saves both space and time. Besides, each symbol table node also contains much other information about each symbol. This information is calculated at the time of parsing the grammar file and stored for later use. The definition for a symbol table node is:

symbol	value	symbol_type	TP	seq	ruleIDList	next_ptr

Here 'symbol' is the actually storage location of the symbol string. 'value' specifies an integer value representing this symbol to be used in the parsing table. 'symbol_type' can be Terminal, Non-Terminal or None. 'seq' specifies the parsing table column number for this symbol, so given a symbol we immediately know which column in the parsing table it belongs to. 'ruleIDList' gives the list of rules whose LHS contains this symbol. 'next_ptr' is the pointer to the next symbol node. 'TP' starts for Terminal Property and is defined as:

is_quoted	precedence	associativity

'is_quoted' is a boolean value, means whether it is in quoted form (e.g., 'a' instead of a) in the input grammar. Precedence and associativity have the standard meanings.

Linked list, statically and dynamically allocated arrays are all used. Linked lists are used where the number of entries is not known initially and only sequential access is needed. Dynamic arrays are used where indexed access is needed for fast retrieval. If the array is an array of objects, then usually the type of the array is pointers to such objects, instead of the object itself. This saves space. Static arrays are used only when the number of entry is known initially and it won't waste much space.

Sometimes linked lists and arrays are used for the same set of objects. For example, the State_collection struct stores states as a linked list of states. All the states in the parsing machine are stored in a State_collection list. However sometimes indexed access is preferred for states, so a separate State_array object is used, which is a dynamically allocated array of State pointers storing pointers to the entries in the State_collection list. Besides, to make searching states fast, a hash table is used to store hash values of the states. Under difference circumstances different objects are used to expedite the operations.

In the parsing table, the rows index the states (row 1 represents actions of state 1, etc.), and the columns are the lookahead symbols (both terminals and non-terminals) upon which shift/goto/ reduce/accept actions happen. The parsing table is implemented as a one dimensional integer array. Each entry [row, col] is accessed as entry [row * column_size + col]. In the parsing table positive numbers are for shifts, negative numbers are for reductions, -10000000 is for Accept and otherwise 0 is for errors. Assuming an integer takes 4 bytes, for a parsing machine of 1000 states and 600 symbols (terminals plus non-terminals), this can be 600 * 1000 * 4 = 2.4 MB. Usually for a grammar of this size, about 90% of the parsing table cells would contain zeros. But today memory is cheap, and 2.4MB is nothing. So this one-dimension array is kept for its ease of implementation. This is how the parsing table is represented in Hyacc.

Multi-rooted trees are used when doing unit production eliminations. Binary trees are not used since a hash table is more suitable: only insertion and find operations are needed in most cases.

There is no size limit for any data structures, they can grow until they consume all the memory. But usually this can hardly happen. So far the largest memory usage happens for the grammar of C++ 5.0 when no optimization is used, in which case it uses about 120 MB of memory. Most computers today can handle that. The program, however, artificially sets an upper limit of 512 characters for the max length of a symbol. The program also sets an upper limit of 65536 for the number of UnitProdState objects used in the unit production elimination (UPE) algorithm. Reaching this many combined states during the UPE process usually means that some error has occured. This is because the number of states in a parsing machine is usually in the order of thousands, and creation of tens of thousands of combined states when removing unit productions is usually unlikely to happen.

Chapter 4

LR(1) Parser Generation

4.1 Overview

Hyacc has implemented these algorithms related to LR(1) parsing:

1) The original Knuth LR(1) algorithm
2) The PGM algorithm (weak compatibility)
3) The UPE algorithm
4) Extension to the UPE algorithm
5) LR(0) algorithm
6) LALR(1) based on the first phase of the lane-tracing algorithm
7) The LR(1) lane-tracing algorithm.

There are lots of issues involved in the designs and implementations. This chapter will explain these issues in detail.

4.2 Knuth's Canonical Algorithm

4.2.1 The Algorithm

This algorithm was introduced in section 2.2.1 (on page 8). A easier to understand summary of this algorithm is in the dragon book. According to the dragon book, the functions used in the Knuth LR(1) parser generation algorithm are as shown in algorithms 4.1, 4.2 and 4.3 (adapted from [15]).

Two major steps are involved when generate a LR(1) parsing machine:

1) closure(): get the closure of a state

2) transition(): make a transition from a state to one of its successors.

The items() procedure is the backbone procedure of the LR(1) parser generation algorithm. Initially state 0, which is for the goal production, is inserted into the collection C. Then function closure() obtains all the successor configurations from the core configurations. Next transition() makes successor states from the current state, and inserts the new states into the collection C. The program then processes the next unprocessed state. This works in a cycle until no new states are created.

Algorithm 4.1: Knuth LR(1) parser generation: function closure(I)

Input: Item set I

Output: Item set I with closure generated

```
repeat
    foreach item [A → α • Bβ, a] in I do
        foreach production B → γ in G' do
            foreach terminal b in FIRST(βa) do
                If [B → • γ, b] is not in I, add it to I;
until I no longer changes ;
return I;
```

Algorithm 4.2: Knuth LR(1) parser generation: function transition(I, X)

Input: Item set I; Symbol X

Output: Item sets obtained by X-transition on I

```
Let J be the set of items [A → α • Xβ, a] such that [A → αX • β, a] is in I;
return closure(J);
```

Algorithm 4.3: Knuth LR(1) parser generation: procedure items(G')

Input: An augmented grammar G'

Output: A collection C of Item sets for grammar G'

```
C ← {closure({[S' → • S, $]})};
repeat
    foreach set of items I in C do
        foreach grammar symbol X do
            if transition(I, X) is not empty and not in C then
                add transition(I, X) to C;
until C no longer changes ;
```

4.2.2 Implementation Issues

The algorithms 4.1, 4.2 and 4.3 only represent the big picture of the Knuth LR(1) parser generation algorithm. There are a lot of details and complications in the details. Here we talk about some techniques used to make it more efficient.

As the big picture, a linked list is used to represent the collection C. C starts with state 0. A pointer is used to point to the current state. More states are added to C as the transition() function obtains successor states from an existing state. The pointer then traverses the linked list, until it reaches the end, at which point no more states is added.

Closure: Incrementally combining new configurations

At the beginning, state S contains only core configurations. In the end it contains all the configurations. A configuration is determined by three factors: the production, the marker position, and the context set. However it is easy to see that one can combine those configurations with the same production and marker position, and differ only by context set. We call such configurations *compatible configurations*. This combination of *compatible configurations* will not change any characteristics of the parsing machine. However the combination can save a lot of storage space for the state objects as well as operation time, which is easy to see even when applying the algorithm by hand. Thus we prefer to combine it.

One straightforward way of combining *compatible configurations* is to do it after all the configurations of a state are generated. Another way is to combine them along the way when new configurations are generated.

It is easy to see that the second method is always faster. Assume that there are n different configurations in a state, and m *compatible configuration sets*. Here a *compatible configuration set* is the set of all the configurations that are compatible, i.e., differ only by context. Then using the first method, the complexity is always $O(n^2)$. Using the second method, the complexity is about $O(m^2/4)$. The difference is $4 * (n/m)^2$. The ratio can be big when m is much smaller than n.

Table 4.1 shows the ratio $4 * (n/m)^2$ for some grammars. The ratio can be around 400 in some cases. That will cause a substantial difference in performance.

Grammar	states	n/states	m/states	$4 * (n/m)^2$
Algol60	1712	141.2	13.24	454.94
C	1605	147.72	15.98	341.81
Cobol	1401	2.68	2.63	4.15
Ftp	201	1.83	1.78	4.23
Grail	719	9.11	4.49	16.47
Matlab	783	121.64	13.73	313.96
Pascal	2244	23.21	7.56	37.70

Table 4.1: The ratio $4 * (n/m)^2$ for some grammars

Figure 4.1 is an example on state 0 of the parsing machine of grammar G3.1. On the left side is the one without combining compatible configurations, and contains ten configurations. On the right side is the one after combining compatible configurations, and contains five configurations only.

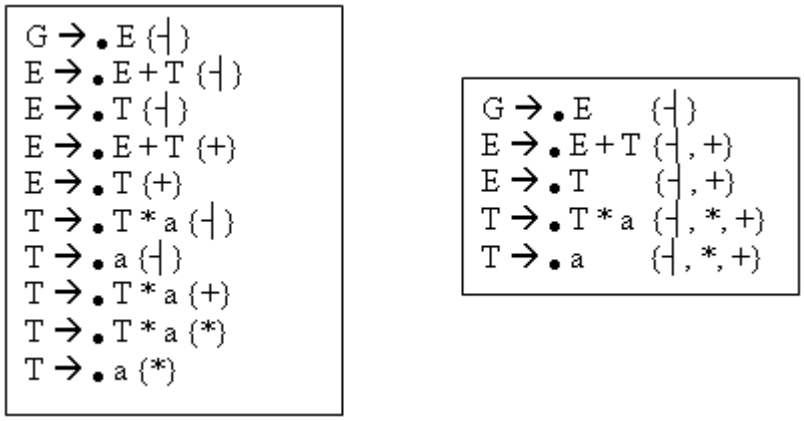


Figure 4.1: State 0 of the parsing machine of grammar G3.1

Below is the algorithm that implements the second method using a queue.

Algorithm 4.4: procedure getClosure(S)

Input: A state S with core configurations only

Output: State S with all configurations obtained

```
1  Let integer queue Q = {0, 1, 2, ..., N − 1}, where N is the number of core
   configurations in S;
2  while Q.count > 0 do
3      i = Q.pop();
4      Let the i − th configuration in S be C : [A → α • Xβ, ω];
5      if Xβ ≠ ϵ and X is non-terminal then
6          ψ = getContext(C);
7          foreach production X → γ in grammar G' do
8              if ∃j, s.t. the j − th configuration of S is D : [X → • γ, ϕ] then
9                  ϕ = ϕ ∪ ψ;
10                 if any new symbol is added from ψ to ϕ, and j is not in Q yet then
11                     Q.insert(j);
12             else
13                 Add a new configuration [X → • γ, ψ] to state S;
14                 Let j be the index of this new configuration;
15                 Q.insert(j);
```

The function getContext(C) gets the context from a configuration $[A \rightarrow \alpha \bullet X\beta, \omega]$ to be passed to successor configurations. Below is the function getContext(C).

Algorithm 4.5: getContext(C)

Input: Configuration C: $[A \rightarrow \alpha \bullet X\beta, \omega]$

Output: Context set ψ obtained from the string $\beta\omega$

```
1  Let ψ = ∅;
2  Let configuration C be [A → α • Xβ, ω];
3  if Xβ == ϵ then
4      ψ = ω;
5  else
6      theads = getTHeads(β);
7      if theads == ∅ then
8          ψ = ω;
9      else
10         if ϵ ∈ theads then
11             ψ = ω ∪ (theads − {ϵ});
12         else
13             ψ = theads;
14 return ψ;
```

The function getTHeads() gets the terminal heads of a string and returns it.

One common approach to getTheads() is shown in algorithms 4.6 and 4.7.

A second approach which is faster is shown in algorithms 4.8 to 4.10. Note that in the function getTheads_v2(), the step for each production S in the grammar can be expedited if we precalculate for each non-terminal S the grammar productions whose LHS is S and record it. We actually do this and record the production list for each non-terminal in the symbol node. The function getTheads_v2() thus avoid going over each production in the grammar including those irrelevant ones. The second approach is faster also because we separated the sets for non-terminals and terminals, so we do not need to check terminals for successor productions, and we also do

not need to remove terminals from the combined set of terminals and non-terminals, like we do in getTheads_v1().

Algorithm 4.6: getTHeads_v1(α)

Input: string α

Output: terminal heads list obtained from α

```
1 φ = getHeads(α);
2 remove non-terminals from φ;
3 return φ;
```

Algorithm 4.7: getHeads(α)

Input: string α

Output: heads list (terminals and non-terminals) obtained from α

```
1  φ = ∅;
2  foreach symbol s in α do
3      φ = φ ∪ {s};
4      If s is not vanishable, break out of foreach loop;
5  if all the symbols in α are visited then
6      φ = φ ∪ {ϵ};
7  repeat
8      foreach production A → γ in grammar G' do
9          if A is in φ then
10             foreach symbol s in γ do
11                 φ = φ ∪ {s};
12                 If s is not vanishable, break out of foreach loop;
13 until no new symbol is added to φ ;
14 return φ;
```

Algorithm 4.8: getTHeads_v2(α)

Input: string α

Output: terminal heads list obtained from α

```
1 φ = ∅;
2 ψ = ∅;
3 insertAlphaToHeads(α, φ, ψ);
4 foreach symbol S in φ do
5     foreach production S → γ in grammar G' do
6         insertRhsToHeads(γ, φ, ψ);
7 return φ;
```

Algorithm 4.9: insertAlphaToHeads(α, ϕ, ψ)

Input: string α, non-terminal set ϕ and terminal set ψ

Output: non-terminal set ϕ and terminal set ψ

```
1 foreach symbol S in α do
2     if S is non-terminal then
3         φ = φ ∪ {S};
4     else
5         ψ = ψ ∪ {S};
6     return;
7 ψ = ψ ∪ {ϵ};
```

Algorithm 4.10: insertRhsToHeads(α, ϕ, ψ)

Input: string γ, non-terminal set ϕ and terminal set ψ

Output: non-terminal set ϕ and terminal set ψ

```
1 foreach symbol S in γ do
2     if S is vanishable then
3         φ = φ ∪ {S};
4     else
5         if S is non-terminal then
6             φ = φ ∪ {S};
7         else
8             ψ = ψ ∪ {S};
9         return;
```

Transition: Search for existing state using a hash table of states.

Given a state in the parsing machine, we need to use the transition() procedure to get its successor states, and then check these successor states against those already in the parsing machine. If a successor state does not exist in the parsing machine yet, we add it to the parsing machine. Otherwise just discard it.

Algorithm 4.11: findExistingState(S)

Input: state S

Output: state number of the found state, or -1 if not found

```
1 find a state T that is the same as S;
2 if T is found then
3     return the state number of T;
4 else
5     return -1;
```

One can use a sequential search to find whether a state already exists. A faster approach is to store references to the states in a hash table. This in concept can reduce the search cost from O(n) to O(1).

A state can be hashed by a function of the production index (in the grammar's production list) and the marker position of the state's core configurations. Let ruleID be the index of the core configuration's production in the grammar's production list; let marker be the position of marker on the core configuration's RHS; and let H_SIZE be the size of the hash table. Then a hash function for a state can be defined as:

Algorithm 4.12: Hash function for a state S

Input: state S

Output: hash value of the state

```
1 val = 0;
2 foreach core configuration r of state S do
3     val = (val + r.ruleID * 97 + r.marker * 7 + i) % H_SIZE;
4 return val;
```

In general, for a parsing machine with n states, from the first to the last state as they are added, in average the number of searches is given by:

$$\sum_{k=1}^{n} \frac{k}{2} = \frac{n(n+1)}{4} \tag{4.2.1}$$

Using the state hash table, for the same parsing machine, assume in average there are m states per list, then the number of searches is given by:

$$(1 + m) * n \tag{4.2.2}$$

where the 1 is the cost of getting the hash value for a state.

For example, for the grammar of C, there are 1605 states in the Knuth canonical LR(1) parsing machine. The above formular gives 644,407 searches. But using this state hash table, using the above hash function, in average there are 5.02 states per list (from StateHashTbl_dump() output), so the cost is (1 + 5.02) * 1605 = 9,662. This is 644,407/9,662 = 67 times faster.

4.3 Pager's Practical General Method

4.3.1 The Algorithm

This algorithm was introduced in section 2.2.4 (on page 9). This is a straightforward add on to the backbone algorithm of the Knuth canonical LR(1) algorithm. When a new state is generated, it is compared against all the existing states to see if it is "compatible" with an existing state. If a match is found, the new state is merged with the compatible state.

There are two kinds of compatibility defined in [48]. The first and commonly used one is weak compatibility. It is the default when we talk about compatibility. The second one, employing more computation and possibly resulting in a smaller parsing machine, is strong compatibility. In our discussion, unless specifically mentioned, compatibility always means weak compatibility.

The definition for weak compatibility [48] is given below.

Let S and S' be two states with a common set of core configurations. Let U_r and U'_r be the r-th core configuration of S and S', respectively. We call S and S' *weakly compatible* if and only if at least one of the following is true for all i and j, $1 \leq i < j \leq n$, where n is the number of core configurations in S and S':

1) $U_i \cap U'_j = \oslash$ and $U'_i \cap U_j = \oslash$

2) $U_i \cap U_j \neq \oslash$

3) $U'_i \cap U'_j \neq \oslash$

An example of how the algorithm works in given below.

We just need to change the findExistingState() function at the end of section 4.2.2 (on page 48) to add the PGM algorithm to the Knuth LR(1) algorithm:

Algorithm 4.13: findExistingState(S) for the PGM algorithm

Input: state S

Output: the state number of a same or compatible state, or -1 if not found

```
1  find a state T which is the same as S;
2  if T is found then
3      return the state number of T;
4  else
5      find a compatible state T of S;
6      if T is found then
7          combineCompatibleStates(T, S);
8          return state number of T;
9      else
10         return -1;
```

So when we run transition() on a state, we obtain a set of successor states. For each of these successors, we need to find out if it should be inserted into the parsing machine as a new state, or if it is already in the parsing machine, or whether there exists a compatible state in the parsing machine that we can combine it with.

Two state states have the same production, the same marker position, and the same contexts. Two compatible state have the same production and the same marker position, but different contexts.

4.3.2 Implementation Issues

The main issue here is to propagate state context change.

After combining compatible states, if the combined state has successor states, then we need to propagate the context change to the successors. This can be a recursive procedure and is where complications occur. This is also a computationally expensive process. But ways exist to make it more efficient, as explained below.

One way of propagating context change is to run the closure() operation on a successor state again when the parent state's context has changed. The process is shown in algorithms 4.14 and 4.15. The problem with this is that one always regenerates the entire state. If only the contexts of a small number of configurations are changed, then we will need a lot of extra work on those configurations without change.

We can do it in a more efficient way by propagating context changes only on those configurations whose contexts actually change, thus only affected successor states are updated. This is combineCompatibleStates_v2() as shown in algorithms 4.16, 4.17 and 4.18. A queue is used to store the index of those configurations that are actually changed. Then when propagating to successor states, we only check those configurations on the queue. This way we can save computation time.

Algorithm 4.14: combineCompatibleStates_v1(S, S')

Input: Destination state S, source state S'

Output: The combined state S

```
1  is_changed = false;
2  foreach pair of corresponding core configurations C in S and C' in S' do
3      combine the context of C' to the context of C;
4      if the context of C is changed then
5          is_changed = true;
6      if is_changed == true then
7          get closure on state S;
8          propagateContextChange_v1(S);
```

Algorithm 4.15: propagateContextChange_v1(S)

Input: state S

Output: context change of state S is propagated to its successor states

```
1  is_changed = false;
2  foreach successor state T of state S do
3      foreach core configuration C of T do
4          find the corresponding configuration C' in S';
5          combine the context of C' to the context of C;
6          if the context of C is changed then
7              is_changed = true;
8      if is_changed == true then
9          get closure on state S;
10         propagateContextChange_v1(S);
```

Algorithm 4.16: combineCompatibleStates_v2(S, S')

Input: Destination state S, source state S'

Output: The combined state S

```
1  is_changed = false;
2  config_queue = ⊘;
3  foreach pair of corresponding core configurations C in S and C' in S' do
4      combine the context of C' to the context of C;
5      if the context of C is changed then
6          is_changed = true;
7          Let i be the index of C in the configuration list of S;
8          queue_push(config_queue, i);
9          getConfigSuccessors(S);
10     if is_changed == true then
11         propagateContextChange_v2(S);
```

Algorithm 4.17: propagateContextChange_v2(S)

Input: state S

Output: context change of S is propagated to its successor states

```
1  is_changed = false;
2  config_queue = ⊘;
3  foreach successor state T of state S do
4      foreach core configuration C of T do
5          find the corresponding configuration C' in S;
6          combine the context of C' to the context of C;
7          if the context of C is changed then
8              is_changed = true;
9              Let i be the index of C in the configuration list of S;
10             queue_push(config_queue, i);
11             getConfigSuccessors(T);
12     if is_changed == true then
13         propagateContextChange_v2(S);
```

Algorithm 4.18: getConfigSuccessors(S)

Input: state S

Output: context change of the configurations in the config_queue are propagated to successor configurations

```
1  while config_queue.count() > 0 do
2      C = config_queue.pop();
3      Let C be in the format [A → α • Xβ, ω];
4      if Xβ ≠ ϵ and X is non-terminal then
5          ψ = getContext(C);
6          foreach production X → γ in grammar G′ do
7              if ∃ j, s.t. the j-th configuration of S is D : [X → • Yγ, ϕ] then
8                  ϕ = ϕ ∪ ψ;
9                  if any new symbol is added from ψ to ϕ, D is not final
                   configuration, Y is not terminal, and j ∉ config_queue then
10                     config_queue.insert(j);
11                 else
12                     add a new configuration [X → • γ, ψ] to state S;
13                     let j be the index of this new configuration;
14                     config_queue.insert(j);
```

4.4 Pager's Unit Production Elimination Algorithm

4.4.1 The Algorithm

Abundant existence of unit productions would unnecessarily increase parsing table size and waste parsing time, sometimes by 20$\sim$30%. There have been proposed algorithms on eliminating unit productions from different researchers.

Pager's unit production elimination algorithm is described in [46]. It is applied after the PGM algorithm to further reduce the state space to achieve a more compact parsing machine. However, this should not be the final restriction as to where it can be applied. For example, it should be ok to apply it to a LALR(1) parsing machine obtained using a LALR(1) algorithm.

A unit production is defined as a production $x \rightarrow y$ where both x and y are single symbols. A leaf is defined as a symbol that only appears on the RHS of any unit production but never on the LHS of any unit production.

As in [46], the algorithm is composed of 5 rules:

1) For each state S of the parsing machine (including new states added in step 2), and for each leaf x where the x-successor of S contains a unit reduction, do step 2. Go to step 3 after finish.

2) For $1 \leq i \leq n$, let x_i be the symbols such that $x_i \Rrightarrow x$ (including x itself), and for which shift/goto actions are defined at state S. Let the x_i-successor of S be T_i. If any state R is or at a earlier time has been a combination of states T_1, ..., T_n, then let R be the new x-successor of state S; otherwise combine states T_1, ..., T_n into a new state T and make T the new x-successor of S.

3) Delete all such transitions where the transition symbol is on the left-hand side of a unit productions.

4) Delete all states that now can not be reached from state 0.

5) Replace all such reductions $y \rightarrow w$ by $x \rightarrow w$, where y is the left-hand side symbol of a unit production, and x is a randomly selected leaf such that $y \Rrightarrow x$.

Example. Use grammar G3.1 (on page 26). Its parsing machine is shown in Figure 3.4. An example is given for this in Figure 4.2. First we need to find the leaves of the grammar. This is achieved by constructing a multi-rooted tree, which is $E \rightarrow T \rightarrow a$ for G3.1. So a is the only leaf in this case. Then following the algorithm rule 1, we see that only states 0 and 4 have a-successor that has a unit production: state 0's a-successor state 3 has a unit production $T \rightarrow a$, state 4's a-successor is also state 3. Thus we follow rule 2 to combine successor states of state 0 and state 4. These are shown in steps (b) and (c) in Figure 4.2. Next, step (d) follows rule 3, step (e) follows rule 4, and step (f) follows rule 5 and also rearranges the states in a better-looking configuration.

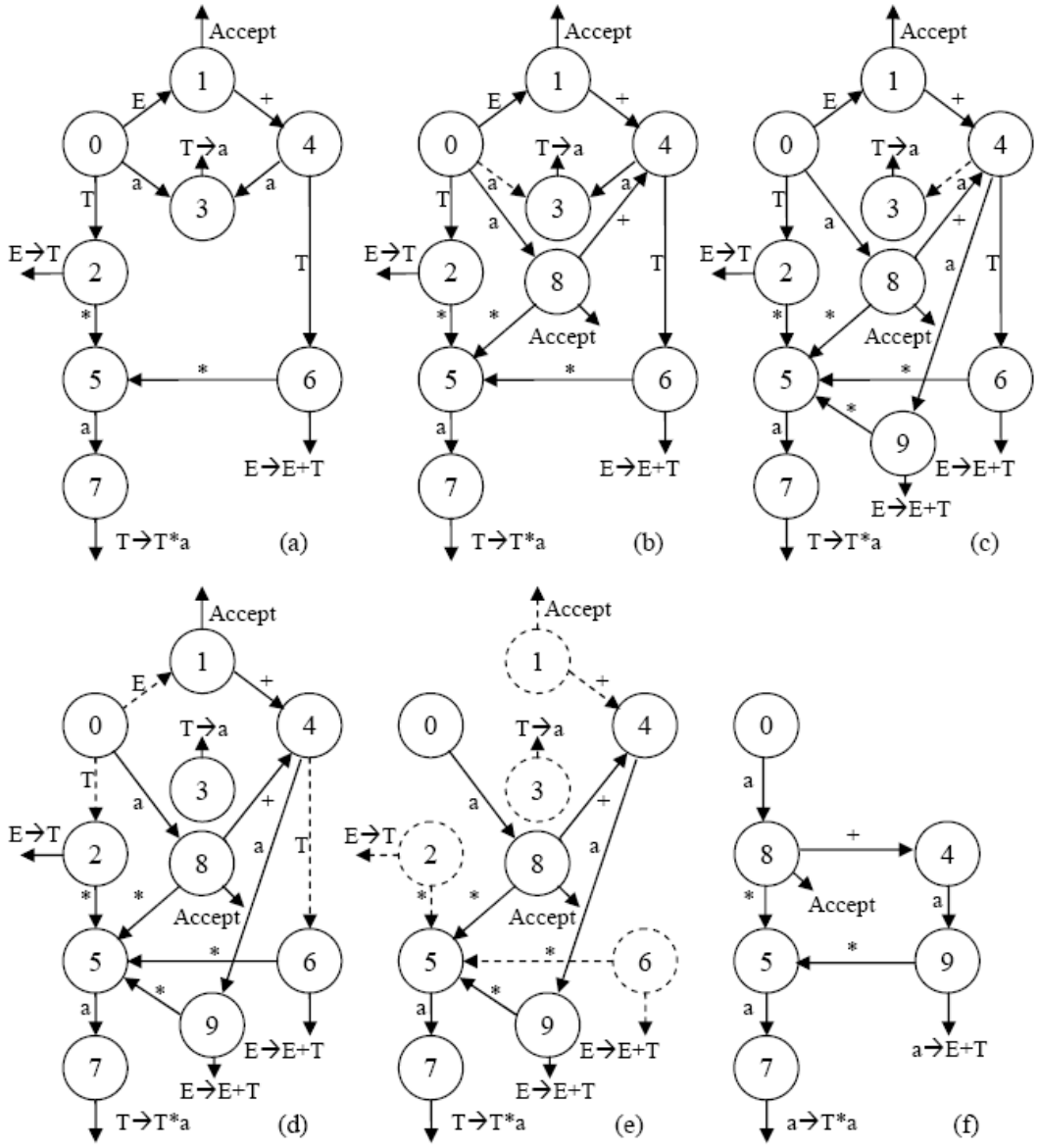


Figure 4.2: Unit Production Elimination on the parsing machine of grammar G3.1

(a) Original parsing machine. (b) Combine states 1, 2 and 3 to state 8. Remove link 0 → 3 because there can be only one a-successor for state 0. (c) Combine states 3 and 6 to state 9. Remove link 4 → 3 because there can be only one a-successor for state 2. (d) Remove transitions corresponding to LHS of unit production: E, T. (e) Remove all states unreachable from state 0, and remove their associated action links. (f) Replace LHS of reductions to corresponding leaf.

4.4.2 Implementation Issues

Implementation using the parsing table

The unit production elimination procedure seems complicated in the parsing machine automata. However it is easy to manipulate the parsing table to achieve the same effect.

Figure 4.3 is an example showing this process.

(a) shows the original LR(1) parsing table.

(b) and (c) shows how to combine states 1, 2 and 3 into state 8, and combine states 3 and 6 into state 9. The unit productions associated with states 2 and 3 are replaced by accept, shift and non-unit production reduction actions of other states in the combination process. The combination is done by copying actions of an old state into the combined state. If an action already exists at a location, then ignore the one that belongs to the unit production. A concern may be raised that two actions that both are not unit-productions may get in conflict. Such situation can never happen. Seen Theorem 4.4.1 and its proof (on page 61).

(d) removes transitions corresponding to LHS of unit productions. This is equivalent to remove goto actions of all non-terminals that are parent nodes in the multi-rooted tree. It can be noted that new combined states will not have such goto actions, and so can save the work on new combined states. As a matter of fact, this step is not even necessary for a parsing table implementation, because after getting all the parent symbols we can just ignore to output their action when writing the final parsing table.

(e) remove all states unreachable from state 0 and associated links. This is easily done by starting from state 0, recursively find those reachable states, then remove states(rows) not on the list from the parsing table. Again, the only thing we actually need is to find those reachable states and we don't really need to remove the rows corresponding to the unreachable states. We can just ignore these rows when writing the final parsing table.

(f) The last step is not conducted on the parsing table, but on the grammar rules data structure. This step is also not necessary, since in the parse engine those LHS symbols just serve as place holders when popping symbols from the symbol stack upon reductions. This step is conducted only to show the change to the human reader.

State	$	+	*	a	E	T
0	0	0	0	s3	g1	g2
1	a0	s4	0	0	0	0
2	r2	r2	s5	0	0	0
3	r4	r4	r4	0	0	0
4	0	0	0	s3	0	g6
5	0	0	0	s7	0	0
6	r1	r1	s5	0	0	0
7	r3	r3	r3	0	0	0

(a) The original parsing table.

State	$	+	*	a	E	T
0	0	0	0	**s8**	g1	g2
1	a0	s4	0	0	0	0
2	r2	r2	s5	0	0	0
3	r4	r4	r4	0	0	0
4	0	0	0	s3	0	g6
5	0	0	0	s7	0	0
6	r1	r1	s5	0	0	0
7	r3	r3	r3	0	0	0
8	**a0**	**s4**	**s5**	**0**	**0**	**0**

(b) Combine states 1, 2 and 3 to state 8. Notice how accept and shift actions overide reductions, where the reductions are associated with unit productions.

State	$	+	*	a	E	T
0	0	0	0	**s8**	g1	g2
1	a0	s4	0	0	0	0
2	r2	r2	s5	0	0	0
3	r4	r4	r4	0	0	0
4	0	0	0	**s9**	0	g6
5	0	0	0	s7	0	0
6	r1	r1	s5	0	0	0
7	r3	r3	r3	0	0	0
8	**a0**	**s4**	**s5**	**0**	**0**	**0**
9	**r1**	**r1**	**s5**	**0**	**0**	**0**

(c) Combine states 3 and 6 to state 9. Note that r1 (E→E+T) overides r4 (T→a) since r4 is a unit production.

State	$	+	*	a	E	T
0	0	0	0	**s8**	**0**	**0**
1	a0	s4	0	0	0	0
2	r2	r2	s5	0	0	0
3	r4	r4	r4	0	0	0
4	0	0	0	**s9**	0	**0**
5	0	0	0	s7	0	0
6	r1	r1	s5	0	0	0
7	r3	r3	r3	0	0	0
8	**a0**	**s4**	**s5**	**0**	**0**	**0**
9	**r1**	**r1**	**s5**	**0**	**0**	**0**

(d) Remove transitions corresponding to LHS of unit productions. Just find those columns corresponding to parent nodes in multi-rooted trees and clear the entries: g1, g2, and g6.

State	$	+	*	a	E	T
0	0	0	0	**s8**	**0**	**0**
~~1~~	~~a0~~	~~s4~~	~~0~~	~~0~~	~~0~~	~~0~~
~~2~~	~~r2~~	~~r2~~	~~s5~~	~~0~~	~~0~~	~~0~~
~~3~~	~~r4~~	~~r4~~	~~r4~~	~~0~~	~~0~~	~~0~~
4	0	0	0	**s9**	0	**0**
5	0	0	0	s7	0	0
~~6~~	~~r1~~	~~r1~~	~~s5~~	~~0~~	~~0~~	~~0~~
7	r3	r3	r3	0	0	0
8	**a0**	**s4**	**s5**	**0**	**0**	**0**
9	**r1**	**r1**	**s5**	**0**	**0**	**0**

(e) Remove all states unreachable from state 0 and associated links. This is easily done by starting from state 0, recursively find those reachable states, then remove states (rows) not on the list from the parsing table.

(f) This last step is not conducted on the parsing table, but on the grammar rules.

Figure 4.3: Applying Unit Production Elimination on the parsing table

Theorem 4.4.1. *In the parsing table implementation of unit production elimination, assume* T_1, T_2, ..., T_n *are the successor states of state S by transition of tokens* $x_1, x_2, ..., x_n$ *respectively, where we have the unit production relationship:* $x_1 \Rightarrow x_2 \Rightarrow \cdots \Rightarrow x_n$ *Then any two of the states* $T_1, T_2, \ldots, T_n$ *will not have the same action on the same token.*

Proof. Assume two states T_a and T_b have the same action on the same token y. Let the transition symbol from S to T_a be x_a, and the transition symbol from S to T_b be x_b, as shown in Figure 4.4. Let's also assume that the relationship between x_a and x_b is $x_a \Rightarrow \cdots \Rightarrow x_b$. Then the input string $\alpha \bullet x_b y\beta$, where α is the string before the marker $\bullet$, and b is the scanned symbol, will have a shift/reduce conflict here. This is because we can shift by x_b to state T_b, then shift be action y, but we can also reduce following the relationship chain $x_a \Rightarrow \cdots \Rightarrow x_b$ to obtain $\alpha \bullet x_a y\beta$, then shift by x_a to state T_a, and then shift by action y. However, the condition that the unit production elimination algorithm can be applied is that the grammar is a LR(1) grammar, and no shift/reduce conflict can exist in a LR(1) grammar's parsing machine. This shows that our assumption that the two states T_a and T_b have the same action on the same token y is invalid. □

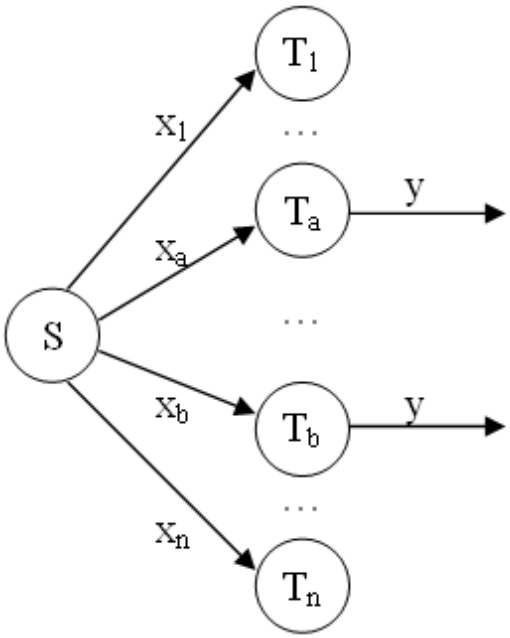


Figure 4.4: Assume states T_a and T_b have the same action on token y

A conclusion that can be derived from Theorem 4.4.1 is: In the parsing table implementation of the unit production elimination, for those successor states obtained from a state by transitions by symbols involved in a chain of unit productions, we can combine these successor states without worrying about conflicts caused by different actions (neither is a unit-production reduction, since if one is then we can use the other for the action under this token) under the same token.

Implementation algorithms on parsing table

This section shows the actual algorithms in the implementation. These algorithms manipulate the parsing table, as shown in Figure 4.3 (on page 60).

Algorithm 4.19 shows the entire UPE flow process.

Algorithm 4.20 shows how to build the multi-rooted tree. A way of constructing a multi-rooted tree is used to find out all the leaves in a grammar. To implement this data structure, an array MRT is used. MRT is an array of leaf nodes. Each leaf has a pointer that points to its parent nodes, if any.

Algorithm 4.21 shows the first two steps in UPE algorithm. These two can't be separated actually.

Algorithm 4.22 shows step 3 in UPE algorithm.

Algorithm 4.23 shows step 4 in UPE algorithm.

Algorithm 4.24 shows step 5 in UPE algorithm.

