The Lane Table Method Of Constructing LR(1) Parsers

David Pager
Department of Information and Computer Science
University of Hawaii at Manoa
POST Building 317, 1680 East-West Road
Honolulu, HI 96822
+1 (808) 292-5629
pagerd001@hawaii.rr.com

Xin Chen
Department of Information and Computer Science
University of Hawaii at Manoa
POST Building 317, 1680 East-West Road
Honolulu, HI 96822
+1 (808) 226-3584
chenx@hawaii.edu

ABSTRACT
The lane-tracing algorithm is a reduced-space LR(1) parser generation algorithm. The previous version of lane-tracing algorithm regenerates states involved in reduce/reduce conflict by employing the practical general method. In this paper we describe an alternative lane-tracing approach, which regenerates states based on the lane table method. We discuss the details of this new algorithm, study its efficiency compared to existing methods, and point out that this method is better suited when extending from LR(1) to LR(k) parser generation.

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

General Terms
Algorithms, Languages, Theory.

Keywords
LR(1), Parser Generation, Lane Table, Lane-tracing, Performance.

1. INTRODUCTION

1.1 Overview
LR parser generation is more advantageous in many aspects than SLR, LALR and LL methods, but was once too expensive in performance. With each significant increment in computer processing speed and storage capacity in the last 30 years, there have been wide press discussion on how such developments could be put to use. LR(1) parser generators employing reduced-space methods, rather than the original Knuth canonical LR(1) algorithm [1], can now work efficiently in space and time not much more expensive than LALR(1) parser generators [10].

But just as the speed and capacity of computers have risen, so have the complexity and scope of computer languages, from the early versions of Fortran and Basic to the languages that are current today. Examples may include gigantic grammars employed in combination with other techniques for the purpose of natural language processing applications.

This work describes a new efficient lane-tracing LR(1) algorithm based on the lane table method, which improves our understanding of LR(1) parser generation, and opens a new door to efficient LR(k) parser generation [11].

1.2 Related Algorithms
The first practical application of the LR algorithm was by DeRemer [2] for the LALR(1) subset of LR(1) grammars.

The practical general method (PGM) of Pager [6] is a well-known LR(1) parser generation algorithm. It uses a merging process, is conceptually simple, and has been implemented into a number of parser generators, such as LR (1979) [12], LRSYS (1985) [13], Menhir (2004) [14] and Hyacc (2008) [9].

The lane-tracing algorithm is another LR(1) parser generation algorithm by Pager [4][5]. It generates a LR(1) parser generator using a splitting approach. It is more complex than the PGM method. The only known implementations are by Pager in assembly language on OS 360 (1977) [5], and in Hyacc (2008) recently.

Other works along the same lines of splitting approach include those by Spector [7][8], which was once implemented in the Muskox parser generator (1994) [15].

The partitioning algorithm of Korenjak [3] is different from the above methods. It divides a grammar into multiple parts, applies LR(1) parser generation to each part, and then combines the output together. Korenjak used Knuth's canonical method in this framework, although in theory he can also use other methods such as those by Pager and Spector.

2. THE LANE TABLE METHOD
We define the following terms for the discussion in this paper. A *grammar* for a language *L* is defined as a 4-tuple $G = (N, \Sigma, P, S)$. Here *N* is a set of *nonterminal* symbols, Σ is a set of *terminal* symbols disjoint from the set *N*, *P* is a set of *productions*, and *S* is the *start symobl* from which the production rules originate from. Following the LR algorithm, we can obtain a *parsing machine* for the grammar, which is composed of *states*. A state contains one or multiple configurations. A *configuration* is of the form $[A \rightarrow \alpha \bullet X \beta, \psi]$, where A is a nonterminal, X is a nonterminal or terminal, α and β are strings of terminals or nonterminals. The dot $\bullet$ is called the *marker* of the configuration. We optionally use square brackets ([and]) here only to make it visually more clear. The symbol on the right side of the marker, in this case X, is called the

APPLC'12, June 14, 2012, Beijing, China.

scanned symbol of the configuration. A → α X β is the *production* part of the configuration. ψ is the set of terminals that can appear immediately after X, and is called the *context* of the configuration. The symbol involved in the transition from one state to the other is called the *transition symbol*. If the transition symbol is X, the target state is called the *X-successor* of the source state. For the configuration A → α • X β, a configuration of the form A → α X • β is called its *transition successor*, and a configuratin of the form X → • η is called its *immediate successor*. A *lane* in a parsing machine is a sequence of configurations ξ_1, ξ_2, …, ξ_n, where for i = 0 to n-1, ξ_{i+1} is the immediate or transition successor of ξ_i. The state containing ξ_1 is called a *lanehead state*. For example, in Figure 6, configurations G → x • W a, W → • U X C, W → U • X C, X → • k t, X → k • t, X → k t • form a lane, and state B is a lanehead state because it contains the first configuration of the lane. Figure 2 and Figure 6 are examples of (part of a complete) parsing machines.

