An Extension Of The Unit Production Elimination Algorithm

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

Abstract - *Removing unit productions from LR parsing machines can reduce the space and time cost of parsing. The unit production elimination algorithm of Pager may result in redundant states in the generated parsing machine. This work introduces an extension to remove the redundancy and thus minimize the parsing machine. We implemented the unit production elimination algorithm and its extension algorithm into the Hyacc parser generator. We study its performance and discuss relevant issues here. Theoretical analysis and experiement result show that when the extension is used, the parser generation process uses the same amound of memory, but more processing time. The resulted parsing machine can be much more compact.*

Keywords: unit production elimination, algorithm, extension, LR

1 Introduction

1.1 Overview

A unit production in a grammar is a production of the form x → y, where symbol x is a non-terminal, and symbol y is a terminal or non-terminal. The existence of unit productions in a LR parsing machine can increase parsing table size and waste significant amount of parsing space and time [1]. Eliminating unit productions is among the most attractive approaches to optimize LR parsers.

Pager's unit production elimination algorithm is among the most discussed. It is found that the unit production elimination algorithm of Pager, when applied, can possibly lead to redundant states. This work extended the unit production elimination algorithm of Pager [5] by further eliminating redundant states and thus minimizing the parsing machine. The extension algorithm was implemented in the parser generator Hyacc [18][19][20]. Here we present the algorithm, analyze its complexity, conduct empirical study on its performance and discuss relevant issues.

We use these acronyms for the algorithms involved in this discussion: PGM (Pager's practical general method) [6], UPE (Pager's unit production elimination algorithm) [5], UPExt (Extension algorithm to Pager's unit production elimination algorithm). In addition, LHS stands for Left Hand Side, and RHS stands for Right Hand Side. The discussion will be based on LR(1), but should in general apply to LR(k).

1.2 Related work

There have been various studies to eliminate unit productions. Anderson, Eve and Horning [1] presented a unit production elimination method, but the method can increase the number of states in the parsing machine significantly. Joliat [8] gave suggestions to simplify the method of Anderson, Eve and Horning. Tokuda [15] presented a method on bypassed LR(k) parsers, which can naturally derive the algorithm of Anderson, Eve and Horning. The methods of Aho and Ullman [2] and Demers [9] can avoid increasing the number of states in the parsing machine, but require restriction on the grammar that no two unit productions should have the same left hand side. Pager [3][4][5] described an algorithm that can avoid the above problems. Backhouse [10] and Lalonde [7] developed variations of Pager's method. Koskimies [11][12] discussed that Pager's method cannot be used during the construction process of a SLR parser and need to be used on a fully constructed SLR parser. Soisalon-Soininen [13] discussed applying Pager's technique only when it does not affect the use of default reductions. Soisalon-Soininen [14] described that Pager's method can possibly cause increase in the size of the parsing machine and presented a fix. Heilbrunner [16] and Schmitz [17] discussed practical conditions needed to correctly eliminate unit productions.

2 Pager's unit production elimination algorithm

Pager's unit production elimination algorithm [3] is applied to a LR parsing machine to further reduce the number of states to achieve a more compact LR parsing machine.

A unit production is a production x → y where both x and y are single symbols. A *leaf* is a symbol that only appears on the RHS of any unit production but never on the LHS of any unit production. The algorithm [3] takes five steps: "1) For each state S in the parsing machine (including new states added in step 2), and for each leaf x where the x-successor of S contains a unit reduction, do step 2. Go to step 3 after finish. 2) For $1 \le i \le n$, let x_i be the symbols such that $x_i \equiv> x$ (including x itself), and for which shift/goto actions are

defined at state S. Let the x_i-successor of S be T_i. If any state R is or at a earlier time has been a combination of states T_1, …, T_n, then let R be the new x-successor of state S; otherwise combine states T_1, …, T_n into a new state T and make T the new x-successor of S. 3) Delete all the transitions where the transition symbol is on the LHS of a unit production. 4) Delete all states that now cannot be reached from state 0. 5) Replace all such reductions $y \rightarrow w$ by $x \rightarrow w$, where y is the LHS symbol of a unit production, and x is a randomly selected leaf such that $y \equiv> x$."

