Full LR(1) Parser Generator Hyacc And Study On The Performance of LR(1) Algorithms

Xin Chen
Department of Information and Computer Science
University of Hawaii at Manoa
POST Building 317, 1680 East-West Road
Honolulu, HI 96822
(808) 226-3584
chenx@hawaii.edu

David Pager
Department of Information and Computer Science
University of Hawaii at Manoa
POST Building 317, 1680 East-West Road
Honolulu, HI 96822
(808) 292-5629
pagerd001@hawaii.rr.com

ABSTRACT
Despite the popularity of LALR(1) parser generators such as Yacc/Bison and LL parser generators such as ANTLR, robust and effective LR(1) parser generators are rare due to expensive performance and implementation difficulty. This work employed relevant algorithms, including the Knuth canonical algorithm, Pager's practical general method, lane-tracing algorithm, unit production elimination algorithm and its extension, and the edge-pushing algorithm, implemented an efficient, practical and Yacc/Bison-compatible open-source parser generator Hyacc, which supports full LR(0)/LALR(1)/LR(1) and partial LR(k). Based on the implementation, an empirical study was conducted comparing different LR(1) parser generation algorithms and 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.

Categories and Subject Descriptors
D.3.4 [**Programming Languages**]: Processors – *code generation, parsing, translator writing systems and compiler generators.*

General Terms
Algorithms, Performance.

Keywords
LR(1), Parser Generator, Hyacc, Algorithm, Performance.

1. INTRODUCTION

1.1 Overview
In 1965 Knuth proposed the canonical LR(k) algorithm [1], which is a very powerful parser generation algorithm for context-free grammars. But it was criticized for being too expensive in time and space to be practical at that time. Despite some relevant research to reduce the cost of LR(k) algorithm, people soon get used to parser generators such as Yacc [2] and later Bison [3], which used the less powerful but more practical LALR(1) algorithm. LL parser generators such as ANTLR [4] started to gain popularity since 1990s. However robust and practical LR(1) parser generators keep scarce, not to say the more expensive case of LR(k).

To address this issue, this work has developed Hyacc [5], an efficient, practical and Yacc/Bison-compatible open source full LR(0)/LALR(1)/LR(1) and partial LR(k) parser generation tool in ANSI C, based on the canonical algorithm of Knuth [1], the lane-tracing algorithm of Pager [6][7], the practical general method of Pager [8], the unit production elimination algorithm of Pager [9] and its extension [10], and a new partial LR(k) algorithm called the edge-pushing algorithm [10].

Based on the implementation of Hyacc, we investigated details in the existing LR(1) algorithms, and compared the performance of LR(1) algorithms (Knuth's canonical algorithm [1], Pager's lane-tracing algorithm [6][7] and Pager's practical general method [8]) as implemented in Hyacc with the LALR(1) algorithm as implemented in Bison with regard to the size of parsing machine, conflict resolution, as well as running time and space. The performance study was conducted on 13 programming languages, including Ada, ALGOL60, COBOL, Pascal, Delphi, C, C++, Java and more.

1.2 Relevant Parser Generation Algorithms
The three LR(1) algorithms used in Hyacc are actually all LR(k) algorithms. We only employed the case k = 1, because when k > 1 it is computationally expensive and also hard to implement. 1) The canonical algorithm of Knuth (1965) [1]. This algorithm is more powerful than LALR(1) and LL parser generation algorithms, and has less restrictions on the structure of the grammar. However it was criticized in the past for being practically infeasible, since its worse case computational complexity grows exponentially. 2) The lane-tracing algorithm of Pager (1977) [6][7]. This algorithm first generates a LR(0) parsing machine, then splits those states that cause conflicts. If the grammar is LR(1), such splitting would resolve all the conflicts. 3) The practical general method of Pager (1977) [8]. In contrast to the lane-tracing algorithm, the practical general method solves the conflict problem by merging instead of splitting. It generates all the states in the full LR(1) parsing machine. But when it does so, it merges compatible states along the way. The merging used in Hyacc is based on the concept of weak compatibility.

The LR(0) algorithm used in Hyacc is the traditional LR(0) algorithm. The LALR(1) algorithm used in Hyacc is based on the

C³S²E-11 2011, May 16–18, 2011, Montreal [QC, CANADA]
Editors: Abran, Desai, Mudur

first phase of the lane-tracing algorithm. Both are implemented because Pager's lane-tracing algorithm depends on these as the first step.