Algorithm 4.19: UPE()

Input: The parsing table of a grammar G'

Output: The parsing table with unit productions eliminated

```
1 build_multi_rooted_tree();
2 UPE_step1and2();
3 UPE_step3();
4 UPE_step4();
5 UPE_step5();
```

Algorithm 4.20: build_multi_rooted_tree()

Input: grammar G'

Output: A multi-rooted tree MRT built from the unit productions of G'

```
1  Let MRT be the multi-rooted tree, which is empty at the beginning;
2  foreach grammar rule r : [X → Y] in G' do
3      if r is a unit production then
4          nLHS ← the node for symbol X in the MRT;
5          nRHS ← the node for symbol Y in the MRT;
6          if only node nLHS is found then
7              Insert Y to MRT as a child of node X;
8          else if only nRHS is found then
9              Insert X to MRT as a parent of node Y;
10         else if both nLHS and nRHS are NOT found then
11             Insert X and Y to MRT as a new tree, X is the parent of Y;
12         else
13             Insert the parent/child relation of X and Y in MRT;
```

Algorithm 4.21: UPE_step1and2()

```
1 foreach state S in the parsing table do
2     foreach leaf x in the multi-rooted tree MRT do
3         find the set T of successor states of S, where each state in T is a y-successor
          of S, where y is a parent node of leaf x;
4         if T ≠ ∅ then
5             let R be the state combined from states in T;
6             if R does not exist then
7                 create a new state R, which is obtained by combining states in T;
8             insert actions of states in T to R in the parsing table;
```

Algorithm 4.22: UPE_step3()

```
1 foreach row in the parsing table do
2     foreach non-terminal Y do
3         if Y is a parent node in the multi-rooted tree then
4             clear the goto action on Y;
```

Algorithm 4.23: UPE_step4()

```
1 allocate an empty array ReachableStates[];
2 put each shift action target state of state 0 to ReachableStates[];
3 while next state S in ReachableStates[] exist do
4     foreach shift action target state T of state S do
5         if T is not in ReachableStates[] then
6             add T to the end of ReachableStates[];
7 go through the parsing table, label each state not in ReachableStates[] as removed;
```

Algorithm 4.24: UPE_step5()

```
1 foreach grammar rule r in grammar G' do
2     if r is unit production: [X → Y] then
3         if X is a parent in the multi-rooted tree then
4             change X to an associated leaf symbol;
```

4.5 Extension To The Unit Production Elimination Algorithm

4.5.1 Introduction and the Algorithm

It can be noted that after removing unit productions, the parsing machine can possibly contain repeated states (with the same actions). These repeated states can be combined to result in a more compact parsing machine. This is a natural extension of Pager's unit production elimination algorithm.

Definition 4.5.1. *Same-action states* are those states in a parsing machine that have exactly the same actions (accept, shift, goto and reduce) on each token symbol (including both terminals and non-terminals).

The following is the algorithm to remove redundant copies of *same-action states*, which is an extension to the unit production elimination algorithm.

Algorithm 4.25: UPE_Ext()

Input: Parsing Machine M

Output: A parsing machine M' where all the same-action states in M are removed

```
1  let Shift(X, k) → Y be a Shift transition from state X to state Y on token symbol k;
2  foreach state S in M do
3      find the set Σ of all the same-action states of state S;
4      foreach state S' in Σ do
5          foreach Shift(R, k) → S' in M do
6              replace it by Shift(R, k) → S;
7          end
8      end
9      remove Σ from M ;
10 end
```

In practice, this algorithm is $O(1)$ in space and does not increase the amount of memory used, since it operates on the existing parsing machine. But it takes quite a large percentage of the execution time, because it looks through each entry of the entire parsing table for each state. The worst

time performance is $O(n^2 * m)$, where n is the number of states, and m is the number of tokens (both terminals and non-terminals).

Derivation of $O(n^2 * m)$ using the best implementation scenario: the step of finding the set of all the same-action states can be done in $O(n)$ time by inserting all states into a hash table based on its actions. Since each state has m actions, this actually is $O(n * m)$. Then for the next step of replacing relevant transitions, assume those same-action states are $\{S_i | i \in [0, n]\}$, assume the number of actions transit to S_i is X_i, it is obvious that in the worst case all the other states transit to X_i and all the actions of each state transit to X_i (although that is unlikely in practice), so:

$$0 \leq X_i \leq n * m \tag{4.5.1}$$

then the number of transitions to replace is

$$0 \leq X_1 + \ldots + X_k \leq n * (n * m) \tag{4.5.2}$$

thus

$$O(n * m + n * (n * m)) = O(n * m * (1 + n)) = O(n^2 * m). \tag{4.5.3}$$

This is the theoretical upper bound, which happens when in the parsing machine each state transits to all the other states for all the lookaheads. This is impossible in practice. A empirical study maybe can be used to show the situation in practice.

One concern of the unit production elimination algorithm is that it was designed for LR(k) grammars. For non-LR(k) grammars, more conflicts, including shift/shift conflicts, can be derived. Under such situations, the unit production elimination algorithm and this extension should not be used.

Example. Given grammar G4.1 [41]:

S → d i A
A → A T | ϵ
T → M | Y | P | B
M → r | c
Y → x | f
P → n | o
B → a | e

In Figure 4.5 (on page 69), (a) is the parsing machine obtained using the practical general method, (b) is the parsing machine after applying the unit production elimination algorithm based on (a),

(c) is the parsing machine after applying the extension to the unit production elimination algorithm based on (b). In (b), states 18 to 25 all have the same action A $\rightarrow$ AT for each of the lookahead symbols in $\Sigma = \{a, c, e, f, n, o, r, x, \dashv\}$ Thus states 18 to 25 are same-action states and they can be combined into one state, i.e., state 18 in (c).

In this example, the parsing machine in (a) has 18 states, in (b) has 13 states, and in (c) has only 6 states. So by applying the extension algorithm after the unit production elimination, (13-6) / 13 = 54% reduction in parsing machine size is achieved.

The following table compares the number of states, shift/goto, reduction and accept actions in the parsing machine after applying each of the practical general method (PGM), unit production elimination (UPE) and UPE extension (UPE Ext) algorithms.

	state	shift/goto	reduction	accept
(a) PGM	18	17	15	1
(b) UPE	13	12	10	1
(c) UPE Ext	6	5	3	1

Example. Given grammar G4.2 [41]:

$E \rightarrow E + T \mid T$
$T \rightarrow a \mid n \mid (E)$

Figure 4.6 (on page 70) shows the parsing machines. The LR(1) Parsing machine is in (a). The parsing machine after applying UPE algorithm is (b). Obviously states 10 and 11, 12 and 13, 14 and 15 are identical pairs because they have the same actions, and can be combined together. The parsing machine after applying UPE Ext algorithm is (c). The following table compares the number of states, shift/goto, reduction and accept actions in the parsing machine after each step.

	state	shift/goto	reduction	accept
(a) PGM	10	17	5	1
(b) UPE	10	16	3	2
(c) UPE Ext	7	9	2	1

Thus we see substantial decrease in the number of states, transitions and reductions in the LR(1) parsing machine by using the UPE Ext algorithm.

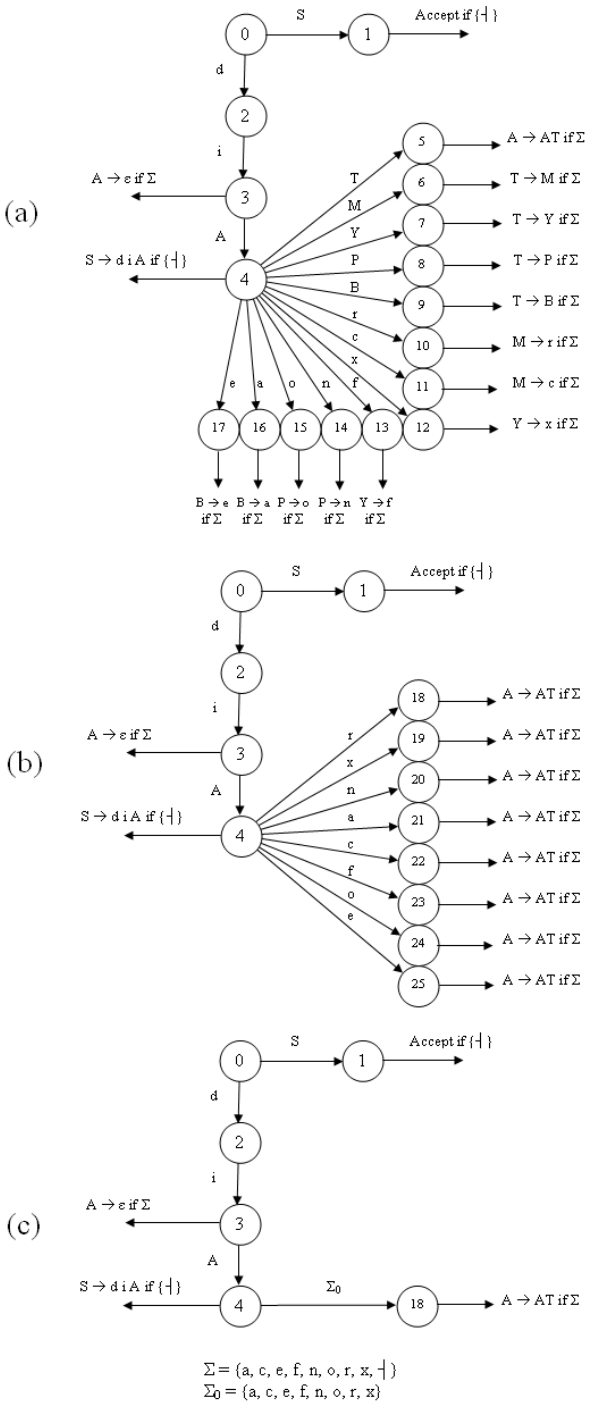


Figure 4.5: Remove same-action states after unit production elimination

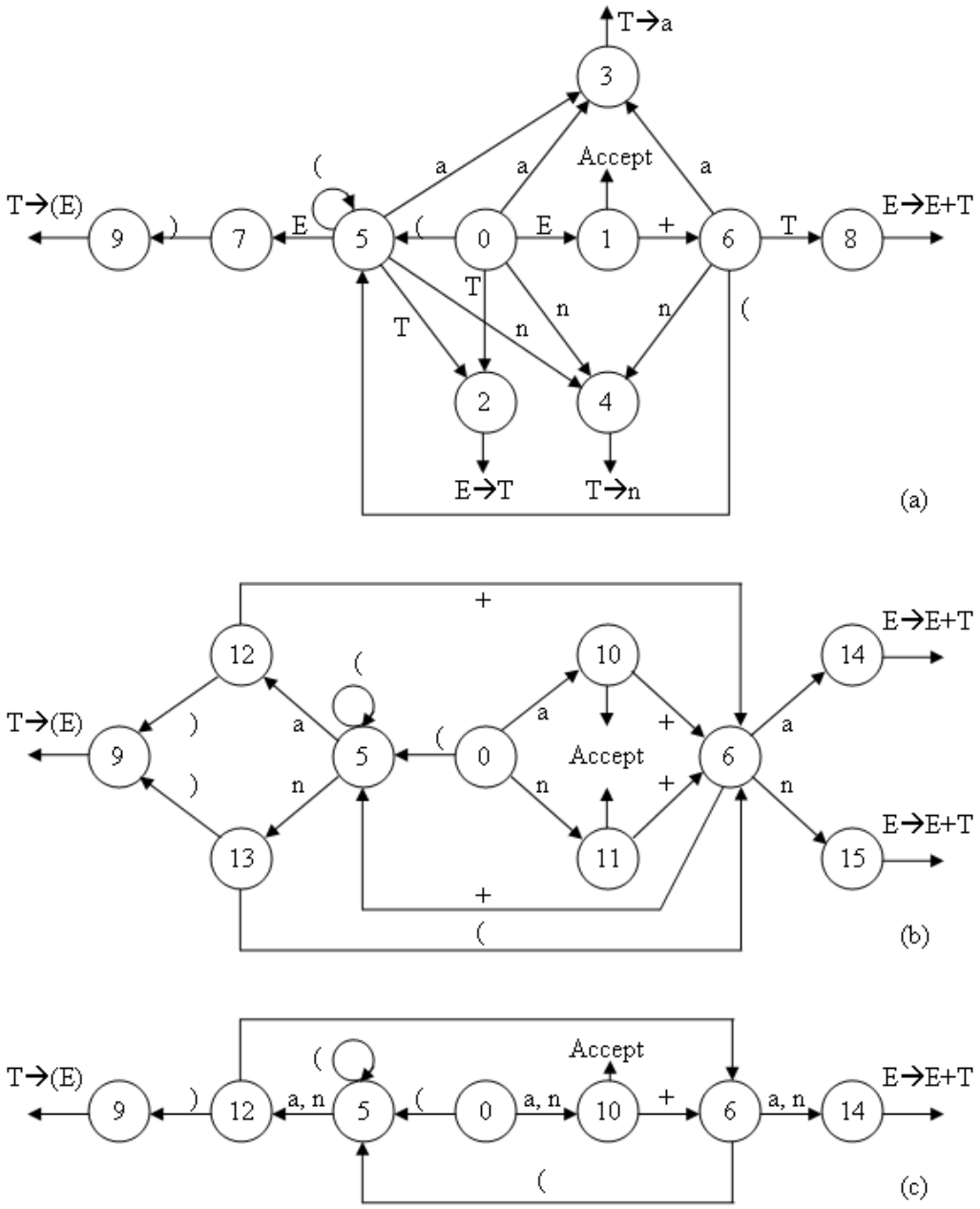


Figure 4.6: Apply UPE and UPE Ext on Grammar G4.2

4.5.2 Implementation Issues

Implementation can be on the level of 1) the parsing machine automata, or 2) the parsing table. It was found that manipulating the parsing table is easier. In Hyacc since the UPE algorithm was implemented by manipulating the parsing table, the extension algorithm to UPE is also implemented this way. The alternative of working on the level of the parsing machine automata, however, can possibly be more efficient from a high level point of view.

If the implementation uses the parsing table, then another optimization can be used. It seems that all *same-action states* are adjacent to each other (rows adjacent in the parsing table). Using this observation, it is faster to go through the cycle.

To see the validity of this optimization, we need to check how same-action states happen. In our examples, such same-action states come from grammar rules like this: X $\rightarrow$ t_1 | t_2 | ... t_n, where t_i is a terminal symbol, i = 1, 2, ... n. Since the UPE algorithm first builds a multi-rooted tree and then combines states involved in unit productions in a round-robin approach (Algorithm 4.21), that procedure results in the occurrence of same-action states, and same-action states obtained this way are always adjacent to each other. So far this is the only source of same-action states we know. There is no way to rule out other possible sources of same-action states so far, but such situation should be rare. This is because same-action states have the same actions usually because they have the same set of (core) configurations, which is the case for same-action states obtained in the UPE procedure above (such resulted same-action states actually are different by the t_i symbols in the configurations, but such t_i symbols can all be replaced by X). Otherwise states with the same set of (core) configurations would have been combined in the PGM algorithm. If two states have different sets of (core) configurations but the same actions, that is quite unusual. Therefore we argue that even though there is no way to rule out other sources of same-action states, the UPE procedure should be the primary source of same-action states, if not the only source.

Now refer back to Algorithm 4.25 (on page 66), taking advantage of the adjacency property of same-action states, the cost savings is in line 3: "find the set Σ of all the same-action states of state S". Previously we needed to search through the entire parsing table (at least from the current row to the end of the parsing table), but now only need to check the next few rows (from the current row on until the next different row). This does not change the $O(n^2 * m)$ nature since line 3 is not the dominant operation in determining the cost, but it does help.

4.6 Pager's Lane-tracing Algorithm

4.6.1 The Algorithm

The lane-tracing algorithm was introduced in section 2.2.3 (on page 9). It contains two phases, as shown in Figure 4.7. The first phase starts from inadequate states (contains reduce/reduce conflicts) in the LR(0) parsing machine, and traces back the configurations until a configuration where only non-NULL contexts are generated. Phase 1 ends up with a LALR(1) parsing machine. If this resolves the reduce/reduce conflicts then we stop here. Otherwise, the second phase is used to split states to resolve the conflicts, and results in a LR(1) parsing machine.

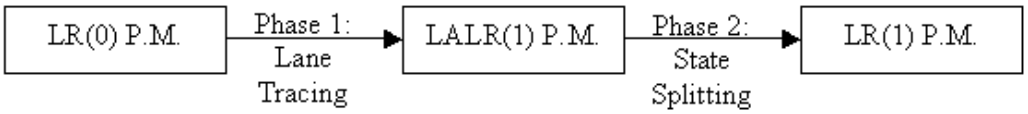


Figure 4.7: The Two Phases of Lane-Tracing Algorithm

The lane-tracing algorithm can also be shown by the following pseudo-code:

Algorithm 4.26: lane_tracing()

```
1 lane_tracing_phase1();
2 resolve_LALR1_conflicts();
3 if not all inadequate states are resolved then
4     lane_tracing_phase2();
5     resolve_LALR1_conflicts();
```

The lane_tracing_phase1() function just does lane tracing. This is followed by a call to the function resolved_LALR1_conflicts() to resolve reduce/reduce conflicts according to the generated context during the lane-tracing process. When this is done, we check if there are still any unresolved states that contain reduce/reduce conflicts. If so we go ahead with lane_tracing_phase2(), at the end of which we make another call to resolve_LALR1_conflicts().

The algorithm used in the first phase is thoroughly discussed and presented in [47]. But the second phase is only briefly mentioned in the same source. This work implemented the second phase using two approaches, which are discussed in details here.

It should be noted that the dragon book describes two ways of generating an LALR(1) parsing machine. The lane-tracing algorithm phase 1 (here it is the call to lane_tracing_phase1() followed by a call to resolve_LALR1_conflicts()) does the same thing. But this fact seems to have been ignored by most literature. This work implements LALR(1) in Hyacc using phase 1 of the lane-tracing algorithm. The result on parsing grammars is compared to Bison (see Chapter 5) and shows the same outcome, which validates this approach.

4.6.2 Lane-tracing Phase 1

The Phase 1 algorithm basically creates a LALR(1) parsing machine based on the LR(0) parsing machine, and find out the lanes that trace back to the states where conflicting contexts are generated. The paper [47] is a pretty good summation of the algorithm. It lacks details on Phase 2 of lane-tracing though, which will be discussed in the next section.

Example. Given grammar G4.3 [45]:

E → a A d | b A c | a B c | b B d
A → e A | e
B → e B | e

The LR(0) parsing machine is shown in Figure 4.8:

In state 6, final configuration 1 $[A \rightarrow e \bullet]$ and final configuration 3 $[B \rightarrow e \bullet]$ both reduce. It is a reduce/reduce conflict. Thus we need to trace back these two configurations to generate contexts for them.

How we trace these two configurations are shown in Figure 4.9, where (a) shows the lanes generating contexts for $[A \rightarrow e \bullet]$, and (b) shows the lanes generating contexts for $[B \rightarrow e \bullet]$. Use (a) as example. In state 2, configuration 1 in state 6 can be traced back to its originator in state 2, which is configuration 4 in state 2 ($[A \rightarrow \bullet\ e]$). In state 2, configuration 4 does not generated any contexts, so we trace back to configuration 1 ($[E \rightarrow a \bullet A\ d]$), which has string "d" after scanned symbol A and generates context set $\{d\}$. So we stop tracing here, and add context $\{d\}$ to configuration 4, which then propagates to configuration 1 in state 6. Another lane traces back to configuration 1 in state 3, which generates the context set $\{c\}$. As the result of lane-tracing, configuration 1 in state 6 now has context set $\{c, d\}$. Similarly, configuration 3 in state 6 has context set $\{c, d\}$ after lane-tracing.

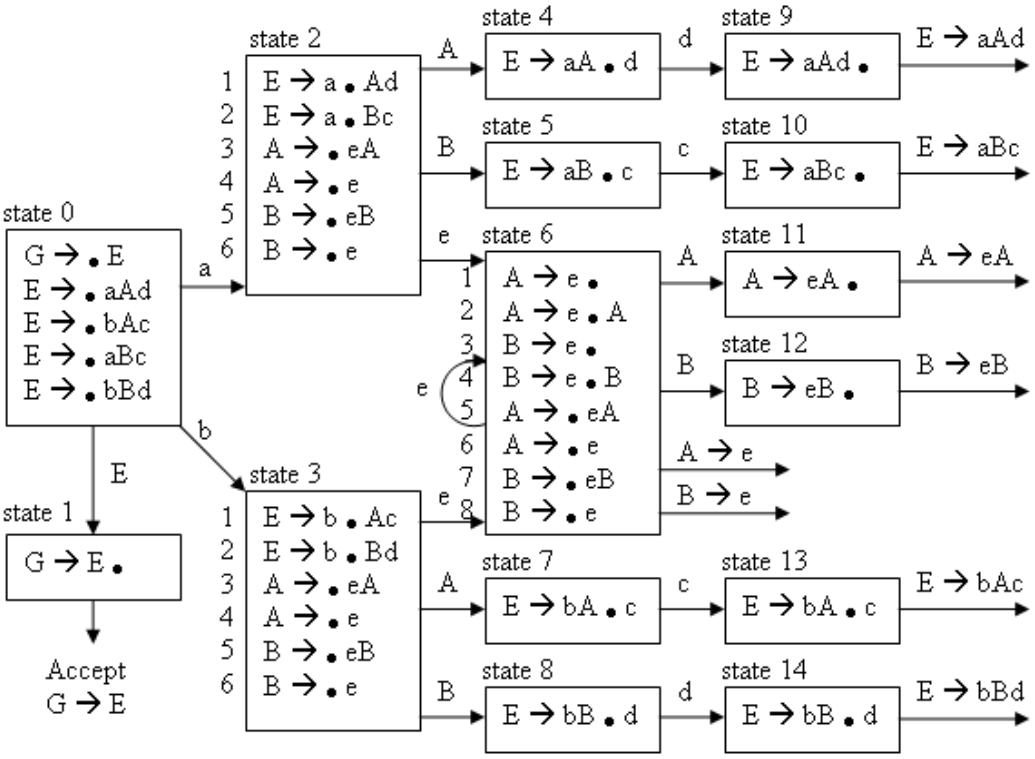


Figure 4.8: LR(0) parsing machine for grammar G4.3

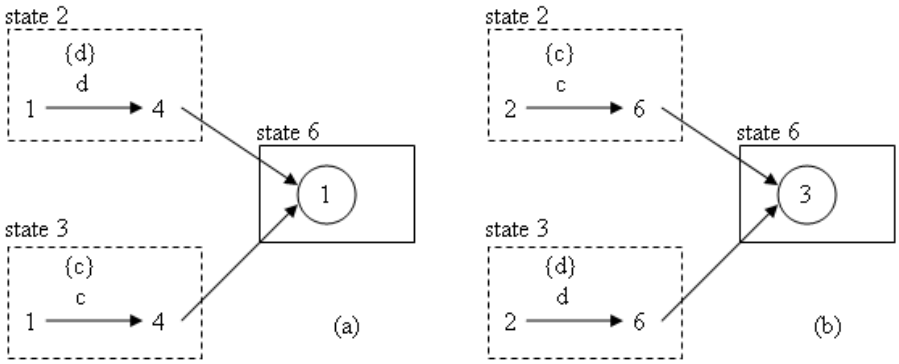


Figure 4.9: Lane tracing on conflict configurations

After tracing using the phase 1 algorithm, the LALR(1) parsing machine we obtain is shown in Figure 4.10 (only relevant states are shown, the rest ignored as they keep the same). Now that in state 6 both configurations 1 and 3 have the same context set $\{c, d\}$, the reduce/reduce conflict still exist. So we need to go to phase 2 of the lane-tracing algorithm to see if it is possible to resolve this conflict by splitting states.

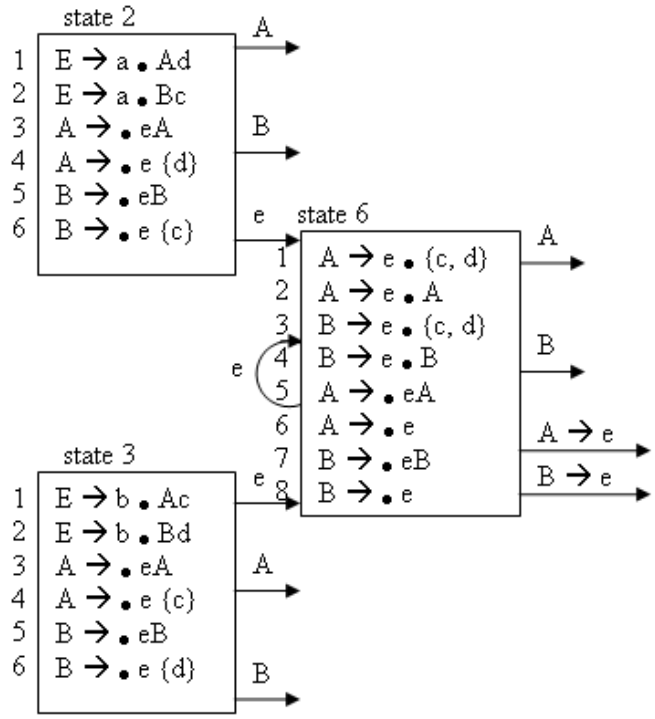


Figure 4.10: LALR(1) parsing machine for G4.3 generated by lane tracing

4.6.3 Lane-tracing Phase 2

The purpose of phase 2 is to split the states that cause the reduce/reduce conflicts. There are two approaches for this as discussed here. The first is based on the practical general method (PGM) as suggested in [47]. The second is based on a lane-tracing table, which is a table constructed from the conflicting lanes [50].

Phase 2 process is:

Algorithm 4.27: lane_tracing_phase_2()

```
1 get_LaneHead_list();
2 phase2_PGM() or phase2_laneTable();
```

4.6.4 Lane-tracing Phase 2 First Step: Get Lanehead State List

The first step of phase 2 is to get the lanehead state list.

After phase 1, we have obtained a parsing machine with conflict states. Then we need to find out a list of states, from which lanes start and eventually lead to the unresolved states. We need to find out these states, because then we need to regenerate the involved states to remove the reduce/reduce conflicts.

We obtain these lane head states by tracing back conflicting lanes a second time. Actually a list of the lane head states can be obtained in phase 1 tracing. The reason we do not do it there but now, is that we don't know back then which lane head states eventually lead to inadequate states. Suppose there are 1000 inadequate states at the beginning, there could be 2000 lane head states for all these inadequate states. But 999 of these inadequate states are resolved after phase 1, with only one inadequate state left and 2 lane head states lead to this inadequate state. It is quite obvious that we only need these 2 lane head states and can ignore the other 1998. It is true that by getting lane head states a second time can double the computation, but it is at most twice as expensive. Besides the fact that the second trace may work on a fewer number of states, the second trace back also can take advantage of the first one: we can mark those configurations where traces end and don't need a second computation on the contexts to determine if we need to stop tracing at a configuration.

Algorithm 4.28: get_laneHead_list()

```
1  foreach state s with reduce/reduce conflict do
2      let A = getStateConflictLaneHead(s);
3      laneHead_List = laneHead_list ∪ A;
4  remove pass_thru states from laneHead_list;
```

Algorithm 4.29: getStateConflictLaneHead(S)

Input: state S

```
1  foreach final configuration C of state S do
2      trace_back(NULL, c);
```

Algorithm 4.30: trace_back(o, c)

Input: Configuration c; An originator configuration of c: o

```
1  c.LANE_CON = 1;
2  if c.LANE_END == true then
3      add c's owner state to laneHead_list;
4      return laneHead_list;
5  else if c has no originators then
6      return laneHead_list;
7  else
8      foreach originator o of c do
9          set_transitors_pass_thru_on(c, o);
10         if o.LANE_CON == 0 then
11             if c.owner != o.owner then
12                 c.owner.PASS_THRU = 1;
13             trace_back(c, o);
14     return laneHead_list;
```

Algorithm 4.31: set_transitors_pass_thru_on(cur_config, o)

Input: Current configuration cur_config; Originator configuration o

Output: Each intermediate state is labeled as PASS_THRU

```
1 foreach transitor c of cur_config do
2     if c is one of the originators of o then
3         LT_tbl_entry_add(c.owner, cur_config.owner);
4         if c.owner != cur_config.owner AND c.owner != o.owner then
5             c.owner.PASS_THRU = 1;
6         set_transitors_pass_thru_on(c, o);
7 if o.owner == cur_config.owner then
8     getTheContext(o);
```

4.6.5 Lane-tracing Phase 2 Based on PGM

Once the lanehead state list is obtained, regenerating states seems mechanical. The idea is to start from lanehead states, regenerate those states on the conflicting lanes. Note there is no need to regenerate states not on the conflicting lanes. Conflicting lanes are those lanes we traced in phase 1. When new states are generated we combine them using the PGM algorithm.

The algorithm Phase2_PGM() below gives the procedure.

Algorithm 4.32: Phase2_PGM()

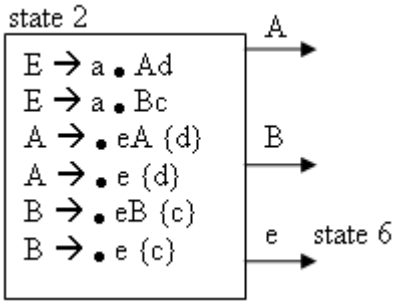
```
   Input: laneHead_list
 1 foreach state s in laneHead_list do
 2     getClosure(s);
 3     updateStateReduceAction(s);
 4     coll = getStateSuccessors(s);
 5     foreach state Y in coll do
 6         let Y0 be the original corresponding successor of s;
 7         if Y0.PASS_THRU == TRUE then
 8             if Y0.regenerated == false then
 9                 regenerateStateContext(Y0, Y);
10                 let Y0.regenerated = true;
11                 If Y0 is not in laneHead_list, add it to laneHead_list;
12             else
13                 if Y0 and Y are weakly compatible then
14                     combine context of Y0 to Y;
15                     add Y0 to laneHead_list if not yet on the list;
16                 else
17                     add new split state Y to the entire states list;
18     if any new state was added to the state list then
19         do PGM on these new states;
```

An example is given below.

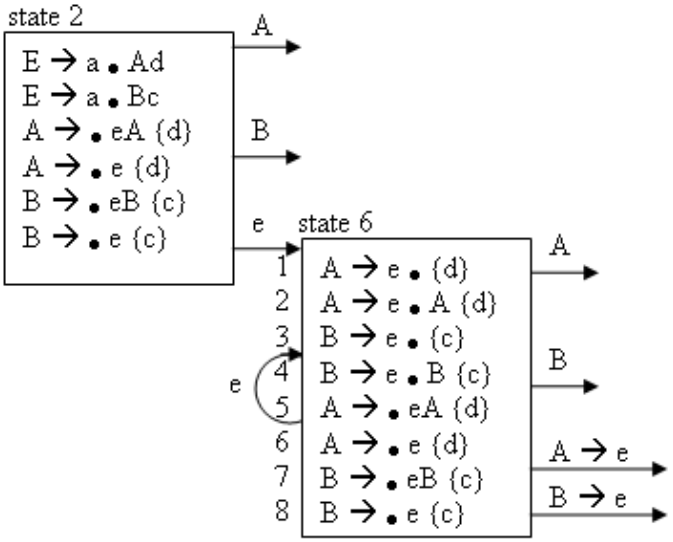
Example. For grammar G4.3, after lane-tracing phase 1 we have obtained the LALR(1) parsing machine as shown in Figure 4.8 (on page 74). Now we need to split states based on the PGM algorithm. We already obtained the set of lane head states, which is $\{2, 3\}$. Following the algorithm Phase2_PGM, the following steps apply:

At the beginning, the laneHead_list is $\{2, 3\}$. The set of states that are on the conflicting lanes (where PASS_THRU is on) is $\phi = \{2, 3, 6\}$.

Step 1. Get state 2 from lanehead_list. Get its closure:

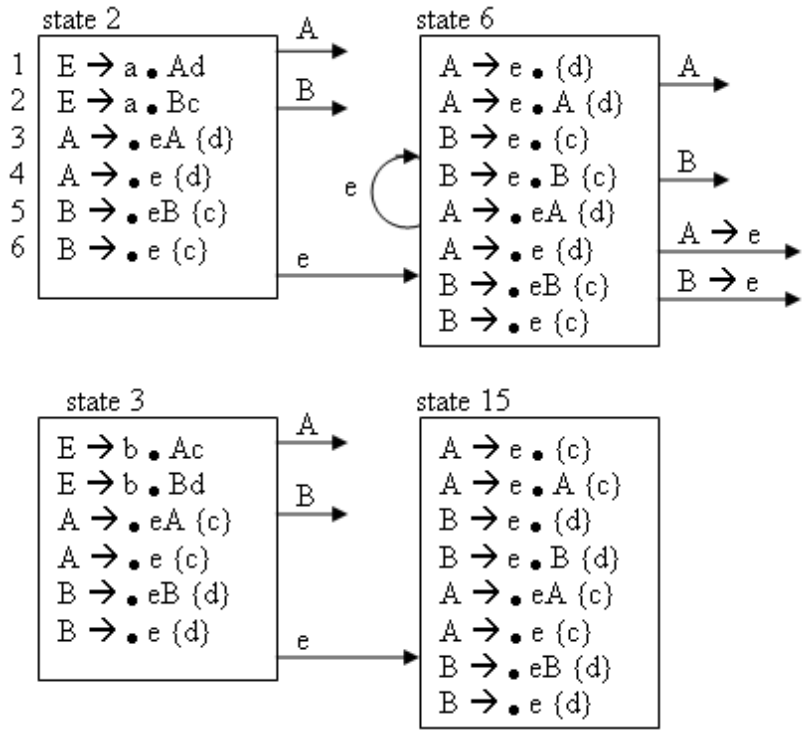


Then we get the successors of state 2 on tokens A, B and e. Only e successor state 6 is on conflicting lane. At this time state 6's *regenerated* label is off, so we regenerate its context to replace the old state 6, and set its *regenerated* label on. Since it is not in the lanehead_list, we add state 6 to lanehead_list.



Now lanehead_list is $\{2, 3, 6\}$.

Step 2. Get the next state 3 from laneHead_list, and get its closure:

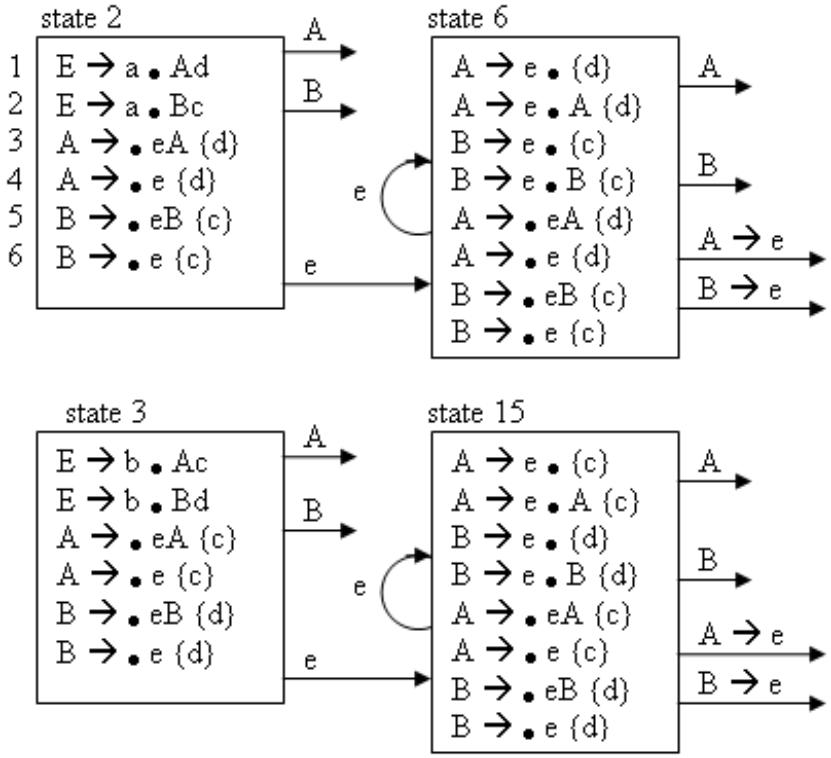


Then we get the successors of state 3 on tokens A, B and e. Only e successor state 6 is on conflicting lane. At this time the *regenerated* label of state 6 is on, so we check if 3's e successor is compatible with the regenerated state 6. We find the answer is no. So we have to make the e successor of state 3 a new state and insert it in the state list. The new state is the 15th state, so we label it as state 15. Now lanehead_list is $\{2, 3, 6\}$.

Step 3. Get the next state 6 from lanehead_list, and get its closure. Here it has no change. In other cases, it is possible that there are compatible states combined into it, so the context will change after getting closure. State 6 has successors on tokens A, B and e. Only the e successor is in the conflict lane, which is state 6 itself. Since the regenerated label is on, we check if the new copy of state 6 is compatible with state 6. Since it is compatible, we combine them (actually no change in this case). Since state 6 is already in lanehead_list, we don't insert it again.

At this time the lanehead_list is exhausted, so we exit the loop on lanehead_list.

Step 4. Next we need to apply the PGM algorithm on the new states added to the state list. We have added one such state, which is state 15. After applying the PGM algorithm, we obtain the following state machine. Here the A and B successors of state 15 are the same as those of state 6.



4.6.6 Lane-tracing Phase 2 Based on A Lane-tracing Table

Lane-tracing based on a lane-tracing table is another way of splitting states to remove inadequate states. The idea is that, using the lane-tracing table constructed here, we check the local context information of the regenerated state group to see if there is a need to split. The following algorithm is derived from [50].

Algorithm

Let the actions of the state considered, where conflicts occur, be π_1, π_2, ..., π_r. If a state S contains configurations that for $1 \leq i \leq r$, generate a set of contexts C_i along a lane leading to π_i, then the collection of contexts generated by S is defined as the set $\{(C_i, i)|1 \leq i \leq r\}$.

Note that if the sets $\{C_i|1 \leq i \leq r\}$ are not pairwise disjoint, then the grammar is not LR(1).

The collection of contexts associated with any state S is initially the collection of contexts it generates.

The manner in which regenerated states are combined is as follows.