2.1 The Lane-Tracing Algorithm

Pager's lane-tracing algorithm [4][5] employs a two-phase approach, as shown in Algorithm 1. It starts by generating the LR(0) parsing machine, then proceeds to lane-tracing to resolve conflicts. This splitting process is divided into two phases, as shown in Figure 1 below. The first phase starts from inadequate states (states that contain shift/reduce and reduce/reduce conflicts) in the LR(0) parsing machine, 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 all the conflicts then we stop here. Otherwise, phase 2 is used to split remained inadequate states to resolve reduce/reduce conflicts, and results in a LR(1) parsing machine.

Figure 1. The two phases of the Lane-Tracing Algorithm

Algorithm 1. The Lane-Tracing Algorithm

```
Lane_Tracing_Phase1();
Resolve_Conflicts();
If not all inadequate states are resolved then
    Lane_Tracing_Phase2();
    Resolve_ Conflicts();
```

The purpose of phase 2 is to split the states that contain reduce/reduce conflicts. The phase 2 in Pager's previous work [4][5] is based on the practical general method (PGM). However, here we will present an alternative algorithm for phase 2, which is based on a lane table: a table constructed from the conflicting lanes during lane-tracing. Taken together with phase 1, this new method of phase 2 will, as before, produce a conflict-free LR(1) parser for all LR(1) grammars. Let us first look at an example on how the phase 1 of lane-tracing algorithm works.

Example 1. Given grammar G1: E → a X d | b X c | b Y d, X → e X | e, Y → e Y | e. The relevant part of LR(0) parsing machine is shown in Figure 2. State D is inadequate, because it contains a reduce/reduce conflict on two configurations: [X → e •] and [Y → e •]. To apply lane-tracing phase 1 to resolve the conflict, we generate relevant contexts. The conflict can be resolved if the generated contexts are different.

Lane-tracing for phase 1 is shown in Figure 3. The lanes that generate contexts for [X → e •] is shown in (a). The lanes that generate contexts for [Y → e •] is shown in (b). After lane-tracing, we obtain the corresponding LALR(1) parsing machine, the relevant part of which is shown in Figure 4. Now the context for configuration [X → e •] is {c, d}, the context for configuration [Y → e •] is also {c, d}. The conflict still exists, awaiting further processing by Phase 2.

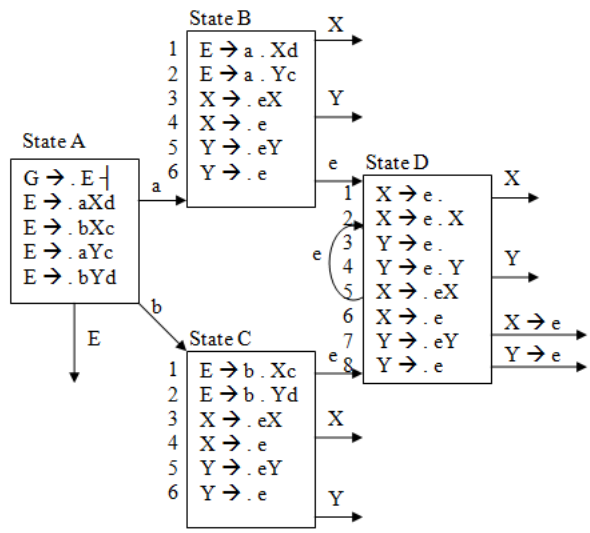


Figure 2. LR(0) parsing machine of grammar G1

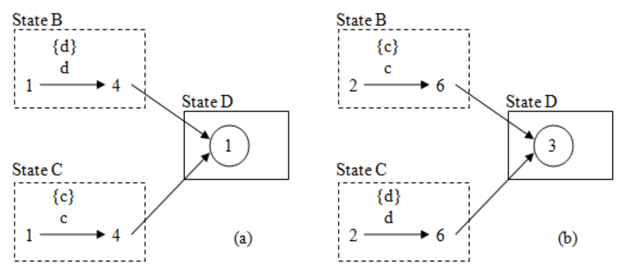


Figure 3. Lane-tracing on conflict configurations

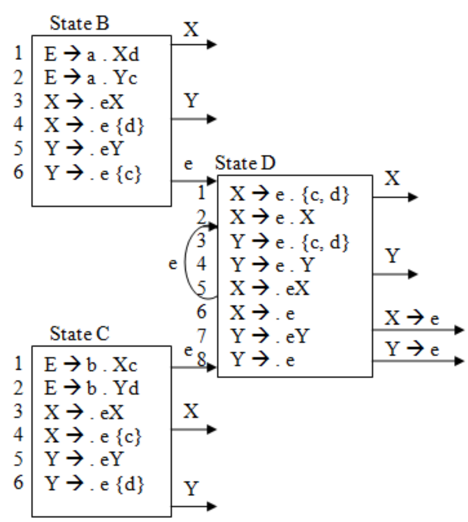


Figure 4. LALR(1) parsing machine of grammar G1

2.2 Phase 2 Of The Lane-Tracing Algorithm

Algorithm 2. Lane_Tracing_Phase2()
Get_LaneHead_List();
Phase2_PGM() or Phase2_LaneTable();

As shown in Algorithm 2, the first step of phase 2 is to get a list of lanehead states. After phase 1, we have obtained a parsing machine with inadequate states. Next we need to find out a list of states, from which lanes start and eventually lead to the unresolved reduce/reduce conflicts in the inadequate states. Then we need to regenerate the states on the conflicting lanes (the lanes that we have traced in Phase 1).