Unit productions are those with only one symbol on the right hand side of the production. Removal of unit productions from a grammar decreases the size of the parsing machine and increases parsing speed. Parsing machines generated by Pager's unit production algorithm are found to contain duplicate states. Therefore an extension algorithm [10] is designed to remove the redundancy. Pager's unit production elimination algorithm and the extension algorithm are both implemented into Hyacc.

Finally, the edge-pushing LR(k) algorithm [10] is based on recursively applying the lane-tracing process to split further on the unresolved conflict states.

Besides the algorithms used in Hyacc, we also briefly review a couple of relevant ones below.

Korenjak's Partitioning Algorithm in 1969 [13] 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.

Spector first proposed his Splitting Algorithm in 1981 [11], based on splitting the inadequate states of an LR(0) parsing machine. In that sense it is similar to the lane-tracing algorithm of Pager. He further refined his algorithm in 1988 [12]. He did not have a formal proof of the validity of the algorithm, and only gave some examples to show how it worked.

In addition, 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.

1.3 The Need For Revisiting LR(1) Parser Generation

First is the inadequacy 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 [14] 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 trade-off in efficiency and recognition power. Yacc and Bison are popular open source LALR(1) parser generators.

Second is 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) [15], 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: i) Although a subset of LR(1), LALR(1) can cover most programming language grammars. ii) The size of the LALR(1) parsing machine is smaller than the LR(1) parsing machine. iii) Each shift/reduce conflict in a LALR(1) parsing machine also exists in the corresponding LR(1) parsing machine. iv) "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 can be of the same size as a LALR(1) parsing machine for LALR(1) grammars. Only for LR(1) grammars that are not LALR(1), LR(1) parsing machines are much bigger. Further, there exist algorithms that can reduce the running time and parsing table size, such as those by Pager [6][7][8] and Spector [11][12].

Third is the current status of LR(1) parser generators. 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.

Considering all the advantages that LR(1) parser generation can provide, we feel it is beneficial to revisit the LR(1) parser generation problem, conduct a thorough investigation and provide a practical solution, so as to bring the power of LR(1) parser generation to life.

2. THE HYACC PARSER GENERATOR

2.1 Overview

Hyacc is an efficient, practical and Yacc/Bison-compatible open source parser generator, written from scratch in ANSI C. It supports full LR(0)/LALR(1)/LR(1) and partial LR(k). Hyacc is pronounced as "HiYacc", means Hawaii Yacc. Current features of Hyacc include:

1) Implements the original Knuth canonical algorithm.

2) Implements the practical general method based on weak compatibility.

3) Implements the unit production elimination algorithm.

4) Implements the extension to the unit production elimination algorithm.

5) Implements the lane-tracing algorithm.

6) Implements LALR(1) based on the lane-tracing algorithm phase 1.

7) Implements the traditional LR(0) algorithm.

8) Implements partial 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) Is compatible to Yacc and Bison in input file format, ambiguous grammar handling and error handling.

13) Works together with Lex. Or a customized yylex() function can be provided.

14) If requested, can generate a graphviz input file for the parsing machine.

15) If requested, the generated parser can record the parsing steps in a file, which makes it easy for debugging and testing.

16) Is ANSI C compliant, thus easy to port to other platforms.

17) Rich information in its debug output.

What is left to be implemented is that Hyacc does not support these Yacc directives: %nonassoc, %union, %type.

Hyacc is released under the GPL license, but the LR(1) parse engine file hyaccpar and LR(k) parse engine file hyaccpark come under the BSD license. This guarantees 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 manual [16][17].

Hyacc version 0.9 was released to the open source community in January 2008 [5]. Version 0.95 was released in April 2009. Version 0.97 was released in January 2011.

2.2 Architecture Of The Hyacc Parser Generator

Fig. 1 shows the steps on how the Hyacc parser generator works.

Hyacc first gets command line switch options, then reads from the grammar input file. Next, the step "Generate parsing machine" creates the parsing machine according to different algorithms as specified by the command line switches. Fig. 2 and Fig. 3 show the relationship of these algorithms.

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

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

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

Fig. 2 shows the relationship of the algorithms used in Hyacc from the point of view of grammar processing. The input grammars can be processed by taking the left side merging path, first be processed by the Knuth LR(1) algorithm, then end here or be processed by the PGM LR(1) algorithm.

