The Edge-Pushing LR(k) Algorithm

X. Chen[1], **D. Pager**[1]
[1] Department of Information and Computer Science, University of Hawaii at Manoa, Honolulu, HI, USA

Abstract - *The original LR(k) parser generation algorithm of Knuth in 1965 is very expensive in time and space. Different approaches have been proposed to achieve LR(k) with better practical performance. This work designed a new LR(k) algorithm called the edge-pushing algorithm, which is based on recursively applying the lane-tracing process. The algorithm has been implemented into the parser generator Hyacc. Here we first present the background and related work on LR(k) parser generation, next we introduce the edge-pushing algorithm's design and implementation, its LR(k) parse engine and corresponding storage parsing table. Relevant issues are discussed. Finally we give some applicable LR(k) grammar examples.*

Keywords: LR(k), Parser Generation, Edge-Pushing, Lane-Tracing, Algorithm

1 Introduction

The canonical LR(k) algorithm proposed by Knuth in 1965 [1] is a very powerful parser generation algorithm for context-free languages. However it is too expensive in time and space costs to be practical. Subsequent research on reduced-space LR(1) algorithms, which can reduce the state space and thus improve the performance of canonical LR(1) parser generation, were made by researchers such as Pager [2][3][4] and Spector [5][6]. LR(k) parser generation is in general much 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. It would be of value to study practical LR(k) parser generation algorithms. Here we show the design and implementation of a LR(k) algorithm called the edge-pushing algorithm, which is based on the lane-tracing LR(1) algorithm. The edge-pushing algorithm has been implemented into the parser generator Hyacc [7][8].

In this discussion, 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 $\dashv$ stands for the end of an input stream, and Ø stands for the empty set. The concepts of *state*, *configuration* and *theads*(α, k) used here are equivalent to "item set", "item" and $\text{FIRST}_k(\alpha)$ correspondingly in some other literature.

2 Background And Related Work On LR(k) Parser Generation

The 1965 paper of Knuth [1] introduced LR(k) parser generation. After that, a lot of works were done to reduce performance cost.

The work of Pager in the 1970s [2][3][4] were about reduced-space LR(k) parser generation. Pager reported LR(k) analysis on the grammars of real languages such as ALGOL for LR(2) and LR(3) [3]. M. Ancona et. al. published their work on LR(k) parser generation from 1980s to 1990s [9][10][11][12][13]. They proposed a method [13] in which non-terminals are not expanded to terminals in contexts, and expansion is not done until absolutely needed to resolve inadequacy. This defers the calculation of $\text{FIRST}_k(\alpha)$ until absolutely necessary. They claim savings in both time and storage space by deploying this method when trying 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" [14] 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). Parr'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. Parr's PhD thesis proposed that adding semantic actions to a LR(k) grammar degrades its recognition power to that of a LL(k) grammar. Based on this proposition he worked on LL(k) parser generation only. Josef Grosch worked on a LR(1)/LR(k) parser generator in 1995 [15]. In case of LR(1), it was practical only for small to medium size grammars, and LR(k) is certainly more expensive. Bob Buckley worked on a LR(1) parser generator called Gofer in 1995 [16]. 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 a LR(k) implementation [17]. There was no more news so far. Chris Clark worked on a LALR(k)/LR(1)/LR(k) parser generator

Yacc++ [18][19]. Its LR(1)/LR(k) implementation was loosely based on Spector's paper [5][6]. But there was an infinite loop problem on LR(k). Thus the LR(k) feature of Yacc++ was only used internally and did not go public. Ralph Boland once worked on LR(k) [20], but report on his results was not found. Paul Mann mentioned that Ron Newman's Dr. Parse [21] works on LR(k) for k = 2 or maybe 3. The only claimed successful efficient LR(k) parser generator is the MSTA parser generator in the COCOM tool set [22]. The author Vladimir Makarov said it generates fast LALR(k) and LR(k) grammar parsers with "acceptable space requirements".