When new states are generated we check if we can combine them, or need to split by making a new copy. This process will remove the reduce/reduce conflicts if the grammar is LR(1). The previous lane-tracing algorithm uses the PGM algorithm for this purpose, which we call Phase2_PGM(). Phase2_LaneTable() is the new approach. We give an example of applying Phase2_PGM() below.

Example 2. Apply Phase2_PGM() on grammar G1. The lanehead state list is {B, C}, because lanes originate from states B and C, and lead to the unresolved reduce/reduce conflict in state D.

When applying Phase2_PGM(), we generate context closure for each of the lanehead states, and propagate the context change to successor states involved in reduce/reduce conflicts, applying the PGM method to combine or split successor states as necessary.

The e-successor of state B and state C is the inadequate state that contains the reduce/reduce conflict in the LALR(1) parsing machine. We generate a new e-successor of state B, which is state D. Next we generate a new e-successor of state C, which is state D′. We apply the PGM method to see if we can combine state D′ into state D. The answer is no because that causes a reduce/reduce conflict. Therefore we keep state D′ as a split new state. The resulted LR(1) parsing machine is shown in Figure 5. In this LR(1) parsing machine, the reduce/reduce conflict is resolved.

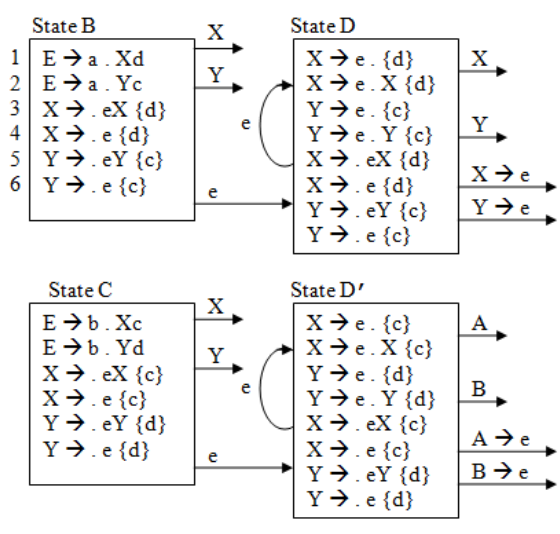


Figure 5. LR(1) parsing machine of grammar G1 after applying Phase2_PGM()

2.3 The Lane Table Method

Now we discuss the new approach: Phase2_LaneTable(). Lane-tracing based on a lane table is another way of splitting states to remove inadequate states. The idea is that, using the lane table constructed, we check the local context information of the regenerated state group to see if there is a need to split.

Let the conflicting actions at the inadequate state be $\pi_1, \pi_2, ..., \pi_r$. If a state S contains configurations that for $1 \le i \le r$, 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 $\{(C_i, i) \mid 1 \le i \le r\}$. Note that if the sets $\{C_i \mid 1 \le i \le r\}$ are not pair-wise 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 criterion according to which regenerated states may be 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 $S_1, \ldots, S_t$ be $\{(A_i, i) \mid 1 \le i \le r\}$ in each case. Now we regenerate a state T that is a successor of one of $S_1, \ldots, S_t$:

1) If there is an existing copy of state T whose associated collection of contexts is $\{(B_i, i) \mid 1 \le i \le r\}$ and the collection of the set of states $\{(A_i \cup B_i) \mid 1 \le i \le r\}$ are pair-wise disjoint, then this existing copy of state T is taken as the successor involved, and the collection of contexts associated with $\{S_1, \ldots, S_t, T\}$ is defined to be $\{(A_i \cup B_i, i) \mid 1 \le i \le r\}$.
2) Otherwise, a new copy T' of T is regenerated as the successor involved, and if the collection of contexts generated by T is $\{(B_i', i) \mid 1 \le i \le r\}$, then the collection of contexts associated with {S1, …, St, T'} is defined to be $\{(A_i \cup B_i', i) \mid 1 \le i \le r\}$.

Note that if the sets $\{(A_i \cup B_i', i) \mid 1 \le i \le r\}$ are not pair-wise disjoint, then the grammar is not LR(1).

Next we show two examples applying the lane table method.

Example 3. Given grammar G2: 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, P → z. Let us derive its LR(1) parsing machine using the lane table method

Part of the LR(0) parsing machine of grammar G2 involved in conflicting lanes is shown in Figure 6. It contains reduce/reduce conflicts at state I on configurations π_1, π_2, π_3. At state I, configurations π_1, π_2, π_3 are defined as: π_1 : U → U K t, π_2 : Y → k t, π_3 : X → k t.

Figure 6 shows the states that result in the reduce/reduce conflict at state I: A, B, … J. The configurations of each state are also shown. Arrows represent the transitions between the states. Transition symbols are labelled on the arrows. Text on the left side of some states, for example, "a for π_3" on the left side of state B, means that the configuration to the right of it [G → x • W a) generates context {a} for configuration π_3.

Figure 7 depicts the states involved in traced lanes and shows how they are connected to each other. The lanes are shown in the forward direction. The information collected is stored into a lane table as in Table 1.