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

On the splitting side, it always generates the LR(0) parsing machine first. It then can generate the LALR(1) parsing machine based on the first phase of the lane-tracing algorithm. 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 [3], the second is based on a lane table [18]. Then if 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. In this case, the UPE algorithm can be applied. The UPE Ext algorithm can be used to further remove redundant states after the UPE step.

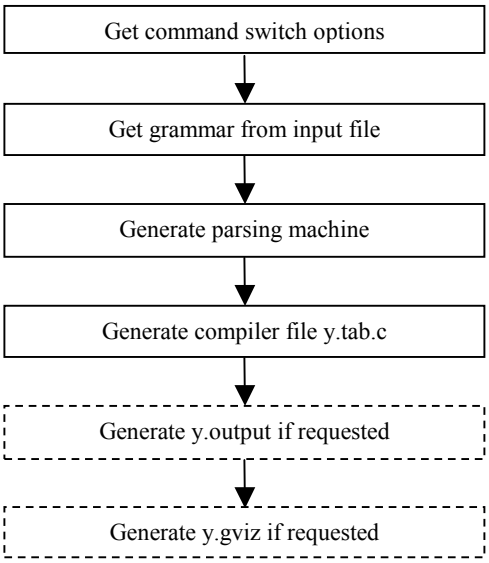

Figure 1. Overall architecture of the Hyacc parser generator

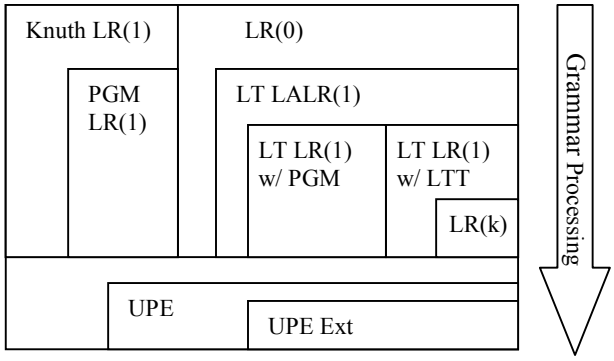

Figure 2. Relationship of algorithms from the point of view of grammar processing [1]

[1] Knuth LR(1) – Knuth canonical algorithm, PGM LR(1) – Pager's practical general method, LT LALR(1) – LALR(1) based on lane-tracing phase 1, LT LR(1) w/ PGM – lane-tracing LR(1) algorithm based on Pager's practical general method, LT LR(1) w/ LTT – lane-tracing LR(1) algorithm based on Pager's lane table method, UPE – Pager's unit production elimination algorithm, UPT Ext – Extension algorithm to Pager's unit production elimination algorithm.

2.3 Performance

The performance of Hyacc is compared to efficient LR(1) parser generators Menhir [19] and MSTA [20]. Menhir implements Pager's Practical General Method in Caml. MSTA is an open source parser generator implemented in C++.

Table 1 and Table 2 show a comparison of the generated parsing table on C++ and C grammars. 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 is significantly smaller. Obviously Menhir uses some other optimizations to compress its generated parsing table. MSTA also compresses its generated parsing machine, but not as much as Menhir.

Table 3 and Table 4 show the running time comparison on C++ and C grammars. It is similar for all the three parser generators. There is no significant difference in the measurement.

We conclude that Hyacc is a very efficient parser generator. Furthermore, it should be a favourable choice for compiler programmers with reduced-space LR(1) algorithms, which are not usually available in other LR(1) parser generators. MSTA does not use reduced-space LR(1) algorithms. For Menhir, the implementation language Caml is not so popular in industry. That said, Menhir and MSTA are both very efficient and useful parser generators in their domains.

Table 1. Parsing table size comparison of Menhir, MSTA and Hyacc on C++ grammar.

	Knuth LR(1)	PG MLR(1)	LALR(1)
Menhir	4325	1238	-
MSTA	9724/8413 [2]	-	1237/1196 [3]
Hyacc	9723	1384	1236

Table 2. Parsing table size comparison of Menhir, MSTA and Hyacc on C grammar.

	Knuth LR(1)	PG MLR(1)	LALR(1)
Menhir	1172	351	-
MSTA	1575/1572	-	352/338
Hyacc	1574	351	351

Table 3. Running time (sec) comparison of Menhir, MSTA and Hyacc on C++ grammar.

	Knuth LR(1)	PG MLR(1)	LALR(1)
Menhir	1.971	1.484	-
MSTA	5.319	-	1.175
Hyacc	3.529	1.779	1.101

Table 4. Running time (sec) comparison of Menhir, MSTA and Hyacc on C grammar.