To conclude, practical result on LR(k) parser generation is scarce.

3 LR(k) Parser Generation Based On Edge-Pushing

This section discusses practical LR(k) parser generation based on the edge-pushing algorithm as implemented in Hyacc.

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 such as those by Pager can solve the first problem. The second problem can be solved following the way of Terence Parr [14], 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. The edge-pushing algorithm does this by recursively applying the lane-tracing procedure.

Three problems need to be solved when extending lane-tracing to LR(k) in a practical way: 1) LR(k) algorithm. This part should extend the lane-tracing LR(1) parser generation algorithm, such that it can be recursively applied to states 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 together with the LR(k) algorithm in 1). 3) LR(k) parse engine. After the LR(k) parsing table is generated, the Hyacc LR(1) parse engine should be extended so it can use the LR(k) parsing table for parsing LR(k) grammars.

3.1 The Edge-Pushing LR(k) algorithm

For the ease of discussion, we define the terms and functions below. A *final configuration* is one where the marker (the dot) is at the right most position on the right hand side, e.g., $E \rightarrow E + T$ ▪. A *head configuration* is a configuration 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(α, k)* to return a set of terminal strings obtained from α. The length of these strings is k, and this function is potentially exponential on k. *theads(α, k)* is the same as $\mathrm{FIRST}_k(\alpha)$ in many other literature.

3.1.1 LR(k) parser generation based on recursive lane-tracing

Each time after LR(k) lane-tracing for a certain k, we need to check if conflicts are resolved, and further trace LR(k+1) only for states with unresolved reduce/reduce conflicts. In order to resolve conflicts, we may need to go back multiple levels. 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 for resolving conflicts: 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 for more efficient computation of LR(k+1) theads.

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. It is also possible to do LR(k) lane-tracing recursively on one inadequate state, increase k on this state only until its inadequacy is resolved or found not resolvable, then start on another inadequate state. For these two methods, the second may be easier from a practical point of view, since if we want to take advantage of the result of LR(k) for LR(k+1), the second method allows us to remember less intermediate information.

Below is an intuitive and straightforward solution for LR(k) lane-tracing. Algorithm 1 *Conflicts_Resolved(S, k)* checks if the reduce/reduce conflicts on a state S can be solved by LR(k) lane-tracing. Algorithm 2 *Intuitive_Lane_Tra-cing_LRk(S)* achieves LR(k) by calling Algorithm 1 and increasing the parameter k from 2 to 3, 4, … until *Conflicts_Resolved(S, k)* returns true. The problem of Algorithm 2 is that it always repeats the computation for each LR(k) context, even for those configurations that do not have associated conflict. E.g., suppose two reduction configurations 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 Algorithm 2 also computes the configurations that generate 'ac' and 'ad'. To avoid this problem and achieve more delicate control, we can improve by only computing the configurations that generate 'ab'. We call such a method "edge-pushing" and show it below in a conceptual example.

Algorithm 1: Conflicts_Resolved(S, k)

```
foreach final configuration C_f of state S do
    C_S ← head configuration of C_f;
    get LR(k) context of C_S;
if all reduce/reduce conflicts are resolved then
    add context obtained to parsing table;
    return true;
else
    return false;
```

Algorithm 2: Intuitive_Lane_Tracing_LRk(S)

```
k ← 2;
repeat
    resolved ← Conflicts_Resolved(S, k);
    k ← k + 1;
until resolved == true
```

3.1.2 Edge-pushing – a conceptual example

Below we will show a conceptual description of LR(k) lane-tracing with edge-pushing.

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) to resolve the conflict eventually. In each graph, the circles with bold 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. 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(β, k')* for a configuration $A \rightarrow \alpha \bullet B \beta$. $k' = k - z$, where k is the k in LR(k), and z is the local z. The figures below show a conceptual example of the calculation of these values, and how LR(k) lane-tracing is pushed at the cutting edge.