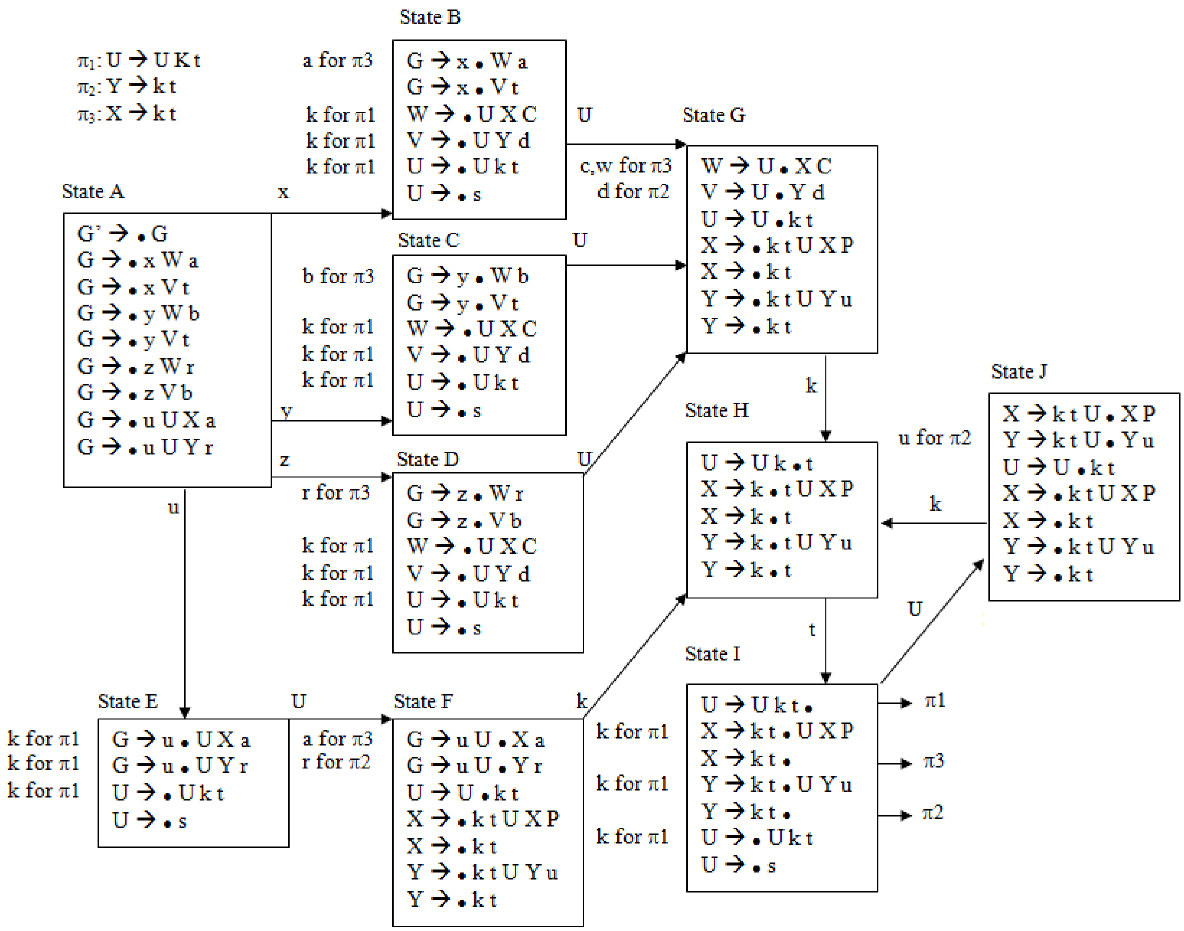


Figure 6. States on the conflicting lanes of the LR(0) parsing machine of grammar G2

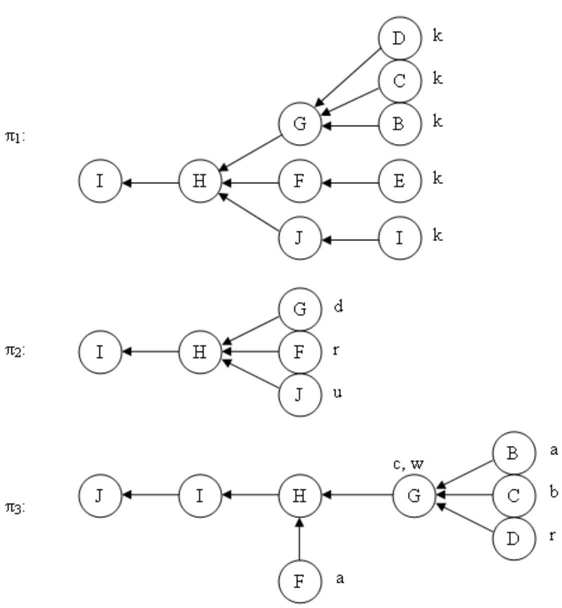


Figure 7. Conflicting lanes from lane-tracing of grammar G2

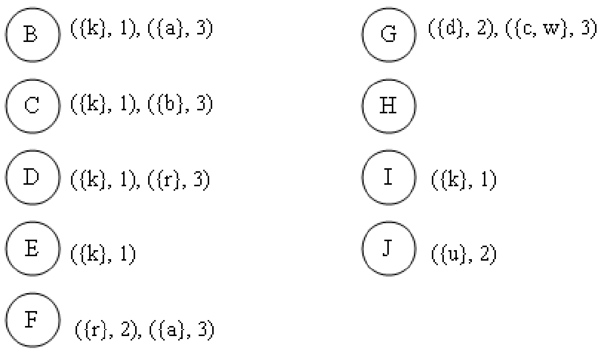
Table 1. Lane table constructed from lane-tracing in Figure 7