Let $\{S_1, \ldots, S_t\}$ be a set of connected regenerated states, and let the same collection of contexts associated with each of $S_1, \ldots, S_t$ be $\{(A_i, i)|1 \leq i \leq r\}$. If we now regenerate a state T that is a successor of one of $S_1, \ldots, S_t$, then:

1) If there is an existing copy of state T whose associated collection of contexts is $\{(B_i, i)|1 \leq i \leq r\}$ and the collection $\{(A_i \cup B_i)|1 \leq i \leq r\}$ are pairwise disjoint, then this existing copy of state T is taken as the successor, and the collection of contexts associated with $\{S_1, \ldots, S_t, T\}$ is defined to be $\{(A_i \cup B_i, i)|1 \leq i \leq r\}$.

2) Otherwise, a new copy T' of T is regenerated as the successor, and if the collection of contexts generated by T is $\{(B'_i, i)|1 \leq i \leq r\}$, then the collection of contexts associated with $\{S_1, \ldots, S_r, T'\}$ is defined to be $\{(A_i \cup B'_i, i)|1 \leq i \leq r\}$.

Note that if the sets $\{(A_i \cup B'_i, i)|1 \leq i \leq r\}$ are not pairwise disjoint, then the grammar is not LR(1).

Example. This example is from [50]. Given grammar G4.4:

G → x W a | x V t | y W b | y V t | z W r | z V b | u U X a | u U Y r
W → U X C
V → U Y d
X → k t U X P | k t
Y → k t U Y u | k t
U → U k t | s
E → a | b | c | v
C → c | w | ϵ
P → ϵ

The states involved in the conflicting lanes are as shown in Figure 4.11.

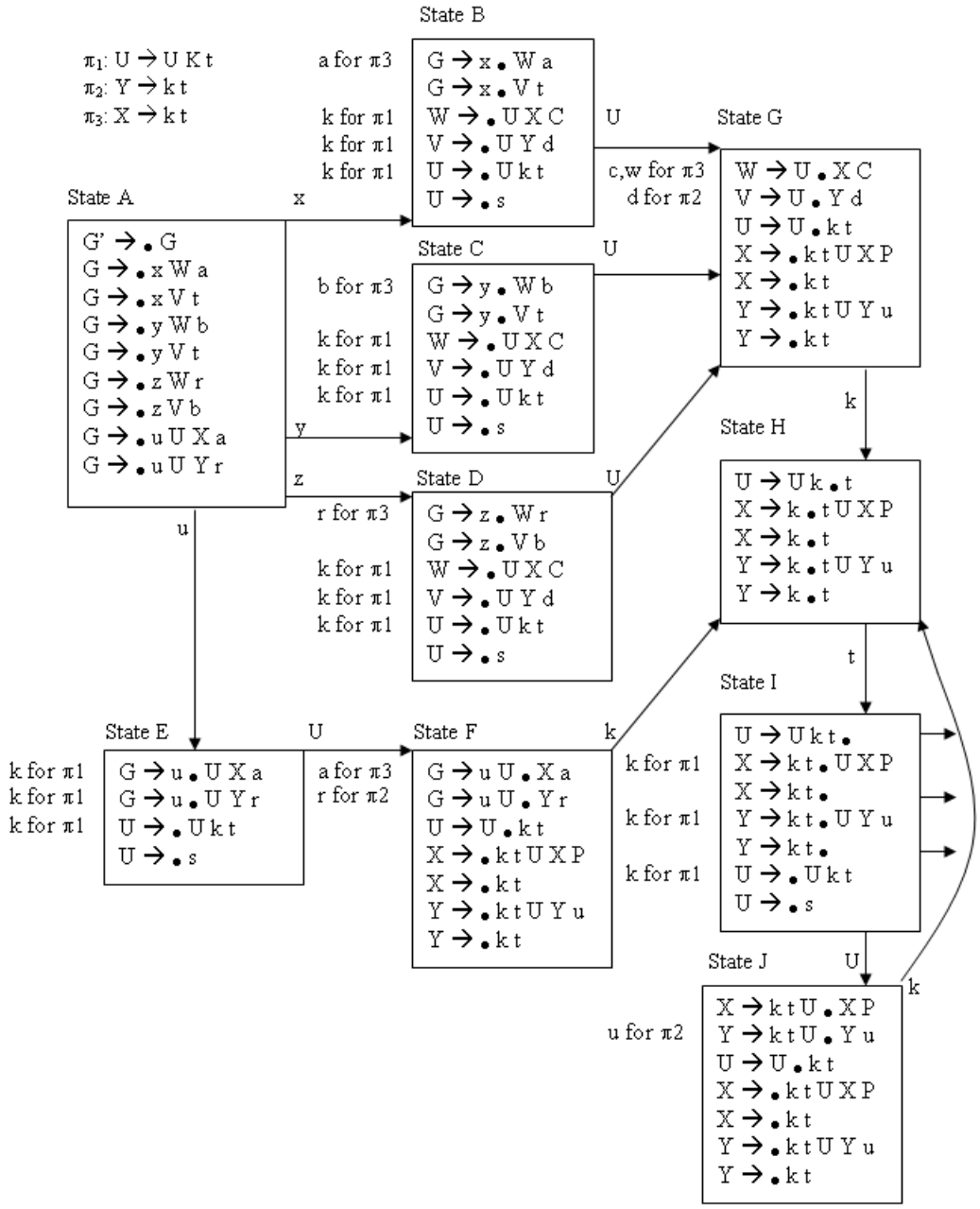


Figure 4.11: Grammar G4.4: states on the conflicting lanes

Figure 4.12 is a depiction of the states involved in traced lanes and how they are connected to each other. The lanes are shown in the forward direction.

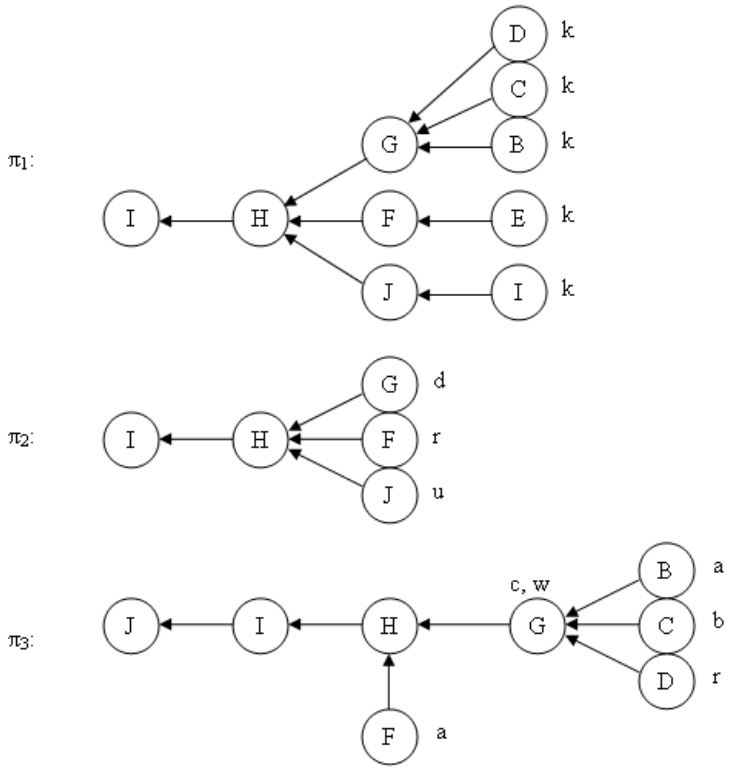


Figure 4.12: Grammar G4.4: conflicting lanes traced in lane-tracing

The information collected is stored into a lane table as in Table 4.2.

State	π_1	π_2	π_3	Connected to
B *	k		a	{G}
C *	k		b	{G}
D *	k		r	{G}
E *	k			{F}
F		r	a	{H}
G		d	c, w	{H}
H				{I}
I *	k			{J}
J		u		{H}

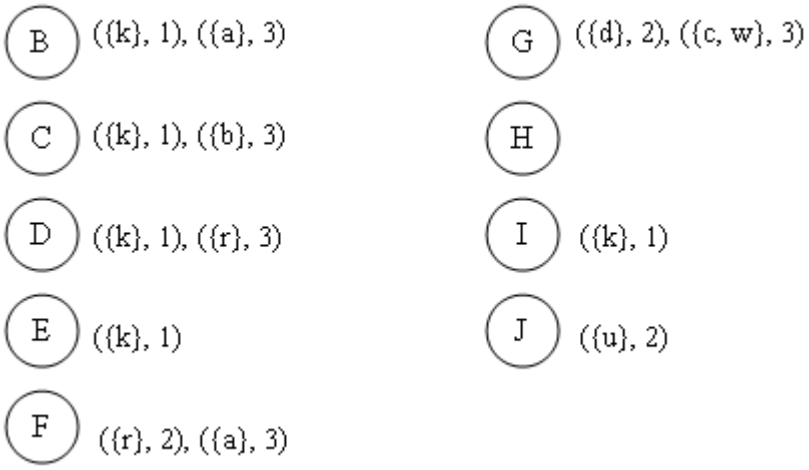
Table 4.2: Grammar G4.4: lane table constructed in lane-tracing

* : means lanes start, but does not pass through the state involved.

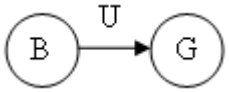
The example of combining regenerated states is given below.

The example sequence of regeneration shown is one where the states along the lanes from state D are regenerated last, as this best illustrates the algorithm. Note that no states other than states B, C, . . . , J are regenerated. For example, the X and Y successors of all the copies of state J are the original X and Y successors of the original state J.

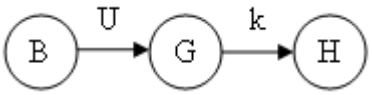
Step 1: Initially show the collection of contexts associated with each state (i.e., the collection of context generated by the state).



Step 2: Start from state B, first add state G to the collection.
The collection of contexts associated with {B, G}: ({k}, 1), ({d}, 2), ({a, c, w}, 3).

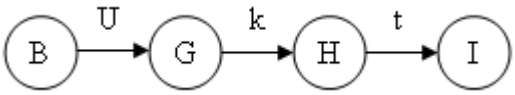


Step 3: Add the successor of state G: state H.
The collection of contexts associated with {B, G, H}: ({k}, 1), ({d}, 2), ({a, c, w}, 3).



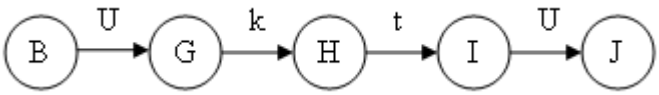
Step 4: Add the successor of state H: state I.

The collection of contexts associated with {B, G, H, I}: ({k}, 1), ({d}, 2), ({a, c, w}, 3).



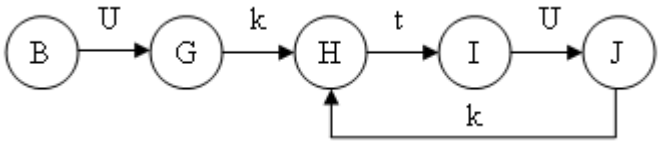
Step 5: Add the successor of state I: state J.

The context sets associated with {B, G, H, I, J}: ({k}, 1), ({d, u}, 2), ({a, c, w}, 3).

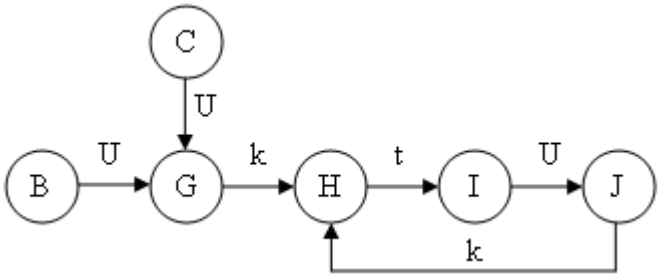


Step 6: Add the successor of state J: state H. State H is already in the set.

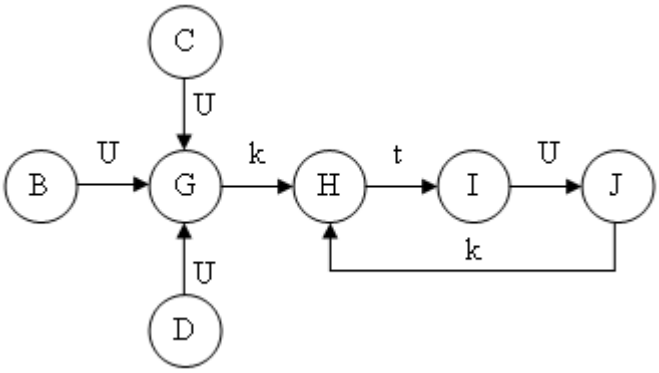
The context sets associated with {B, G, H, I, J}: ({k}, 1), ({d, u}, 2), ({a, c, w}, 3).



Step 7: State H is already in this set of states. So find the next state after B in the table and see if it's possible to add it to this set of states, which is state C.
The context sets associated with {B, C, G, H, I, J}: ({k}, 1), ({d, u}, 2), ({a, b, c, w}, 3).

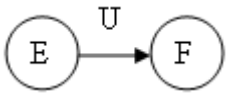


Step 8: Successor state G of state C is in this set of states already. So find the next state after C in the table and see if it's possible to add it to this set of states, which is state D.
The context sets associated with {B, C, D, G, H, I, J}: ({k}, 1), ({d, u}, 2), ({a, b, c, r, w}, 3).

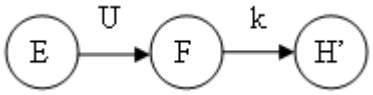


Step 9: Successor state G of state D is in this set of states already. So find the next state after D in the table and see if it is possible to add it to this set of states, which is state E. But adding E to this set will cause a conflict in the associated context sets. So E must be put into a new set of states.
The collection of contexts associated with {E}: ({k}, 1).

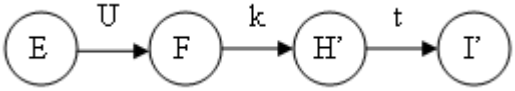
Step 10: Add the successor of state E: state F. The collection of contexts associated with {E, F}: ({k}, 1), ({r}, 2), ({a}, 3).



Step 11: Add the successor of state F: state H. Now state H is already in the first set of states. So adding H to the current set of states means we need to combine the new set of states with the old one. But then the combined contexts is: ({k}, 1), ({d, r, u}, 2), ({a, b, c, r, w}, 3). This is not pairwise disjoin because "r" is in sets for configurations 2 and 3. So we have to keep the current set separate from the old one, and create a copy of state H to insert into the new set.
The collection of contexts associated with {E, F, H'}: ({k}, 1), ({r}, 2), ({a}, 3).

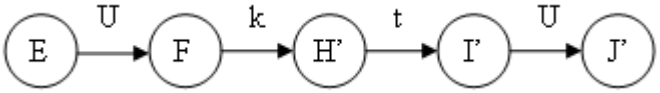


Step 12: Add the successor of state H', which is I. Similarly, we have to create a copy of state I to insert into the new set to avoid merging with the old set, which causes the failure of pairwise disjointness of the context sets.
The collection of contexts associated with {E, F, H', I'}: ({k}, 1), ({r}, 2), ({a}, 3).



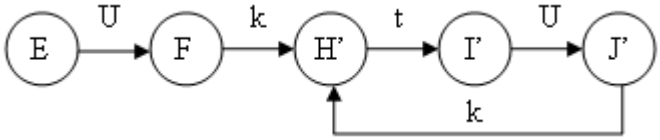
Step 13: Add the successor of state I', which is J. For the same reason, we need to create a copy of J.

The collection of contexts associated with {E, F, H', I', J'}: ({k}, 1), ({r, u}, 2), ({a}, 3).



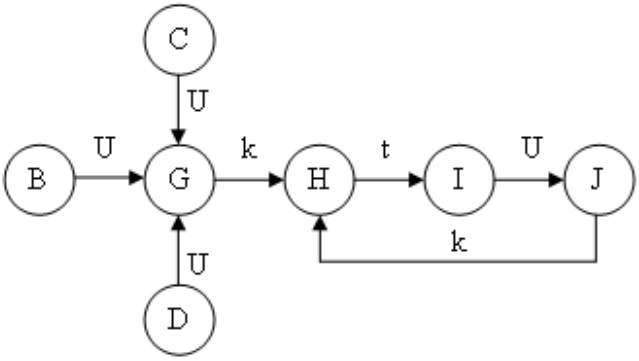
Step 14: Add the successor of state J', which is H. For the same reason, we need a copy of H. But there exists a copy of H in this set already, so we just use it.

The collection of contexts associated with {E, F, H', I', J'}: ({k}, 1), ({r, u}, 2) ({a}, 3).



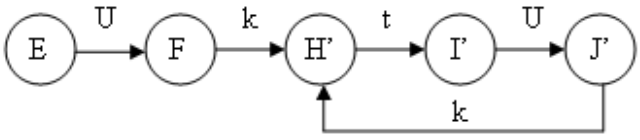
So finally the result of combining the regenerated states is:

Set 1:



Associated context sets: ({k}, 1), ({d, u}, 2), ({a, b, c, w}, 3)

Set 2:



Associated context sets: ({k}, 1), ({r, u}, 2) ({a}, 3)

Implementation

In the implementation, these procedures are used:

Algorithm 4.33: phase2_laneTable()

```
1 all_clusters = ⊘ ;
2 foreach entry e in LT_tbl do
3     if e.processed == false then
4         new_cluster = cluster_create(e);
5     clear_regenerate(e.from_state);
6     let e.processed = true;
7     is_new_chain = cluster_trace_new_chain_all(e.from_state, e);
8     if is_new_chain == true then
9         add new_cluster to the all_clusters list;
```

Algorithm 4.34: cluster_trace_new_chain_all(parent_state, e)

Input: state parent_state; LT_tbl_entry e

```
1 let is_new_chain = true;
2 foreach state s in the to_states list of e do
3     val = cluster_trace_new_chain(parent_state, s);
4     if ret_val == false then
5         is_new_chain = false;
6 return is_new_chain;
```

Algorithm 4.35: cluster_trace_new_chain(parent_state, s)

```
Input: state parent_state; state s
1  let c be the current new_cluster;
2  let is_new_chain = true;
3  find the entry e for s in LT_tbl;
4  let e_ctxt be the associated context sets of e, or null if e is not found;
5  let ret_state be the copy of s (original s or its split copy) contained in c;
6  if ret_state exists then
7      inherit_propagate(ret_state, parent_state, new_cluster, e);
8      replaceSuccessor;
9      return true;
10 else
11     in the all_clusters list, search for a cluster 'container' that contains s or its split
       copy;
12     if container is found, and it and new_cluster are pairwise disjoint in context then
13         combine container into new_cluster;
14         is_new_chain = false;
15         set new_cluster = container;
16     else
17         // A context-pairwise container is not found;
18         if a non-context-pairwise container exist then
19             make a copy of s and add it to new_cluster;
20             clear_inherit_regenerate;
21         else
22             // s is not in any cluster yet;
23             add s to new_cluster, combine its context to new_cluster;
24             clear_inherit_regenerate;
25     if e is null null then
26         is_new_chain = cluster_trace_new_chain_all(ret_state, e);
```

4.7 Framework of Reduced-Space LR(1) Parser Generation

From the available literature it seems that all the existing LR(1) algorithms that compress the parsing table (and thus can be called *reduced-space* LR(1) algorithms) can be divided into three categories:

1) Combining. For this category, the algorithm combines new generated states with old ones when appropriate. The key is the judging criteria of when is "appropriate".

   Examples of this kind of algorithms include the practical general method of Pager and the partially working MLR/Honalee algorithm of Tribble.

   LALR(1) is a widely used algorithm that is considered a good tradeoff between efficiency and recognition power. It is also widely know that the difference between LALR(1) and LR(1) is the existence of reduce/reduce conflicts in the LALR(1) parsing machine. A shift/reduce conflict existing in a LALR(1) parsing machine would also exist in an LR(1) parsing machine. An LALR(1) parsing machine can be obtained by combining all those LR(1) states with the same core configurations. But this would "over-compress" the LR(1) parsing machine so reduce/reduce conflicts would occur as a result. Intuitively to compress the canonical LR(1) parsing machine, at the same time not to "over-compress" it and introduce reduce/reduce conflicts is the key to the solution. According to what we know Pager's practical general method is the only one that works for this category. Finding a sound criteria with proven correctness is not easy.

2) Splitting. This kind of algorithm starts from an LR(0) or LALR(1) parsing machine, finds those states where reduce/reduce conflicts occur, and split relevant states to resolve the conflicts.

   Examples of this category include the lane-tracing algorithm of Pager and the splitting method of Spector.

   There are a lot of people aware of this idea and many implemented this idea in their parser generators. The start point of splitting can be from an LR(0) parsing machine or from an LALR(1) parsing machine. The method to get an LR(0) parsing machine is quite unified, but there are different approaches to getting an LALR(1) parsing machine. The dragon book mentions two, one is considered practical and the other of combining LR(1) states with the same core set of configurations is considered not practical. Pager's lane-tracing is another

way. The concrete way of find states relevant to reduce/reduce conflicts also vary, although they should more or less be similar to the lane-tracing algorithm. This is the most commonly thought of and used approach. The difficult part is in splitting and regenerating those states involved in reduce/reduce conflicts.

3) Partitioning. This category of algorithms uses the approach of divide conquer by partitioning the original grammar into small parts, find an LR(1) parsing table for each of them, then combine these small tables together into a large LR(1) parsing table for the original grammar.

   An example is the partitioning algorithm of Korenjak. Another example may be the MSTA parser generator, whose author Vladimir Makarov said: "MSTA does a lot of optimizations. Even if you use LR(k) algorithm, it finds and uses LALR and regular parts of grammar".

   This method is not as popular as splitting. It is also less intuitive as combining or splitting. The tough part here is the strategy and heuristic to do the partitioning.

   Also it should be pointed out that this algorithm by itself may or may not be reduced-space. It just provides this strategy of divide and conquor. In the process of finding LR(1) parsing table for each small part, it can either use the combining/splitting approach, or just use the original Knuth canonical algorithm.

This framework can be summarized in figure 4.13 below.

For the combining approach, it generally should start with the canonical LR(1) parsing machine, combining states along the way. For the splitting approach, it generally should start from the LR(0) parsing machine, then to the LALR(1) parsing machine, and then work on those inadequate states with reduce/reduce conflicts in the LALR(1) parsing machine. The partitioning approach as discussed by Korenjak deals with canonical LR(1) parsing machine, but it would be interesting to see if the reduced-space LR(1) algorithms can be used instead of the canonical LR(1) algorithm.

4.8 Conclusion

In this chapter we showed the details of design and implementation of some LR(1) algorithms.

The Knuth canonical LR(1) algorithm is implemented with optimized algorithms and efficient data structures. The practical general method is based on this by adding a state-combining step.

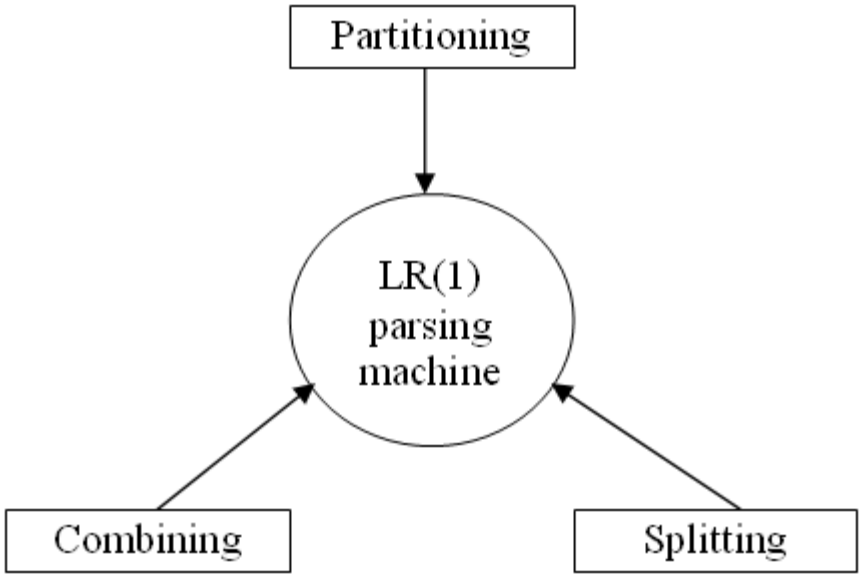


Figure 4.13: The approaches to LR(1) parsing machine

The unit production elimination algorithm is implemented using the parsing table, instead of on the parsing machine automata. This is easier to handle.

We extended the unit production elimination algorithm to remove redundant states and obtain a minimal parsing machine.

Phase 2 of the lane-tracing algorithm was not described with details before. Here we have shown the full details of two approaches: one based on the practical general method, the other based on a lane-tracing table.

Finally we summarized approaches to LR(1) parser generation in a framework. Here we have worked on the combining and splitting approaches, but not the partitioning approach.

Chapter 5

Measurements and Evaluations of LR(1) Parser Generation

5.1 About the Measurement

In this chapter empirical study on performance is discussed. State numbers and conflict numbers in the parsing machine of a grammar always keep the same. But time and memory usage can change according to different environment. However, the relative trend of time and memory usage of different algorithms keeps the same, and allows us to watch the relative performance of different algorithms.

5.1.1 The Environment and Metrics Collection

The data are collected on a laptop computer with 1.7 GHz Intel Pentium processor and 1 GB RAM. The laptop runs Fedora core 4.0. The version of Bison [3] is 2.3. For all the measurements, time is in sec (second), and memory is in MB (megabyte).

The execution time is measured by the "time" command line utility. Since the execution time varies a little bit each time, we did ten meansurements on each execution and took the average of the values.

The memory usage is read from Fedora Core system monitor. To obtain a stable value, a "scanf" function call is used at the end of the C program, so the program hangs there until the user presses

a key. When the program hangs, the memory usage reading is stable in the system monitor. There is no variation in the reading, therefore there is no need to average multiple readings. Note that theoretically this way we are reading the memory usage at the end of program execution, but not the peak memory usage. But at least visually from the system monitor, the memory usage value increases monotonously. Therefore we just take this final memory usage as the peak memory usage.

5.1.2 The Algorithms

The following algorithms are measured in this study. We have included the acronyms for them. These acronyms are used in later discussion.

There are four LR(1) algorithms, of these the latter three are reduced-space:

Knuth LR(1): Knuth canonical LR(1) algorithm.
PGM LR(1): LR(1) algorithm based on the practical general method.
LT LR(1) w/ PGM: LR(1) algorithm based on lane-tracing, use PGM in phase 2.
LT LR(1) w/ LTT: LR(1) algorithm based on lane-tracing, use lane-tracing table in phase 2.

These are followed by two LALR(1) algorithms:

LT LALR(1): LALR(1) algorithm based on lane-tracing phase 1.
Bison LALR(1): LALR(1) algorithm implemented in Bison.

Then a LR(0) algorithm:

LR0: The traditional LR(0) algorithm.

There are also two algorithms used to optimize LR(1) parsing machine:

UPE: The unit production elimination algorithm.
UPE Ext: The extension algorithm to the unit production elimination algorithm.

5.1.3 The Grammars

17 simple grammars were used to test the correctness of Hyacc. These grammars are shown in Appendix B. The grammars of 13 real programming languages were used to check the performance of Hyacc. These real language grammars were obtained from [13] with minor modifications to fit in Yacc grammar input format. Table 5.1 shows the grammar statistics of these 30 grammars.

	Grammar statistics		
Grammar	Terminal #	Non-Terminal #	Rule #
G1	3	3	5
G2	3	7	10
G3	3	7	10
G4	4	3	5
G5	5	3	6
G6	5	4	8
G7	10	8	16
G8	4	6	10
G9	5	3	6
G10	4	4	7
G11	3	5	6
G12	8	10	17
G13	2	5	7
G14	13	10	18
G15	14	15	24
G16	21	19	36
G17	7	10	19
Ada	94	239	459
Algol 60	55	77	169
C	82	64	212
Cobol	184	181	401
C++ 5.0	101	186	665
Delphi	95	169	358
Ftp	52	16	74
Grail	42	32	74
Java 1.1	96	97	266
Matlab	44	35	93
Pascal	65	135	257
Turbo Pascal	71	99	222
Yacc	25	58	103

Table 5.1: Number of terminals, non-terminals and rules in the grammars

5.2 LR(1), LALR(1) and LR(0) Algorithms

The four LR(1) algorithms here include the Knuth canonical LR(1) algorithm, the PGM algorithm and the two lane-tracing algorithms based on PGM and LTT, all implemented in Hyacc. The two LALR(1) algorithms here include the one in Bison and the one in Hyacc (based on lane-tracing phase 1). The LR(0) algorithm is the common one described by textbooks and implemented in Hyacc.

5.2.1 Parsing Table Size Comparison

A comparison of parsing table sizes of the 30 grammars is shown in Table 5.2. Figure 5.1 contains the 13 real language grammars only, and is the graphic version that better visualizes the comparison.

Observations:

1) The size of Knuth canonical LR(1) parsing machine is much bigger than the rest.

2) For the three reduced-space LR(1) algorithms, the generated parsing machines are only slightly bigger than LALR(1) parsing machines. LT LR(1) w/ PGM always produces the smallest parsing machine. For parsing machines generated by LT LR(1) w/ LTT and PGM LR(1), sometimes the former is bigger, sometimes the latter is bigger.

3) For Bison, its state number is always one more than Hyacc. This is because Bison adds a "$end" symbol to the end of the goal production, so it always has one more accept state than Hyacc LALR(1) parsing machine. Considering this, LT LALR(1) gives the same number of states as Bison. This validates our implementation.

Summary:

1) For given grammars, reduced-space LR(1) algorithms bring down the parsing machine size significantly from the Knuth LR(1) parsing machine, and not much bigger than LALR(1) parsing machine. Actually, if the parsing machine contains no reduce/reduce error (shown in Table 5.3 on page 104) then the reduced-space LR(1) parsing machine has the same size as the LALR(1) parsing machine.

2) LT LR(1) w/ PGM results in slightly more compact LR(1) parsing machine than PGM LR(1). This is possibly due to the use of *weak compatibility* in the PGM algorithm. Use of the strong compatibility can result in a most compact parsing machine [48]. The strong compatibility algorithm has not been implemented though, so we cannot compare it at this time.

3) LT LALR(1) works properly.

	Hyacc						Bison
Grammar	Knuth LR(1)	PGM LR(1)	LT LR(1) w/ PGM	LT LR(1) w/ LTT	LR0	LT LALR(1)	LALR(1)
G1	8	8	8	8	8	8	9
G2	21	20	20	20	19	19	20
G3	21	20	20	20	19	19	20
G4	16	9	9	9	9	9	10
G5	20	11	11	11	11	11	12
G6	35	14	14	14	14	14	15
G7	18	18	18	18	18	18	19
G8	13	13	13	13	13	13	14
G9	18	10	10	10	10	10	11
G10	17	10	10	10	10	10	11
G11	9	9	9	9	9	9	10
G12	19	19	19	19	19	19	20
G13	13	13	13	13	13	13	14
G14	82	40	40	40	40	40	41
G15	53	53	53	53	53	53	54
G16	130	73	73	73	73	73	74
G17	51	32	32	32	32	32	33
Ada	12786	873	860	860	860	860	861
Algol 60	1538	274	272	294	272	272	273
C	1572	349	349	349	349	349	350
Cobol	2398	657	657	657	657	657	658
C++ 5.0	9785	1404	1261	1496	1256	1256	1257
Delphi	4215	609	609	945	609	609	610
Ftp	210	200	200	200	200	200	201
Grail	719	193	193	193	193	193	194
Java 1.1	2479	439	428	428	428	428	429
Matlab	588	174	174	174	174	174	175
Pascal	2245	418	412	412	412	412	413
Turbo Pascal	1918	394	386	386	386	386	387
Yacc	153	128	128	128	128	128	129

Table 5.2: Parsing table size comparison

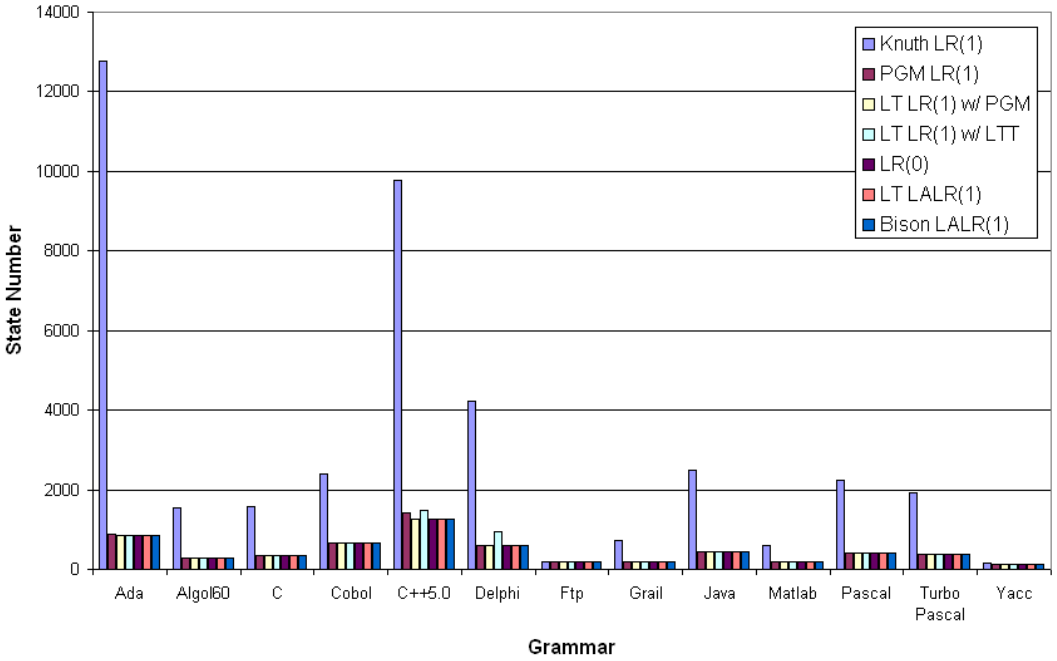


Figure 5.1: Parsing Table Size Comparison

5.2.2 Parsing Table Conflict Comparison

A comparison of parsing table conflicts is shown in Table 5.3.

Observations:

1) LT LALR(1) and Bison LALR(1) produce the same number of shift/reduce and reduce/reduce conflicts for all the grammars (except for Delphi grammar, which may have some problem).

2) LT LR(1) w/ PGM and PGM LR(1) give the same number of conflicts as LALR(1) (except for the Delphi grammar). Algol60 and C++ have reduce/reduce conflicts in LR(1) parsing machine, and therefore are not LR(1) grammars. The other grammars do not have reduce/reduce conflicts in LALR(1) parsing machine, so no such conflicts in LR(1) parsing machine too. G2 and G3 are LR(1) grammars, and their reduce/reduce conflicts in LALR(1) parsing machine are resovled in LR(1) parsing machine.

3) LT LR(1) w/ LTT has more reduce/reduce conflicts than the other two reduced-space algorithms. This is because it splits those states with reduce/reduce conflicts and count these conflicts repeatedly.

Summary:

1) LR(1) algorithms can resolve reduce/reduce conflicts in LALR(1) parsing machine (although only G2 and G3 are such LR(1) grammars here). The parsing machines of some programming language grammars (Algol60, C++, Delphi) contain reduce/reduce conflicts that can not be resolved by LR(1) algorithms, and are not LR(1) grammars.

2) LT LR(1) w/ LTT may have more reduce/reduce conflicts in the generated parsing machine than the other two reduced-space LR(1) algorithms, due to the fact that it splits relevant states and count such conflicts repeatedly.

	Hyacc												Bison	
	Knuth LR(1)		PGM LR(1)		LT LR(1) w/ PGM		LT LR(1) w/ LTT		LR(0)		LT LALR(1)		LALR(1)	
Grammar	s/r	r/r	s/r	r/r	s/r	r/r	s/r	r/r	s/r	r/r	s/r	r/r	s/r	r/r
G1	0	0	0	0	0	0	0	0	2	0	0	0	0	0
G2	0	0	0	0	0	0	0	0	1	4	0	1	0	1
G3	0	0	0	0	0	0	0	0	1	4	0	1	0	1
G4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
G5	0	0	0	0	0	0	0	0	2	0	0	0	0	0
G6	7	0	4	0	4	0	4	0	12	0	4	0	4	0
G7	0	0	0	0	0	0	0	0	8	0	0	0	0	0
G8	0	0	0	0	0	0	0	0	2	0	0	0	0	0
G9	0	0	0	0	0	0	0	0	0	0	0	0	0	0
G10	0	0	0	0	0	0	0	0	3	0	0	0	0	0
G11	0	0	0	0	0	0	0	0	0	4	0	0	0	0
G12	0	0	0	0	0	0	0	0	3	27	0	0	0	0
G13	0	0	0	0	0	0	0	0	1	0	0	0	0	0
G14	0	0	0	0	0	0	0	0	5	0	0	0	0	0
G15	0	0	0	0	0	0	0	0	1	0	0	0	0	0
G16	0	0	0	0	0	0	0	0	6	0	0	0	0	0
G17	0	0	0	0	0	0	0	0	4	0	0	0	0	0
Ada	0	0	0	0	0	0	0	0	260	2565	0	0	0	0
Algol 60	0	4	0	1	0	1	0	3	133	336	0	1	0	1
C	2	0	1	0	1	0	1	0	214	0	1	0	1	0
Cobol	6	0	5	0	5	0	5	0	349	1480	5	0	5	0
C++ 5.0	280	31	24	18	24	18	24	250	7140	10812	24	18	24	18
Delphi	316	1191	60	174	58	139	109	578	578	1344	15	121	60	174
Ftp	0	0	0	0	0	0	0	0	6	0	0	0	0	0
Grail	0	0	0	0	0	0	0	0	117	0	0	0	0	0
Java 1.1	2	0	1	0	1	0	1	0	236	582	1	0	1	0
Matlab	25	0	14	0	14	0	14	0	142	45	14	0	14	0
Pascal	0	0	0	0	0	0	0	0	222	264	0	0	0	0
Turbo Pascal	25	0	1	0	1	0	1	0	263	288	1	0	1	0
Yacc	8	0	8	0	8	0	8	0	60	0	8	0	8	0

Table 5.3: Parsing table conflict comparison

5.2.3 Running Time Comparison

Table 5.4 shows the time comparison, and Figure 5.2 is the graphic view. The simple grammars are not included since they run too fast (actually about 0.14 second for all the simple grammars).