	Knuth LR(1)	PG MLR(1)	LALR(1)
Menhir	1.640	0.557	-
MSTA	0.918	-	0.130
Hyacc	1.047	0.420	0.189

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

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

3. MEASUREMENTS AND EVALUATIONS OF LR(1) ALGORITHMS

Since Hyacc has implemented LR(0), LALR(1) and several LR(1) parser generation algorithms, it is natural to conduct a performance study on them.

The data in this study are collected on a Dell Inspiron 600M computer, with 1.7GHz Intel Pentium processor and 1GB RAM. The operating system is Fedora core 4.0. The version of Bison is 2.3. For all the measurements, time is in sec (second), memory is in MB (megabyte). All the algorithms are implemented in Hyacc, except for the Bison LALR(1) algorithm, which is implemented in Bison.

The following algorithms are measured in this study. Their acronyms are introduced here and used in later discussion. There are three LR(1) algorithms, of these the latter two are reduced-space: 1) Knuth LR(1): Knuth canonical algorithm, 2) PGM LR(1): LR(1) based on the practical general method (PGM), 3) LT LR(1) w/ PGM: LR(1) based on the practical general method, use PGM in phase 2. There are two LALR(1) algorithms: 1) LT LALR(1): LALR(1) based on lane-tracing phase 1, 2) Bison LALR(1): LALR(1) as implemented in Bison. Finally, a LR(0) algorithm: The traditional LR(0) algorithm.

Table 5. Number of terminals, non-terminals and rules in the 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

17 simple grammars were used to test the correctness of Hyacc. The grammars of 13 real programming languages were used to check the performance of Hyacc. These real language grammars were obtained from [21] with minor modifications to fit in Yacc-style grammar input. Table 5 shows the statistics of these 30 grammars.

3.1 Parsing Table Size Comparison

A comparison of parsing table sizes of the 30 grammars is shown in Table 6. Fig. 3 contains the 13 real language grammars only, and is the graphic version of the comparison that better visualizes the comparison.

We can see that the size of Knuth canonical LR(1) parsing machine is much bigger than the rest. 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 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.

We can conclude that 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 then the reduced-space LR(1) parsing machine has the same size as the LALR(1) parsing machine. 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 [8].

3.2 Parsing Table Conflict Comparison

A comparison of parsing table conflicts is shown in Table 7. 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). LT LR(1) w/ PGM and PGM LR(1) give the same number of conflicts as LALR(1) (except for 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 resolved in LR(1) parsing machine. The parsing machines of some programming language grammars (Algol60, C++, Delphi) contain reduce/reduce conflicts that cannot be resolved by LR(1) algorithms, and are not LR(1) grammars.

3.3 Running Time Comparison

Table 8 shows running time comparison of the 13 programming language grammars. Fig. 4 is the graphic view. The Knuth LR(1) algorithm takes the longest time. As expected, reduced-space LR(1) algorithms are faster than Knuth LR(1), and close to Bison LALR(1), or even faster. And quite understandable, the LR(0) algorithm runs the fastest.

We can conclude that even though more expensive than the rest here, Knuth LR(1) parser generation is still practical in running time, since it takes just a few seconds at most for the given grammars.

3.4 Memory Usage Comparison

Table 9 shows memory usage comparison of the 13 programming language grammars. Fig. 5 is the graphic view. The Knuth LR(1) algorithm always uses more or much more memory than the rest. Reduced-space LR(1) algorithms use much less memory than Knuth LR(1), and often not much more than LALR(1). Here although Knuth LR(1) parser generation requires much more memory than LT LR(1) and PGM LR(1), it is still acceptable for today's personal computers, even for grammars as complex as that of C++ 5.0.

3.5 Conclusion

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 it.

Despite this, 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 two reduced-space LR(1) algorithms (PGM, LT LR(1) w/ PGM), LT LR(1) w/ PGM in general creates a smaller parsing machine.

The current implementation 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.

The practical general method based on weak compatibility is also 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. 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.

Table 6. Parsing table size comparison.