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}

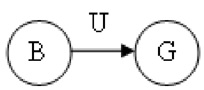
* means the labeled state is a lanehead state, i.e., it is a state from which lane(s) start, but do not pass through. For example, states B, C, D and E are where lane(s) start, and they are not in the middle or end of any path. Therefore they are all lanehead states. State F is at the start of lanes for π_1 and π_2, but it is in the middle of lanes for π_1. Therefore it is not a lanehead state.

The example of combining regenerated states is given below. The regeneration starts from lanehead states B, C, D and E. Note that no states other than states B, C, …, J are regenerated, except for their copies when split is needed.

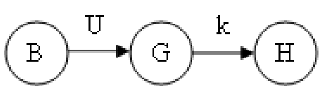
Step 1: Initially show the collection of contexts associated with each state (i.e., the collection of context generated by the state). For example, for state B, its contexts are ({k}, 1), ({a}, 3). This means state B generates context set {k} for configuration $\pi 1$, and context set {a} for configuration $\pi 3$.



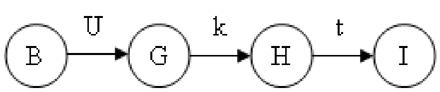
Step 2: Start from state B, first add its successor state G to the collection. The collection of contexts associated with {B, G} is: ({k}, 1), ({d}, 2), ({a, c, w}, 3).



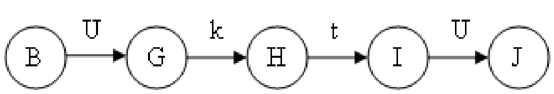
Step 3: Add the successor of state G: state H. The collection of contexts associated with {B, G, H} is: ({k}, 1), ({d}, 2), ({a, c, w}, 3).



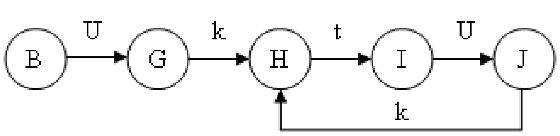
Step 4: Add the successor of state H: state I. The collection of contexts associated with {B, G, H, I} is: ({k}, 1), ({d}, 2), ({a, c, w}, 3).



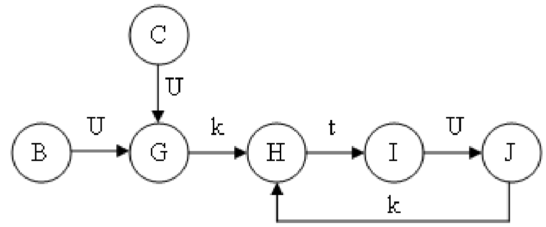
Step 5: Add the successor of state I: state J. The context sets associated with {B, G, H, I, J} is: ({k}, 1), ({d, u}, 2), ({a, c, w}, 3).



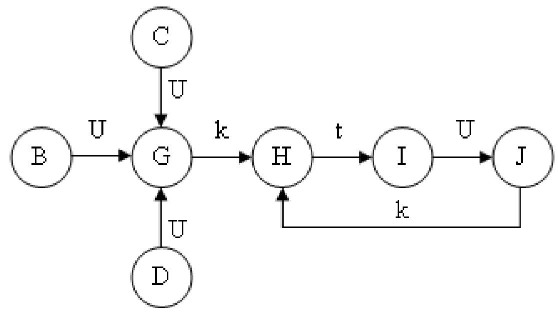
Step 6: Add the successor of state J: state H. State H is already in the set. The context sets associated with {B, G, H, I, J} is: ({k}, 1), ({d, u}, 2), ({a, c, w}, 3).



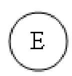
Step 7: State H is already in this set of states. So find the next lanehead state after B in the lane table and see if it is possible to add it to this set of states, which is state C. The context sets associated with {B, C, G, H, I, J} is: ({k}, 1), ({d, u}, 2), ({a, b, c, w}, 3).



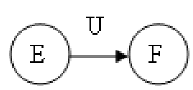
Step 8: Successor state G of state C is in this set of states already. So find the next lanehead state after C in the lane table and see if it is possible to add it to this set of states, which is state D. The context sets associated with {B, C, D, G, H, I, J} is: ({k}, 1), ({d, u}, 2), ({a, b, c, r,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



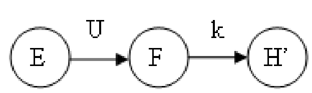
Step 9: Successor state G of state D is in this set of states already. So find the next state after D in the table and see if it is possible to add it to this set of states, which is state E. But adding E to this set will cause a conflict in the associated context sets, which becomes evident in step 11 below. So E must be put into a new set of states. The collection of contexts associated with {E} is: ({k}, 1).