Example 1. Given grammar G1: E → E + T | T, T → T * a | a. The LR(1) parsing machine of grammar G1 is shown in Fig. 1. We have two unit productions that are the candidates of elimination: E → T and T → a.

An example of applying the unit production elimination algorithm on the LR(1) parsing machine of grammar G1 is shown in Fig. 2. First we need to find the leaves of the grammar. This is achieved by constructing a multi-rooted tree, which is E → T → a for G1. In this case a is the only leaf. Then following the algorithm step 1, we see that only states 0 and 4 have a-successor that has a unit production: state 0's a-successor state 3 has a unit production T → a, state 4's a-successor is also state 3. Thus we follow step 2 to combine successor states of state 0 and state 4. These are shown in (b) and (c) of Fig. 2. Next, (d) follows step 3, (e) follows step 4, (f) follows step 5 and also rearranges the states in a better-looking layout.

3 Extension to the unit production elimination algorithm

3.1 The extension algorithm

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

Definition 1. *Equivalent states* are those states in a parsing machine that have exactly the same actions (*accept*, *shift*, *goto* and *reduce*) on each token symbol (including both terminals and non-terminals).

Algorithm 1 (UPExt) is shown in the next page. It removes redundant *equivalent states* from the parsing machine obtained from Pager's unit production elimination algorithm.

One concern of the unit production elimination algorithm is that it was designed for LR(k) grammars. For non-LR(k) grammars, more conflict complications can be derived. Under such situations, the unit production elimination algorithm and this extension should not be used. Another concern is for unit productions with semantic actions, these should not be removed so as to retain the associated semantic actions.

3.2 Complexity analysis

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

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

Assume the action of accessing one action of one state is O(1). Derivation of upper bound $O(n^2 * m)$ using the best scenario: the step of finding the set of all the *equivalent states* can be done in linear time by inserting all states into a hash table based on its actions. Since there are n states, and assume each state has m actions in average, this is $O(n * m)$. The next step replaces relevant transitions. Assume those *equivalent states* are $S_1, S_2, \ldots, S_k$ ($k \leq n$). Let the number of actions transiting into S_i be X_i ($i = 1, \ldots, k$). In the worst case all the n other states transiting to S_i and all the m actions of each state transit to S_i (although this is unlikely in practice), so $0 \leq X_i \leq n * m$. The total number of transitions to replace is $0 \leq X_1 + \ldots + X_k \leq n * (n * m)$. Thus $O(n * m + n * (n * m)) = O(n^2 * m)$.

For the lower bound $O(n * m)$, just notice that the first step of finding the set of all the *equivalent states* always takes $O(n * m)$, and the second step of replacement in the best case takes no time when no *equivalent states* are found.

3.3 Implementation in Hyacc

Pager's unit production elimination algorithm and the extension algorithm here are implemented into LR(1) parser generator Hyacc [18][19][20].

Implementation can be on the level of 1) the parsing machine automata, or 2) the parsing table. In Hyacc the UPE algorithm is implemented by manipulating the parsing table, so the extension algorithm UPExt is implemented based on this way. We think that manipulating the parsing table is easier. The alternative of working on the level of the parsing machine automata, however, may be more intuitive from a human point of view.

Hyacc uses reduced-space LR(1) parser generation algorithms, such as Pager's PGM algorithm. In general the user of Hyacc applies the UPE and UPExt algorithms on a parser generated from the PGM algorithm or other reduced LR(1) algorithms. In the example and empirical studies below we assume this scenario.