	Hyacc					Bison
Grammar	Knuth LR(1)	PGM LR(1)	LT LR(1) w/ PGM	LR(0)	LT LALR(1)	LALR(1)
G1	8	8	8	8	8	9
G2	21	20	20	19	19	20
G3	21	20	20	19	19	20
G4	16	9	9	9	9	10
G5	20	11	11	11	11	12
G6	35	14	14	14	14	15
G7	18	18	18	18	18	19
G8	13	13	13	13	13	14
G9	18	10	10	10	10	11
G10	17	10	10	10	10	11
G11	9	9	9	9	9	10
G12	19	19	19	19	19	20
G13	13	13	13	13	13	14
G14	82	40	40	40	40	41
G15	53	53	53	53	53	54
G16	130	73	73	73	73	74
G17	51	32	32	32	32	33
Ada	12786	873	860	860	860	861
Algol 60	1538	274	272	272	272	273
C	1572	349	349	349	349	350
Cobol	2398	657	657	657	657	658
C++ 5.0	9785	1404	1261	1256	1256	1257
Delphi	4215	609	609	609	609	610
Ftp	210	200	200	200	200	201
Grail	719	193	193	193	193	194
Java 1.1	2479	439	428	428	428	429
Matlab	588	174	174	174	174	175
Pascal	2245	418	412	412	412	413
Turbo Pascal	1918	394	386	386	386	387
Yacc	153	128	128	128	128	129

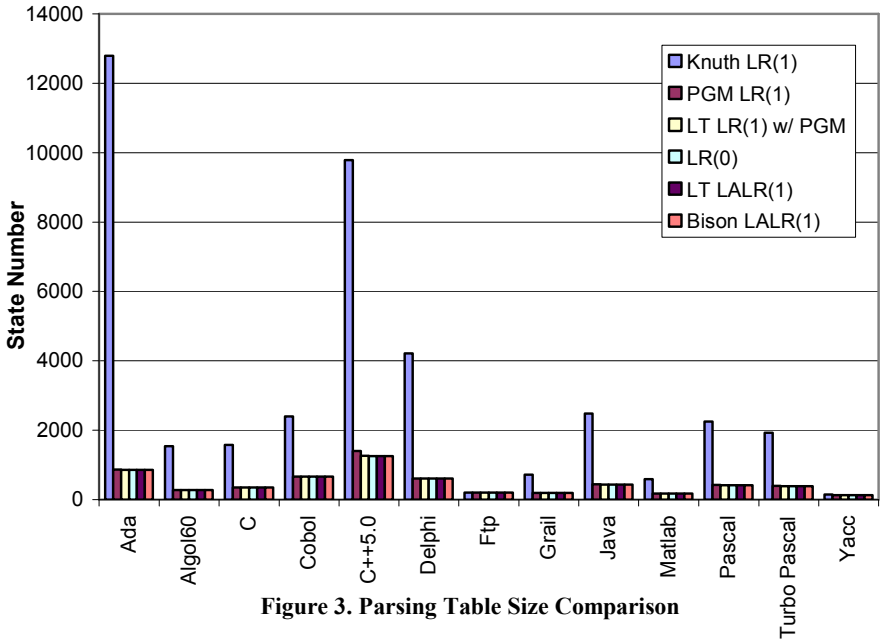

Figure 3. Parsing Table Size Comparison

Table 7. Parsing table conflict comparison

	Hyacc										Bison	
Grammar	Knuth LR(1)		PGM LR(1)		LT LR(1) w/ PGM		LR(0)		LT LALR(1)		LALR(1)	
	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	2	0	0	0	0	0
G2	0	0	0	0	0	0	1	4	0	1	0	1
G3	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
G5	0	0	0	0	0	0	2	0	0	0	0	0
G6	7	0	4	0	4	0	12	0	4	0	4	0
G7	0	0	0	0	0	0	8	0	0	0	0	0
G8	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
G10	0	0	0	0	0	0	3	0	0	0	0	0
G11	0	0	0	0	0	0	0	4	0	0	0	0
G12	0	0	0	0	0	0	3	27	0	0	0	0
G13	0	0	0	0	0	0	1	0	0	0	0	0
G14	0	0	0	0	0	0	5	0	0	0	0	0
G15	0	0	0	0	0	0	1	0	0	0	0	0
G16	0	0	0	0	0	0	6	0	0	0	0	0
G17	0	0	0	0	0	0	4	0	0	0	0	0
Ada	0	0	0	0	0	0	260	2526	0	0	0	0
Algol 60	0	4	0	1	0	1	133	336	0	1	0	1
C	2	0	1	0	1	0	214	0	1	0	1	0
Cobol	6	0	5	0	5	0	349	1480	5	0	5	0
C++ 5.0	280	31	24	18	24	18	7140	10812	24	18	24	18
Delphi	316	1191	60	174	58	139	578	1344	15	121	60	174
Ftp	0	0	0	0	0	0	6	0	0	0	0	0
Grail	0	0	0	0	0	0	117	0	0	0	0	0
Java 1.1	2	0	1	0	1	0	236	582	1	0	1	0
Matlab	25	0	14	0	14	0	142	45	14	0	14	0
Pascal	0	0	0	0	0	0	222	264	0	0	0	0
Turbo Pascal	25	0	1	0	1	0	263	288	1	0	1	0
Yacc	8	0	8	0	8	0	60	0	8	0	8	0