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

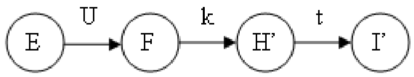


Step 11: Add the successor of state F: state H. Now state H is already in the first set of states. So adding H to the current set of states means we need to combine the new set of states with the old one. But then the combined contexts is: ({k}, 1), ({d, r, u}, 2), ({a, b, c, r, w}, 3). This is not pair-wise disjoint because the terminal symbol "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'} is: ({k}, 1), ({r}, 2), ({a}, 3).

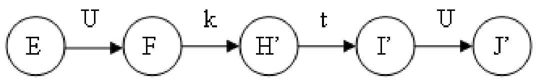


Step 12: Add the successor of state H', which is I. Similarly, we have to create a copy of state I to insert into the new set to avoid merging with the old set, which causes the failure of pair-wise

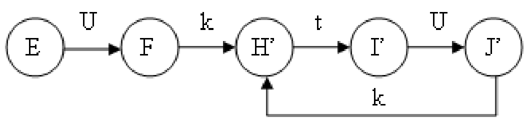
disjointness of the context sets. The collection of contexts associated with {E, F, H’, I’} is: ({k}, 1), ({r}, 2), ({a}, 3).



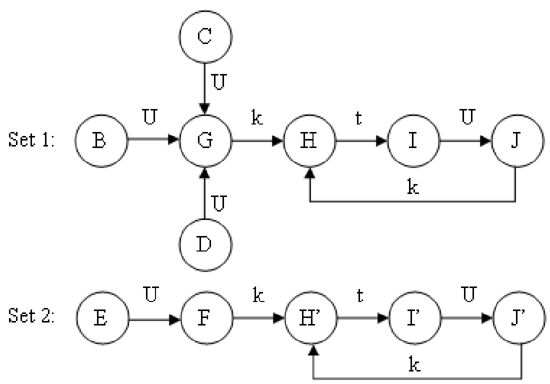
Step 13: Add the successor of state I’, which is J. For the same reason, we need to create a copy of J. The collection of contexts associated with {E, F, H’, I’, J’} is: ({k}, 1), ({r, u}, 2), ({a}, 3).



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’} is: ({k}, 1), ({r, u}, 2) ({a}, 3).



So finally the result of combining the regenerated states is shown below. The associated context sets is ({k}, 1), ({d, u}, 2), ({a, b, c, w}, 3) for set 1, and ({k}, 1), ({r, u}, 2), ({a}, 3) for set 2.



The portion of the parsing machine involved in lane-tracing given previously has now been transformed into Figure 8.

Example 4. Back to given grammar G1, we have seen how its reduce/reduce conflict can be resolved using the PGM method in phase 2. Let us apply the lane-table method of phase 2 to its LALR(1) parsing machine. The obtained conflicting lanes are shown in Figure 9.

The lane table is shown in Table 2. Follow the same procedure as in Example 3, we can obtain two sets for the finally LR(1) parsing machine of grammar G1, as shown in Figure 10. The associated context sets is ({d}, 1), ({c}, 2) for set 1, and ({c}, 1), ({d}, 2) for set 2. This is identical to the result of Phase2_PGM()method (as shown in Figure 5).

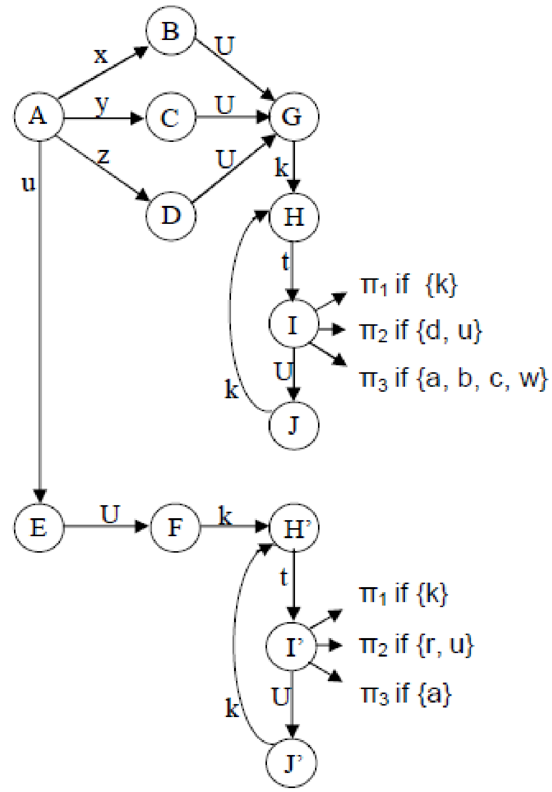


Figure 8. Portion of LR(1) parsing machine involved in lane-tracing of grammar G2

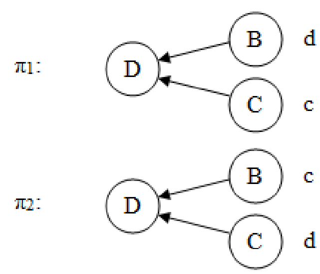


Figure 9. Conflicting lanes from lane-tracing of grammar G1

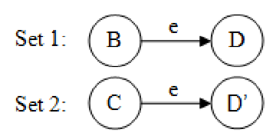


Figure 10. LR(1) parsing machine obtained by lane table method of grammar G1

Table 2. Lane table constructed in lane-tracing of Figure 9

State	π_1	π_2	Connected to
B *	d	c	{D}
C *	c	d	{D}
D			

3. IMPLEMENTATION IN HYACC AND EXTENSION TO LR(K)

We have implemented the lane table based lane-tracing algorithm into LR(1) parser generator Hyacc. It proves correct and efficient. It is also used to implement LR(k) [11], as shown in Figure 11. The acronyms used in Figure 11 are defined in section 4.