Observations:

1) Knuth LR(1) takes the longest time.

2) Reduced-space LR(1) algorithms are faster than Knuth LR(1), and close to Bison LALR(1).

3) For LT LR(1) w/ PGM and LT LR(1) w/ LTT, the latter takes longer time. This is especially obvious when there is a lot of states to split in the LR(0) parsing machine. The grammars of Algol60, C++ and Delphi are such examples.

4) LR(0) runs the fastest, as expected.

Summary:

1) Even though more expensive, Knuth LR(1) parser generation is still practical in running time, since it takes just a few seconds at most for given grammars.

2) Reduced-space LR(1) algorithms are more efficient than Knuth LR(1), and close to Bison LALR(1), sometimes even faster.

3) LT LR(1) w/ LTT takes more time than LT LR(1) w/ PGM when many states need to be splitted in the LR(0) parsing machine.

	Hyacc						Bison
Grammar	Knuth LR(1)	PGM LR(1)	LT LR(1) w/ PGM	LT LR(1) w/ LTT	LR(0)	LT LALR(1)	LALR(1)
Ada	1.883	0.406	0.172	0.173	0.136	0.173	0.155
Algol 60	0.606	0.290	0.509	0.531	0.039	0.499	0.174
C	1.047	0.420	0.192	0.192	0.067	0.189	0.225
Cobol	0.234	0.127	0.115	0.113	0.117	0.113	1.690
C++ 5.0	3.529	1.779	1.261	2.045	0.544	1.101	0.705
Delphi	1.141	0.335	0.364	0.945	0.093	0.137	0.638
Ftp	0.016	0.017	0.017	0.017	0.016	0.017	0.268
Grail	0.051	0.024	0.020	0.021	0.017	0.021	0.156
Java 1.1	1.552	1.026	0.350	0.352	0.097	0.350	0.339
Matlab	0.351	0.189	0.117	0.117	0.034	0.116	0.120
Pascal	0.504	0.174	0.066	0.067	0.05	0.066	0.246
Turbo Pascal	0.305	0.098	0.053	0.053	0.042	0.054	0.204
Yacc	0.018	0.026	0.016	0.017	0.015	0.017	0.157

Table 5.4: Time performance comparison (sec)

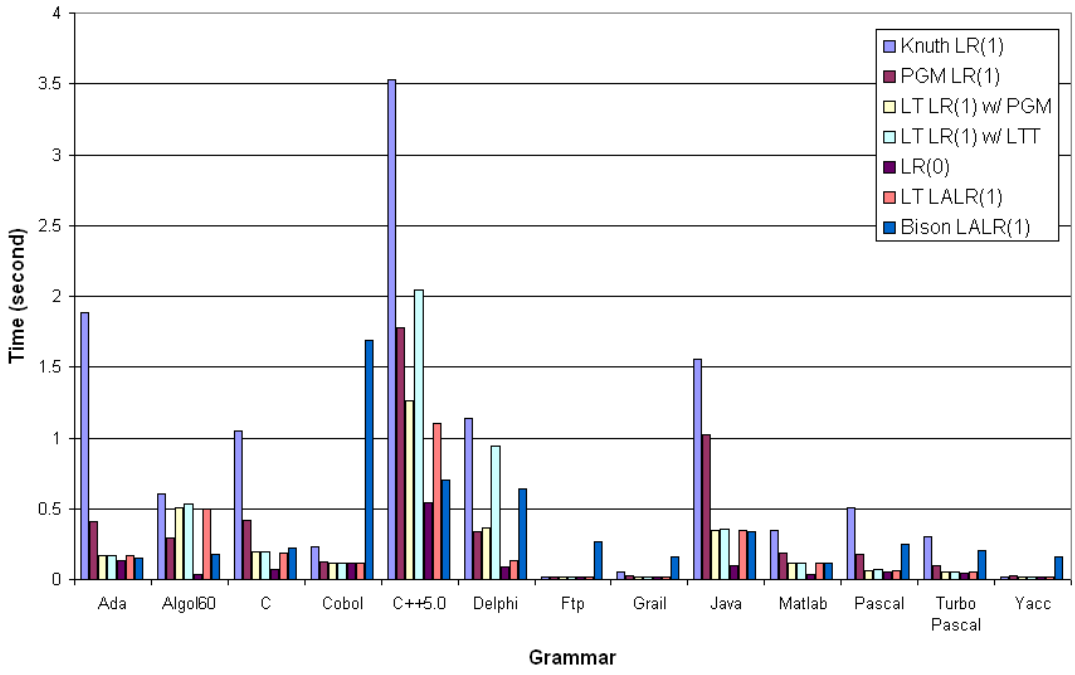


Figure 5.2: Running Time Comparison

5.2.4 Memory Usage Comparison

Table 5.5 shows the memory usage comparison, and Figure 5.3 is the graphic view. The simple grammars are not included since their memory usage are all very low (below 1 MB).

Observations:

1) Knuth LR(1) always use more or much more memory than the rest.

2) reduced-space LR(1) algorithms use much less memory than Knuth LR(1), and often not much more than LALR(1). There is no definite order which of the 3 reduced-space LR(1) algorithms is always smaller though.

3) If only compare LT LR(1) w/ PGM and LT LR(1) w/ LTT, the latter always takes equal or less memory.

Summary:

1) Knuth LR(1) parser generation requires much more memory than LT LR(1) and PGM LR(1). However it is still acceptable for today's personal computers even for grammars as complex as that of C++ 5.0.

2) Reduced-space LR(1) algorithms use not much more memory than LALR(1).

3) LT LR(1) w/ LTT uses equal or less memory than LT LR(1) w/ PGM.

	Hyacc						Bison
Grammar	Knuth LR(1)	PGM LR(1)	LT LR(1) w/ PGM	LT LR(1) w/ LTT	LR(0)	LT LALR(1)	LALR(1)
Ada	95.1	7.9	7.9	7.9	6.9	7.9	4.0
Algol 60	16.0	4.2	6.4	5.4	3.6	5.1	3.9
C	18.9	6.0	5.2	5.2	4.3	5.2	4.0
Cobol	19.1	6.3	6.5	6.5	6.0	6.5	4.0
C++ 5.0	122.7	23.9	39.1	34.4	12.5	19.9	4.3
Delphi	37.4	6.5	14.5	11.0	5.5	6.4	3.9
Ftp	2.8	2.8	2.8	2.8	2.7	2.8	3.9
Grail	5.3	2.9	2.9	2.9	2.8	3.0	3.8
Java 1.1	35.6	7.8	6.3	6.3	5.0	6.3	3.8
Matlab	7.8	3.9	3.5	3.5	3.0	3.5	3.8
Pascal	18.6	4.9	4.8	4.8	4.4	4.8	3.9
Turbo Pascal	13.8	4.3	4.5	4.5	4.2	4.5	3.9
Yacc	2.6	2.6	2.6	2.6	2.5	2.6	3.9

Table 5.5: Memory performance comparison (MB)

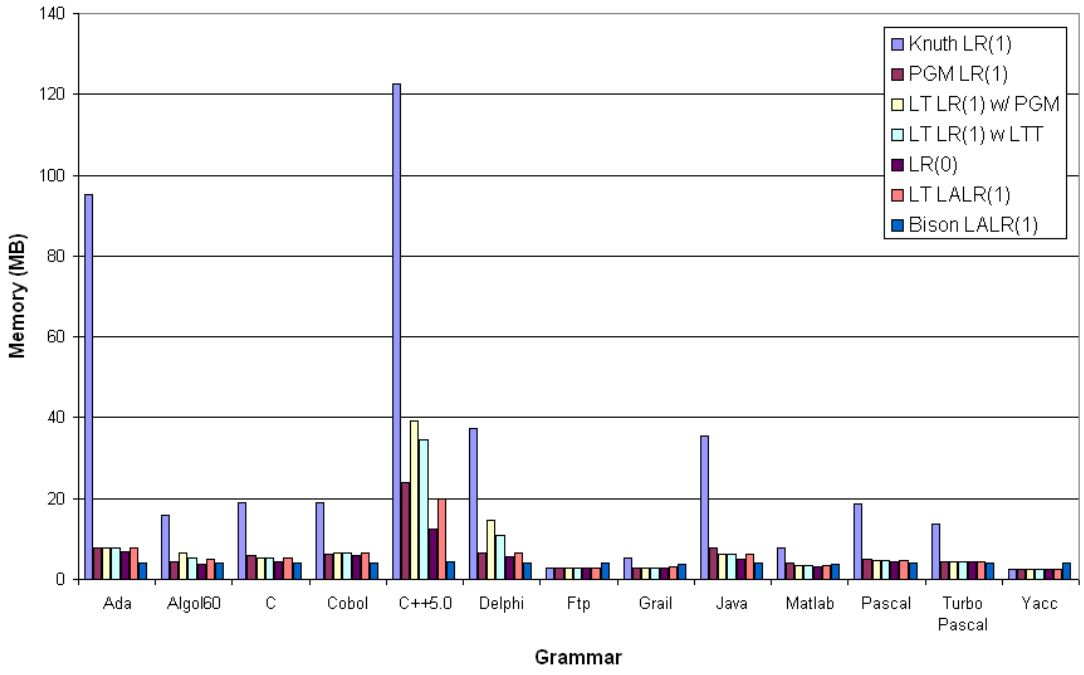


Figure 5.3: Memory Usage Comparison

5.3 Extension Algorithm to Unit Production Elimination

5.3.1 Parsing Table Size Comparison

Table 5.6 shows the parsing table size comparison. Figure 5.4 is the graphic view. Figure 5.5 is the graphic view of the comparison of parsing machine size change after applying UPE and UPE Ext algorithms versus the parsing machine obtained by only using the PGM LR(1) algorithm.

Observations:

1) Unit production elimination may decrease the number of states, but in most cases (12 out of 13) increases it.

2) Using the extension to the unit production elimination algorithm, in 3 out of the 13 real language grammars the number of states is less than that of Bison. For the other grammars, although the size of the parsing machine is bigger, it is bigger only by a small margin.

Summary:

1) Unit production elimination may increase the number of states in the LR(1) parsing machine.

2) Extension algorithm to the unit production elimination algorithm decreases the size of the parsing machine to not much bigger than LALR(1) parsing machine. Therefore it is desirable to apply the extension algorithm.

	PGM LR(1)	UPE	UPE Ext	
Grammar	State #	State #	State #	Rule #
G1	8	6	6	3
G2	20	11	11	5
G3	20	11	11	5
G4	9	7	7	3
G5	11	9	9	4
G6	14	11	11	5
G7	18	13	6	4
G8	13	11	9	6
G9	10	10	7	3
G10	10	7	7	4
G11	9	7	7	4
G12	19	11	11	8
G13	13	12	12	6
G14	40	48	31	9
G15	53	57	45	16
G16	73	93	65	26
G17	32	26	26	13
Ada	873	1074	805	262
Algol 60	274	498	412	92
C	349	786	380	116
Cobol	657	646	528	268
C++ 5.0	1404	3573	2255	443
Delphi	609	1195	669	200
Ftp	200	211	211	71
Grail	193	247	204	54
Java 1.1	439	1174	673	142
Matlab	174	374	178	53
Pascal	418	844	427	119
Turbo Pascal	394	649	353	116
Yacc	128	134	134	87

Table 5.6: Parsing table size comparison

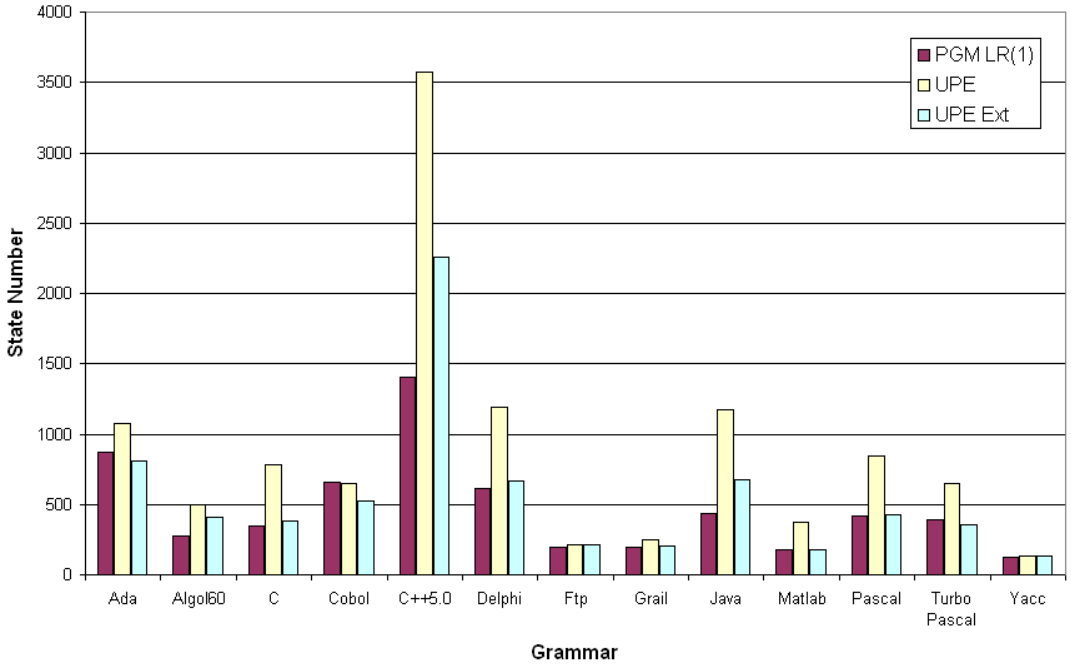


Figure 5.4: Parsing Table Size Comparison

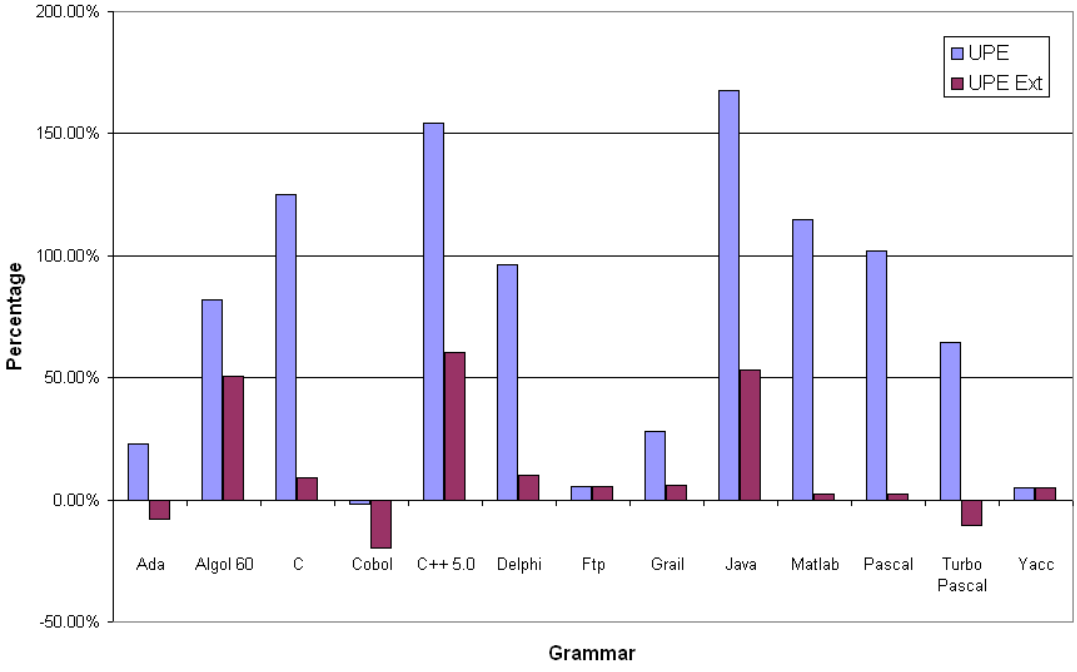


Figure 5.5: Parsing Table Size Change Percentage

5.3.2 Parsing Table Conflict Comparison

Table 5.7 shows the parsing table conflict comparison.

Observations:

1) There may be more shift/reduce and reduce/reduce conflicts when UPE and UPE Ext are used for real language grammars, which in general are not LALR(1) or even LR(1) grammars. But for the 17 simple grammars which are LALR(1) (except for G6), there is not increase in number of conflicts.

2) Shift/shift conflicts happen abundantly to some complex grammars when UPE (and so UPE Ext as well) is used. This is because such grammars are ambiguous and thus not LR(1), so that UPE cannot be used for such grammars. Users should try their grammars on UPE. It is fine only when no shift/shift conflict occurs.

3) Note that some grammars (Ada, Matlab) do not have conflicts when use Knuth LR(1) or PGM LR(1), but do have shift/shift conflicts when use UPE or UPE Ext. No conflicts when using Knuth LR(1) or PGM LR(1) does not mean it has no conflicts, just that those conflicts may have been solved using precedence and associativity rules. The grammar still can be non-LR(1). So when using UPE or UPE Ext, it would cause shift/shift conflicts even though there is no conflict when using Knuth LR(1) and PGM LR(1).

Summary:

1) UPE and its extension may lead to more shift/reduce and reduce/reduce conflicts for grammars that are not LALR(1) and not LR(1).

2) UPE and its extension sometimes would lead to shift/shift conflicts. This is an error and in such situation these two optimizations should not be used.

	PGM LR(1)		UPE			UPE Ext		
Grammar	s/r	r/r	s/r	r/r	s/s	s/r	r/r	s/s
G1	0	0	0	0	0	0	0	0
G2	0	0	0	0	0	0	0	0
G3	0	0	0	0	0	0	0	0
G4	0	0	0	0	0	0	0	0
G5	0	0	0	0	0	0	0	0
G6	7	0	4	0	4	0	4	0
G7	0	0	0	0	0	0	0	0
G8	0	0	0	0	0	0	0	0
G9	0	0	0	0	0	0	0	0
G10	0	0	0	0	0	0	0	0
G11	0	0	0	0	0	0	0	0
G12	0	0	0	0	0	0	0	0
G13	0	0	0	0	0	0	0	0
G14	0	0	0	0	0	0	0	0
G15	0	0	0	0	0	0	0	0
G16	0	0	0	0	0	0	0	0
G17	0	0	0	0	0	0	0	0
Ada	0	0	57	146	4	57	146	4
Algol 60	0	1	5	1	1	5	1	1
C	1	0	1	0	959	1	0	959
Cobol	5	0	10	0	0	10	0	0
C++ 5.0	24	18	42	43	500	42	43	500
Delphi	60	174	81	186	336	81	186	336
Ftp	0	0	0	0	0	0	0	0
Grail	0	0	0	0	0	0	0	0
Java 1.1	1	0	1	0	144	1	0	144
Matlab	14	0	162	232	68	162	232	68
Pascal	0	0	0	0	70	0	0	70
Turbo Pascal	1	0	1	0	32	1	0	32
Yacc	8	0	8	0	0	8	0	0

Table 5.7: Parsing table conflict comparison

5.3.3 Running Time Comparison

Table 5.8 shows the running time comparison, and Figure 5.6 is the graphic view.

Observations and summary: Compared to PGM LR(1), UPE and UPE Ext use longer running time, sometimes significantly longer. such as for the C++ grammar.

5.3.4 Memory Usage Comparison

Table 5.9 shows the running time comparison, and Figure 5.7 is the graphic view.

Observations:

1) For memory usage, Bison does not change much for different grammars. For Hyacc, the difference for different grammars can be big.

2) For Hyacc, using PGM LR(1) leads to a bigger reduction in memory than using Knuth LR(1). When using UPE and UPE Ext, there is a slight increase in memory.

Summary:

1) Compared to PGM LR(1), UPE and UPE Ext require slight increase in memory.

2) UPE and UPE Ext use the same amount of memory, the same as shown by theoretical analysis.

Table 5.10 shows the memory increase percentage after using the UPE algorithm compared to only using the PGM LR(1) algorithm. UPE Ext uses the same amount of memory as UPE. It shows that for complex grammars such as Ada, C++, Delphi and Pascal, the increase in memory usage can be big.

Table 5.11 shows the state number change percentage of using UPE and UPE Ext versus only using the PGM LR(1) algorithm. It shows that the UPE algorithm often increases the size of the parsing machine significantly, but the use of UPE Ext algorithm can bring this back down.

Grammar	PGM LR(1)	UPE	UPE Ext
Ada	0.406	1.342	3.452
Algol 60	0.290	0.566	0.931
C	0.420	1.142	1.418
Cobol	0.127	1.205	1.206
C++ 5.0	1.779	5.680	33.986
Delphi	0.335	1.347	4.371
Ftp	0.017	0.035	0.035
Grail	0.024	0.066	0.119
Java 1.1	1.026	1.563	3.328
Matlab	0.189	0.307	0.637
Pascal	0.174	1.061	1.787
Turbo Pascal	0.098	0.587	1.159
Yacc	0.026	0.043	0.043

Table 5.8: Time performance comparison (sec)

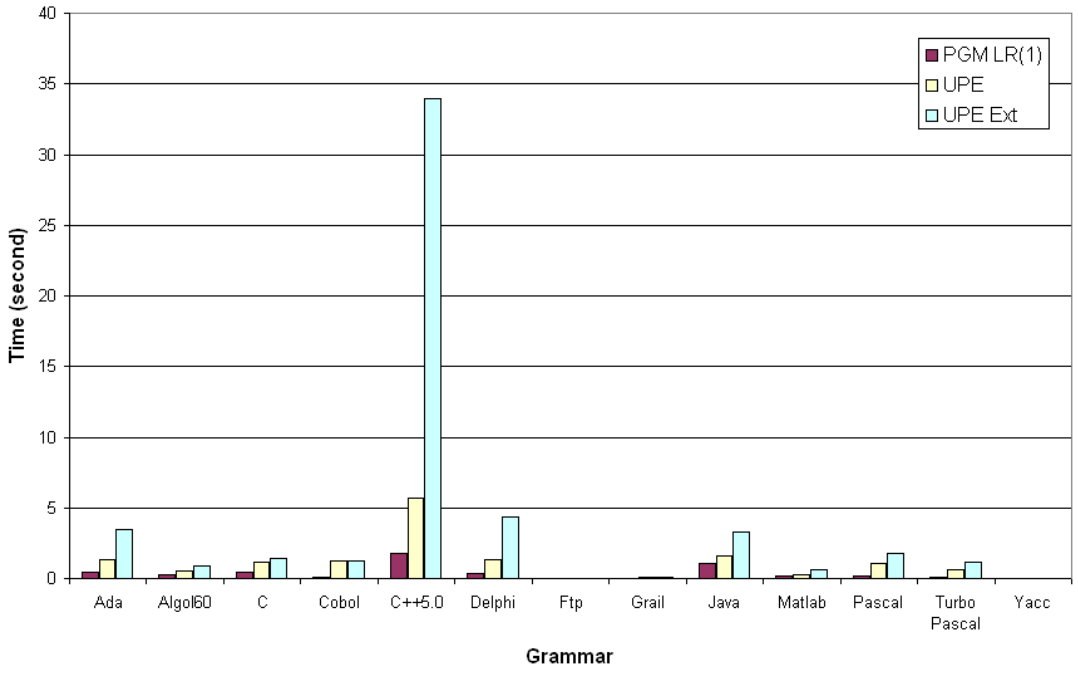


Figure 5.6: Running Time Comparison

Grammar	PGM LR(1)	UPE	UPE Ext
Ada	7.9	9.4	9.3
Algol 60	4.2	4.2	4.2
C	6.0	6.0	6.0
Cobol	6.2	6.4	6.4
C++ 5.0	23.9	30.9	30.9
Delphi	6.5	7.5	7.5
Ftp	2.8	2.9	2.9
Grail	2.9	2.9	2.9
Java 1.1	7.8	8.6	8.6
Matlab	3.9	3.9	3.9
Pascal	4.9	5.7	5.7
Turbo Pascal	4.3	4.3	4.3
Yacc	2.6	2.7	2.7

Table 5.9: Memory usage comparison (MB)

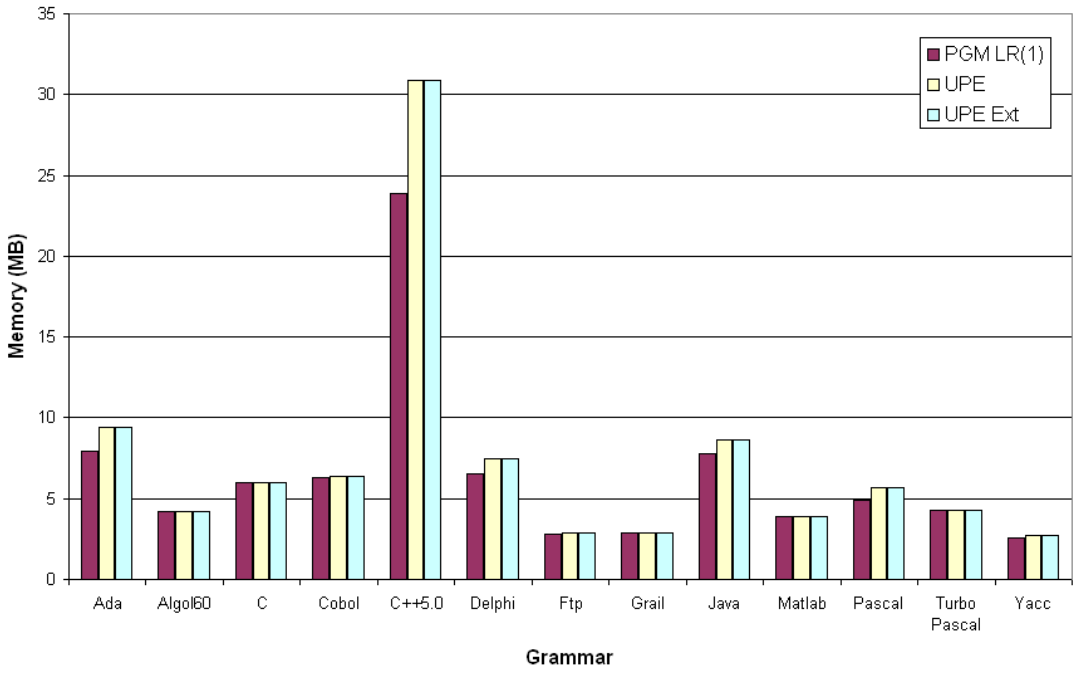


Figure 5.7: Memory Usage Comparison

Grammar	UPE (and UPE Ext)
Ada	18.99%
Algol 60	0.00%
C	0.00%
Cobol	1.59%
C++ 5.0	29.29%
Delphi	15.38%
Ftp	3.57%
Grail	0.00%
Java 1.1	10.26%
Matlab	0.00%
Pascal	16.33%
Turbo Pascal	0.00%
Yacc	3.85%

Table 5.10: Memory increase percentage of UPE (and UPE Ext) v.s. PGM LR(1)

Grammar	UPE	UPE Ext
Ada	23.02%	-7.79%
Algol 60	81.75%	50.36%
C	125.21%	8.88%
Cobol	-1.67%	-19.63%
C++ 5.0	154.49%	60.61%
Delphi	96.22%	9.85%
Ftp	5.50%	5.50%
Grail	27.98%	5.70%
Java 1.1	167.43%	53.30%
Matlab	114.94%	2.30%
Pascal	101.91%	2.15%
Turbo Pascal	64.72%	-10.41%
Yacc	4.69%	4.69%

Table 5.11: Percentage of state number change compared to PGM LR(1)

5.4 Comparison with Other Parser Generators

5.4.1 Comparison to Dragon and Parsing

Three LR(1) parser generators: Dragon [9], Parsing [10] and Hyacc are compared using grammars of similar complexity. See Table 5.12. The performance data of Dragon and Parsing are from communications with the authors.

It is obvious that Hyacc has a significant advantage in time and memory performance over the other two, especially in running time. Using the Knuth LR(1) canonical algorithm, Dragon takes 20 minutes 30 seconds, but Hyacc uses only 3.529 seconds. The ratio is about 500:1. Use the PGM algorithm, Parsing takes 5 minutes 17 seconds, but Hyacc only uses 1.779 seconds. The ratio is about 180:1.

Memory wise, the difference is not so big. For the Knuth LR(1) algorithm, Hyacc uses 4-times more memory than Dragon. For PGM LR(1), Hyacc uses one tenth the memory of Parsing. For the latter, it is quite advantageous.

Parser Generator	Implemented in	Environment	Testing grammar	Algorithm	Time (sec)	Memory (MB)
Dragon (2007)	C++	iMac Dual Core 2GHz 1GB RAM	Cego: 120 Terminals 350 productions	Knuth LR(1)	20m30s	30
Parsing module (2007)	Python	Opteron 2.2GHz	Lyken: 104 Terminals 117 N.T. 525 productions	PGM LR(1)	5m17s	220
Hyacc (2008)	C	Intel Pentium 1.7GHz 1GB RAM Fedora Core 4.0	C++ 5.0: 101 Terminals 164 N.T. 643 productions	Knuth LR(1)	3.529	122.7
				PGM LR(1)	1.779	23.9
				LT LR(1) w/ PGM	1.261	39.1
				LT LR(1) w/ LTT	2.045	34.4

Table 5.12: Comparison with other parser generators

5.4.2 Comparison to Menhir and MSTA

Menhir and MSTA are also efficiently implemented and run in speeds comparable to Hyacc.

Table 5.13 shows a comparison of the generated parsing table. MSTA and Hyacc generate the same number of states in canonical LR(1) parsing machine. However, the corresponding canonical LR(1) parsing machine generated by Menhir can be significantly smaller. Obviously Menhir used some other optimizations, such that its "canonical LR(1) parsing machine" is not the actual "canonical LR(1) parsing machine", but compressed. MSTA also compresses its generated parsing machine, but not as much as Menhir.

Table 5.14 shows the comparison of conflicts. MSTA and Hyacc always generate the same number of shift/reduce and reduce/reduce conflicts for the same algorithms. But Menhir uses different notations, and has smaller values.

Table 5.15 shows the running time comparison. It is similar for all the three parser generators. There is no significant difference in the measured values.

	C++			C		
	Knuth LR(1)	PGM LR(1)	LALR(1)	Knuth LR(1)	PGM LR(1)	LALR(1)
Menhir	4325	1238	-	1172	351	-
MSTA	9724/8413 [1]	-	1237/1196 [2]	1575/1572	-	352/338
Hyacc	9723	1384	1236	1574	351	351

Table 5.13: Parsing table size comparison

	C++			C		
	Knuth LR(1)	PGM LR(1)	LALR(1)	Knuth LR(1)	PGM LR(1)	LALR(1)
Menhir	15/4/15/12 [3]	8/2/8/6	-	1/0/1/0	1/0/1/0	-
MSTA	280 s/r, 31 r/r	-	24 s/r, 18 r/r	2 s/r	-	1 s/r
Hyacc	280 s/r, 31 r/r	24 s/r, 18 r/r	24 s/r, 18 r/r	2 s/r	1 s/r	1 s/r

Table 5.14: Conflict comparison

	C++			C		
	Knuth LR(1)	PGM LR(1)	LALR(1)	Knuth LR(1)	PGM LR(1)	LALR(1)
Menhir	1.971s	1.484s	-	1.640s	0.557s	-
MSTA	5.319s	-	1.175s	0.918s	-	0.130s
Hyacc	3.529s	1.779s	1.101s	1.047s	0.420s	0.189s

Table 5.15: Running time comparison

[1] For MSTA, a/b means this in output: a canonical LR-sets, b final states

[2] For MSTA, a/b means this in output: a LALR-sets, b final states

[3] For Menhir, a/b/c/d means this in the output: a states have shift/reduce conflicts, b states have reduce/reduce conflicts, c shift/reduce conflicts were arbitrarily resolved, d reduce/reduce conflicts were arbitrarily resolved.

5.5 Conclusion

5.5.1 LR(1) and LALR(1) Algorithms

As expected, the Knuth canonical LR(1) algorithm is still quite expensive in both running time and space. The generated parsing machine is big. That said, for the given 13 programming language grammars, it is practical on today's hardware. The most complex grammar of these, the grammar of C++ 5.0, contains 101 terminals, 186 non-terminals and 665 rules. It costs less than 4 seconds and about 120 MB memory to generate the parsing machine for the C++ 5.0 grammar. More complex grammars may contain thousands of rules. But for such grammars, it is still acceptable for the typical hardware configuration of a personal computer.

That said, considering the theoretical implication and actual performance advantage of reduced-space LR(1) algorithms, we should always use reduced-space algorithms for faster running speed and less memory usage, as well as a smaller generated parsing machine.

The practical general method and the lane-tracing algorithm are such reduced-space LR(1) algorithms. For the given programming language grammars, they both generate parsing machines with size close to those of LALR(1) parsing machines, and time and space requirements not much more expensive than LALR(1). For LALR(1) grammars, these reduced-space LR(1) algorithms generate the same parsing tables as those by LALR(1) algorithm. Only for LR(1) grammars it is more expensive. In this sense, we can adequately replace LALR(1) parser generators with LR(1) ones, with no worry in modifying existing projects, and less worry for projects to come.

Comparing the three redueced-space LR(1) algorithms (PGM, LT LR(1) w/ PGM and LT LR(1) w/ LTT), LT LR(1) w/ PGM in general creates a smaller parsing machine than the other two.

Comparing LT LR(1) w/ PGM and LT LR(1) w/ LTT, measurements show that the latter takes more running time when there are many states to split, but less running space.

The current implemention of practical general method is based on the concept of weak compatibility. The strong compatibility may obtain more compression, but requires more computation and is harder to implement. It should be satisfying to use weak compatibility. On the other hand, the practical general method based on weak compatibility is much easier to understand and implement than the lane-tracing algorithm. From the point of view of a LR(1) parser generator author, there is no reason to go for lane-tracing instead of the practical general method.

However, the advantage of the lane-tracing algorithm is that it is easier to extend to LR(k), since it only works on those configurations and states relevant to resolve reduce/reduce conflicts. This is especially true for the LT LR(1) w/ LTT approach. The practical general method, however, has to handle the entire context tuples for all the configurations and states, and thus becomes more expensive for increasing k.

5.5.2 The Unit Production Elimination Algorithm and Its Extension Algorithm

The goal of the unit production elimination algorithm is to remove those intermediate states relevant to unit productions only, obtain a smaller parsing machine, and thus reduce parsing time. However this goal in some way is weakened by the existence of extra redundant states in the resulted parsing machine, and the percentage of redundancy can be big. The extension algorithm brings down the size of the generated parsing machine significantly.

Although the extension algorithm does not require extra space to run other than needed by the unit production elimination procedure itself, may need much longer running time. This is the disadvantage. This is evident for those complex grammars in this survey, such as the grammar of C++ 5.0. However since parser generation is a one-time process, it should be worth such an effort.

Another "problem" with the unit production elimination algorithm is that for non-LR(1) grammars that contains shift/reduce or reduce/reduce conflicts, even though these may be hidden by using associativity and precedence directives, may cause abundant shift/shift conflicts. Thus this algorithm should be applied only to pure LR(1)/LALR(1) grammars. In reality, most grammars are not pure LR(1)/LALR(1). Whether the unit production elimination algorithm and its extension algorithm can be used for these cases can only be determined by experiment. If shift/shift conflicts occur then the answer is no, otherwise it is possible to do so.

5.5.3 Hyacc and Other Parser Generators

Hyacc is a highly efficient parser generator. Compared to many not so carefully implemented LR(1) parser generators, or LR(1) parser generators implemented in less efficient programming languages, Hyacc has obvious performance advantages. Furthermore, with reduced-space LR(1) algorithms, which are not usually available in other LR(1) parser generators, Hyacc should be a quite favorable choice.

Menhir and MSTA are both efficient parser generators. The Menhir parser generator also implements the practical general method. It is just in a not so popular programming language Caml, thus the potential user base is smaller. MSTA, implemented in C++, should be another handy choice for industry users. However it does not include the reduced-space LR(1) algorithms, thus always comes out with larger parsing machines. Thus Hyacc has advantages compared to both Menhir and MSTA.

MSTA also implements LR(k). It is the only parser generator that declares full LR(k) functionality in the industry setting. Current casual testing did not shown any problem with its LR(k) functionality. But without any careful study and analysis on its LR(k) algorithm (which is not available to our literature review), it is not known whether its generated output is always correct.

Hyacc takes its LR(k) approach based on lane-tracing. So far it works well for some frequently used LR(k) grammar examples. It still can not handle situations where lane-tracing upon increasing k contains cycle, we will work on this later.