Table 8. Time performance comparison (second)

	Hyacc					Bison
Grammar	Knuth LR(1)	PGM LR(1)	LT LR(1) w/ PGM	LR(0)	LT LALR(1)	LALR(1)
Ada	1.883	0.406	0.172	0.136	0.173	0.155
Algol 60	0.606	0.290	0.509	0.039	0.499	0.174
C	1.047	0.420	0.192	0.067	0.189	0.225
Cobol	0.234	0.127	0.115	0.117	0.113	1.690
C++ 5.0	3.529	1.779	1.261	0.544	1.101	0.705
Delphi	1.141	0.335	0.364	0.093	0.137	0.638
Ftp	0.016	0.017	0.017	0.016	0.017	0.268
Grail	0.051	0.024	0.020	0.017	0.021	0.156
Java 1.1	1.552	1.026	0.350	0.097	0.350	0.339
Matlab	0.351	0.189	0.117	0.034	0.116	0.120
Pascal	0.504	0.174	0.066	0.050	0.066	0.246
Turbo Pascal	0.305	0.098	0.053	0.042	0.054	0.204
Yacc	0.018	0.026	0.016	0.015	0.017	0.157

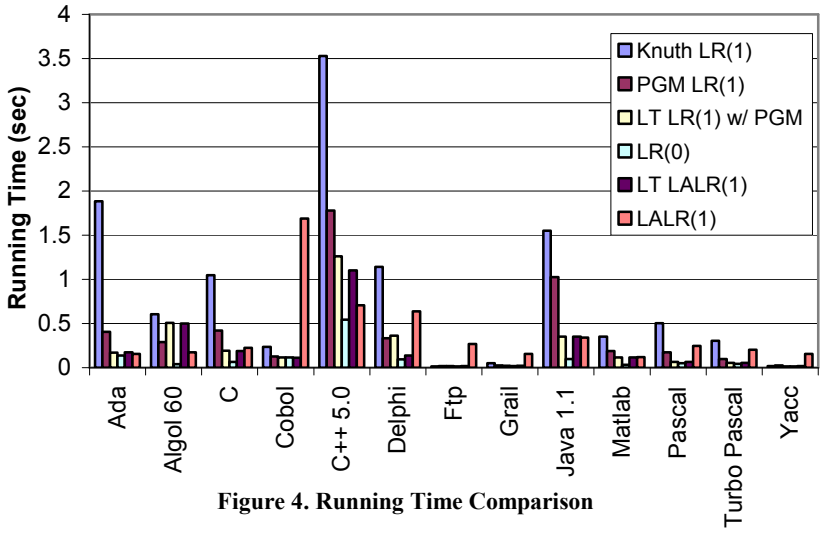

Figure 4. Running Time Comparison

Table 9. Memory usage comparison (MB)