Fig. 1 shows the initial conflict state obtained by LR(1) lane-tracing:

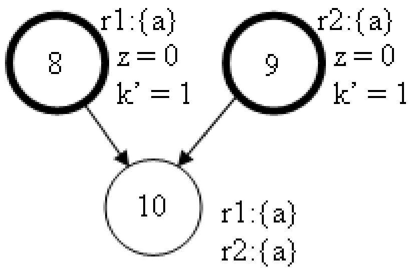

Fig. 1. LR(1) state with reduce/reduce conflict.

In Fig. 2, states 8 and 9 are at the cutting edge of lane-tracing. For state 8, on its right side "r1: {ab, ac}" 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 configuration*s 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.

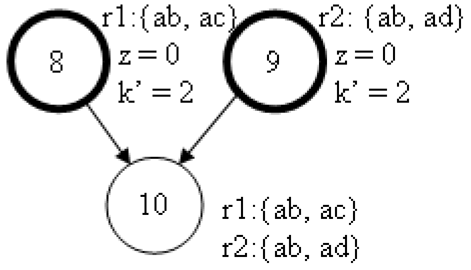

Fig. 2. LR(2) states.

In Fig. 3, 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.

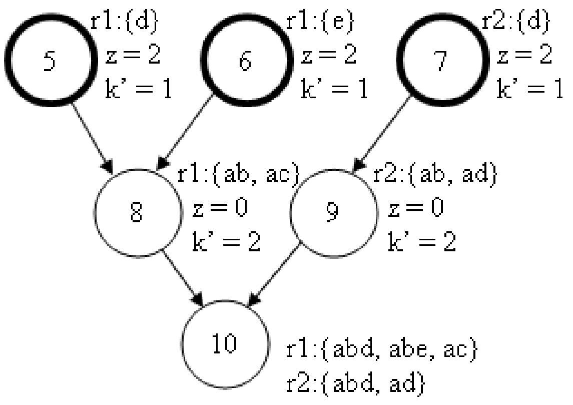

Fig. 3. LR(3) states.

See Fig. 4, in state 5 we get two consecutive context symbols "dd" by doing *theads(β, k'₅)* calculation where $k'_5 = k - z_5 = 4 – 2 = 2$. In state 7, $k'_7 = k - z_7 = 4 - 2 = 2$, *theads*(*β*, 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*(*β*, k'_4) to obtain context 'd'.

In Fig. 5, 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, $k'_4 = k - z_4 = 5 - 3 = 2$.

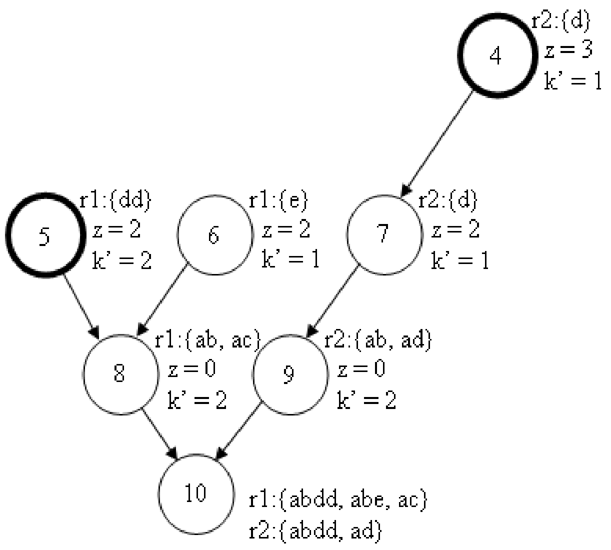

Fig. 4. LR(4) states.