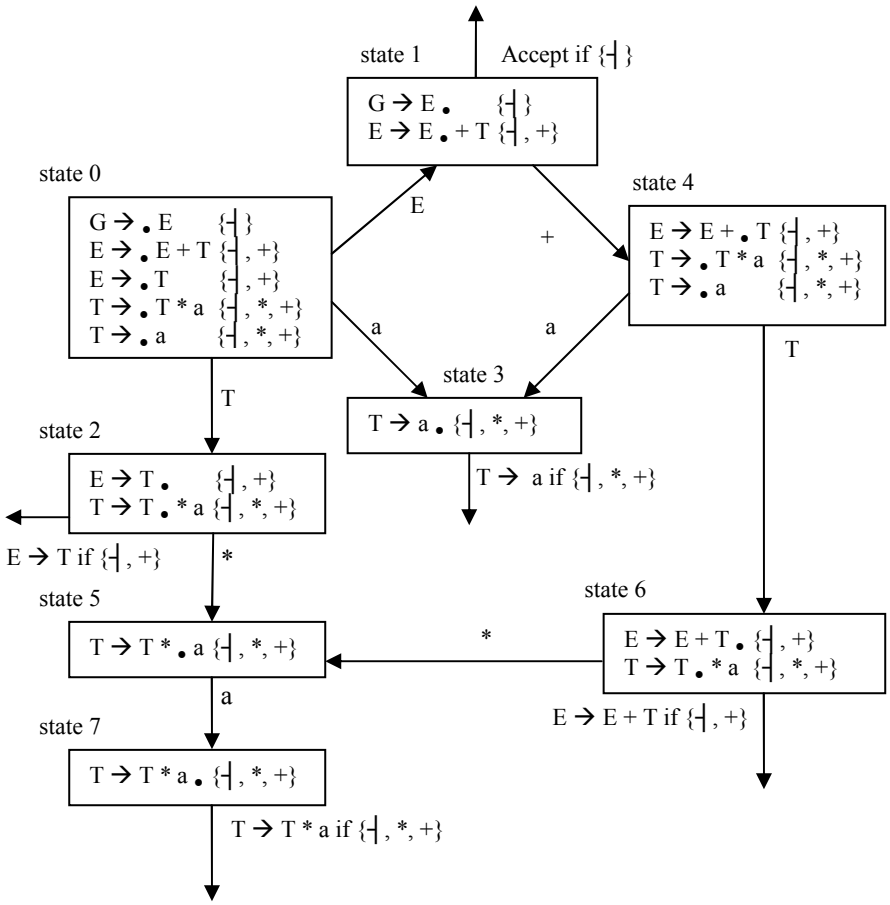

Fig. 1. Parsing machine of grammar G1

Algorithm 1 (UPExt):

```
Input: Parsing Machine M
Output: A parsing machine M' where all the
        equivalent states in M are removed;

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

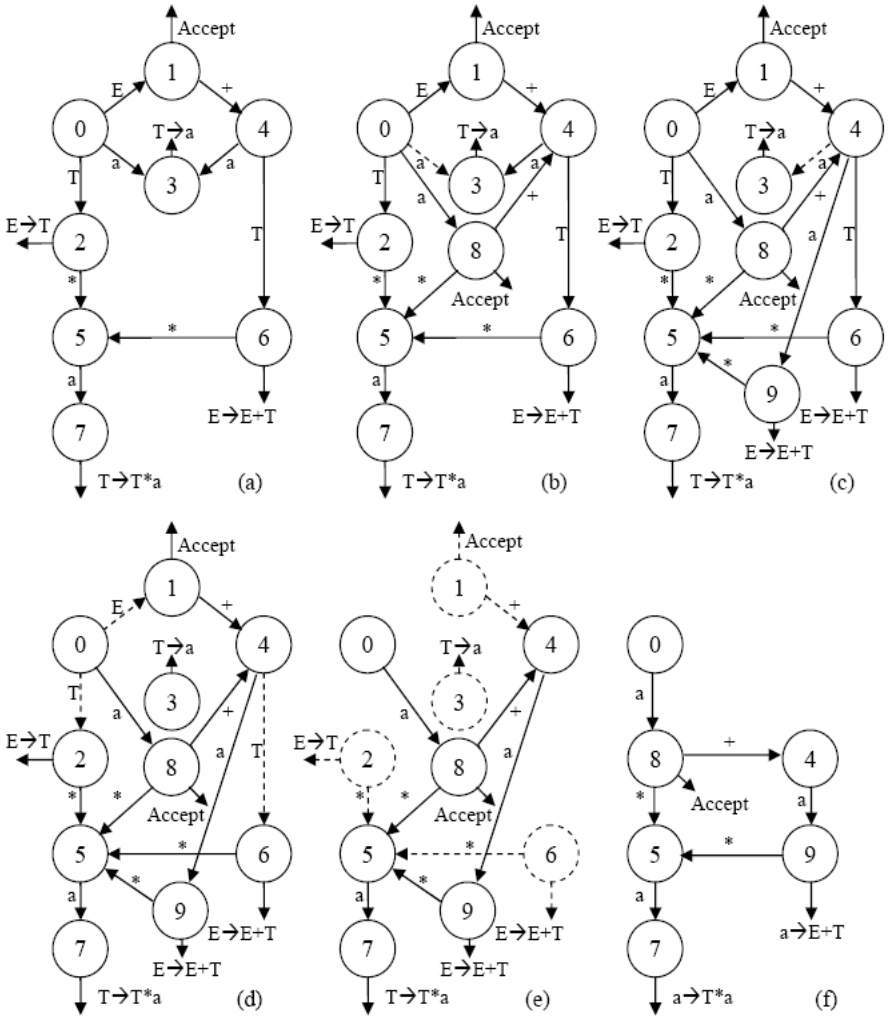

Fig. 2. Apply Unit Production Elimination on the LR(1) parsing machine of grammar G1

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

3.4 An example

Example 2. Given grammar *G2: S → d i A, A → A T | ϵ, T → M | Y | P | B, M → r | c, Y → x | f, P → n | o, B → a | e.* In Fig. 3, (a) is the parsing machine obtained using the practical general method, (b) is the parsing machine after applying the unit production elimination algorithm based on (a), (c) is the parsing machine after applying the extension to the unit production elimination algorithm based on (b). In (b), states 18 to 25 all have the same action A → A T for each of the lookahead symbols in ∑ = { a, c, e, f, n, o, r, x, ⊣ }. Thus states 18 to 25 are equivalent states and they can be combined into one state, i.e., state 18 in (c).

In this example, the parsing machine in (a) has 18 states, in (b) has 13 states, and in (c) has only 6 states. So by applying the extension algorithm after the unit production elimination, (13 - 6) / 13 = 54% reduction in parsing machine size is achieved. The following table compares the number of states, '*shift*/*goto*', '*reduce*' and '*accept*' actions in the parsing machine after applying each of the PGM, UPE and UPExt algorithms.