This algorithm is advantageous to extend to LR(k) because it only works on those configurations and states relevant to resolving reduce/reduce conflicts. The practical general method, however, needs to handle the entire context tuple for all the configurations and states, and thus infeasible for increasing k.

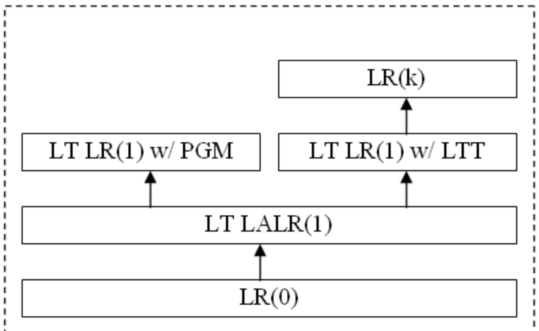


Figure 11. The LR(k) implementation stack In Hyacc

4. PERFORMANCE STUDY

In this section we study the performance of the lane-table based lane-tracing algorithm by comparing it with other relevant algorithms. The test machine has a 1.7 GHz Intel Pentium CPU and 1 GB RAM, and OS is Fedora core 4.0. For the measurements, time is in *sec* (second) and memory is in *MB* (megabyte). The grammars of 13 programming languages are used as input. These grammars are from [16] and are modified slightly to fit in input format.

The 5 algorithms compared are: 1) Knuth LR(1): Knuth canonical LR(1) algorithm. 2) PGM LR(1): LR(1) algorithm based on the practical general method. 3) LT LR(1) w/ PGM: LR(1) algorithm based on lane-tracing, use PGM in phase 2. 4) LT LR(1) w/ LTT: LR(1) algorithm based on lane-tracing, use lane table in phase 2. 5) Bison LALR(1): LALR(1) algorithm as in Bison 2.3 [3]. The 3 algorithms in 2), 3) and 4) are called *reduced-space LR(1) parser generation algorithms*, as opposed to the original Knuth canonical LR(1) algorithm in 1). Except LT LR(1) w/ LTT, the data of the other 4 algorithms were shown previously [10].

4.1 Parsing Table Size Comparison

Table 3 shows comparison of the size of the generated parsing tables. Figure 12 visualizes the data. The parsing machine generated by LT LR(1) w/ LTT has the same size as by LT LR(1) w/ PGM in most cases. LT LR(1) w/ PGM always results in the smallest LR(1) parsing machine. LALR(1) parser machine size is similar. Knuth canonical LR(1) parsing machine is much bigger.

4.2 Running Time Comparison

Table 4 shows the running time comparison. Figure 13 visualizes the data. LT LR(1) w/ LTT takes slightly longer time than LT LR(1) w/ PGM. Both are faster than PGM LR(1) in most cases. As expected, the running time of the three reduced-space LR(1) algorithms are on the same level as Bison LALR(1), and much faster than Knuth LR(1).

4.3 Memory Usage Comparison

Table 5 shows the memory usage comparison. Figure 14 visualizes the data. LT LR(1) w/ LTT always uses equal or less memory than LT LR(1) w/ PGM. PGM LR(1) memory usage can be lower or higher than the two lane-tracing methods. The three reduced-space LR(1) algorithms often use slightly more memory than LALR(1), and use much less memory than Knuth LR(1).