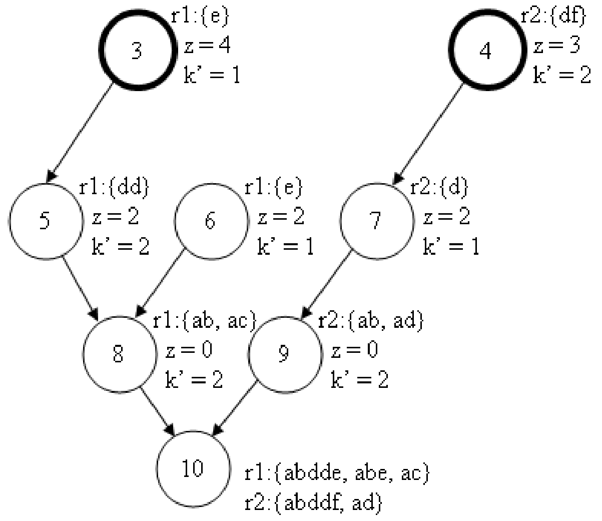

Fig. 5. LR(5) states.

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.

LR(k) parsing tables 1 to 5 are the corresponding LR(1) to LR(5) parsing tables. Symbol Θ means a reduce/reduce conflict. See section 3.3 for the exact notations on storage of LR(k) parsing tables. These tables are created when we do the LR(k) lane-tracing. Whenever a conflict is found: e.g., when inserting action r2 to a field we find an r1 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.

Table 1. LR(1) parsing table.

state / token	…	a	…
…			
10		Θ	
…			

Table 2. LR(2) parsing table.

(state, LR(1) lookahead) / token	b	c	d
(10, a)	Θ	r1	r2

Table 3. LR(3) parsing table.

(state, LR(2) lookahead) / token	d	e
(10, b)	Θ	r1

Table 4. LR(4) parsing table.

(state, LR(3) lookahead) / token	d
(10, d)	Θ

Table 5. LR(5) parsing table.

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

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 Θ 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 Θ 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.

3.1.3 The Edge-pushing algorithm

We summarize the skeleton of the edge-pushing algorithm below as Algorithm 3.

There are lots of details involved, but two major components are lane-tracing and calculation of *theads(α, k)*. The edge-pushing algorithm uses iteration and avoids recursion. Two configuration sets are used between iterations, such that during a round of iteration, we draw an element from the working set (Set_C), process it and add new derived configurations to the derived set (Set_C2); 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 stops when lane-tracing is back to state 0, line 16 "Σ ← lane_tracing(C)" will return an empty set. So eventually Set_C2, and thus Set_C, becomes an empty set.

Algorithm 3: Edge_Pushing(S)

```
Input: Inadequate state S

Set_C ← Ø; Set_C2 ← Ø
k ← 1;
foreach final configuration T of S do
   T.z ← 0;
   Let C be the head configuration of T,
     and X be the context generated by C;
   Add triplet (C, X, T) to set Set_C;
while Set_C ≠ Ø do
   k ← k + 1;
   foreach (C:A → α • B β, X, T) in Set_C do
     k′ ← k - C.z;
     calculate ψ ← theads(β, k′);
     foreach context string x in ψ do
       if x.length == k′ then
         Insert (S, X, last symbol of
           string x, C, T) to Set_C2
           and add to LR(k) parsing table;
       else if x.length == k′ - 1 then
         Σ ← lane_tracing(C);
         //Σ: set of head configurations
         foreach configuration σ in Σ do
           σ.z ← C.z + k′;
           Let m be the generated context
             symbol in σ;
           Insert(S, X, m, σ, T) to Set_C2
           and add to LR(k) parsing table;
   Set_C ← Set_C2;
   Set_C2 ← Ø;
```

At the beginning we initialize the variable *z* of relevant final configurations to 0, and 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 1 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 do not 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 do not cause conflict.

Due to the exponential nature of *theads(α, 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.

3.1.4 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. Fig. 6 (a)), or when cycles are involved in the tracing (e.g. Fig. 6 (b)).