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

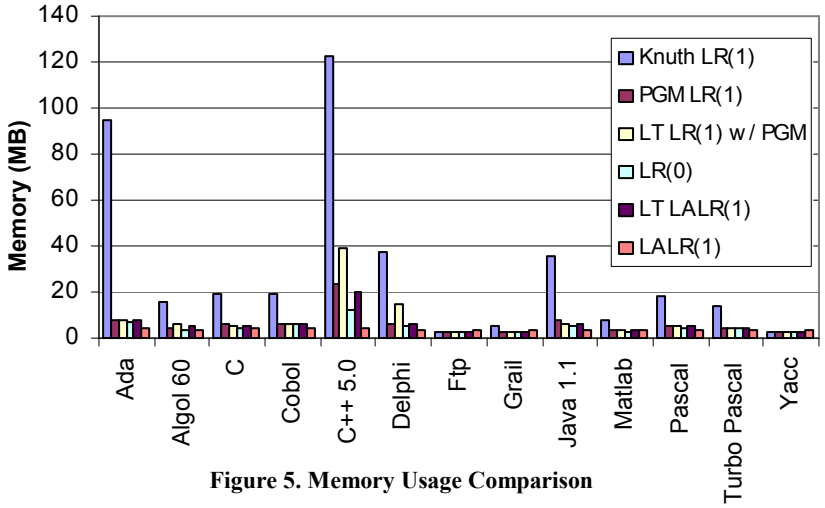

Figure 5. Memory Usage Comparison

4. RELATED WORK

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

We introduce six LR(1) parser generators that implement Pager's practical general method. These six parser generators 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 [22]. It was ported to more than nine platforms, and was used to develop compilers and system utilities. However, it is rarely used today, and not known 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. Its rigid and weird input format also limited its popularity. In addition, the use of LR is not open to the public and not free.

LRSYS was developed in the Pascal language around 1985, also at the Lawrence Livermore National Laboratory [23]. 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 CFTFORTRAN 77.

It is reported [24] 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. It is stated that Pager's algorithm was also used in GDT_PC (Grammar Debugging Tool and Parser Constructor) in about 1988.

The Menhir program in Objective Caml was developed around 2004 in France [19], and the source code is actively maintained. It implemented Pager's algorithm with slight modifications. It has since been widely used in the Caml language community, quickly replacing the previous Caml parser generator ocamlyacc.

The Python Parsing module was developed at the beginning of 2007 [25]. Its author felt that an LALR(1) parser generator could not meet his needs in developing a grammar in his work. A wide literature survey led him to Pager's PGM algorithm. 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 in March 2007. The author estimated the Python implementation to be about 100 times slower than a C counterpart, which is kind of close to the measurement here.

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

The lane-tracing algorithm was implemented by Pager in the 1970s [6][7] However the implementation was done in Assembly for OS 360, and thus not portable to other platforms. We did not find other available lane-tracing algorithm implementations.

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

Spector created a splitting LR(1) algorithm in the 1980s, which in concept is similar to Pager's lane-tracing algorithm. He implemented the algorithm in an experimental, incomplete parser generator as described in his 1988 paper [12]. Later, in 1994 the Muskox parser generator [26] implemented a version of Spector's algorithm. The author Boris Burshteyn said that the 1988 paper of Spector was short of implementation details, so he implemented the algorithm in a modified way according to his understanding.

4.4 Other LR(1) Parser Generators

More efforts were done along this line. But most of these other approaches are not formally available in literature, are implemented in proprietary products and details unknown, or sometimes are not fully working.

Yacc++ [27][28] 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.

Dr. Parse [29] is another commercial product that claimed to use LALR(1)/LR(1). Its implementation details are unknown.

MSTA [30] is a part of the open source COCOM toolset, and is believed to take the splitting approach. It claims LR(k)/LALR(k), but does not use reduced-space algorithms such as those by Pager and Spector.

In addition, most recently the IELR(1) algorithm [31][32] was proposed to provide LR(1) solution to non-LR(1) grammars with specifications to solve conflicts. The authors implemented this as an extension of Bison.

5. CONCLUSIONS

LR(1) is a powerful parser generation algorithm for context-free grammars. However LR(1) parser generation were long regarded as computationally infeasible. The community has seen various parser generators using LALR(1) and LL algorithms, but LR(1) parser generators are still rare. There are however LR(1) algorithms that can reduce the number of states in a parsing machine, making LR(1) parser generation practical.

In this work we investigated LR(1) parser generation algorithms and implemented a parser generator Hyacc, which supports LR(0)/LALR(1)/LR(1) and partial LR(k). These three LR(1) algorithms are used: 1) the Knuth canonical algorithm, 2) Pager's practical general method, 3) Pager's lane-tracing algorithm. Hyacc implemented the traditional LR(0) algorithm, and implemented LALR(1) based on the first phase of the lane-tracing algorithm. The partial LR(k) algorithm used is called the edge-pushing algorithm. Hyacc also implemented Pager's unit production elimination algorithm and an extension of it.

Hyacc has been released to the open source community. The usage of Hyacc is highly similar to the widely used LALR(1) parser generators Yacc and Bison, which makes it easy to learn and to be used. Hyacc is written in ANSI C and can be easily ported to different platforms. In summary, Hyacc is unique in its wide span of algorithm coverage, efficiency, portability, usability and availability.