In addition to LR(1) and LR(k), Hyacc also implements LALR(1) based on lane-tracing phase 1. The comparison study shows it generates the same parsing machines as generated by LALR(1) algorithm in Bison. This validates our approach, and provides an implementation to another LALR(1) approach unaware of by the general public.

Hyacc also implements the LR(0) algorithm. Although not very useful for real world situations, it nevertheless has education value for academic users.

Besides the algorithms used, Hyacc has an advantage over many other parser generators in that it uses a command line interface very similar to Yacc and Bison. Many parser generators choose to invent new interface features and new grammar input formats, and for new users there is a steep learning curve barrier. With a user interface familiar to a large existing user base, Hyacc should be easier to use and gain a wider user acceptance.

Finally, Hyacc is written in ANSI C, so can be easily ported to most platforms. It has been released to the open source community so is easy to obtain.

In conclusion, Hyacc is unique in its wide span of algorithm coverage, efficiency, portability, usability and availability. In the parser generator community there have been abundant LALR(1) and LL(k) implementations, Hyacc aims at becoming a practical tool to the community from the LR(1)/LR(k) side.

Chapter 6

LR(k) Parser Generation

This chapter discusses LR(k) parser generation based on lane-tracing.

The exponential behavior of LR(k) parser generation comes from two sources: 1) the number of states in the parsing machine, and 2) the number of context tuples of the configurations. The reduced-space LR(1) algorithms we have discussed can solve the first problem. The second problem can be solved following the way of Terence Parr [52], or as in the edge-pushing algorithm discussed here by only working on those configurations that actually lead to reduce/reduce conflicts and ignore the rest.

These problems should be solved when extending lane-tracing to LR(k).

1) Algorithm for LR(k) extension. This part should extend the lane-tracing LR(1) parser generation algorithm, such that it can be recursively applied where reduce/reduce conflicts cannot be resolved by available lookaheads.

2) Storage of LR(k) parsing table. This part should efficiently represent the LR(k) lookaheads in LR(k) parsing table and work well together with the LR(k) algorithm.

3) Parse engine change. After the LR(k) parsing table is generated, the LR(1) parse engine should be modified so it can make use of the LR(k) parsing table for parsing LR(k) grammars.

6.1 LR(k) Algorithm

For the ease of discussion, we define the terms and functions below.

By *head configuration* we mean those configurations at the start of a lane where we stop in lane-tracing. On the contrary, by *tail configuration* we mean those configurations at the end of a lane from which we start the lane-tracing.

Define the *function* $theads(\alpha, k)$ to return a set of terminal strings obtained from α. The length of these strings is k. This function is potentially exponential on k. $theads(\alpha, k)$ is the same as $FIRST_k(\alpha)$ defined in many other literature.

6.1.1 LR(k) Parser Generation Based on Recursive Lane-tracing

Each time after LR(k) lane-tracing for a fixed k, we need to check if conflicts are resolved, and further trace LR(k+1) only for states with unresolved reduce/reduce conflicts.

It is possible to do LR(k) lane-tracing on a specific k for each inadequate state, then increase k and do this again on all unresolved states. This is as shown in Algorithm 6.1.

Algorithm 6.1: LRk_v1()

```
1 k ← 1;
2 while the inadequate states list is not empty do
3     k ← k + 1;
4     foreach inadequate state S do
5         do_LRk_lane_tracing(S, k);
6         if the inadequacy of state S is resolved OR lane_tracing ends at state 0 then
7             remove S from inadequate states list;
```

It is also possible to do LR(k) lane-tracing recursively on one inadequate state until its inadequacy is resolved or found not resolvable, then start on another inadequate state. This is as shown in Algorithm 6.2. The only complication is on states involved in a cycle.

Algorithm 6.2: LRk_v2()

```
1 foreach inadequate state S do
2     k ← 1;
3     while inadequacy of S is not resolved AND lane_tracing does not end at state 0
      do
4         k ← k + 1;
5         do_LRk_lane_tracing(S, k);
```

For both algorithms above, the complications occur when states are involved in a cycle and states shared by lane-tracing originated from different inadequate states. The second may be easier since if we want to take advantage of the result of LR(k) for LR(k+1), the second algorithm allows us to remember less intermediate information.

Note that checking LR(k) conflict resolution can be done when inserting the LR(k) entry into the extended LR(k) parsing table: when inserting a entry, if a different reduction action already exists in the same location, we know a reduce/reduce conflict occurs. This is the same as for LR(1) in Hyacc, in which conflicts are detected when inserting into the parsing table. There we may have shift, reduce, accept actions, but here we only have reduce actions, so it is easier to handle here.

The purpose here is to trace back all the way until we find relevant contexts of length k that solve the reduce/reduce conflicts of inadequate states. The word "relevant" here means we only get contexts that are useful to resolving conflict: only for those contexts that cause conflicts, we trace further. This is needed because the number of LR(k) contexts can increase exponentially with k. We also should better cache the computation of LR(k) theads, so computation of LR(k+1) theads can start from those relevant edges.

In order to get enough contexts to resolve conflicts, we may need to go back several levels of states. Two essential procedures are: 1) lane-tracing, 2) calculation of theads on the string following the scanned symbol of the relevant configurations.

Here are three possible solutions for LR(k) lane-tracing:

1) Fixed k.

This is shown in Algorithm 6.3.

Algorithm 6.3: FixedK(S, k)

Input: state S; number k as in LR(k)

Output: true - conflict resolved, false - failed to resolve conflict

```
1  foreach final config C_f of state S do
2      C_s ← head configuration of C_f;
3      get_LRk_context(C_s, k);
4  if all reduce/reduce conflicts are resolved then
5      add context obtained to LR(k) parsing table;
6      return true;
7  else
8      return false;
```

Algorithm 6.4: get_LRk_context(C, k)

Input: Start configuration C; integer k: length of context to get

Output: an array of context strings

```
1  result ← {};
2  c_traced ← false;
3  let C be: A → α • B β;
4  ctxt ← theads(β, k);
5  foreach string x in ctxt do
6      if x.length == k then
7          add x to the result array;
8      else
9          if c_traced == false then
10             do lane-tracing on C to obtain a set ψ of head configurations of C;
11             c_traced ← true;
12         foreach configuration D in ψ do
13             array Y = get_LRk_context(D, k - x.length);
14             concatenate x to each string in Y and add to the result array;
15 return result;
```

The need to increase k naturally leads to the next algorithm.

2) Auto-inc k based on Fixed k.

This is dependent on algorithm FixedK(S, k).

Algorithm 6.5: AutoIncK(S)

```
Input: State S
1 resolved ← false;
2 k ← 2;
3 repeat
4 |   resolved ← FixedK(S, k);
5 |   k ← k + 1;
6 until resolved == true ;
```

The problem with this algorithm is that it always repeats the computation for each LR(k) context, even for those that do not have a conflict associated. E.g., suppose two reductions r1 and r2 both have conflicted LR(1) context {'a'}. Then for the LR(2) context, r1 has context set {'ab', 'ac'}, r2 has context set {'ab', 'ad'}. Then we only need to continue with LR(3) on the configurations that generate conflict 'ab'. But the current algorithm also computes the configurations that generate 'ac' and 'ad'.

To avoid this problem, we can improve by only computing the configurations that generate 'ab'. We call such a method "edge-pushing" and show it below.

3) Auto-inc k based on edge-pushing.

See the next section for this algorithm.

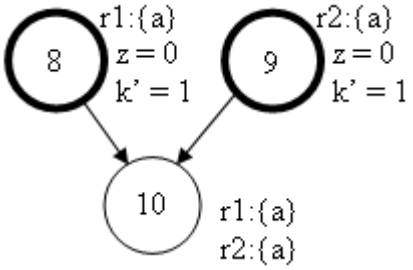
6.1.2 Edge-pushing Algorithm: A Conceptual Example

Below we will show a conceptual description of LR(k) lane-tracing with edge-pushing. Real examples will be shown in section 6.6 (on page 150).

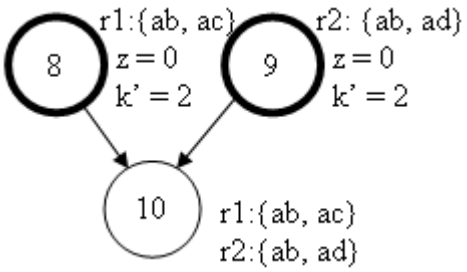
Here the starting state is state 10 with a LR(1) reduce/reduce conflict. Four steps are carried out from LR(2) to LR(5).

In each graph, the circles with bolded edge are at the cutting edge of lane-tracing. Circle 10 stands for state 10. Circles 3 to 9 actually should mean a *head configuration* in states 3 to 9, and here by saying state 3 we actually mean a head configuration in state 3. at the tail of a lane from which we start the lane-tracing. z is a local variable associated with a *head configuration*, whose value is obtained by adding together z and k' of the *tail configuration*. k' is the value used in the calculation of $theads(\beta, k')$ for a configuration A $\rightarrow \alpha \bullet$ B β. k' = k - z, where k is the k in LR(k), and z is the local z. The graphs below shows an example of the calculation of these values, and how LR(k) lane-tracing is pushed at the cutting edge.

LR(1) state with reduce/reduce conflict, obtained by LR(1) lane-tracing:

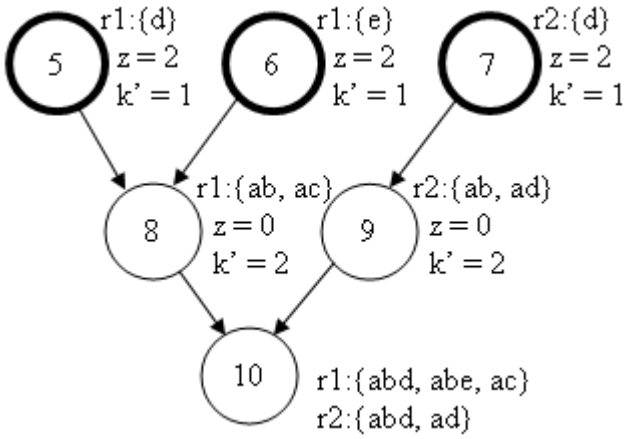


LR(2) states:



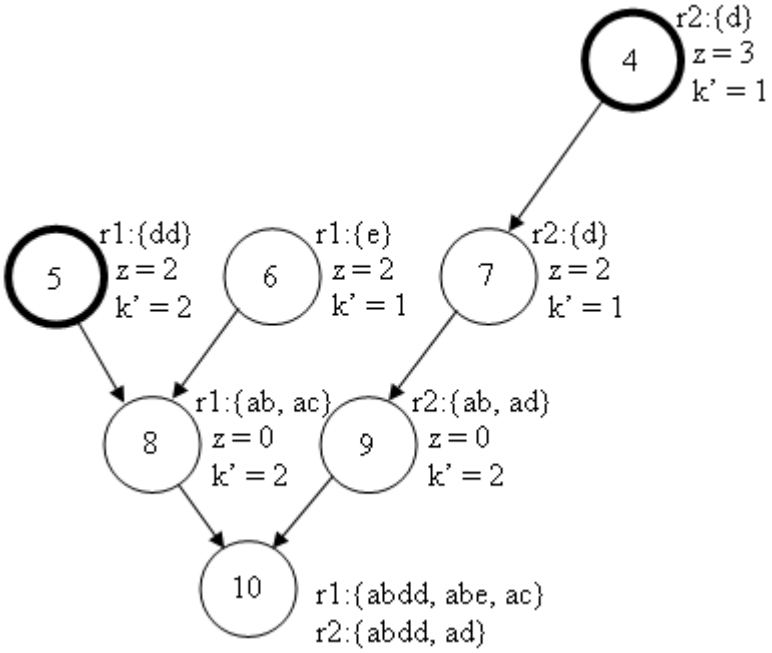
Here states 8 and 9 are at the cutting edge of lane-tracing. For state 8, on its right side "r1: {b, c}" means that 'b' and 'c' are the context symbols generated by a configuration in state 8 for reduction r1. Local "z = 0". It is always 0 for head configurations in LR(1) and LR(2) lane-tracing. The value of local k' is obtained by subtracting the local z from the k in LR(k): k'_8 = k - z_8 = 2 - 0 = 2. The meanings of notations are similar for state 9.

LR(3) states:



Here states 5, 6 and 7 are at the cutting edge of LR(3) lane-tracing. The only thing that needs explanation is the value of z. For state 5, z is obtained by the sum of k' and z in its tail configuration in state 8: $z_5 = k'_8 + z_8 = 2 + 0 = 2$. Similarly z is obtained this way in states 6 and 7.

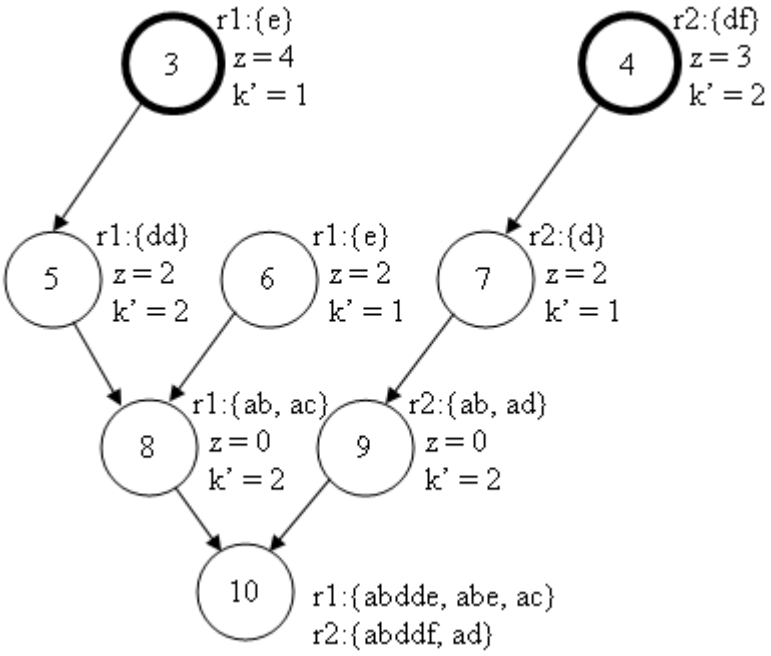
LR(4) states:



In state 5 we get two consecutive context symbols "dd" by doing $theads(\beta, k'_5)$ calculation where $k'_5 = k - z_5 = 4 - 2 = 2$. In state 7, $k'_7 = k - z_7 = 4 - 2 = 2$, $theads(\beta, k'_7)$ returns 'd' whose length is less than 2, so we have to do lane-tracing here to obtain state 4 as shown. In the head

configuration in state 4 z_4 = k'$_7$ + z_7 = 2 + 1 = 3, k'$_4$ = k - z_4 = 4 - 3 = 1, and we do $theads(\beta, k'_4)$ to obtain context 'd'.

LR(5) states:



Here states 3 and 4 are at the cutting edge of lane-tracing, and we obtain the values of z and k' for them in the same way as before:

z_3 = z_5 + k'$_5$ = 2 + 2 = 4, k'$_3$ = k - z_3 = 5 - 4 = 1.
z_4 = 3 was obtained in the last step, k'$_4$ = k - z_4 = 5 - 3 = 2.

Now, we don't need to add any context to state 10's final configurations, because the LR(k) parsing tables (LR(1) parsing table, LR(2) parsing table, ..., LR(5) parsing table) suffice for both storage of contexts as well as conflict detection. These LR(k) parsing tables are shown in the next page.

These are the corresponding LR(1) to LR(5) parsing tables. $\bigotimes$ means a reduce/reduce conflict. See section 6.3 for the exact notations on storage of LR(k) parsing tables.

LR(1) parsing table:

state/token	...	a	...
...			
10		$\bigotimes$	
...			

LR(2) parsing table:

(state, LR(1) lookahead)/token	b	c	d
(3, a)	$\bigotimes$	r1	r2

LR(3) parsing table:

(state, LR(2) lookahead)/token	d	e
(3, b)	$\bigotimes$	r1

LR(4) parsing table:

(state, LR(3) lookahead)/token	d
(3, d)	$\bigotimes$

LR(5) parsing table:

(state, LR(4) lookahead)/token	e	f
(3, d)	r1	r2

These tables are created when we do the LR(k) lane-tracing. Whenver a conflict is found: e.g., when inserting action r2 to a field we find a r2 action already exists in the same cell, then we know a conflict occurs, and then we pass the two relevant configurations to the next round of lane-tracing.

When parsing an input string, we follow the LR(1), ... LR(k) parsing tables to find a match. Suppose during a parse we are in state 10, and the next few lookaheads of the input string are "abddf". The first lookahead symbol 'a' gets us a $\bigotimes$ action which denotes a reduce/reduce conflict, so we take the second symbol 'b' and go to the LR(2) parsing table. There we find another $\bigotimes$ action so need to take the third symbol 'd' and go to the LR(3) parsing table. This chain of actions stops at the LR(5) parsing table, where the 5th lookahead 'f' denotes an action 'r2', which means to reduce by rule 2.

6.1.3 The Edge-pushing Algorithm

Next let us summarize the *edge-pushing algorithm* below.

Algorithm 6.6: edge_pushing(S)

Input: Inadequate state S

```
1  Let Set_C and Set_C2 be ⊘ (i.e., empty sets);
2  k ← 1;
3  foreach final configuration T of S do
4      T.z ← 0;
5      Let C be T's head config, and X be the context generated by C;
6      add triplet (C, X, T) to set Set_C;
7  while Set_C ≠ ⊘ do
8      k ← k + 1;
9      foreach triplet (C: A → α • B β, X, T) in Set_C do
10         k' ← k - C.z;
11         calculate ψ ← theads(β, k');
12         foreach context string x in ψ do
13             if x.length == k' then
14                 insert_LRk_PT(S, X, last symbol of string x, C, T, Set_C2);
15             else if x.length == k' - 1 then
16                 Σ ← lane_tracing(C); // Σ is a set of head configurations
17                 foreach head configuration σ in Σ do
18                     σ.z ← C.z + k';
19                     let m be the generated context symbol in σ;
20                     insert_LRk_PT(S, X, m, σ, T, Set_C2);
21     Set_C ← Set_C2;
22     Set_C2 ← ⊘;
```

Function insert_LRk_PT() is defined below.

Algorithm 6.7: insert_LRk_PT(s, X, col, γ, config T, Set_C2)

Input: State s; token_list X; token col; configuration γ; final configuration T; set Set_C2

```
1  foreach token x in token_list X do
2      Let c0 be the LR(k) parsing table entry at [(s, x), col];
3      if c0 does not exist then
4          add entry (γ, T) to LR(k) parsing table entry [(s, x), col];
5      else
6          if c0.tail is ⊗ then
7              add triplet (γ, col, T) to Set_c2;
8          else
9              add entry (γ, T) to LR(k) parsing table entry [(s, x), col];
10             add triplet (γ, col, T) to Set_c2;
11             add triplet (c0.head, col, c0.tail) to Set_c2;
```

Note that here each LR(k) parsing table entry is a pair (head, tail). The tail is the configuration where conflict occurs and from which we start lane-tracing. The head is the configuration where lane-tracing ends. Each element of Set_c2 contains the triple (C, X, T), where C is the head configuration, T is the tail configuration, and X is the context generated by C.

Some discussion of the algorithm follows.

This edge-pushing algorithm uses iteration and avoids recursion. This is achieved with the use of a queue by putting the configurations to be processed in the next round on the queue. Two configuration sets are used between iterations, such that during a round of iteration, we draw an element from the working set, process it and add new derived configurations to the derived set; then at the end of the iteration, pass the elements of the derived set to the working set and start the next iteration.

The edge-pushing algorithm eventually stops because when lane-tracing back to state 0, line 16 (see Algorithm 6.6 last page) "$\Sigma \leftarrow$ lane_tracing(C)" will return an empty set. So eventually Set_C2, and thus Set_C, becomes an empty set.

At the beginning we initialize the variable z of relevant final configurations to 0, then obtain the z values for derived configurations recursively.

Each time lane-tracing is done, we only use the LR(1) theads of the new head configurations. This is easier. In comparison, in algorithm FixedK() when lane-tracing is involved, we may need to go several rounds of lane-tracing recursively to get all the LR(k) theads, which is much harder.

We don't need to attach any context to the initial inadequate states' final configurations, because here the LR(k) parsing tables can store the LR(k) context symbols as well as detect possible conflicts.

Here since we only push the cutting edge of lane-tracing, we avoid recalculating those edges that don't cause conflict. But due to the exponential nature of $theads(\alpha, k)$, we potentially still may have an exponential problem.

However, for the entire parsing machine, the number of inadequate states is small. Further, those configurations that cause conflicts for increasing k may be just a portion of all such initial configurations, i.e., configurations that we should trace further for LR(k+1) usually are just a small portion of the configurations that we trace for LR(k). Thus we should expect a below exponential increase in most cases. This is similar to what is achieved by Terrence's LL(*) parser generator ANTLR.

In summary, conceptually this algorithm works well. However there are lots of details other than what we have discussed, and implementation is not easy. Two major components involved are lane-tracing and calculation of $theads(\alpha, k)$.

6.1.4 Edge-pushing Algorithm on Cycle Condition

We have described the edge-pushing algorithm on one state. Complication is involved when lane-tracing of different inadequate states come into the same configurations (e.g. Figure 6.1 (a) joint), or when cycles are involved in the tracing (e.g. Figure 6.1 (b) cycle).

Using Figure 6.1 (b) as an example, if LR(k) tracing ends at state B and can't resolve the reduce/reduce conflict, LR(k+1) tracing ends at state C and can't resolve the conflict, LR(k+2) tracing ends at state D and still can't resolve the conflict, then LR(k+3) tracing will come back to state B. This forms an infinite cycle: B $\rightarrow$ C $\rightarrow$ D $\rightarrow$ B ...

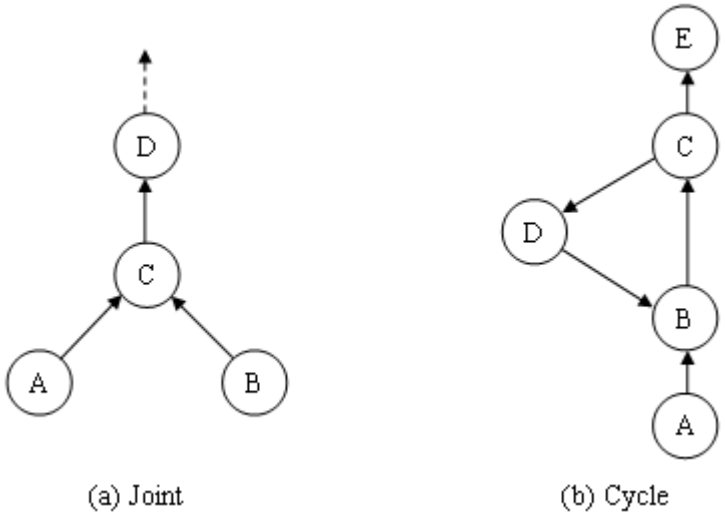


Figure 6.1: LR(k) lane-tracing: joint and cycle conditions

Then we need to answer two questions: 1) Do we need to cache a previous computation? 2) what to do with the parameters k' and z?

For 1), it is obvious that caching a previous computation has advantages. Once we do cache, then we also don't need to worry about 2), since we can refer to the cache for contexts generated. We don't need to care about k' and z, since these are used only if we do the computation.

So every time after lane-tracing and we obtain a set of head configurations, we then search in the cache. If it already exists in the cache, then we get the contexts generated from the cache, and also the next round of head configurations from the cache.

The problem of cycles can be solved by using a cache as described above. The only unique problem with cycles is cycle detection: how to avoid tracing down the cycle infinitely.

The cache part so far has only partially been implemented in Hyacc, and will be among the future work.

6.2 Computation of $theads(\alpha, k)$

6.2.1 The Problem

The algorithm of lane-tracing for LR(k) grammars depends on the calculation of $theads(\alpha, k)$.

Let $theads(\alpha, k)$ be the set of thead (terminal head) symbols of length k for string α. One of the difficult parts of LR(k) is calculating $theads(\alpha, k)$.

Figure 6.2 shows an example of the need for getting more context for increasing k in LR(k) lane-tracing.

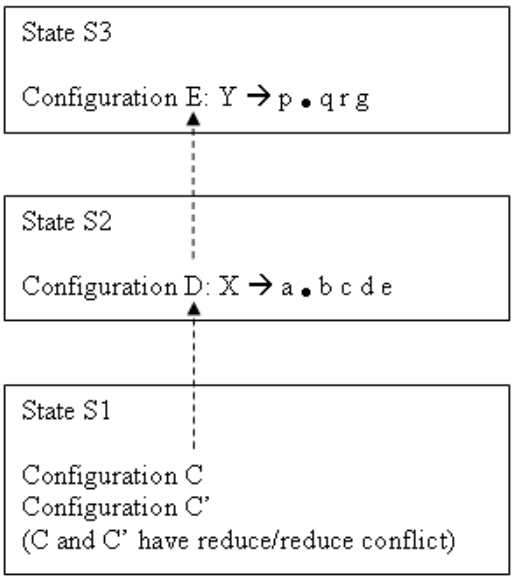


Figure 6.2: The need of getting more context for increasing k in LR(k) lane-tracing

Assume that for an inadequate state S1, one of its final configurations is C. When we do LR(1) lane-tracing on configuration C, we end up at configuration D in state S2. Let configuration D be: X $\rightarrow$ a • b c d e, which generates context c, and we stop lane-tracing here with context {"c"}.

If the grammar is LR(2), then we stop lane tracing here with context {"cd"}.

If the grammar is LR(3), then we stop lane tracing here with context {"cde"}.

If the grammar is LR(4), then we cannot stop here, we need to get more lookahead symbols. From configuration D, we calculate $theads(\alpha, 4)$, where α is "cde". We get "cde", whose actual

length is 3 and less than 4. So we have to do lane-tracing on D to get more lookahead symbols from its predecessor configurations. Let's say the lane-tracing ends at configuration E in state S3: Y → p • q r g. So we get the extra lookahead symbol: "r". Thus for LR(4) grammar, we stop lane tracing with context set {"cder"}.

If the grammar is LR(5), then we stop lane tracing with context {"cderg"}.

In summary, the non-trivial key component of LR(k) is the algorithm to calculate $theads(\alpha, k)$.

An algorithm to get $theads(\alpha, 1)$ was given as Algorithm 4.6, which first calculates $heads(\alpha, 1)$, then removes non-terminals from the list to get $theads(\alpha, 1)$. A more efficient way used in Hyacc was given as Algorithm 4.8.

For $theads(\alpha, k)$, a new algorithm is needed. There is a mechanical way of doing it, but it is not efficient, and sometimes may not be practical. The mechanical way is based on expanding all non-terminals to terminals, but this is not always possible.

Example 1. α is "Nbc", where N → s t. Just plug "st" into "Nbc" so it is "stbc". Calculation of $theads(\alpha, k)$ is easy.
$theads(\alpha, 1)$ = {"s"},
$theads(\alpha, 2)$ = {"st"},
$theads(\alpha, 3)$ = {"stb"},
$theads(\alpha, 4)$ = {"stbc"}.

Example 2. α is "Nbc", where N → N s | ϵ, and ϵ is empty string. Then actually N is s*.
$theads(\alpha, 1)$ = {"s", "b"},
$theads(\alpha, 2)$ = {"ss", "sb", "bc"},
$theads(\alpha, 3)$ = {"sss", "ssb", "sbc"},
$theads(\alpha, 4)$ = {"ssss", "sssb", "ssbc"}.

Example 3. α is "NMbc", where N → N s | ϵ, M → M t | ϵ. Then actually N is S*, and M is t*.
$theads(\alpha, 1)$ = {"s", "t", "b"},
$theads(\alpha, 2)$ = {"ss", "st", "sb", "tt", "tb", "bc"},
$theads(\alpha, 3)$ = {"sss", "sst", "ssb", "stt", "stb", "sbc", "ttt", "ttb", "tbc"},
$theads(\alpha, 4)$ = {"ssss", "ssst", "sssb", "sstt", "sstb", "ssbc", "sttt", "sttb", "stbc", "tttt", "tttb", "ttbc"}.

It is easy to see the complexity of $theads(\alpha, k)$ increases very fast. Thus if the algorithm of lane-tracing of LR(k) grammars depends on $theads(\alpha, k)$, its complexity also increases very fast.

Also if the grammar rules for N and M in examples above are not so simple, say if it's a "formular" element in a grammar for math expressions, or it's a "sentence" element in a grammar for the English language, then it may be hard to calculate $theads(\alpha, k)$.

The calculation of $theads(\alpha, k+1)$ also may not be able to take advantage of the result of $theads(\alpha, k)$. For example, let A $\rightarrow$ c d e, B $\rightarrow$ f g. $theads(AB, 4)$ may be able to take advantage of the result of $theads(AB, 3)$ in that we can store $theads(A, 3)$ = "cde" and get $theads(B, 1)$ = "f", then concatenate these two parts to get $theads(AB, 4)$ = "cdef". However, $theads(AB, 3)$ can not take any advantage from the result of $theads(AB, 2)$ this way.

In summary, the question is: the algorithm of lane-tracing on LR(k) grammar depends on the calculation of $theads(\alpha, k)$. But it is not always practical to do, since evaluation of the k-heads of strings can involve determining sets whose maximum size increases exponentially with increasing value of k.

On the other hand, we see that in practical real grammars, the set of k-heads may not be impractically large for any k. In lane-tracing, lanes that lead back to state 0 cannot be traced further, and no larger context is possible beyond this point.

Also if the evaluation of contexts is carried out at compile time, we can terminate the tracing of lanes that do not generate the k-head of the remaining input. This is relevant in the discussion of lane-tracing at compile time in section 6.7.

6.2.2 Literature Review on $theads(\alpha, k)$ Calculation

Our study of the LR(k) algorithm based on lane-tracing shows that the calculation of LR(k) lookahead is one of the major steps. We proposed the edge-pushing LR(k) algorithm (Algorithm 6.6), which depends on a function $theads(\alpha, k)$ to calculate k-lookahead of a string α. $theads(\alpha, k)$ is also referred to as $FIRST_k(\alpha)$ in some other literature. A literature survey gave the following information.

The work of DeRemer and Pennello [25] on "Efficient computation of LALR(1) look-ahead sets" and the work of Kristensen and Madsen [17] on "Methods for computing LALR(k) lookahead" are for LALR grammars.

The PhD thesis of Terrence [52] proposed a way to compute $FIRST_k(\alpha)$. This seems to be the only one that we found to work. We believe it works since it is the one used in ANTLR. Basically, a data structure called GLA (Grammar Lookahead Automata) is used to represent grammars. To calculate LR(k) lookahead, just do a bounded walk of a GLA, and the lookaheads are stored as a lookahead DFA (Deterministic Finite Automata). Terrence's PhD thesis uses one chapter to introduce the GLA grammar representation, and another chapter to explain lookahead computation and representation. This algorithm needs different fundamental data structures from the current ones of Hyacc.

Some people at the comp.compiler discussion group pointed out that the lecture notes of Kwang-Moo Choe [21] includes a method to compute LR(k) lookahead. The algorithm quoted by Choe is actually given by Aho and Ullman as Algorithm 5.5 in [14].

Eventually a $FIRST_k(\alpha)$ algorithm is given by Pager [49] and used in Hyacc. Details of this algorithm is given in the next section. Compared to the algorithm of Aho and Ullman, which uses a bottom-up process, this algorithm takes a top-down approach and is quite different.

6.2.3 The $theads(\alpha, k)$ Algorithm Used in Hyacc

We define $theads(\alpha, k)$ to be the set of terminal heads of string α, where the length of each terminal head string is k.

To ease illustration of the algorithm, let's define the following functions:

For string $\alpha = x_1x_2...x_n$, $h_v(\alpha, k)$ is a substring of α that consists of the head of α up to the k-th symbol that does not vanish, or the entire α string if it contains less than k symbols that do not vanish.

For string $\alpha = x_1x_2...x_n$, let $prod(\alpha, j)$ be the set of strings obtained by applying all possible productions to the j-th symbol x_j of α.

The $theads(\alpha, k)$ algorithm used in Hyacc is given below [49].

Algorithm 6.8: theads(α, k)

Input: string α: $x_1x_2...x_n$; integer k: length of theads

Output: the k-head set H of α: $\{\beta \mid \beta = theads(\alpha, k)\}$

```
1  Let H be the set of k-heads that we want to obtain;
2  Let S be the set of heads whose length is less than k;
3  Let L be a list of strings;
4  Initially, L is empty, add h_v(α, k) to L;
5  for j = 1 to k do
6      foreach string β in L do
7          φ = prod(α, j);
8          foreach string γ in φ do
9              add h_v(γ, k) to the end of L if it is not in L yet;
10     remove from L all strings whose j-th symbol is a non-terminal;
11     remove from L all strings whose k-heads are all terminals, and add these
       k-heads to H;
12     remove from L all strings of length less than k that consist of terminals only,
       and add these strings to set S if needed;
```

The following example is from [49].

Example. Given grammar G6.1 below, find $theads(XYZU, 2)$.

X $\rightarrow$ Y | x | ϵ

Y $\rightarrow$ Z | y | ϵ

Z $\rightarrow$ X | z | ϵ

U $\rightarrow$ u

First round of operation for j = 1 is:

j	i-th string to process	string added to L	string seq. no.
1	1	XYZU	1
		YYZU	2
		xYZU	3
		YZU	4
	2	ZYZU	5
		yYZU	6
	3		
	4	ZZU	7
		yZU	8
		ZU	9
	5	zYZU	10
	6		
	7	XZU	11
		zZU	12
	8		
	9	XU	13
		zU	14
		U	15
	10		
	11	xZU	16
	12		
	13	YU	17
		xU	18
	14		
	15	u	19
	16		
	17	yU	20
	18		
	19		
	20		

Remove all strings with non-terminals in the j-th (first) position, remove all strings containing 2-heads, and remove all strings of length less than 2 and contains only terminal strings, we have H = {}, S = {u}.

Now we start the next round where j = 2:

j	i-th string to process	string added to L	string seq. no.
2		xYZU	1
		yYZU	2
		yZU	3
		zYZU	4
		zZU	5
		zU	6
		xZU	7
		xU	8
		yU	9
	1	xZZU	10
		xy	11
	2	yZZU	12
		yy	13
	3	yXU	14
		yz	15
	4	zZZU	16
		zy	17
	5	zXU	18
		zz	19
	6	zu	20
	7	xXU	21
		xz	22
	8	xu	23
	9	yu	24
	10	xXZU	25
	11		
	12	yXZU	26
	13		
	14	yYu	27
		yx	28
	15		
	16	zXZU	29
	17		
	18	zYU	30
		zx	31
	19		
	20		
	21	xYU	32
		xx	33

j	i-th string to process	string added to L	string seq. no.
(cont.)	22		
	23		
	24		
	25		
	26		
	27		
	28		
	29		
	30		
	31		
	32		
	33		

Remove all strings with non-terminals in the j-th (second) position, remove all strings containing 2-heads, and remove all strings of length less than 2 and contains only terminal strings, we have H = {xy, yy, zy, zz, zu, xu, xz, yz, yx, yu, zx, xx}, S = {u}.

6.3 Storage of LR(k) Parsing Table

We keep the current 2-dimensional matrix format of the LR(1) parsing table. For LR(k) extension, a 2-dimensional matrix is used for each fixed k. For an LR(k) grammar, there is an array of parsing tables for each of LR(1), LR(2), ..., LR(k). When resolving conflicts, if the LR(i) parsing table can't do it, we consult the LR(i+1) parsing table. This repeats until the conflict is resolved.

A special symbol ($\bigotimes$) is used to denote a reduce/reduce conflict in the parsing table.

In the LR(k) parsing table ($k \geq 2$), each column represents a lookahead token as in the LR(1) parsing table. Each row represents a (state, token) pair, where the token is a lookahead token that causes reduce/reduce conflict in LR(k - 1) parsing table. By doing this we avoid repeating lookaheads for LR(1), LR(2), ... LR(K - 1) in the LR(k) parsing table, and can save space.

An example of the tentative way of storing the LR(k) parsing table in a file is given below.

Here is the LR(1) Parsing table.

state/token	$	a	b	c	d	E	T
0			s1			g2	g3
1	acc			s2			
2	r1	s3	s4	r1		g4	
3		s2	$\bigotimes$	r1	r2		
4	r2	r2	s4				
...							

In state 3, a reduce/reduce conflict occurs on token 'b' and needs more lookaheads to resolve.

Assume state 3 contains the following reduce/reduce conflict:

```
...
reduction r1: A → B • {b, c}
reduction r2: C → D • {b, d}
...
```

Now assume the LR(2) lane-tracing results in state 3 as follows:

```
...
reduction r1: A → B • {ba, bd, c}
reduction r2: C → D • {bc, bd, d}
...
```

Then the corresponding LR(2) parsing table is:

(state, token) \ LR(2) token	$	a	b	c	d	E	T
(3, b)		r1		r2	⊗		

Assume that after LR(3) lane-tracing on 'bd' (to do this efficiently, record those new lane head states/configurations that generate 'bd' and start further lane-tracing there), we get this state 3:

...
reduction r1: A → B • {ba, bda, c}
reduction r2: C → D • {bc, bdb, d}
...