Using Figure 6 (b) as an example, if LR(k) tracing ends at state B and cannot resolve the reduce/reduce conflict, LR(k+1) tracing ends at state C and cannot resolve the conflict, LR(k+2) tracing ends at state D and still cannot resolve the conflict, then LR(k+3) tracing will come back to state B. This forms an infinite cycle: B → C → D → B …

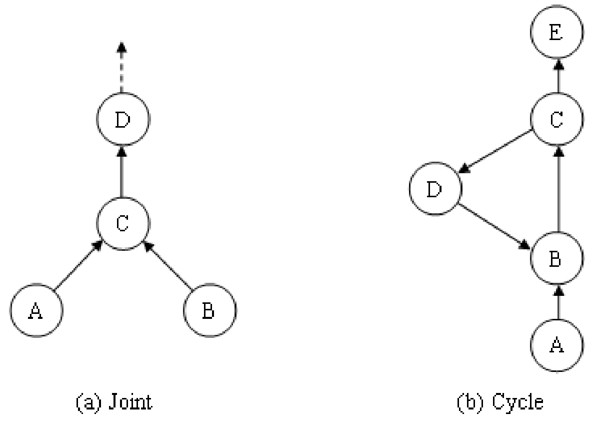

Fig. 6. LR(k) lane-tracing: joint and cycle conditions.

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 do not 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 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 cache this way. The only left issue is cycle detection: how to avoid tracing down the cycle infinitely. This will be among the future work.

3.2 Computation of theads(α, k)

The *theads*(*α*, *k*), i.e., FIRST$_k$(*α*) algorithm used in the edge-pushing algorithm is given by Pager [23]. Compared to the FIRST$_k$(*α*) algorithm of Aho and Ullman [24], which uses a bottom-up process, this algorithm takes a top-down approach.

3.3 Storage of LR(k) parsing table

In Hyacc, 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 goes on until the conflict is resolved.

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.

The Hyacc parser generator uses the hyaccpar file for LR(1) parse engine [7]. For the LR(k) extension, Hyacc uses another parse engine file hyaccpark, which is based on hyaccpar with extension. Table 6 shows the additional variable and arrays used to represent the LR(k) parsing tables besides those for LR(1) in hyaccpark.

In additional, the LR(1) part of the parsing table should have this modification: wherever a reduce/reduce conflict occurs, the previous default reduction number should be replaced by a special value (Θ) labeling the occurrence of a reduce/reduce conflict. In the hyaccpark parse engine's arrays, this would mean the update of corresponding values in arrays yyfs[] and yyptblact[].

Table 6. LR(k) parse engine arrays

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. For each parsing table, there may be multiple states, and each state may have multiple tokens.
yy_lrk_cols[]	Each row starts with two fields for each (state, token) pair, followed by one entry for each token.
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: i) -2 means reduce/reduce conflict, ii) a positive number means no conflict and is the corresponding production's ruleID, iii) -1 labels the end of each row.
yyPTC[]	The values of parsing table column tokens.

3.4 LR(k) parse engine

In Hyacc, the LR(k) parse engine is an extension to the LR(1) parse engine [7]. 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 Θ symbol, which leads to the following extension. The LR(k) parse engine extension is shown in Algorithm 4 below. The while loop eventually ends 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.

Algorithm 4: LR(k) parse engine extension.