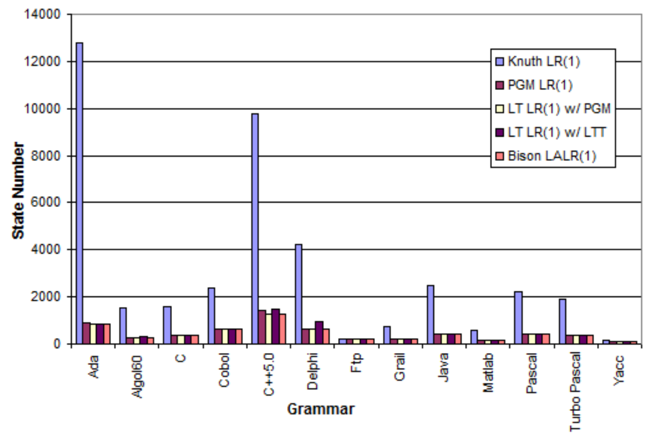


Figure 12. Parsing table size comparison

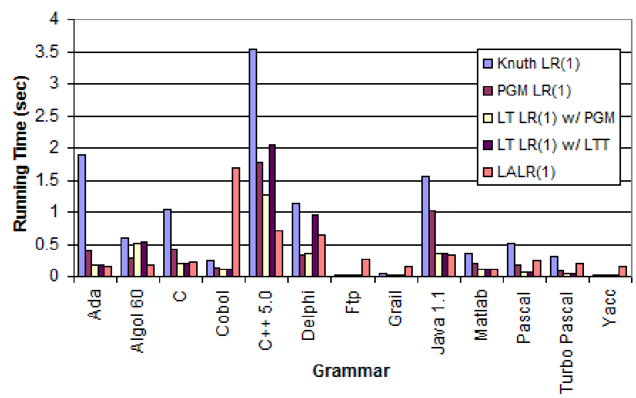


Figure 13. Running time comparison

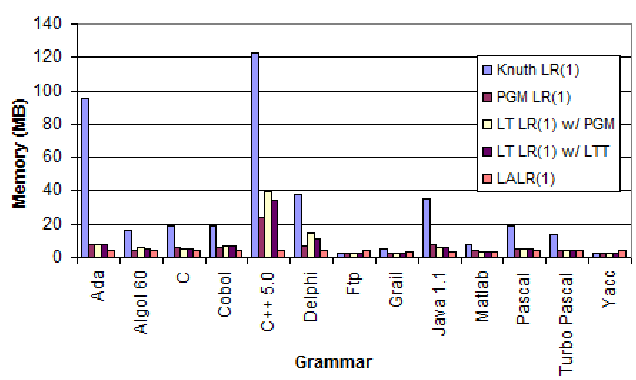


Figure 14. Memory usage comparison

Table 3. Parsing table size comparison

	Hyacc				Bison
Grammar	Knuth LR(1)	PGM LR(1)	LT R(1) w/ PGM	LT LR(1) w/ LTT	LALR(1)
Ada	12786	873	860	860	861
Algol 60	1538	274	272	294	273
C	1572	349	349	349	350
Cobol	2398	657	657	657	658
C++ 5.0	9785	1404	1261	1496	1257
Delphi	4215	609	609	945	610
Ftp	210	200	200	200	201
Grail	719	193	193	193	194
Java 1.1	2479	439	428	428	429
Matlab	588	174	174	174	175
Pascal	2245	418	412	412	413
Turbo Pascal	1918	394	386	386	387
Yacc	153	128	128	128	129

Table 4. Running time comparison (sec)

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

Table 5. Memory usage comparison (MB)

	Hyacc				Bison
Grammar	Knuth LR(1)	PGM LR(1)	LT LR(1) w/ PGM	LT LR(1) w/ LTT	LALR(1)
Ada	95.1	7.9	7.9	7.9	4.0
Algol 60	16.0	4.2	6.4	5.4	3.9
C	18.9	6.0	5.2	5.2	4.0
Cobol	19.1	6.3	6.5	6.5	4.0
C++ 5.0	122.7	23.9	39.1	34.4	4.3
Delphi	37.4	6.5	14.5	11	3.9
Ftp	2.8	2.8	2.8	2.8	3.9
Grail	5.3	2.9	2.9	2.9	3.8
Java 1.1	35.6	7.8	6.3	6.3	3.8
Matlab	7.8	3.9	3.5	3.5	3.8
Pascal	18.6	4.9	4.8	4.8	3.9
Turbo Pascal	13.8	4.3	4.5	4.5	3.9
Yacc	2.6	2.6	2.6	2.6	3.9

As a summary, compared to the other two reduced space LR(1) algorithms, in a small number of cases LT LR(1) w/ LTT may generate a parsing machine slightly larger and use a little more time, but it generally uses less memory. Overall it is another efficient reduced-space LR(1) parser generation algorithm.

5. CONCLUSION

In this paper we have presented the lane table method to construct LR(1) parsers, which is an alternative lane-tracing algorithm. We described the details of the algorithm and showed examples to demonstrate its application. Empirical study shows that the algorithm is another efficient reduced-space LR(1) parser generation algorithm. The lane table method is also more suitable when extend from LR(1) to LR(k) parser generation. It has been implemented into parser generator Hyacc for LR(1) and LR(k).

6. REFERENCES

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

[2] Frank L. DeRemer. Practical translators for LR(k) languages. Ph.D. thesis, MIT, Cambridge, 1969.

[3] 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.

[4] 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.

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

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

[7] David Spector. Full LR(1) parser generation. *ACM SIGPLAN Notices*, 58-66, 1981

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

[9] Xin Chen. Open Source Project: LR(1) Parser Generator Hyacc. 2008. Available: http://sourceforge.net/projects/hyacc

[10] Xin Chen and David Pager. Full LR(1) parser generator Hyacc and study on the performance of LR(1) algorithms. In *Proceedings of The Fourth International C* Conference on Computer Science and Software Engineering* (C3S2E '11), 83-92. Montreal, QC, Canada, May 2011. ACM Press.

[11] Xin Chen, David Pager. The Edge-Pushing LR(k) Algorithm. In *Proceedings of International Conference on Software Engineering Research and Practice*, pages 490-495. Las Vegas, USA, July 2011.

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

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

[14] Francois Pottier and Yann Regis-Gianas. Parser Generator Menhir. (2004) Available: http://cristal.inria.fr/~fpottier/menhir/

[15] Boris Burshteyn. MUSKOX Algorithms. (1994) Available: http://compilers.iecc.com/comparch/article/94-3-067

[16] "Yacc-keable" Grammars. Available: http://www.angelfire.com/ar/CompiladoresUCSE/COMPILERS.html