Then the corresponding LR(3) parsing table is:

(state, token) \ LR(3) token	$	a	b	c	d	E	T
(3, d)		r1	r2				

Or it is possible that we end LR(3) lane-tracing of state 3 at state 0 and can no longer trace back, but still cannot resolve the reduce/reduce conflict:

...
reduction r1: A → B • {ba, bd$, c}
reduction r2: C → D • {bc, bd$, d}
...

Then the corresponding LR(3) parsing table entry is:

(state, token) \ LR(3) token	$	a	b	c	d	E	T
(3, d)	⊗						

Alternatively we can also specify a default reduction for this unresolved conflict.

In an LR(k) parsing table, we can also keep only those columns that are involved in reduce/reduce conflicts, and omit the rest to save space.

Actual LR(k) Parsing Table Storage in Hyacc

In section 3.3.2 we discussed the actual storage of LR(1) parsing table in the hyaccpar file of Hyacc.

For the LR(k) extension, Hyacc uses another parse engine file hyaccpark, which contains data structures and algorithms for manipulating LR(k) parsing tables.

These variables and arrays are actually used to represent the LR(k) parsing tables in addition to those for LR(1).

Variable/Array Name	Explanation
yy_lrk_k	The maximum value of k for this LR(k) grammar.
yy_lrk_rows[]	The number of rows in each LR(k) parsing table. Note that each row is for a (state, token) pair. For each parsing table, there may be multiple states, and each state may have multiple tokens.
yy_lrk_cols	The value is yyPTC.length + 2. Each row starts with two fields for each (state, token) pair, followed by one entry for each token. This is where yyPTC.length + 2 comes from.
yy_lrk_r[]	The actual values in each LR(k) parsing table. The first two entries are for the (state, token) pair. This is followed by (index, value) pairs, where index is the index of a token in the yyPTC[] array, and value is the action for this token: -2 means reduce/reduce conflict, a positive number means no conflict and is the corresponding production's ruleID. Finally, -1 labels the end of each row.
yyPTC[]	The values of parsing table column tokens.

Note that the LR(1) parsing table should have this change: where a reduce/reduce conflict occurs, the previous default reduction number should be replaced by a special value labeling the occurrence of a reduce/reduce conflict. This value is set to -10000010 in Hyacc. In the parse engine's arrays, this would mean the change of corresponding values in arrays yyfs[] and yytblact[].

6.4 LR(k) Parse Engine

The LR(k) parse engine is an extension to LR(1) parse engine, which was shown in section 3.3 on page 24.

In the LR(1) parse engine, the action depends on the value at parsing table entry (S, L) in the LR(1) parsing table, where S is a state and L is a lookahead symbol. For LR(k) the only change to the LR(1) parsing table is the addition of the $\bigotimes$ symbol, which leads to the following extension, based on the LR(k) parsing table data structure in the last section. The LR(k) parser engine extension is shown in Algorithm 6.9 below.

Algorithm 6.9: LR(k) parse engine extension

Input: Current state S; LR(1) lookahead L; action A: action(S, L)

```
1  if A == ⊗ then
2      k = 2;
3      while true do
4          L_next ← next lookahead;
5          if L_next == EOF then
6              Report error and exit; // premature end of input stream.
7          else
8              In LR(k) parsing table, find entry A_next ← ((S, L), L_next);
9              if A_next == ⊗ then
10                 L ← L_next;
11                 k ← k + 1;
12             else
13                 // A_next is reduction, or ⊘ plus default reduction;
14                 do reduce (the same as in LR(1));
15                 break out of while loop;
```

The repeat loop eventually will end when a reduction is found, either the reduction resolves the reduce/reduce conflict, or the reduction is the end of LR(k) lane-tracing in state 0 and a default reduction is used.

6.5 Performance

LR(1) parser generation was believed to be expensive in time and space. We have shown that practically this is not an issue now. LR(k) parser generation is more expensive than LR(1).

Here we approach LR(k) using the edge-pushing algorithm, which recursively calls the lane-tracing LR(1) algorithm on states with reduce/reduce conflicts. The performance bottleneck comes from two aspects: 1) lane-tracing, 2) calculation of $theads(\alpha, k)$.

The performance of LR(k) is not measured here since we only tested the edge-pushing algorithm on simple grammars. However, since after LR(1) there are usually not many states that contain reduce/reduce conflicts, we apply further lane-tracing only on a small portion of all the states. Also each time after increasing k, the states involved in LR(k+1) lane-tracing is only a portion of those involved in LR(k). With the use of a cache, each state should be traced no more than once. This analysis shows that lane-tracing should not cause much problem for the performance.

Besides lane-tracing, the process of finding $theads(\alpha, k)$ is another critical step. Potentially this is still exponential in k. But practically, most LR(k) grammars should end lane-tracing fast, or in the worst case end up at state 0 after some rounds. Unless k is really large and the string α is really long, $theads(\alpha, k)$ should not cost too much time or space.

In summary, in most cases the performance of LR(k) parser generation should be acceptable.

6.6 Examples

Example. Given LR(5) Grammar 6.2 [49]:

sentence → male_the male_subject is male_a male_adjective male_object |
female_the female_subject is female_a female_adjective female_object
male_the → the
female_the → the
male_subject → student | friend
female_subject → student | friend
male_a → a
female_a → a
male_adjective → tall | short
female_adjective → tall | short
male_object → boy | man
female_object → girl | woman

The LR(1) parsing state machine is shown in Figure 6.3. The state that contains a reduce/reduce conflict is state 4. The default reduction to solve the conflict is reduction 3 when the next token is "friend" or "student". The only two states involved in lane-tracing are state 4 and state 0, as shown in Figure 6.4.

In summary, input to this grammar is: "the student/friend is a tall/short boy/man/girl/woman".

at state 0, the parse can go either on the right path for male, or on the left path for female. It is obvious that "male_the" and "female_the", "male_subject" and "female_the", "is", "male_a" and "female_a", "male_adjective" and "female_adjective" are all identical. The only way to disambiguate the parse is the last item: "male_object" for "boy" or "man", or "female_object" for "girl" or "woman".

Therefore in order to solve the reduce/reduce conflict at state 4, one needs to look 5 tokens ahead after the current token "the" to determine whether to use reduction 3 or 4. If the 5th token is "boy" or "man" then reduce by rule 3 and go to state 2. If the 5th token is "girl" or "woman" then reduce by rule 4 and go to state 3. This is an LR(5) grammar.

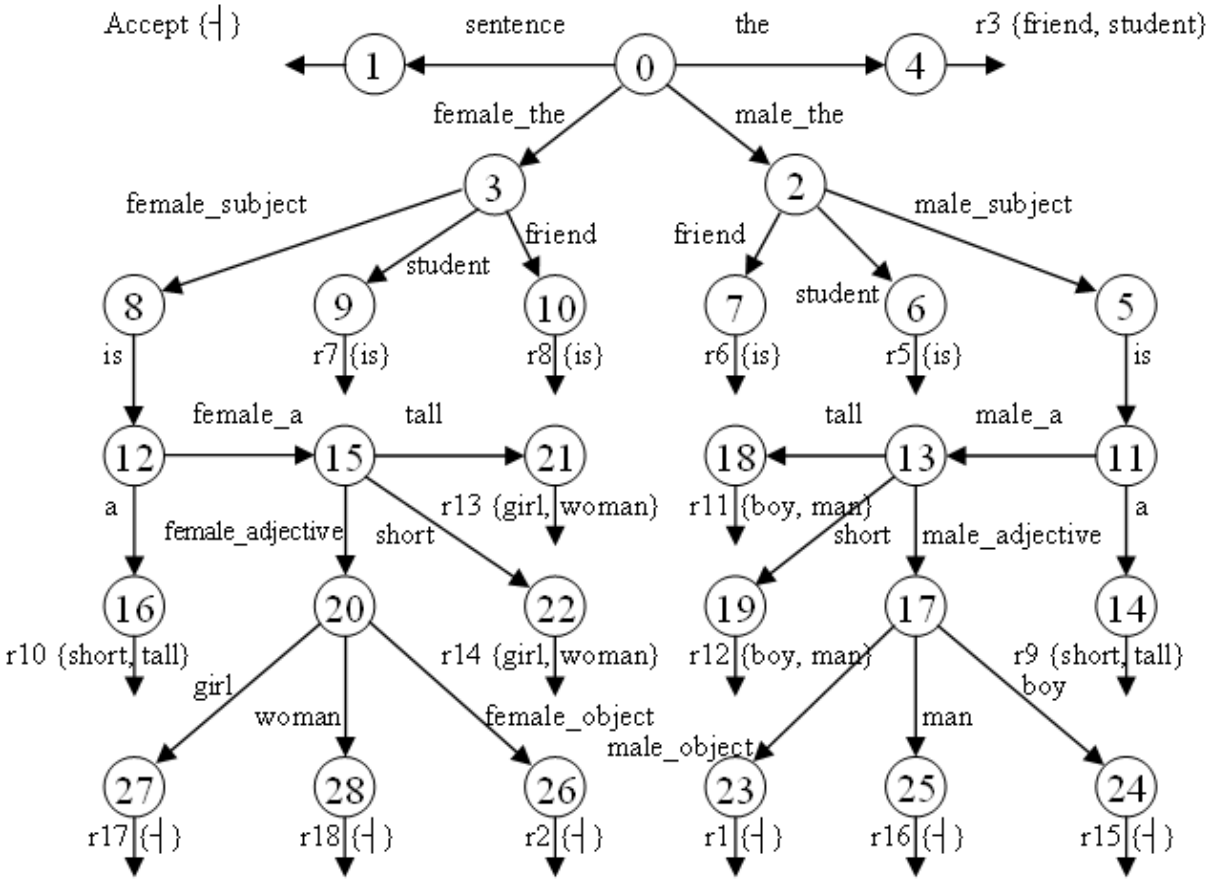


Figure 6.3: Parsing machine of grammar G6.2

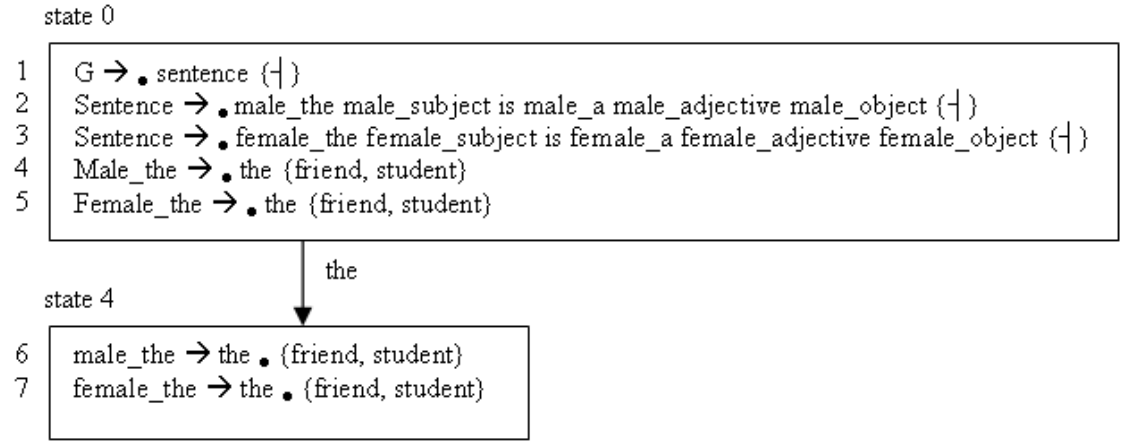
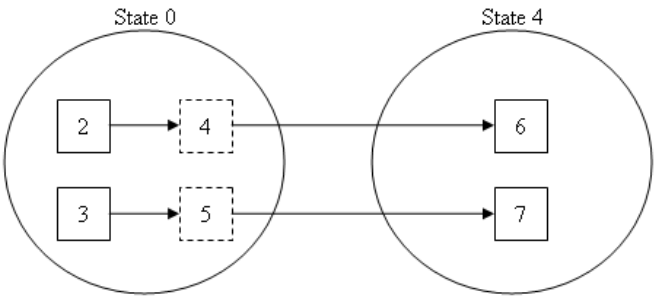


Figure 6.4: Parsing machine of grammar G6.2 - the part relevant to lane-tracing

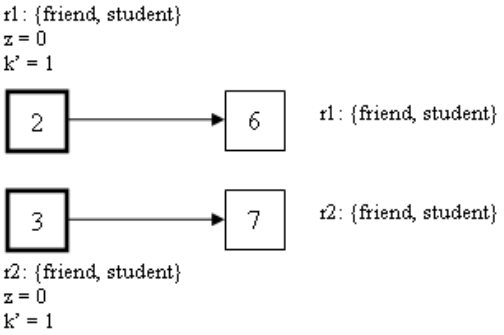
Now let's show the steps of applying the edge-pushing algorithm to this example.

We have labeled the configurations as in Figure 6.4.

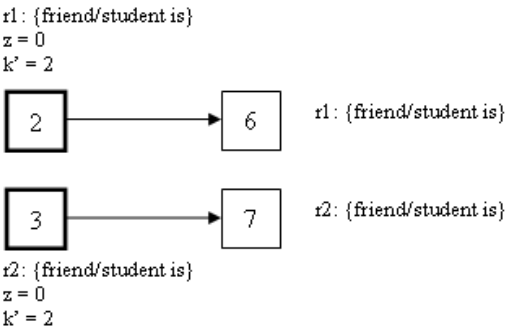
The following graph shows the configurations involved in LR(1) lane-tracing. Configurations 4 and 5 do not generate any contexts and are shown in dashed line circles. These two configurations will be ignored in the later graphs. Configurations 2 and 3 are the head configurations that generate contexts, and configurations 6 and 7 are the corresponding tail configurations where reduce/reduce conflicts occur.



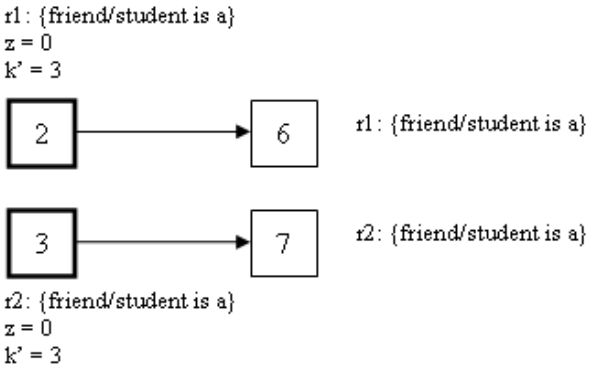
This is the step for LR(1) when applying edge-pushing:



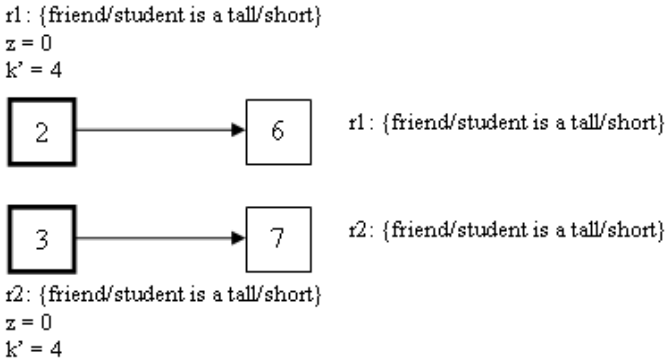
This is the step for LR(2) when applying edge-pushing:



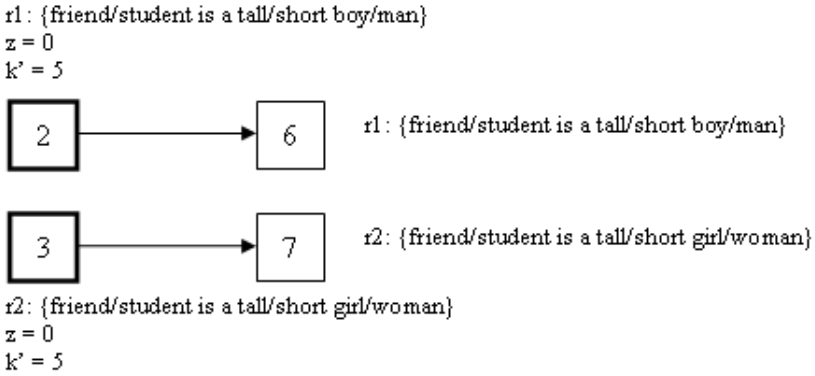
This is the step for LR(3) when applying edge-pushing:



This is the step for LR(4) when applying edge-pushing:



This is the step for LR(5) when applying edge-pushing:



At this time the reduce/reduce conflicts are resolved.

The corresponding LR(1) to LR(5) parsing tables are:

LR(1) parsing table:

state/token	...	friend	student	...
...				
4		⊗	⊗	
...				

LR(2) parsing table:

(state, LR(1) lookahead)/token	is
(4, friend)	⊗
(4, student)	⊗

LR(3) parsing table:

(state, LR(2) lookahead)/token	a
(4, is)	⊗

LR(4) parsing table:

(state, LR(3) lookahead)/token	tall	short
(4, a)	⊗	⊗

LR(5) parsing table:

(state, LR(4) lookahead)/token	man	boy	woman	girl
(4, tall)	r1	r1	r2	r2
(4, short)	r1	r1	r2	r2

In the parse engine, arrays representing the LR(1) parsing table are shown in table 6.1:

```
static YYCONST yytabelem yyfs[] = {
0, 0, 0, 0, -3, 0, -5, -6, 0, -7,
-8, 0, 0, 0, -9, 0, -10, 0, -11, -12,
0, -13, -14, -1, -15, -16, -2, -17, -18};

static YYCONST yytabelem yyptbltok[] = {
257, -1, -2, -3, 0, 258, 259, -4, 258, 259,
-5, -10000001, 267, -10000001, -10000001, 267, -10000001, -10000001, 260, -6,
260, -7, 261, 262, -8, -10000001, 261, 262, -9, -10000001,
263, 265, -10, -10000001, -10000001, 264, 266, -11, -10000001, -10000001,
-10000001, -10000001, -10000001, -10000001, -10000001, -10000001, -10000000};

static YYCONST yytabelem yyptblact[] = {
4, 1, 2, 3, 0, 6, 7, 5, 9, 10,
8, -3, 11, -5, -6, 12, -7, -8, 14, 13,
16, 15, 18, 19, 17, -9, 21, 22, 20, -10,
24, 25, 23, -11, -12, 27, 28, 26, -13, -14,
-1, -15, -16, -2, -17, -18, -10000000};

static YYCONST yytabelem yyrowoffset[] = {
0, 4, 5, 8, 11, 12, 13, 14, 15, 16, 17, 18,
20, 22, 25, 26, 29, 30, 33, 34, 35, 38,
39, 40, 41, 42, 43, 44, 45, 46};

static YYCONST yytabelem yyr1[] = {
0, -1, -1, -2, -3, -4, -4, -5, -5, -6,
-7, -8, -8, -9, -9, -10, -10, -11, -11};
static YYCONST yytabelem yyr2[] = {
0, 13, 13, 3, 3, 3, 3, 3, 3, 3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3};
```

Table 6.1: LR(1) storage tables in y.tab.c for grammar G6.2

The LR(1) parsing machine will always reduce to default production 3 at state 4 and go to the male branch. An error is reported if the last token is “girl” or “woman”.

In comparison, a LR(5) parsing machine can successfully distinguish these two situations and reduce to the correct production according to the 5th token.

In the corresponding LR(k) parse engine, the variables and arrays used to represent LR(k) parsing tables are shown in table 6.2 below:

```
static YYCONST yytabelem yyfs[] = {
0, 0, 0, 0, -10000010, 0, -5, -6, 0, -7,
-8, 0, 0, 0, -9, 0, -10, 0, -11, -12,
0, -13, -14, -1, -15, -16, -2, -17, -18};

static YYCONST yytabelem yyptbltok[] = { ... };

static YYCONST yytabelem yyptblact[] = {
4, 1, 2, 3, 0, 6, 7, 5, 9, 10,
8, -10000010, 11, -5, -6, 12, -7, -8, 14, 13,
16, 15, 18, 19, 17, -9, 21, 22, 20, -10,
24, 25, 23, -11, -12, 27, 28, 26, -13, -14,
-1, -15, -16, -2, -17, -18, -10000000};

static YYCONST yytabelem yyrowoffset[] = { ... };
static YYCONST yytabelem yyr1[] = { ... };
static YYCONST yytabelem yyr2[] = { ... };

static YYCONST yytabelem yy_lrk_k = 5;
static YYCONST yytabelem yy_lrk_rows[] = {2, 1, 1, 2};
static YYCONST yytabelem yy_lrk_cols = 26;
static YYCONST yytabelem yy_lrk_r[] = {
4, 259, 1, -2, -1,
4, 258, 1, -2, -1,
4, 267, 5, -2, -1,
4, 260, 6, -2, 7, -2, -1,
4, 262, 8, 3, 9, 3, 10, 4, 11, 4, -1,
4, 261, 8, 3, 9, 3, 10, 4, 11, 4, -1
};

static YYCONST yytabelem yyPTC[] = {
0, 267, 257, 258, 259, 260, 261, 262, 263, 265,
264, 266, CONST_ACC, -1, -2, -3, -4, -5, -6, -7,
-8, -9, -10, -11
};
```

Table 6.2: LR(k) storage tables in y.tab.c for grammar G6.2

Note that tables that are the same as those in LR(1) are shown by "...". The "-10000010" in bold font in yyfs[] and yyptblact[] arrays labels reduce/reduce conflicts to be solved by consulting the LR(k) parsing tables. It is the $\bigotimes$ symbol we used in the previous sections.

Example. LR(k) grammar G6.3. Here k depends on the value of n: k = n + 1. c^n is the concatenation of n letters 'c'.

S → a A D a | b A D b | a B D b | b B D a

A → a

B → a

D → c^n

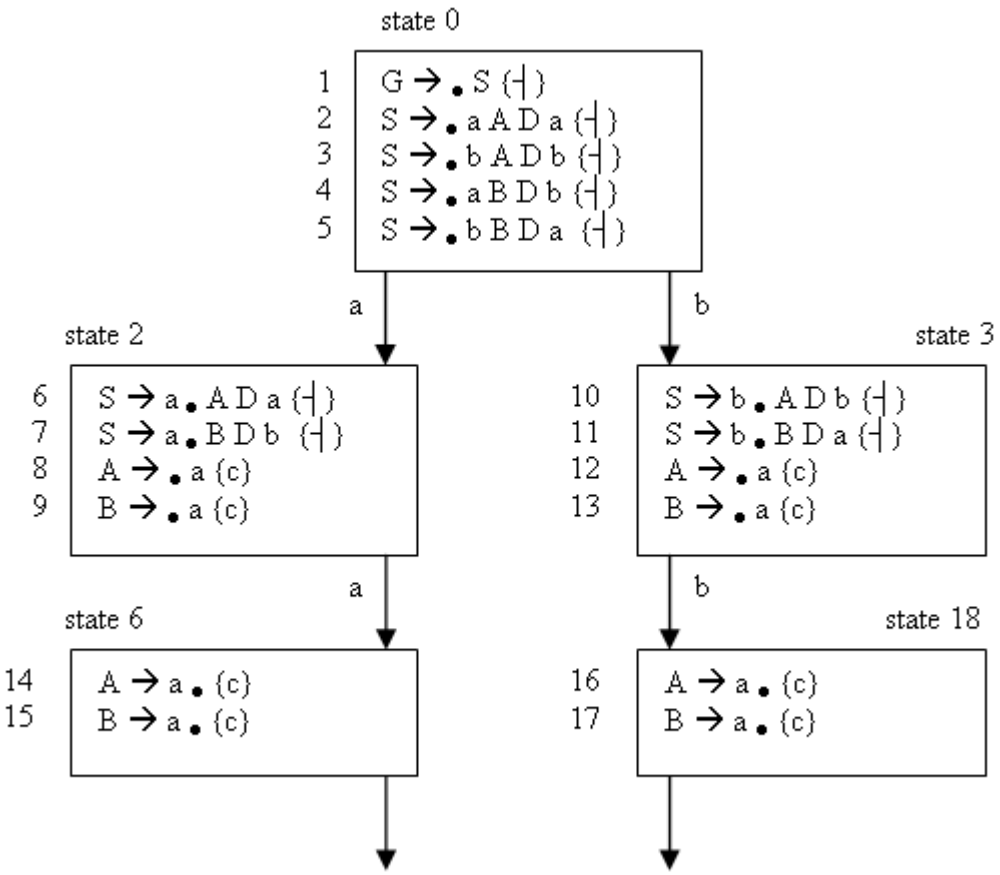
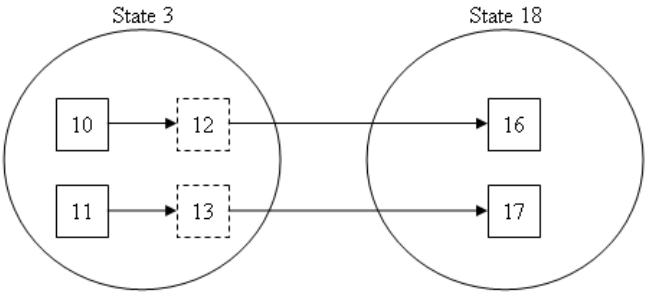


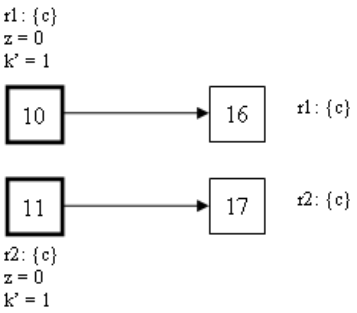
Figure 6.5: Parsing machine of grammar G6.3 - the part relevant to lane-tracing

Figure 6.5 shows the LR(1) parsing machine obtained using lane-tracing. States 6 and 18 both have reduce/reduce conflicts on token 'c'. With LR(k), the parse engine traces back to states 2 and 3 correspondingly, and get through the tokens in non-terminal D which is the string of n letters 'c', then use the next token 'a' or 'b' to make the choice of reduction to use. For example, in state 6, context c^nb means reduce with r5: A → a, context c^na means reduce with r6: B → a.

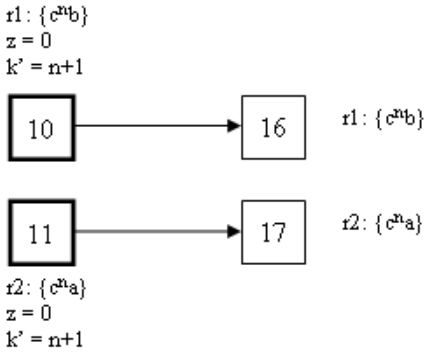
As in the last example, we will show the steps of applying edge-pushing. The following graph shows the configurations involved in lane-tracing on state 18. Again configurations 12 and 13 do not generate contexts. Configurations 10 and 11 are head configurations that generate contexts. Configurations 16 and 17 are corresponding tail configurations that have reduce/reduce conflict.



This is the step for LR(1) when applying edge-pushing:



This is the step for LR(n+1) when applying edge-pushing, which resolves the conflict:



Note that state 6 also has a reduce/reduce conflict. We can apply the same procedure above for configurations 14 and 15 in state 6.

The complete (including both states 6 and state 18) parsing tables are:

LR(1) parsing table:

state/token	...	c	...
...			
6		⊗	
...			
18		⊗	
...			

LR(2) parsing table:

(state, LR(1) lookahead)/token	c
(6, c)	⊗
(18, c)	⊗

Then it repeats with LR(3), LR(4), ..., until LR(n+1) parsing table:

(state, LR(n) lookahead)/token	a	b
(6, c)	r1	r2
(18, c)	r2	r1

Example. LR(3) grammar G6.4:

S → a A D a | b A D b | a A D b | b C E

A → a

B → a

D → e d

C → B e

E → d a

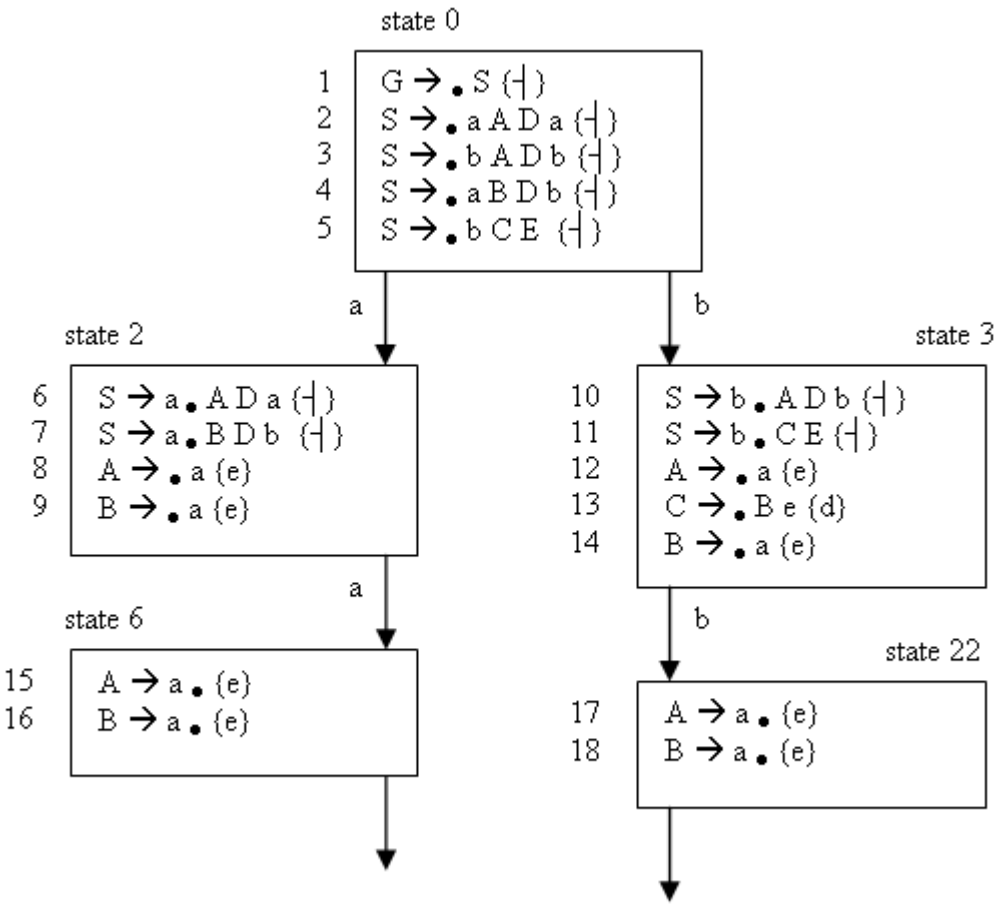
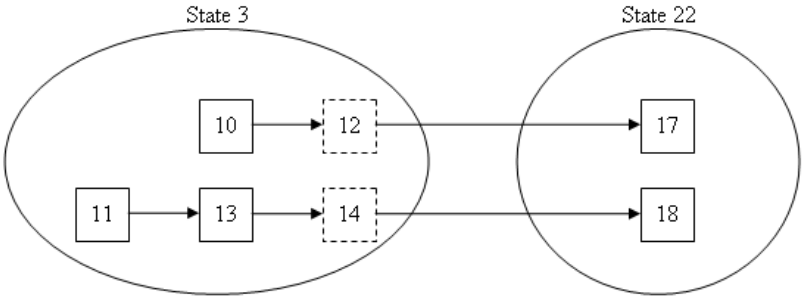
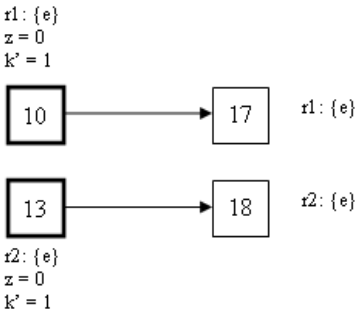


Figure 6.6: Parsing machine of grammar G6.4 - the part relevant to lane-tracing

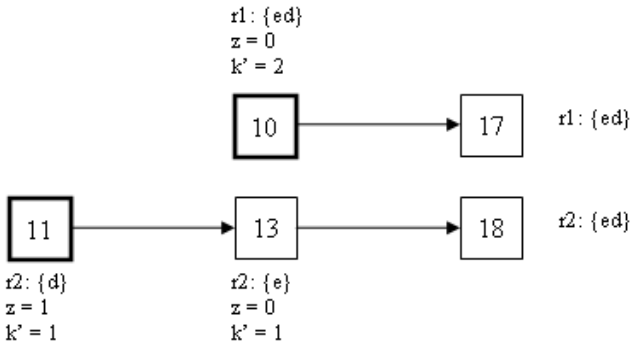
Again we will show the steps of applying edge-pushing. The following graph shows the configurations involved in lane-tracing on state 22. Configurations 12 and 14 do not generate contexts. Configurations 10 and 13 are head configurations that generate contexts. Configurations 17 and 18 are corresponding tail configurations that have reduce/reduce conflict.



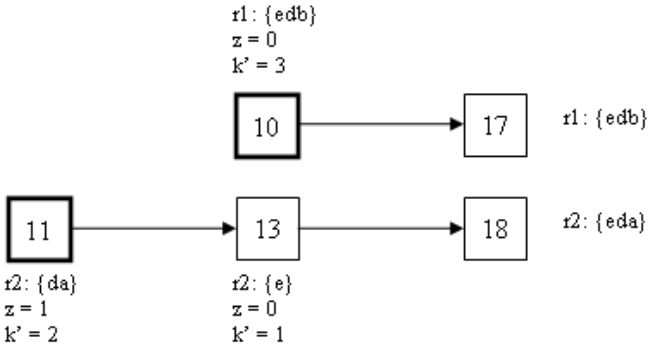
This is the step for LR(1) when applying edge-pushing:



This is the step for LR(2) when applying edge-pushing. Note now configuration 13 can not generate LR(2) context, so we have to trace back one more round to the next head configuration 11:



This is the step for LR(3) when applying edge-pushing:



Now the reduce/reduce conflict is resolved.

State 6 also has a reduce/reduce conflict. Following similar procedure we can obtain contexts to resolve its conflict. The complete LR(k) parsing tables (for both state 6 and state 22) are:

LR(1) parsing table:

state/token	...	e	...
...			
6		⊗	
...			
22		⊗	
...			

LR(2) parsing table:

(state, LR(1) lookahead)/token	d
(6, e)	⊗
(22, e)	⊗

LR(3) parsing table:

(state, LR(2) lookahead)/token	a	b
(6, d)	r1	r2
(22, d)	r2	r1

Note the difference of this example is that we have to do one more round of lane-tracing to resolve the conflicts. The previous two examples only needed one lane-tracing.

Example. Other LR(k) grammars that can be solved by Hyacc.

LR(2) grammar G6.5 [4]:

S → a A a | b A b | a B b | b B a

A → C a

B → D a

C → a

D → a

LR(3) grammar G6.6 [4]:

S → a A a a | b A a b | a B a b | b B a a

A → C a

B → D a

C → a

D → a

6.7 Lane-tracing at Compile Time

This section discusses the possibility of doing lane-tracing at compile time [51].

The logic for this is that, since the number of states involving LR(k) computation is small, we can move lane-tracing to compile time to save the complication of lane-tracing at parser generation time. The parsing table remains the same as LR(1), except that those entries with reduce/reduce conflicts are specially labeled. The intensive part of the work is transferred to the parse engine used at compile time.

Example. Given LR(2) grammar G6.7:

S → a A D a | b A D b | a B D b | b B D a
A → a
B → a
D → c

The part of the LR(1) parsing machine involving LR(2) is:

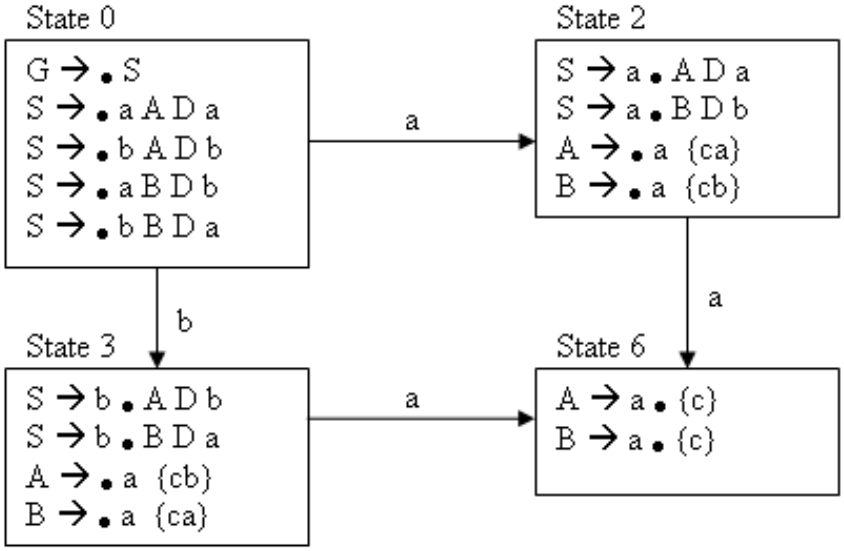


Figure 6.7: LR(2) part of the LR(1) parsing machine for grammar G6.7

It is obvious that state 6 has a reduce/reduce conflict. Lane-tracing back to state 2 and state 3 can resolve the conflict. In LR(2) parsing machine, state 6 should be split into two states. But in LR(1) parsing machine, state 6 is one single state with a reduce/reduce conflict.

Now suppose we are at state 2, the lookahead is ‘a’, the action is shift to state 6.

Next we are at state 6, the lookahead is ‘c’, so we need to reduce. But by ‘A → a’ or ‘B → a’?