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

if A == Θ then
    k = 2;
    while true do
        L_next ← next lookahead;
        if L_next == EOF then
            Report error and exit;
        else
            In LR(k) parsing table, find
              entry A_next ← ((S, L), L_next);
            if A_next == Θ then
                L ← L_next;
                k ← k + 1;
            else
                do reduce;
                break out of while loop;
```

3.5 Examples

The edge-pushing algorithm has been tried and works on the following grammars. These grammars are often used as examples in LR(k) discussion.

LR(k) grammar G1: S $\rightarrow$ a A D a | b A D b | a B D b | b B D a, A $\rightarrow$ a, B $\rightarrow$ a, D $\rightarrow$ c^n. Here k depends on the value of n: k = n + 1. c^n is the concatenation of n letters 'c'.

LR(3) grammar G2: S $\rightarrow$ a A D a | b A D b | a A D b | b C E, A $\rightarrow$ a, B $\rightarrow$ a, D $\rightarrow$ e d, C $\rightarrow$ B e, E $\rightarrow$ d a

LR(2) grammar G3: S $\rightarrow$ a A a | b A b | a B b | b B a, A $\rightarrow$ C a, B $\rightarrow$ D a, C $\rightarrow$ a, D $\rightarrow$ a

LR(3) grammar G4: S $\rightarrow$ a A a a | b A a b | a B a b | b B a a, A $\rightarrow$ C a, B $\rightarrow$ D a, C $\rightarrow$ a, D $\rightarrow$ a

4 Conclusions

We have presented a new LR(k) parser generation algorithm called the edge-pushing algorithm, which is based on the lane-tracing process, recursively traces back on relevant configurations to obtain more contexts to resolve reduce/reduce conflicts. The edge-pushing algorithm, its corresponding LR(k) storage arrays and parse engine have all been designed and implemented into the open source parser generator Hyacc. The edge-pushing algorithm so far works on LR(k) grammars where lane-tracing upon increasing k does not form a cycle.

5 References

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

[2] 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.

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

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

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

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

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

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

[9] Massimo Ancona, Alessandro Paone. Table merging by compatible partitions for LR parsers is NP-complete. Elektronische Informationsverarbeitung und Kybernetik, 30(3):123 – 134, 1994.

[10] Massimo Ancona, Vittoria Gianuzzi. A new method for implementing LR(k) tables. Inf. Process. Lett., 13(4/5):171 – 176, 1981.

[11] Massimo Ancona, Claudia Fassino, Vittoria Gianuzzi. Optimization of LR(k) "Reduced Parsers". Inf. Process. Lett., 41(1):13 – 20, 1992.

[12] Massimo Ancona, Gabriella Dodero, Vittoria Gianuzzi. Building collections of LR(k) items with partial expansion of lookahead strings. SIGPLAN Notices, 17(5):24 – 28, 1982.

[13] Massimo Ancona, Gabriella Dodero, Vittoria Gianuzzi, M. Morgavi. Efficient construction of LR(k) states and tables. ACM Trans. Program. Lang. Syst., 13(1):15 – 178, 1991.

[14] 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.

[15] Josef Grosch, 1995.
http://compilers.iecc.com/comparch/article/95-04-179

[16] Bob Buckley, 1995.
http://compilers.iecc.com/comparch/article/95-05-087

[17] Karsten Nyblad, 2005.
http://compilers.iecc.com/comparch/article/05-03-117

[18] Chris Clark. Yacc++ historical notes, 2005.
http://compilers.iecc.com/comparch/article/05-06-124

[19] Chris Clark. More Yacc++ historical notes, 2000.
http://compilers.iecc.com/comparch/article/00-02-142

[20] Ralph Boland, 1997.
http://compilers.iecc.com/comparch/article/97-11-011

[21] Parser Generator Dr. Parse.
http://www.downloadatoz.com/software-development_directory/dr-parse

[22] Vladimir Makarov. Toolset COCOM & scripting language DINO, 2002.
http://sourceforge.net/projects/cocom.

[23] David Pager. Evaluating Terminal Heads Of Length K. Technical Report No. ICS2009-06-03, University of Hawaii, Information and Computer Sciences Department, 2008. http://www.ics.hawaii.edu/research/tech-reports/terminals.pdf/view.

[24] Alfred V. Aho, Jeffrey D. Ullman. The Theory of Parsing, Translation and Compiling, Vol. 1, Parsing. Prentice-Hall, Englewood Cliffs, N.J, 1972.