We further conducted a performance study of different LR(1) algorithms as implemented in Hyacc with each other, and with LALR(1) algorithm as implemented in 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 programming languages grammars. It is safe to conclude that we can take reduced-space LR(1) as an efficient alternative of its LALR(1) peers.

6. FUTURE WORK

The Yacc directives %union, %type and %nonassoc are not implemented in Hyacc yet. It is also useful to provide the parse engine in programming languages other than C to support more developers, such as in C++ and Java.

7. REFERENCES

[1] Donald E. Knuth. On the translation of languages from left to right. Information and Control, 8(6):607–639, 1965.

[2] Stephen C. Johnson. YACC – yet another compiler compiler. CSTR 32, Bell Laboratories, Murray Hill, NJ, 1975.

[3] GNU Bison. http://www.gnu.org/software/bison/

[4] 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. http://www.antlr.org/

[5] Xin Chen. LR(1) Parser Generator Hyacc. http://hyacc.sourceforge.net. January 2008.

[6] 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.

[7] David Pager. The lane-tracing algorithm for constructing LR(k) parsers and ways of enhancing its efficiency. Information Sciences, 12:19–42, 1977.

[8] David Pager. A practical general method for constructing LR(k) parsers. Acta Informatica, 7:249 – 268, 1977.

[9] David Pager. Eliminating unit productions from LR parsers. Acta Informatics, 9:31 – 59, 1977.

[10] Xin Chen. Measuring and Extending LR(1) Parser Generation. PhD thesis, University of Hawaii, August 2009.

[11] David Spector. Full LR(1) parser generation. ACM SIGPLAN Notices, pages 58 – 66, 1981.

[12] David Spector. Efficient full LR(1) parser generation. ACM SIGPLAN Notices, 23(12):143–150, 1988.

[13] 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.

[14] Masaru Tomita. Efficient Parsing for Natural Language. Kluwer Academic Publishers, Dordrecht, 1986.

[15] M. D. Mickunas, R. L. Lancaster, and V. B. Schneider. Transforming LR(k) Grammars to LR(1), SLR(1), and (1,1) Bounded Right-Context Grammars. J. ACM, 23(3):511-533, 1976.

[16] Charles Donnelly, Richard Stallman. Bison, The YACC-compatible Parser generator (for Bison Version 1.23). 1993.

[17] Charles Donnelly, Richard Stallman. Bison, The YACC-compatible Parser generator (for Bison Version 1.24). May 1995.

[18] 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.

[19] Francois Pottier and Yann Regis-Gianas. Parser Generator Menhir. (2004)

http://cristal.inria.fr/~fpottier/menhir/

[20] Vladimir Makarov. Toolset COCOM & scripting language DINO. (2002)

http://sourceforge.net/projects/cocom

[21] "Yacc-keable" Grammars.

http://www.angelfire.com/ar/CompiladoresUCSE/COMPILERS.html

[22] Charles Wetherell and A. Shannon. LR automatic parser generator and LR(1) parser. Technical Report UCRL-82926 Preprint, July 1979.

[23] LRSYS. (1991) http://www.nea.fr/abs/html/nesc9721.html

[24] Algirdas Pakstas. (1992) http://compilers.iecc.com/comparch/article/92-08-109

[25] Parser Generator Parsing.py: An LR(1) parser generator with CFSM/GLR drivers. (2007) http://compilers.iecc.com/comparch/article/07-03-076

[26] Boris Burshteyn. MUSKOX Algorithms. (1994)

http://compilers.iecc.com/comparch/article/94-03-067

[27] Yacc++ and the Language Objects Library. (1997 – 2004) http://world.std.com/~compres

[28] Chris Clark. (2005) http://compilers.iecc.com/comparch/article/05-06-124

[29] Parser Generator Dr. Parse.

http://www.downloadatoz.com/software-development_directory/dr-parse

[30] Vladimir Makarov. Toolset COCOM & scripting language DINO. (2002)

http://sourceforge.net/projects/cocom

[31] Joel E. Denny, Brian A. Malloy. IELR(1): practical LR(1) parser tables for non-LR(1) grammars with conflict resolution. Proceedings of the 2008 ACM symposium on Applied computing, p.240-245, 2008.

[32] Joel E. Denny, Brian A. Malloy, The IELR(1) algorithm for generating minimal LR(1) parser tables for non-LR(1) grammars with conflict resolution, Science of Computer Programming, v.75 n.11, p.943-979, November, 2010