To resolve the conflict we do lane-tracing.

In parser generation time lane-tracing we need to trace back to both states 2 and state 3. But here since we have state 2 on the state stack, we only need to trace back on state 2. This results in context {‘ca’} for ‘A → a’ and {‘cb’} for ‘B → a’.

So besides the current lookahead ‘c’, we check the next lookahead to make decision on which reduction to use. If it is ‘a’ then we reduce by ‘A → a’, otherwise if it is ‘b’ then we reduce by ‘B → a’.

The conclusions for compile-time lane-tracing are:

1) We only need to trace back on states that are currently on the state stack, and thus save tracing time and it is less complicated as well.

2) We need to keep records of all the configurations in the states. We can limit these states to the ancestor states of the inadequate states, which could be a relatively small portion of all the states. To be more space-efficient, when recording the configurations in each state, we can just record the rule number of the configuration (so we’ll search for the rule in the grammar’s rule list), and also the position of the scanned symbol.

3) We need to reconstruct the involved part of the LR(1) parsing machine for this to work. In comparison to the current LR(1) parse engine, which only needs to make use of the 2-D LR(1) parsing table, a lot of extra information will need to be stored and used. LR(1) parsing is linear in performance to the size of input string. But when compile-time lane-tracing is added, it becomes uncertain.

Implementation Details

We need:

1) the normal LR(1) parsing table. This is already done.

2) Inadequate states and their ancestor states. The question is how to get the ancestor states? A simple solution needs to modify the current data structure for state, so each state not just keeps a list of its children, but also a list of its parents. Maybe this can be done in a separate data structure, so we can construct such a network only when this algorithm is used.

3) In each state we also need to keep a record of the configurations. We can let each configuration keep a list of its parents, just like in 2). This again can be done is a separate data structure to avoid extra work when not using this algorithm.

4) Then when doing the compile-time lane-tracing, we need to reconstruct the parsing machine automata in order to use the existing implementation.

Actually, one design can be like this. Compile time lane-tracing has only a minor difference from the LR(k) parse engine. In the LR(k) parse engine algorithm (on page 148), on line 8, instead of looking into the LR(k) parsing table (which does not exist in this case), we trace back on each head configuration C_1, ..., C_n, in the same manner as edge-pushing. Then check the next generated context symbols (on the edge of tracing) of each front edge, if any two new generated context symbols are the same, then the reduce/reduce conflict is not resolved, and we need to trace further, and only on those new hand configurations involved in conflict. Now for the LR(1) parsing table, those locations where there is a reduce/reduce conflict are labeled by $\bigotimes$. This is all about the parsing table we need, all the rest rely on the edge-pushing process in the parse engine.

6.8 More Issues

In this chapter we have described our design and implementation of the edge-pushing LR(k) algorithm by extending LR(1) lane-tracing. Here are some more thoughts and issues.

1) The role of state splitting in LR(k). In the LR(1) lane-tracing algorithm, we does state-splitting from LALR(1) to LR(1). But from LR(1) to LR(k), there is no more state splitting. We just recursively trace back the configuration lanes to get more context information to solve LR(k) reduce/reduce conflicts. This actually is more the statement of a fact than an issue.

2) Avoid infinite cycling upon increasing k. This remains an issue and should be solved with a cache-like mechanism.

3) LR(k) grammars that don't have reduce/reduce conflicts in the LALR(1) parsing machine. One problem should be noted that it seems some LR(k) grammars have only shift/reduce conflicts in their LR(1) parsing machine, and no reduce/reduce conflicts. In such situations the pre-condition of applying lane-tracing, existence of reduce/reduce conflicts, does not exist. A guess is that for LR(k) we should apply lane-tracing on states involved in shift/reduce conflicts in such situations. We know that shift/reduce conflicts existing in the LALR(1) parsing machine also exist in the LR(1) parsing machine. But for LR(k), this may not hold, and we need to try to resolve shift/reduce conflicts the same way as we do for reduce/reduce conflicts. This issue needs further exploration. Some such LR(2) grammars are shown in [2] by Pete Jinks.

   Example. Given the Yacc grammar G6.8 [2]:

   yacc → rule | yacc rule
   rule → rulebody | rulebody SEMICOLON
   rulebody → RULENAME COLON alt | rulebody BAR alt
   alt → ϵ | alt TOKEN | alt RULENAME

   This should be an LR(2) grammar, since in order to determine whether the next symbol is on the right hand side of a rule, or the left hand symbol of a new rule, we need to look at the second lookahead: if the second lookahead is COLON, we know it is the start of another rule, otherwise it is just another symbol on the right hand side of the current rule. This means we need to look two tokens ahead to parse the Yacc input file, this is thus an LR(2) grammar.

   However, from the parsing machine in Figure 6.8 (the part relevant to shift/reduce conflicts are shown in figure 6.9), we see only two shift/reduce conflicts in states 9 and state 10. There is no reduce/reduce conflict. Furthermore, it seems that even if we trace back on the configurations involved in the shift/reduce conflicts, both will end at state 0, and we obtain no more context that can be used to resolve the shift/reduce conflicts.

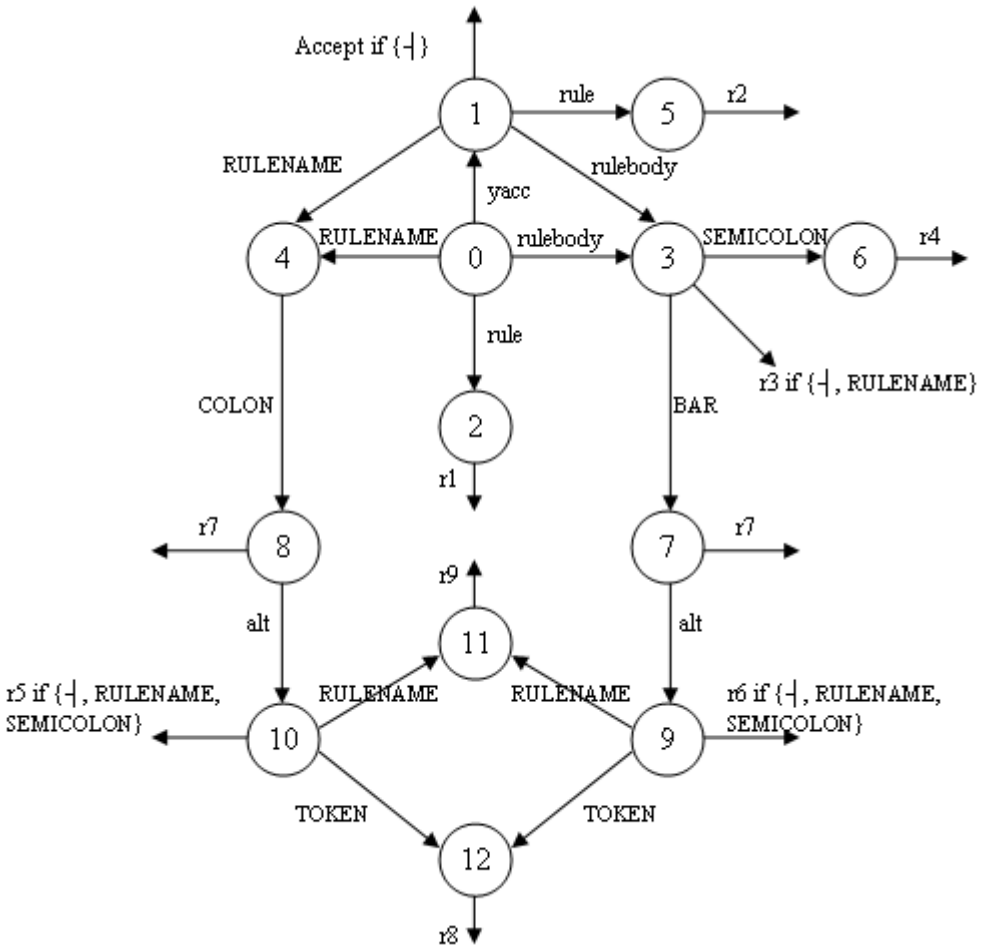


Figure 6.8: Parsing machine of the LR(2) grammar for Yacc

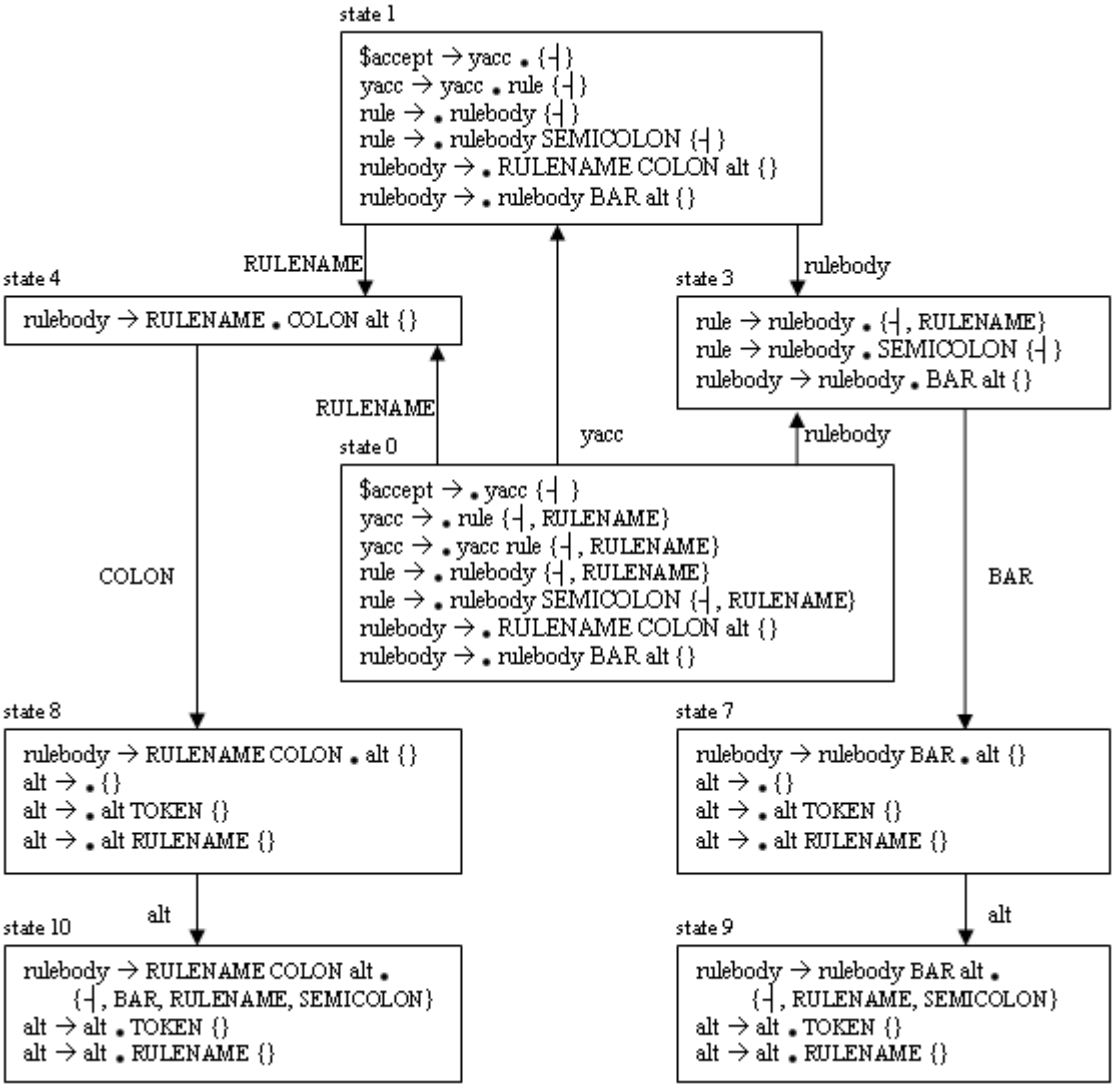


Figure 6.9: The part of the Yacc grammar parsing machine related to shift/reduce conflicts

4) LR(closed) grammars. Chris Clark also mentioned one type of grammar that he calls the LR(closed) grammar [4]. Such grammars can be deterministically parsed, but with an indefinite amount of rules pushed to the stack. He said computing a closure for some such grammars will never terminate due to the halting problem. More investigation is also needed in this. One such example of such grammars given by him is grammar G6.9 [4]:

S → a A a | b A b | a B b | b B a
A → A a
B → B a
A → a
B → a

The part of this grammar's parsing machine related to the reduce/reduce conflicts is shown below in Figure 6.10. It can be seen that lane-tracing produces no context information to help resolving the reduce/reduce conflicts.

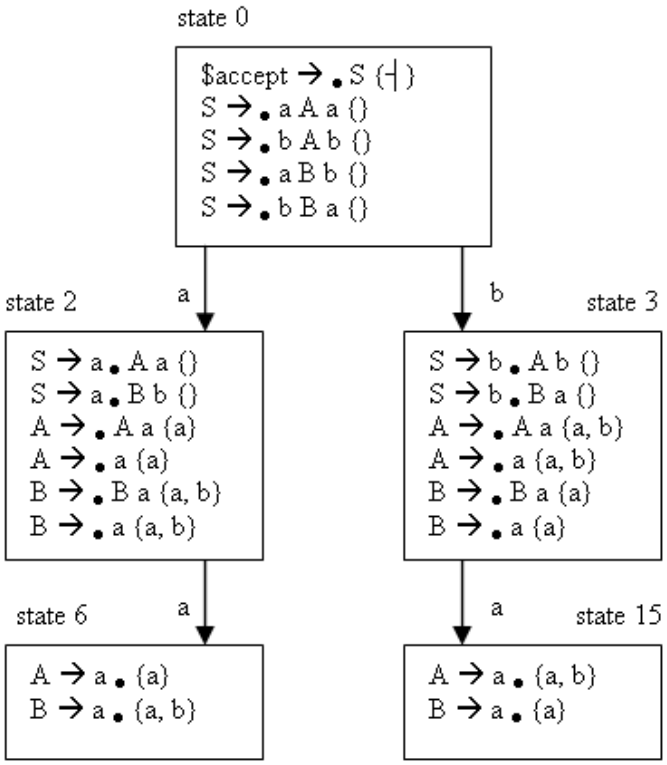


Figure 6.10: The part of Chris Clark's grammar's parsing machine related to reduce/reduce conflicts

Chapter 7

The Latex2gDPS compiler

7.1 Introduction

The latex2gDPS compiler translates Latex source to the gDPS language. There are two motivations for creating the latex2gDPS compiler. The first is to demonstrate the use of the parser generator Hyacc. The second is that the creation of such a tool is of interests by itself.

In the work of Holger [40], a general-purpose dynamic programming problem solver called the DP2PN2Solver was designed and implemented. In this system, a DPFE (Dynamic Programming Functional Expression) is specified in the gDPS (general Dynamic Programming Specification) language. Then a D2P compiler translates a DPFE expressed in gDPS to a Bellman net. The Bellman net, which is represented as a N x N adjacency matrix, is further translated into solver code in three formats: spreadsheet, Java and PN code. They are then solved by spreadsheet program, Java program and Petri Net simulator correspondingly. This structure is shown in Figure 7.1.

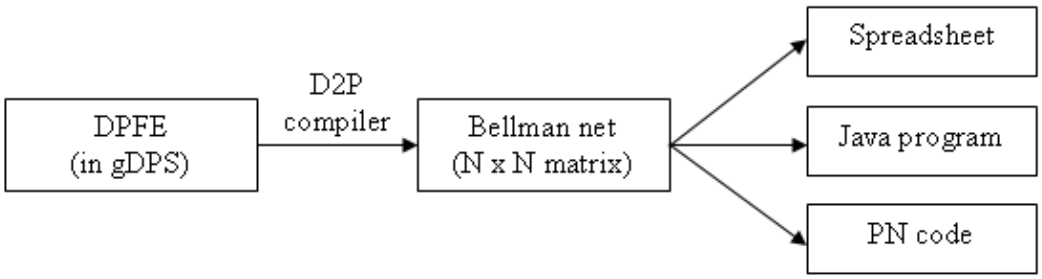


Figure 7.1: Architecture of the DP2PN2Solver

To use this general DP problem solver, one needs to learn the gDPS language. There is a learning curve associated with it. To remove this learning curve, a solution is to allow the user to specify the DPFE in a familiar language. A translator then can translate from this language into gDPS, and make the gDPS language layer transparent to the user. This project is based on this idea. Figure 7.2 shows how the latex2gDPS compiler fits in this system.

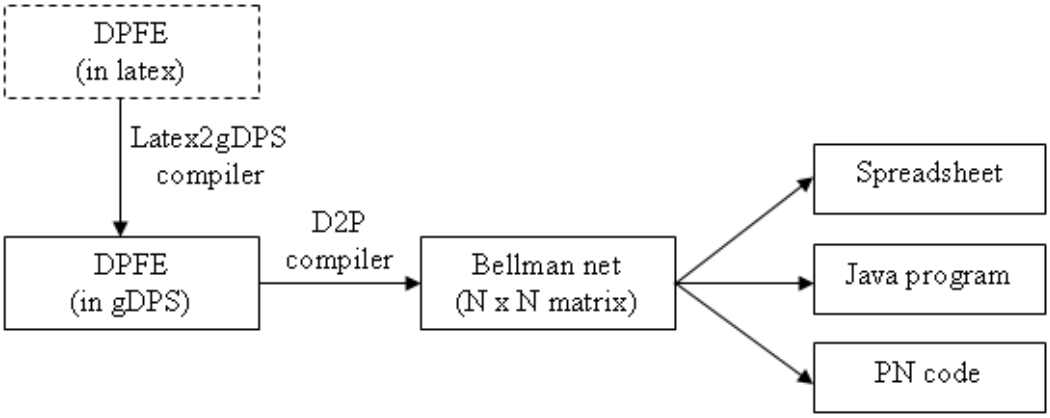


Figure 7.2: Adding the latex2gDPS compiler to the architecture of the DP2PN2Solver

7.2 Design of the Latex2gDPS Compiler

7.2.1 Overall Design

The Latex2gDPS compiler is built on the traditional Lex/Yacc system, and in this case, also Lex/Hyacc. The compiler is written in C. The designed grammar is shown in Appendix C.

A two-pass parse process is used for the compiler. In the first pass the DPFE in latex input file is scanned into a container data structure of DPFE, and each of the functions, variables, operators and literals (numbers, constants) gets one entry in the symbol table. Types (number, variable or set) of the variables and literals are determined from context or use default value. In the second pass, which is the code generation pass, the obtained information of the DPFE is written into the gDPS output file.

Right now the processing includes: 1) Translation of the DPFE formula. 2) Translation of the declarations. 3) Fill in the other sections needed by the gDPS output.

One thing special about dynamic programming problems is that each type is different from the rest. Individual solutions are needed for each type. Thus a comprehensive top-down approach is hard in practice since there are too many exceptions to handle. Consequently, a bottom-up solution is taken, such that different types of dynamic programming problems are handled one by one. For this purpose, a pluggable architecture is used, such that a set of API functions are provided to incorporate each new DPFE type. Some macros are ready to use to ease the work of writing API functions. For unhandled cases the user needs to write customized code. When incorporating a new DPFE type, just provide a C source file and a header file specific for this DPFE type, and compile again.

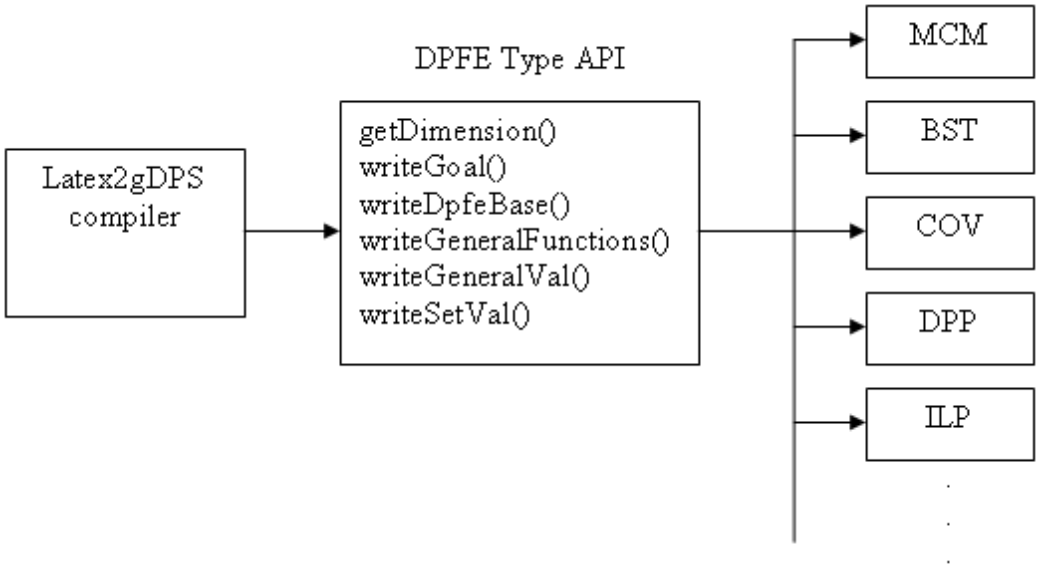


Figure 7.3: DPFE Type API of the latex2DPS compiler

7.2.2 Data Structures

An appropriate data structure is needed to represent the DPFE formula. We need to organize the scanned DPFE formula is such a way that it is easy to translate it to the output gDPS file. The relevant data structure containers are: 1) the DPFE itself, 2) DPFE optimization function, 3) DPFE base condition, 4) the goal function of the DPFE, which can be included in DPFE optimization function or be independent, the current implementation keeps it independent. The data structure of the DPFE itself is a container of the other three.

The goal function needs to keep track of 5) the function name, and 6) the parameter list.

The DPFE optimization function needs to keep track of 7) the optimization (MIN, MAX or others) used, 8) the decision variable, 9) the decision operator, 10) the decision space, 11) the terms of the function, including both the operands and the operators, 12) the constraint condition.

The DPFE base condition needs to keep track of 13) the base value, 14) the constraint condition for this base value. It is possible that more than one base condition is specified, so the DPFE base condition data structure needs to be implemented as a collection (a list or an array, or something similar).

Data structures also are used to represent components of the gDPS file. When the DPFE formula in latex format is scanned in, the components of the DPFE formula are translated into the components corresponding to each section of the gDPS file. Then in the code generation step, the information can be easily written to the gDPS file.

7.2.3 Use of Special Declarations

Some context information are not available in the DPFE itself. To overcome this problem special declarations are used to specify such information. Seven declaration types are used:

1) ModuleType: a one word ID

2) ModuleName: a one word ID

3) ModuleGoal: function name([parameters])

4) ModuleBase: list of “function name([parameters]) = value [when (condition list)]”.

5) Dimension: dimension name : list of numbers (separated by “,” in a single array; arrays are separated by “;”).

6) DataType: list of “type ID” separated by “,”

7) Alias: list of “ID = ID alias” separated by “,”

The order of these declarations does not matter. But only one of each declaration type can be used. In the declaration, the keyword is enclosed by $ and following by a colon, then the declaration content.

7.3 Current Status

The essential functions and framework of the latex2gDPS compiler have already been established.

Problems may exist in some marginal situations, like in the declaration of some bases, general functions etc. The handling of these should be aided with the gDPS compiler. By feeding the gDPS output files from the latex2gDPS compiler to the gDPS compiler, we can see whether they work, and if not, how to correct the problems.

20 DPFE types are collected. The acronyms and full names are listed in Table 7.1. Table 7.2 shows whether they are significant problems (and thus have higher priority to be handled), the source (which books they are from), and the status of whether they have been handled and included in the latex2gDPS compiler.

Type	Full name
ALLOT	
ASMBAL	
BST	Optimal Binary Search Tree Problem
COV (or SCA)	Optimal Covering Problem
DPP (or FPP)	Discounted Profits Problem
EDP	Edit Distance Problem
ILP	Integer Linear Programming Problem
KS01	0/1 Knapsack Problem
LCS	Longest Common Subsequence Problem
LSP	Longest Simple Path Problem
MCM	Matrix Chain Multiplication Problem
ODP (or OFP)	Optimal Distribution Problem
PERM (or OST)	Optimal Permutation Problem
RAP	Production: Reject Allowances Problem
RDP	Reliability Design Problem
SCP	Stagecoach Problem
SPA	Shortest Path in an Acyclic Graph Problem
SPC	Shortest Path in an Cyclic Graph Problem
TSP	Traveling Salesman Problem
WLV	Investment: Winning in Las Vegas Problem

Table 7.1: DPFE types and their full names

Type	Significant	Source [1]	Handled
ALLOT	Y	3	
ASMBAL	Y	1	
BST	Y	1	Y
COV (or SCA)		4	Y
DPP (or FPP)		4	
EDP		4	
ILP	Y	3	Y
KS01	Y	2	Y
LCS	Y	1	
LSP	Y	1	Y
MCM	Y	1	Y
ODP (or OFP)	Y	3	Y
PERM (or OST)		4	
RAP	Y	3	Y
RDP		4	
SCP	Y	3	Y
SPA	Y	2	Y
SPC	Y	2	Y
TSP	Y	2	Y
WLV	Y	3	Y

Table 7.2: DPFE types, significance, sources and status

[1] Source: 1 - Cormen, 2 - unknown, 3 - Hillier & Lieberman, 4 - Other

7.4 An Example

An example for the MCM (Matrix Chain Multiplication) DPFE type is given below.

The MCM DPEF and an example of goal and input data are:

$$f(i,j) = \begin{cases} \min\limits_{k\in\{i,\ldots,j-1\}} \{f(i,k) + f(k+1,j) + d_{i-1}d_k d_j\} & \text{if } i < j \\ 0 & \text{if } i = j. \end{cases}$$

goal=f(1,n)
A_i has dimension $d_{i-1} * d_i$
D={3,4,5,2,2}
n=4

The corresponding latex specification is:

\begindeclaration
$ModuleType$: MCM
$ModuleName$: MultipleChainMultiplication
$ModuleGoal$: f(0, 0)
$ModuleBase$: f(i, j) = 0 if (i = j)
$Dimension$: d = 3, 4, 5, 2, 2
$Alias$: i = firstIndex, j = secondIndex, d = dimension
\enddeclaration

\beginequation
f(i,j)=
\left\
\beginarray ll
\displaystyle \min k \in n i, \ldots, j-1\
\ f(i,k)+f(k+1,j)+d i-1d kd j n & \mboxif i ¡ j \\
0 & \mboxif i = j.\\
\endarray
\right.
\endequation

The output is:

```
BEGIN
    NAME MultipleChainMultiplication;

GENERAL_VARIABLES_BEGIN
private static int[] d = {3, 4, 5, 2, 2};
GENERAL_VARIABLES_END

STATE_TYPE: (int i, int j);

DECISION_VARIABLE: int k;
DECISION_SPACE: decisionSet(i, j) = {i, ..., j - 1};

GOAL:
f(1, 4)

DPFE_BASE:
f(i, j) = 0 WHEN i = j;

DPFE:
f(i, j)
= MIN_{k IN decisionSet}
{
f(t1(i, j, k)) + f(t2(i, j, k)) + r(i, j, k)
};
REWARD_FUNCTION:
r(i, j, k) =
d[i - 1] * d[k] * d[j];
TRANSFORMATION_FUNCTION:
t1(i, j, k) = (i, k);
t2(i, j, k) = (k + 1, j);
END
```

Chapter 8

Conclusion

LR(1) is the most powerful parsing algorithm for context-free languages. However LR(1) parser generation was long regarded as computationally infeasible. The compiler community has seen various parser generators using LALR(1) and LL(k) algorithms, but LR(1) parser generators are still rare. There are LR(1) algorithms that can reduce the number of states in a parsing machine, making LR(1) parser generation not much more expensive than LALR(1). It is meaningful to revisit this field, do an investigation of LR(1) parser generation algorithms and come up with a practical tool. LR(k) is more expensive than LR(1) and little practical work has been done on it due to the difficulty. Despite the fact that LR(1) should suffice for the needs of processing most programming languages, LR(k) should have values in areas such as natural language processing.

In this work we investigated LR(1) parser generation algorithms and implemented a LR(0)/ LALR(1)/LR(1)/LR(k) parser generator Hyacc, which has been released to the open source community in the hope of bringing the power of LR(1) parsing to life. The interface features of Hyacc are highly similar to widely used LALR(1) parser generators Yacc and Bison, which makes it easy to learn and to be accepted by the wide user base of Yacc and Bison. Hyacc is written in ANSI C and can be easily ported to most platforms.

Hyacc contains these LR(1) algorithms: 1) the Knuth canonical algorithm, 2) Pager's practical general method, which combines compatible states during new state generation process of the Knuth canonical algorithm, 3) Pager's lane-tracing algorithm which starts from LR(0) parsing machine and splits states that cause reduce/reduce conflicts in the parsing machine, 4) Pager's unit production elimination algorithm, which is not a backbone LR(1) algorithm but related. A lot of details in the

implementation are discussed, especially for the second phase of lane-tracing which has not been discussed in detail before.

The LALR(1) algorithm in Hyacc is implemented by the first phase of the lane-tracing algorithm. It is an alternative to what most people know from the current literature.

We found that Pager's unit production elimination algorithm leads to redundant states in the resulting parsing machine, and extended the algorithm to remove redundant states, so as to obtain a minimal parsing machine and increase parser generation efficiency.

We measured and compared the performance of different LR(1) algorithms as implemented in Hyacc with each other, and with LALR(1) algorithms as implemented in Hyacc and Bison. The study was done on the grammars of 13 programming languages. We have shown that with reduced-space LR(1) algorithms such as the practical general method and the lane-tracing algorithm, the time and space requirements are not much bigger than the LALR(1) algorithm for these real programming languages grammars. It is safe to conclude that we can disregard the myth about the impracticality of LR(1) performance, and take LR(1) as an efficient alternative of its LALR(1) peers.

On the framework of LR(1) parser generation algorithms, three pathways are found in literature research: the combining path as represented by Pager's practical general method, the splitting path as represented by Pager's lane-tracing algorithm and Spector's splitting algorithm, and the divide and conquer path as represented by Korenjak's partitioning algorithm. We have investigated the combining and the splitting paths in this work.

We have created a new LR(k) parser generation algorithm called the edge-pushing algorithm, which is based on the LR(1) lane-tracing algorithm, recursively traces back on relevant configurations to obtain more contexts to resolve reduce/reduce conflicts. The edge-pushing algorithm itself, corresponding LR(k) storage parsing table and parse engine have all been designed and implemented in Hyacc. The edge-pushing algorithm so far works on LR(k) grammars where lane-tracing upon increasing k does not form a cycle.

We also have developed a latex2gDPS compiler to demonstrate the use of Hyacc.

Chapter 9

Future Work

9.1 Study of More LR(1) Algorithms

Currently we have studied and implemented the practical general method on weak compatibility. An extension of the practical general method is based on strong compatibility, which can possibly generate a more compact parsing machine but is more computationally demanding. It would be interesting to investigate the performance and see whether it might be worthwhile to apply it in addition to weak compatibility in some cases.

The partitioning approach to LR(1) is not studied in this work due to limited time. There are issues worth study related to this approach, such as the partition strategy.

9.2 Issues in LR(k) Parser Generation

There are many issues to explore in LR(k) parser generation.

The current edge-pushing algorithm for LR(k) works for situations where no cycles are involved in lane-tracing upon increasing k. If there are cycles involved, we have to introduce a cache feature to avoid infinitely tracing down the cycles. Branches where two lanes merge into each other do not have the problem of infinitely tracing down cycles, but are similar in that we can use a cache to avoid redundant work.

It will also be interesting to apply LR(k) algorithm to the study of natural languages. We already see one example of applying LR(k) to resolve the reduce/reduce conflict in a simple LR(5) natural language grammar. We can try our algorithm on more complex natural languages grammars. For natural language processing, one common technique currently employed is GLR algorithm with LALR or SLR engine. GLR keeps separate stacks for different parse alternatives and is very expensive in both time and space. It is more expensive than LR(k). It also cannot tell ambiguity apart from deterministic LR(k) grammars due to the way it handles the grammar using multiple stacks. LR(k) can serve as a good alternative to GLR.

For the generated parser using lane-tracing, it is also interesting to see if we can move lane-tracing from parser generation time to compile time, which means imbedding the lane-tracing algorithm into the parse engine. This way we can avoid the cost of generation and storage of the LR(k) parsing table. It will be intriguing to see how well this works.

9.3 Ambiguity

The LR(1) algorithms discussed in this work are designed for LR(1) algorithms only, and cannot be applied to grammars that are not LR(1). For example, the unit production elimination algorithm generates shift/shift conflicts for ambiguous grammars. Since ambiguous grammars exist widely in practice, it is useful if we can extend the current LR(1) algorithms to better handle ambiguous grammars.

9.4 More Work on Hyacc

The Hyacc parser generator is an efficient and practical tool for LR(1) parser generation. It has been released to the open source community for some time. Some features are not yet completed, such as the directives %union, %type and %nonassoc. The LR(k) part also is not yet fully working. We need to finish these and enrich its features.

We hope Hyacc can gain wide acceptance in the industry. We also hope to collaborate with people in the compiler industry on LR(1) issues.

Appendix A

Hyacc User Manual

HYACC User Manual

Created on 3/12/07. Last modified on 3/27/09.

Version 0.97

Department of Information and Computer Science, University of Hawaii
Please send all bug report and comments to chenx@hawaii.edu

This documentation introduces Hyacc and its usage.

1. Overview
1.1 Background
1.2 Feature list
1.3 A little note on the license

2. Compile and Installation
2.1 For Unix/Linux/Cygwin users
2.2 For windows users

3. Usage
3.1 Input file
3.2 Command line switches
3.3 Output file

4. Future perspective

5. References

1. Overview

1.1 Background

Many people have used Yacc. It is a LALR(1) compiler generator, often used with the lexical analyzer Lex to create compilers. There are many variations of Yacc, like Bison and Berkeley yacc.

Hyacc is similar to Yacc in that it is a compiler generator. It is different from yacc in that it is a LR(0)/LALR(1)/LR(1)/LR(k) compiler generator. It can accept all LR(1) grammars. Hyacc also contains LR(0) and LALR(1) algorithms, and a partially working LR(k) algorithm that allows it to accept some LR(k) grammars. This is more powerful than Yacc.

Hyacc is pronounced as “HiYacc”, means Hawaii Yacc.

In the past people think LR(1) compiler generator is hard to implement, and the process of generating a LR(1) parser is very expensive in time and space. However, Hyacc shows that, with optimization algorithms and proper choice of data structures, a LR(1) parser generator can be close to LALR(1) parser generators in compactness and performance in many cases.

Specifically, based on the original LR(1) algorithm [Knuth], the practical general method [Pager77] is used to combine (weakly) compatible states to reduce the state space and increase the performance. Based on this, another optimization of unit production elimination [Pager77b] and its extension are also used in the hope of further reducing the size of the state space.

Besides, Hyacc also implemented LR(1) parser generation based on the lane-tracing algorithm [Pager77c][Pager73].

The LR(k) algorithm in Hyacc is called the edge-pushing algorithm, which recursively applies lane-tracing to obtain more contexts to resolve reduce/reduce conflict.

Hyacc tries to be backward compatible with Yacc and Bison in the format of input file and command line switches.

Hyacc was developed under Cygwin, and has been tested in Fedora Core 4.0, Solaris and Suse 10.3. It is ANSI C compliant, which makes it extremely easy to port to other platforms.

1.2 Feature list

Current features:

1) Implements the original Knuth LR(1) algorithm [Knuth].
2) Implements the practical general method (weak compatibility) [Pager77]. This is a LR(1) algorithm.
3) Removes unit productions [Pager77b].
4) Removes repeated states after removing unit productions.
5) Implements the lane-tracing algorithm [Pager77c][Pager73]. This is a LR(1) algorithm.
6) Supports LALR(1) based on the lane-tracing algorithm phase 1.
7) Supports LR(0).
8) Implements the edge-pushing LR(k) algorithm. So far this algorithm can accept those LR(k) grammars where lane-tracing on increasing k do not involve cycle.
9) Allows empty productions.
10) Allows mid-production actions.
11) Allows these directives: %token, %left, %right, %expect, %start, %prec.
12) In case of ambiguous grammars, uses precedence and associativity to resolve conflicts. When unavoidable conflicts happen, in case of shift/reduce conflicts the default action is to use shift, in case of reduce/reduce conflicts the default is to use the production that appears first in a grammar.
13) Is compatible to yacc and bison in the ways of input file format, ambiguous grammar handling, error handling and output file format.
14) Works together with Lex. Or the users can provide the yylex() function themselves.
15) If specified, can generate a graphviz input file for the parsing machine.
16) If specified, the generated compiler can record the parsing steps in a file.
17) Is ANSI C compliant.
18) Rich information in debug output.

What's not working so far and to be implemented:

1) Hyacc is not reentrant.
2) Hyacc does not support these Yacc directives: %nonassoc, %union, %type.
3) The optimization of removing unit productions can possibly lead to shift/shift conflicts in case of grammars that are ambiguous or not LR(1), and thus should not be applied in such situation.
4) Full LR(k) where the cycle problem can be solved.

1.3 A little note on the license

All the source files of Hyacc comes under the GPL license. The only exception is the file hyaccpar, which comes under the BSD license and is the skeleton parser driver of hyacc output. This should guarantee that Hyacc itself is protected by GPL, but the parser generators created by Hyacc can be used in both open source and proprietary software. This addresses the problem that Richard Stallman discussed in "Conditions for Using Bison" of his Bison 1.23 manual and Bison 1.24 manual.

2. Compilation and installation.

2.1 For Unix/Linux/Cygwin users

These files are included in the package for Unix/Linux/Cygwin users:

lane_tracing.h – The header file for lane-tracing functions.
mrt.h – The header file for multi-rooted tree.
stack_config.h – The header file for configuration stack.
y.h – The header file.
get_options.c – Gets command line switches.
gen_compiler.c – Generates compiler from the parsing machine created in y.c.
gen_graphviz.c – Generates graphviz input file for the parsing machine.
get_yacc_grammar.c – Parses input grammar file and feeds the result to y.c.
hyacc_path.c – Gives path information of hyaccpar and hyaccmanpage.
inst.c – Recreates hyacc_path.c at compilation time.
lr0.c – Contains functions for the LR(0) algorithm.
lrk.c – Contains functions for the LR(k) algorithm.
lrk_util.c – Contains utility functions for the LR(k) algorithm.
lane_tracing.c – Contains functions for the lane-tracing algorithm.
mrt.c – Contains functions for multi-rooted tree used in unit production elimination.
queue.c – A circular, expandable queue of integer.
stack_config.c – Contains functions for configuration stack.
state_hash_table.c – A hash table that makes searching states easy.
symbol_table.c – A hash table that stores information of grammar symbols.
upe.c – contains functions for unit production elimination.
version.c – Gives version information of Hyacc.
y.c – Creates LR(1) parsing machine, and applies the three optimizations.
hyaccpar – The LR(0)/LALR(1)/LR(1) compiler parse engine.
hyaccpark – The LR(k) compiler parse engine.
hyaccmanpage – The man page file.
hyacc.1 – Used to create the man page file.
makefile – The make file.

The makefile file is the utility used for compilation.

The options provided by the makefile are:

1) If compile the first time, type “make release” to compile the source code. This will take the INSTALL_PATH and feed it to inst.c, which recreate hyacc_path.c using the path information. Then it does the compilation to create the executable file hyacc.

2) If NOT compile the first time, type “make” to compile the source code, this is different from “make release” in that it does not create a new hyacc_path.c file.
3) Type “make debug” will do the same thing as “make release”, but also use the –g switch, so the user can use gdb debugger to debug the hyacc executable if anything goes wrong.
4) Type “make clean” will remove the hyacc executable file.
5) Type “make install” will do all the work of “make release”, and then copy executable hyacc, hyaccpar and hyaccmanpage to the destination directory.
6) Type “make uninstall” will tell the user what files to remove. The user needs to follow the instruction and manually remove the files hyacc, hyaccpar and hyaccmanpage from the installation folder.
7) Type “make dist” will create a distribution package in the format of hyacc_mm-dd-yy.tar.gz.

A typical process of compile and install Hyacc is:

If the user wants to install to the current directory:

1) Type “make release” will do all the work.

If the user wants to install to a different directory:

1) Modify the INSTALL_PATH macro at the top of the makefile. This tells where the user wants to install Hyacc. By default, this is the current directory. If you want the files to be installed to another location, make sure you have the permission to copy files there.
2) Type “make install”. That’s all.

2.2 For Windows users

For the files included in the package for windows users, the only difference from the package for Unix/Linux/Cygwin users is that now use the file hyacc_dos_path.c instead of hyacc_path.c, and there is no inst.c.

In hyacc_dos_path.c, if USE_CUR_DIR is defined as 1 (this is the default value), then the compiled binary uses current directory to locate resource files (hyaccpar and hyaccmanpage). If USE_CUR_DIR is defined as 0, then it uses C:\windows as the default installation path. The user should change this file if want to install to different location.

The user should have a C compiler in windows, like Microsoft Visual Studio 6.0. Using Microsoft Visual Studio 6.0 as an example, these steps are required:

1) Unzip the package.
2) Create an empty win32 console application from File → New.
3) Switch to File View, add all the *.c files to "Source Files", and add *.h to "Header Files".
4) From Build → Set Active Configuration, choose "Release".
5) Build hyacc.exe using the Build menu or use the short-cut key F7.
6) Now the hyacc.exe is successfully built.
7) In hyacc_path_dos.c, by default USE_CUR_DIR is defined as 1. So hyacc uses the current directory to locate resource files (hyaccpar and hyaccmanpage). Otherwise, if USE_CUR_DIR is defined as 0, then C:\windows is the default installation path, and the user should copy hyacc.exe, hyaccpar and hyaccmanpage to C:\windows. This finishes the installation. Since C:\windows is on the system search path of windows, the user now can use the command "hyacc" anywhere in a dos window.
8) If the user chooses to install to a different location, he can modify hyacc_dos_path.c to tell hyacc.exe where to look for hyaccpar and hyaccmanpage files. He can use hyacc in the folder in which it resides. If he wants to use hyacc anywhere in the system, he needs to go to "My Computer → Properties → Advanced → Environment variables" (in windows XP) and add hyacc's installation path to the PATH variable.

3. Usage

Type "hyacc –h" will show basic help message.
Type "hyacc –m" will show the man page file.

3.1 Input file

The input file format is compatible with Yacc and Bison. The user can check any Yacc/Bison manual or search on the Internet for the input file format.

The input file by default ends with suffix ".y", but other suffix are allowed too.

$accept, $end and $placeholder are reserved words used by Hyacc. The user should NOT use these variables.

The user can use "/*" and "*/" to quote comments in each section of the input file.

Basically there are three sections, separated by "%%" directives.

Note: all "%%" and "%…" directives should start from the first column of the line.

3.1.1 Declaration section

The first section is declaration section. Between "%{" and "%}" is the declarations used by the output compiler file.

Then these Yacc/Bison-compatible directives declare terminal tokens used by Hyacc:

%token – declares a terminal token to be returned by Lex or user-defined yylex() function.

%start – declares the start symbol of the entire grammar. Hyacc adds an extra rule "$start → start_symbol" as the first rule. The result grammar is called the augmented grammar of the input grammar. The start symbol is a non-terminal.

%left – this declares a terminal token, as well as its precedence and associativity (left).

%right – this declares a terminal token, as well as its precedence and associativity (right).

All tokens declared by %left and %right have precedence defined by the relative location of the declaration. Tokens declared by the same %left or %right declaration have the same precedence. For tokens declared by different %left or %right, those appear later have high precedence. For example

%left '+' '-'
%left '*' '/'

Then '+' and '-' have the same precedence, so are '*' and '/'. But '*' and '/' have higher precedence than '+' and '-'.

These Yacc directives are not supported yet: %nonassoc, %union, %type, %pure_parser. There are more directives used by Bison, those are not supported and ignored by Hyacc at this time. In summary, only %start, %token, %left, %right, %expect and %prec are supported by Hyacc so far.

3.1.2 Rules section

The second section is the rules section, where the user specifies all the grammars. %prec directive can be used in this section at the end of a rule to specify the context-dependent precedence and associativity of the rule. For example: A → - C %prec UNARY declares

this rule's precedence and associativity to be that of token UNARY, which should be declared in the declaration section.
In the rules section the user can put semantic actions of the rules at the end of each rule, quoted by "{" and "}". $$ indicates the value of the current rule. $n (n = 1, 2, 3…) indicates the value of the nth term on the RHS. For example:

A : B '+' C { $$ = $1 + $2; } /* $1 is the value of B, $2 is the value of C. */
;

Mid-action are not supported yet. Mid-actions are semantic actions in the middle of the RHS of a rule. For example:

A : B { $$ = $1; } '+' C
;

3.1.3 Code section

The third section is the code section, where the user puts all the other code he wants to go in the compiler file.

3.1.4 Example

An example hyacc input file is:

```
/* See http://www.gnu.org/software/bison/manual/html_mono/bison.html */
/* Infix notation calculator.  */

%{
       #define YYSTYPE double
       #include <math.h>
       #include <stdio.h>
       #include <stdlib.h>
       #include <ctype.h>
       int yylex (void);
       void yyerror (char const *);
       char * cursor = "#";
%}

/* Bison declarations.  */
%token NUM
%left '-' '+'
%left '*' '/'
%left NEG     /* negation--unary minus */
%right '^'    /* exponentiation */

%% /* Grammar rules and actions follow.  */
```

```
input:    /* empty */
        | input line
;

line:     '\n'          { printf ("\n%s ", cursor);   }
        | exp '\n'      { printf ("\t%.10g\n%s ", $1, cursor); }
        | error '\n'    { yyerrok;                  }
;

exp:      NUM                { $$ = $1;         }
        | exp '+' exp        { $$ = $1 + $3;    }
        | exp '-' exp        { $$ = $1 - $3;    }
        | exp '*' exp        { $$ = $1 * $3;    }
        | exp '/' exp        { $$ = $1 / $3;    }
        | '-' exp  %prec NEG { $$ = -$2;        }
        | exp '^' exp        { $$ = pow ($1, $3); }
        | '(' exp ')'        { $$ = $2;         }
;
%%

/* The lexical analyzer returns a double floating point
   number on the stack and the token NUM, or the numeric code
   of the character read if not a number.  It skips all blanks
   and tabs, and returns 0 for end-of-input.  */

#include <ctype.h>

int
yylex (void)
{
  int c;

  /* Skip white space.  */
  while ((c = getchar ()) == ' ' || c == '\t')
    ;
  /* Process numbers.  */
  if (c == '.' || isdigit (c))
    {
      ungetc (c, stdin);
      //ungetc (c);
      scanf ("%lf", &yylval);
      return NUM;
    }
  /* Return end-of-input.  */
  if (c == EOF)
    return 0;
  /* Return a single char.  */
  return c;
}

int
```

```
main (void)
{
  printf("%s ", cursor);
  return yyparse ();
}

#include <stdio.h>

/* Called by yyparse on error.  */
void
yyerror (char const *s)
{
  fprintf (stderr, "%s\n%s", s, cursor);
}
```

Save this file as “example.y”, type command “hyacc example.y”. This will generate y.tab.c. Note that we have yylex() defined in “example.y” so we don’t need Lex in this case. Now type “gcc y.tab.c –o example” will create the compiler file “example”. Type “./example”, you can enter mathematical expressions like “-1.2 + 3” and the compiler will give results.

3.2 command line switches

Options are given in one or two forms: either a dash followed by a single letter, or two dashes followed by a long option name. Single letter switches are supported for all the options. Long name switches are supported for some options.

```
-b fileprefix
--file-prefix==fileprefix
       Specify  a  prefix  to use for all hyacc output file names.  The
       names are chosen as if the input file were named fileprefix.c.


-c
       Use this switch  to  not  generate  parser  files  (y.tab.c  and
       y.tab.h).  This is useful when the user only wants to use the -v
       and -D switches to parse the grammar and check the y.output file
       about the grammar's information.

       -c generally is used with -v, -D and -C.


-C
       For the unit production removal optimization (when -O2 or -O3 is
       used), if a unit production rule has semantic action, when it is
       removed  the  semantic  action won't be preserved, so the output
       compiler will miss some code.

       To solve this problem, by default HYACC adds a placeholder  non-
       terminal to unit production rules with actions, so they won't be
       removed. E.g., from
       program : expression              {printf("answer = %d\n", $1);}
               ;
       to
       program : expression $PlaceHolder {printf("answer = %d\n", $1);}
               ;
       $PlaceHolder : /* empty */
                    ;

       If the -C switch is used, this default action will not be taken.
       This  is  used when the user wants to just parse the grammar and
       does not care about generating a useful compiler.  Specifically,
       -C is used together with switch -c.


-d
--define
       Write  an extra output file containing macro definitions for the
       token type names defined in the grammar.

       The file is named y.tab.h.

       This output file is essential if you wish to put the  definition
       of  yylex  in  a separate source file, because yylex needs to be
       able to refer to token type codes and the variable yylaval.   In
```

such case y.tab.h should be included into the file containing yylex.

-D

Change the print option to debug file y.output. A user who checks the debug file should assume certain degree of knowledge to the LR(1) compiler theory and optimization algorithms.

If the -v options is used, a debug file y.output will be generated when hyacc parses the grammar file. Use of -D switch will automatically turn on the -v switch, and will allow to specify what kind of information to be included into y.output.

By default, use -v will output the information about the states, plus a short statistics summary of the number of the grammar's terminals, nonterminals, grammar rules and states. like the y.output file of yacc.

-D should be followed by a parameter from 0 ~ 14:

-D0
Include all the information available.

-D1
Include the grammar.

-D2
Include the parsing table.

-D3
Include the process of generating the parsing machine, basically, the number of states and the current state in each cycle.

-D4
This is useful only if at the time of compilation, in y.h USE_CONFIG_QUEUE_FOR_GET_CLOSURE is set to 0. This then will include the information of combining compatible configurations: the number of configurations before and after the combination. -D4 can be used together with -D3.

-D5
Include the information of the multi-rooted tree(s) built for the optimization of removing unit productions.

-D6
Include the information in the optimization of removing unit productions. Specifically, the new states created and the original states from which the new states are combined from.

-D7
Include the information of the step 4 in the optimization of removing unit productions. Specifically, this shows the states reachable from state 0.

-D8
Show the entire parsing table after removing unit productions,

including those states that will be removed.

-D9
Show a list of configurations and the theads of the strings after the scanning symbol.

-D10
Include information of the symbol hash table.

-D11
Include the information of shift/shift conflicts if any. This happens when the input grammar is not LR(1) or ambiguous, and the optimization of removing unit production is used. The occurrence of shift/shift conflicts means the optimization of removing unit productions (-O2 and -O3) cannot be applied to this grammar.

-D12
NOT to include the default information about states when the -v option is used. Use -D12 to show only the short statistics summary, and not the states list.

-D13
Include the statistics of configurations for each state, and also dump the state hash table.

-D14
Include the information of actual/pseudo states. An actual state number is the row number of that state in the parsing table. After the step of unit production removal, some states are removed but their rows still remain in the parsing table, thus the state's pseudo state number (counted by ignoring those removed states/rows) will be different.

-D15
Shows the originator and transitor list of each configuration, as well as the parent state list of each state. This is relevant when lane-tracing is used.

-g
--graphviz
Generate a graphviz input file for the parsing machine.

-h
--help Print a usage summary of hyacc.

-K
--lrk Apply the LR(k) algorithm. The LR(k) algorithm is called the edge-pushing algorithm, which is based on lane-tracing, using a lane-tracing table based method to split states. In other words, this is extension of the option -Q (--lane-tracing-ltt) for LR(k) where k > 1. So far this works for those LR(K) grammars

where lanes involved in lane-tracing upon increasing k do not contain cycle.

-l
--nolines
Don't put any #line preprocessor commands in the parser file. Ordinarily hyacc puts them in the parser file so that the C compiler and debuggers will associate errors with your source file, the grammar file. This options causes them to associate errors with the parser file, treating it as an independent source file in its own right.

-m
--man-page
Show man page. Same as "man hyacc". This is used when the man page file exists in the same directory as the hyacc executable. So if installation moves this man page file to another location, you must use "man hyacc".

-o outfile
--output-file==outfile
Specify the name outfile for the parser file.

The other output files' names are constructed from outfile as described under the v and d switches.

-O
Specify the kind of optimization used to parse the yacc input file.

Basically, three optimizations are used: 1) Combine compatible states based on weak compatibility. 2) Remove unit productions. 3) Remove repeated states after optimization 2).

The -O switch should be followed by a parameter from 0 to 3:

-O0
No optimization is used.

-O1
Optimization 1) is used.

-O2
Optimizations 1) and 2) are used.

-O3
Optimizations 1), 2) and 3) are used.

By default, when -O switch is not specified, the optimization 1) of combining compatible states is used. So "hyacc file.y" is equivalent to "hyacc file.y -O1" or "hyacc -O1 file.y".

-P
--lane-tracing-pgm
Use LR(1) based on the lane-tracing algorithm. The lane-tracing algorithm first obtains the LR(0) parsing machine, then use lane-tracing to obtain the contexts for those states where shift/reduce or reduce/reduce conflicts exist. If conflicts are not resolved for some states, then the involved states are splitted using the practical general method.

-Q
--lane-tracing-ltt
Use LR(1) based on the lane-tracing algorithm. The lane-tracing algorithm first obtains the LR(0) parsing machine, then use lane-tracing to obtain the contexts for those states where shift/reduce or reduce/reduce conflicts exist. If conflicts are not resolved for some states, then the involved states are splitted using a lane-tracing table based method.

-R
--lalr1
Use LALR(1) algorithm based on lane-tracing phase 1.

-S
--lr0 Use LR(0) algorithm.

-t
--debug
In the parser files, define the macro YYDEBUG to 1 if it is not already defined, so that the debugging facilities are compiled. When the generated compiler parses an input yacc file, the parse process will be recorded in an output file y.parse, which includes all the shift/reduce actions, associated state number and lookahead, as well as the content of state stack and symbol stack.

-v
--verbose
Write an extra output file containing verbose descriptions of the parser states and what is done for each type of lookahead token in that state.

This file also describes all the conflicts, both those resolved by operator precedence and the unresolved ones.

The file's name is y.output.

-V
--version
Print the version number of hyacc and exit.

EXAMPLES

Assume the input grammar file is arith.y.

The user wants y.tab.c only:

```
hyacc arith.y
```

The user wants y.tab.c and y.tab.h:

```
hyacc -d arith.y
```

The user wants the generated compiler create y.parse when parsing a program:

```
hyacc -dt arith.y
or
hyacc arith.y -d -t
```

The user wants y.ta.b, y.tab.h, and create a y.output file when parsing the grammar:

```
hyacc -dv arith.y
```

The user wants, y.tab.c, y.tab.h, y.output and wants to include no other information than the short statistics summary in y.output:

```
hyacc -dD12 arith.y
```

Here -D12 will suppress the states list.

The user wants y.tab.c and y.tab.h, use optimization 1) only, and wants a default y.output:

```
hyacc -d -O1 -v arith.y
or
hyacc -dO1v arith.y
```

The user wants to parse the grammar and check y.output for information, and doesn't need a compiler. While use all the optimizations, he wants to keep those unit productions with semantic actions:

```
hyacc -cCv arith.y
```

3.3 output files

y.tab.c and y.tab.h
The output compiler file is y.tab.c. If the –d switch is used, y.tab.h is created to be included by other source files, such as lex.yy.c created by Lex.

y.output
If the –v switch is used, y.output will be created, which contains various information about the LR(1) parser generator depending on the –Dn switch.

y.parse
If the –t switch is used, y.parse will be created when running the created compiler on a source file, explaining step by step the parsing process.

y.gviz
If the –g switch is used, y.gviz will be created, which can be used as the input file for graphviz to generate a graph for the parsing machine.

The user can change the output file names using the –o and –b switches.

4. Future perspective

The future will see Hyacc:

1) More complete in being compatible with the interface of Yacc/Bison.
2) More information in y.output file.
3) More optimizations to increase the performance.
4) More compression of the created parser tables.

5. References

[Aho] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools. (1986)

[Knuth] Donald E. Knuth. On the translation of languages from left to right. Information & Control, 8(6):607–639. (1965)

[Nigel] Nigel P. Chapman. LR Parsing: Theory and Practice. (1987)

[Pager77] David Pager. A Practical General Method for Constructing LR(k) Parsers. Acta Informatica 7, 249 – 268 (1977)

[Pager77b] David Pager. Eliminating Unit Productions from LR Parsers. Acta Informatica 9, 31 – 59 (1977)

[Pager77c] David Pager. The Lane-Tracing Algorithm for Constructing LR(k) Parsers and Ways of Enhancing Its Efficiency. Information Sciences 12, 19 – 42 (1977)

[Pager73] David Pager. The lane tracing algorithm for constructing LR(k) parsers. Proceedings of the fifth annual ACM symposium on Theory of computing, 172 - 181 (1973)

Appendix B

Simple Grammars Used for Testing

These are the 17 simple grammars used for testing in Chapter 5. G1, G4, G5, G6, G7, G8, G9, G10, G14, G15, G16 are from [41], G2 from [1], G3 from[7], G11, G12, G13 from [46], G17 from [47]. Of these grammars, G2 and G3 are actually the same grammar and are LR(1). G6 is a non-LR(1) grammar (contains shift/reduce conflicts in LR(1) parsing machine). The rest are LALR(1) grammars. In the grammar rules, ϵ means empty string.

G1

E → E + T | T
T → T * a | a

G2

def → param_spec return_spec ,
param_spec → type | name_list : type
return_spec → type | name : type
type → ID
name → ID
name_list → name | name , name_list

G3

def→ param_spec return_spec COMMA
param_spec→ type | name_list COLON type
return_spec→ type | name COLON type

type→ ID

name→ ID

name_list→ name | name COMMA name_list

G4

E → E + T | T

T → (E) | n

G5

E → E + T | T

T → T * a | a | (E)

G6

E → E + T n | T

T → a | (E n) | n a

n → ϵ | num

G7

S → d i A

A → A T | ϵ

T → M | Y | P | B

M → r | c

Y → x | f

P → n | o

B → a | e

G8

A → A B | B

B → C | D

C → x y

D → s E

E → E + y | y |

G9

$E \rightarrow E + T \mid T$

$T \rightarrow a \mid num \mid (E)$

G10

$E \rightarrow E\ o\ T \mid T$

$T \rightarrow a \mid (E)$

$o \rightarrow x \mid \epsilon$

G11

$G \rightarrow A\ b \mid B\ d$

$A \rightarrow C$

$B \rightarrow C$

$C \rightarrow e\ e$

G12

$G \rightarrow A\ x \mid D\ y \mid J\ z \mid E$

$A \rightarrow B \mid B\ u$

$B \rightarrow C$

$D \rightarrow C$

$C \rightarrow E \mid K \mid K\ s$

$E \rightarrow f \mid f\ r$

$J \rightarrow K$

$K \rightarrow H$

$H \rightarrow h\ h$

G13

$G \rightarrow E\ c \mid c\ E\ c$

$E \rightarrow A\ c$

$A \rightarrow B \mid B\ E$

$B \rightarrow b\ b$

G14

program → list_statement
list_statement → statement | list_statement statement
statement → assign_statement | input_statement | output_statement | if_statement | error ';'
assign_statement → identifier = expression ';'
input_statement → Input identifier ';'
output_statement → Output expression ';'
if_statement → If (expression = expression) { list_statement } | If (error) { list_statement }
expression → expression + term | term
term → identifier | number

G15

program → Main ';' declaration_list code_start statement_list End Main ';'
declaration_list → declaration_list declaration | declaration
declaration → Int identifier_list ';'
identifier_list → identifier_list ',' Identifier | Identifier
code_start → ϵ
statement_list → statement | statement_list statement
statement → assignment_statement | input_statement | output_statement | do_statement | error ';'
assignment_statement → Identifier = expression ';'
input_statement → Input Identifier ';'
output_statement → Output expression ';'
do_statement → do_head statement_list End Do ';'
do_head → Do Identifier = Number To Number ';'
expression → expression + primary | primary
primary → Identifier | Number

G16

program → Main ';' declaration_list statement_list End Main ';'
| Main ';' statement_list End Main ';'
declaration_list → declaration_list declaration | declaration
declaration → Int identifier_list ';'
identifier_list → identifier_list ',' Identifier | Identifier

statement_list → statement_list statement | statement

statement → assign_statement | input_statement | output_statement | while_statement
| if_statement | error

while_statement → while_prefix statement_list Wend ';'

while_prefix → WHILE condition | WHILE error

WHILE → While

condition → expression Eq expression | expression Ne expression | expression Le expression

if_statement → if_prefix statement_list End If ';'

if_prefix → IF condition | IF error

IF → If

assign_statement → Identifier = expression ';'

input_statement → Input Identifier ';'

output_statement → Output expression ';'

expression → expression + term | expression - term | term

term → Identifier | Number | (expression)

G17

G → G a | a Y d | b Y c | c V e | d W f | e W e

Y → X

X → e | e g

W → P

V → U

U → a Y Z

Z → a Q | P | ϵ

P → b U

Q → c | ϵ

Appendix C

Latex2gDPS Compiler Grammar

program → declaration BEGIN EQN label eqn_body END EQN

declaration → BEGIN DECLARATION declaration_list END DECLARATION

| BEGIN DECLARATION END DECLARATION

declaration_list → declaration_list declaration_item_line

| declaration_item_line

declaration_item_line → declaration_item NEW_LINE_list

| declaration_item

declaration_item → module_type

| module_name

| dimension

| datatype

| goal

| base

| alias

NEW_LINE_list → NEW_LINE_list NEW_LINE

| NEW_LINE

module_type → MODULE_TYPE ':' ID

module_name → MODULE_NAME ':' ID

dimension → DIMENSION ':' dimension_list

dimension_list → dimension_list ',' dimension_item

| dimension_list ',' NEW_LINE dimension_item

| dimension_item

dimension_item → dimension_name EQ ‘{’ array_list2 ‘}’

| dimension_name EQ ‘{’ expr_list ‘}’

| dimension_name EQ term

array_list2 → array_list ‘;’ expr_list

array_list → array_list ‘;’ expr_list

| expr_list

dimension_name → term

datatype → DATA_TYPE ‘:’ datatype_list

datatype_list → datatype_list ‘,’ datatype_item

| datatype_item

datatype_item → ID ID

goal → MODULE_GOAL ‘:’ FXN_NAME expr_list ‘)’

base → MODULE_BASE ‘:’ module_base_list

module_base_list → module_base_list ‘;’ NEW_LINE_list module_base

| module_base_list ‘;’ module_base

| module_base

module_base → fxn EQ expr module_base_cond

module_base_cond → ϵ

| IF module_base_logic_expr

module_base_logic_expr → module_base_relation_expr logic_op module_base_relation_expr

| module_base_relation_expr

module_base_relation_expr → expr relation_op expr

| ‘(’ module_base_logic_expr ‘)’

alias → ALIAS ‘:’ alias_list

alias_list → alias_list ‘,’ alias_item

| alias_item

alias_item → ID EQ ID

label → ϵ

| LABEL ‘{’ ID ‘}’

eqn_body → fxn EQ fxn_val

| fxn NEW_LINE_list EQ fxn_val

fxn → FXN_NAME expr_list ‘)’

fxn_val → LEFT BRACKET_LEFT formular_array RIGHT PERIOD

| formular_item dot

formular_array → BEGIN ARRAY LL formular_lines END ARRAY

| ϵ

formular_lines → formular_lines formular_line

| formular_line

formular_line → formular '&' condition NEW_LINE

formular → style_formular

| expr

style_formular → '{' DISP_STYLE formular_item '}'

formular_item → opt_hd BRACKET_LEFT formular_expr BRACKET_RIGHT

formular_expr → formular_expr operator term

| formular_expr '^' '{' term '}' | formular_expr term

| term

| '|' formular_expr '|'

expr → expr operator term

| expr '^' '{' term '}'

| expr set_operator term

| expr term

| term

| '|' expr '|'

| '{' expr_list '}'

| '-' term %prec NEG

term → indexed_term

| ID

| NUMBER

| INFINITY

| greek_symbol

| BRACKET_LEFT expr BRACKET_RIGHT

| EMPTY_SET

| DOTS

| fxn

| '(' expr_list ')'

| QUAD

greek_symbol → EPSILON

| ALPHA

| BETA

indexed_term → INDEXED_ID expr_list '}'

expr_list → expr_list ',' expr

| expr

opt_hd → m_hdr term set_operator set '}'

| m_hdr term '}'

m_hdr → MIN_HEAD

| MAX_HEAD

set → expr

| BRACKET_LEFT expr ',' DOTS ',' expr BRACKET_RIGHT

| BRACKET_LEFT expr ',' expr BRACKET_RIGHT

set_operator → IN

| NOTIN

| CUP

condition → MBOX '{' IF logic_expr dot '}' dot

| MBOX '{' OTHERWISE dot '}' dot

dot → ϵ

| PERIOD

| ','

logic_expr → relation_expr logic_op relation_expr

| relation_expr

relation_expr → '$' relation_expr_list '$'

relation_expr_list → expr relation_op expr

| relation_expr_list relation_op expr

logic_op → AND | OR

relation_op → LT | GT | EQ | NE | LE | GE

operator → '+' | '-' | '*' | '/'

Bibliography

[1] Bison manual. http://www.gnu.org/software/bison/manual/html_mono/bison.html#Mystery-Conflicts.

[2] Examples of LR(2) grammars. http://www.cs.man.ac.uk/ pjj/complang/g2lr.html.

[3] GNU Bison. http://www.gnu.org/software/bison/.

[4] LR(closed) grammars. http://compilers.iecc.com/comparch/article/08-12-102.

[5] LRSYS, PASCAL LR(1) Parser Generator System. http://www.nea.fr/abs/html/nesc9721.html.

[6] New LR parser generation algorithm. http://compilers.iecc.com/comparch/article/06-05-006.

[7] Ocamlyacc tutorial. http://plus.kaist.ac.kr/∼shoh/ocaml/ocamllex-ocamlyacc/ocamlyacc-tutorial/sec-mysterious-reduce-reduce-conflicts.html.

[8] Parser Generator Dr. Parse. http://www.softpedia.com/get/Programming/Coding-languages-Compilers/Dr-Parse.shtml. Now invalid: http://www.zpnetics.com/.

[9] Parser Generator Dragon. http://freshmeat.net/projects/dragon-pg/.

[10] Parser Generator Parsing.py: An LR(1) parser generator with CFSM/GLR drivers. http://compilers.iecc.com/comparch/article/07-03-076.

[11] Toolset COCOM & scripting language DINO. http://sourceforge.net/projects/cocom/.

[12] Yacc++ and the Language Objects Library. http://www.world.std.com/∼compres/.

[13] "Yacc-keable" Grammars. http://www.angelfire.com/ar/CompiladoresUCSE/COMPILERS.html.

[14] Alfred V. Aho and Jefferey D. Ullman. The Theory of Parsing, Translation, and Compiling. Volumn 1: Parsing. Prentice-Hall, Englewood Cliffs, N. J., 1972.

[15] Alfred V. Aho, Jefferey D. Ullman, and Ravi Sethe. Compilers: Principles, Techniques, and Tools. 1986.

[16] Andrew Appel. Modern Compiler Implementation in C. 1998.

[17] Ole L. Madsen Bent B. Kristensen. Methods for computing LALR(k) lookahead. ACM Transactions on Programming Languages and Systems, 3(1):60–82, January 1981.

[18] Boris Burshteyn. MUSKOX Algorithms. http://compilers.iecc.com/comparch/article/94-03-067.

[19] Xin Chen. Hyacc 0.9 release. http://compilers.iecc.com/comparch/article/08-02-019.

[20] Xin Chen. Parser Generator Hyacc. http://hyacc.sourceforge.net.

[21] Kwang-Moo Choe. cs522 lecture notes - computation of first$_k$.

[22] Chris Clark. More yacc++ historical notes. http://compilers.iecc.com/comparch/article/00-02-142.

[23] Chris Clark. Yacc++ historical notes. http://compilers.iecc.com/comparch/article/05-06-124.

[24] Frank L. DeRemer. Practical translators for LR(k) languages. PhD thesis, MIT, Cambridge, 1969.

[25] Frank L. DeRemer and Thomas Pennello. Efficient computation of LALR(1) look-ahead set. TOPLAS, 4(4), October 1982.

[26] J. Grattage and Nottingham University T. Alterkirch. A compiler for a functional quantum programming language, February 2005.

[27] A. G. Harford, V. P. Heuring, and M. G. Main. A new parsing method for non-LR(1) grammars. Software Practice and Experience, 22(5):419 – 437, May 1992.

[28] M. L. Joliat. On the reduced matrix representation of LR(k) parser tables. Technical Report Tech. Report CSRG-28, Computer Systems Resarch Group, University of Toronto, October 1973.

[29] Donald E. Knuth. On the translation of languages from left to right. Information and Control, 8(6):607–639, 1965.

[30] A. J. Korenjak. Efficient LR(1) processor construction. In Proceedings of the first annual ACM symposium on Theory of computing, pages 191 – 200, Marina del Rey, California, United States, 1969.

[31] A. J. Korenjak. A practical method for constructing LR(k) processors. Commun. ACM, 12:613–623, November 1969.

[32] P. M. Lewis and R. E. Stearns. Syntax-directed transduction. J. ACM, 15(3):465–488, 1968.

[33] M. S. Livstone, R. Weiss, and L. F. Landweber. Automated design and programming of a microfluidic dna computer. Natural Computing, 5:1 – 13, 2006.

[34] Paul Mann. LRGen 8.0. http://compilers.iecc.com/comparch/article/07-09-045.

[35] Alessandro Paone Massimo Ancona. Table merging by compatible partitions for LR parsers is NP-complete. Elektronische Informationsverarbeitung und Kybernetik, 30(3):123–134, 1994.

[36] Vittoria Gianuzzi Massimo Ancona. A new method for implementing LR(k) tables. Inf. Process. Lett., 13(4/5).

[37] Vittoria Gianuzzi Massimo Ancona, Claudia Fassino. Optimization of LR(k) "Reduced Parsers". Inf. Process. Lett., 41(1):13–20, 1992.

[38] Vittoria Gianuzzi Massimo Ancona, Gabriella Dodero. Building collections of LR(k) items with partial expansion of lookahead strings. SIGPLAN Notices, 17(5):24–28, 1982.

[39] Vittoria Gianuzzi M. Morgavi Massimo Ancona, Gabriella Dodero. Efficient construction of LR(k) states and tables. ACM Trans. Program. Lang. Syst., 13(1):15–178, 1991.

[40] Holger Mauch. Automated Translation of Dynamic Programming Problems to JAVA Code and their Solution via an Intermediate Petri Net Representation. PhD thesis, University of Hawaii, March 2005.

[41] David Pager. University of Hawaii at Manoa 2006 Spring Course ICS611: Compiler Theory and Construction.

[42] David Pager. A solution to an open problem by knuth. Information and Control, 17:462–473, 1970.

[43] David Pager. Some ideas for left-to-right parsing. Technical Report Tech. Report No. PE 84, University of Hawaii, Information Sciences Program, October 1970.

[44] David Pager. On the incremental approach to left-to-right parsing. Technical Report Tech. Report No. PE 238, University of Hawaii, Information Sciences Program, January 1972.

[45] David Pager. The lane tracing algorithm for constructing LR(k) parsers. In Proceedings of the fifth annual ACM symposium on Theory of computing, pages 172 – 181, Austin, Texas, United States, 1973.

[46] David Pager. Eliminating unit productions from LR parsers. Acta Informatics, 9:31 – 59, 1977.

[47] David Pager. The lane-tracing algorithm for constructing LR(k) parsers and ways of enhancing its efficiency. Information Sciences, 12:19–42, 1977.

[48] David Pager. A practical general method for constructing LR(k) parsers. Acta Informatica, 7:249 – 268, 1977.

[49] David Pager. Evaluating Terminal Heads Of Length K. Technical Report No. ICS2009-06-03, University of Hawaii, Information and Computer Sciences Department, November 2008. http://www.ics.hawaii.edu/research/tech-reports/terminals.pdf/view.

[50] David Pager. The Lane Table Method Of Constructing LR(1) Parsers. Technical Report No. ICS2009-06-02, University of Hawaii, Information and Computer Sciences Department, May 2008. http://www.ics.hawaii.edu/research/tech-reports/LaneTableMethod.pdf/view.

[51] David Pager. Resolving LR Type Conflicts at Translation or Compile Time. Technical Report No. ICS2009-06-01, University of Hawaii, Information and Computer Sciences Department, 2009. http://www.ics.hawaii.edu/research/tech-reports/Real time evaluation of LR contexts.pdf/view.

[52] Terence Parr. Obtaining practical variants of LL(k) and LR(k) for k $>$ 1 by splitting the atomic k-tuple. PhD thesis, Purdue University, August 1993.

[53] Francois Pottier and Yann Regis-Gianas. Parser Generator Menhir. http://cristal.inria.fr/$\sim$fpottier/menhir/.

[54] Jan Rekers. Parser Generation for Interactive Environments. PhD thesis, University of Amsterdam, 1992.

[55] D. J. Rosenkrantz and R. E. Stearns. Properties of deterministic top-down grammars. Information and Control, 17(3):226–356, 1970.

[56] Peter Selinger. A brief survey of quantum programming languages. http://www.mscs.dal.ca/ selinger/papers/flops04.ps, 2004.

[57] David Spector. Full LR(1) parser generation. ACM SIGPLAN Notices, pages 58 – 66, 1981.

[58] David Spector. Efficient full LR(1) parser generation. ACM SIGPLAN Notices, 23(12):143–150, 1988.

[59] Masaru Tomita. Efficient Parsing for Natural Language. Kluwer Academic Publishers, Dordrecht, 1986.

[60] David Tribble. YACC/M, Yet Another Compiler-Compiler, An LR(1) Parser Generator for Java. http://david.tribble.com/yaccm.html.

[61] David Tribble. The Honalee LR(k) algorithm. http://david.tribble.com/text/honalee.html, 2006.

[62] Lin Wang, Xiao-bo Yue, and Ying-chun Kuang. The research of grammar scanner based on petri net model. Journal of Changsha Communications University, 21(1), March 2005.

[63] Charles Wetherell and A. Shannon. LR automatic parser generator and LR(1) parser. Technical Report UCRL-82926 Preprint, July 1979.

[64] Ya-qin Zhao and Xian-zhong Zhou. Generalized LR syntactic analytic algorithm based on neural network. Computer Applications, 2005(6), June 2005.