4 Measurements And Evaluations

Measurement data are collected on a Dell Inspiron 600M computer with 1.7GHz Intel Pentium CPU and 1 GB RAM. Operating system is Fedora core 4.0. In the measurements, unit of time is in sec (second), and memory is in MB (megabyte). Hyacc version 0.95 is used. In the empirical study, we measure the performance on three algorithms: PGM, UPE, and UPExt. This is because UPE is applied after PGM is applied, and UPExt is applied after UPE is applied. We would like to see the difference of parsing table size, time and memory costs after applying the UPE and UPExt algorithms. The grammars of 13 real programming languages [21] are used for the study.

4.1 Parsing table size comparison

Table 2 shows the parsing table size comparison. Fig. 4 is the graphic view.

UPE may decrease the number of states as in the case of many simple grammars. but in 12 out of the 13 real programming languages here, UPE actually increases it. Applying the UPExt algorithm decreases the parsing machine size significantly: although in 10 out of the 13 real language grammars the number of states are still bigger than that of PGM, they are bigger only by a small margin. Therefore it is desirable to apply the extension algorithm. In addition, the number of rules in the parsing machine is also reduced, since unit productions are removed.

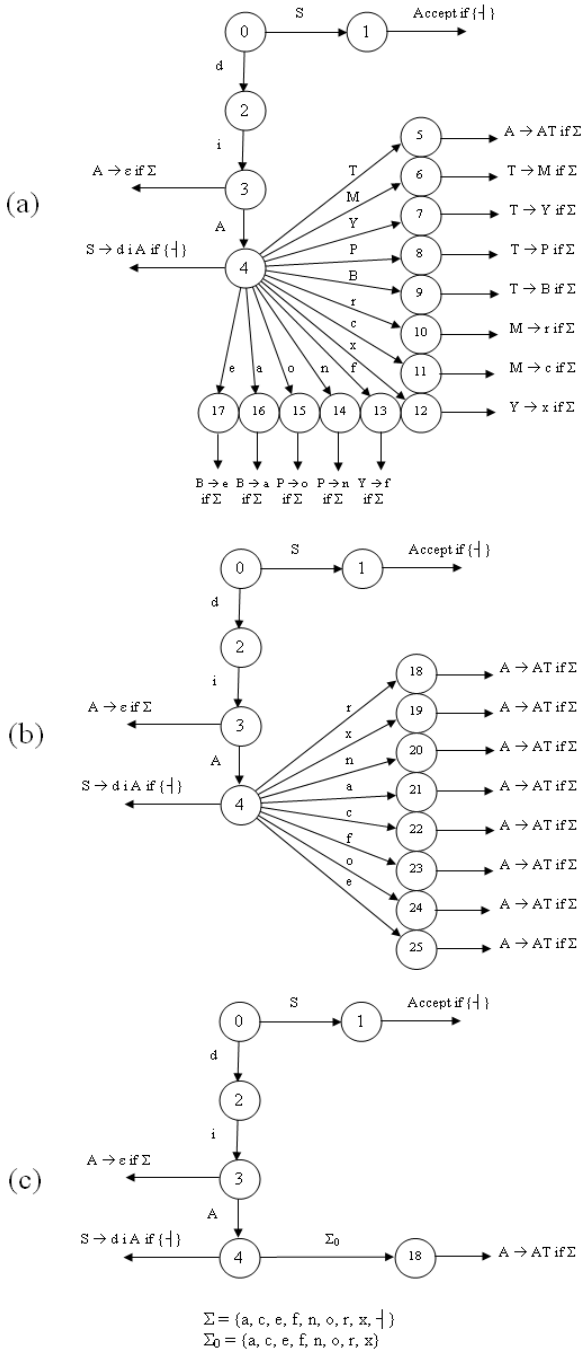

Fig. 3. Remove *equivalent states* after unit production elimination

Table 1. Parsing machine comparison after applying PGM, UPE and UPExt algorithms

	State #	shift/goto	reduce	accept
PGM	18	17	15	1
UPE	13	12	10	1
UPExt	6	5	3	1

Table 2. Parsing table size comparison.

	PGM		UPE		UPExt	
Grammar	State #	Rule #	State #	Rule #	State #	Rule #
Ada	873	459	1074	262	805	262
Algol 60	274	169	498	92	412	92
C	349	212	786	116	380	116
Cobol	657	401	646	268	528	268
C++ 5.0	1404	665	3573	443	2255	443
Delphi	609	358	1195	200	669	200
Ftp	200	74	211	71	211	71
Grail	193	74	247	54	204	54
Java 1.1	439	266	1174	142	673	142
Matlab	174	93	374	53	178	53
Pascal	418	257	844	119	427	119
Turbo Pascal	394	222	649	116	353	116
Yacc	128	103	134	87	134	87

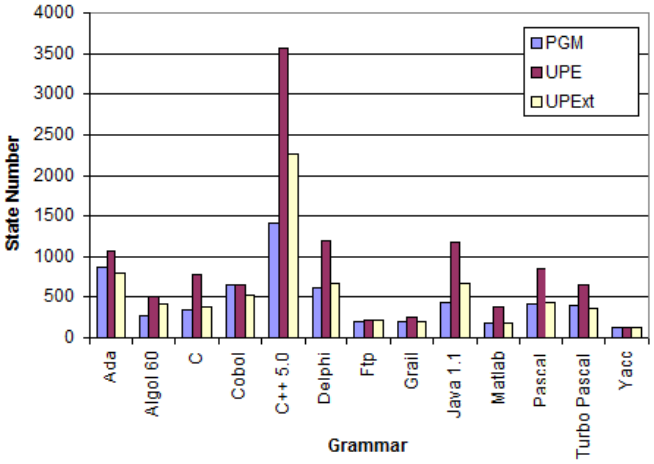

Fig. 4. Parsing table size comparison.

4.2 Running time comparison

Table 3 shows the running time comparison, and Fig. 5 is the graphic view. Compared to PGM, UPE and UPExt use longer running time, sometimes significantly longer, especially for the UPExt algorithm. This is as expected.

Table 3. Running time comparison

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

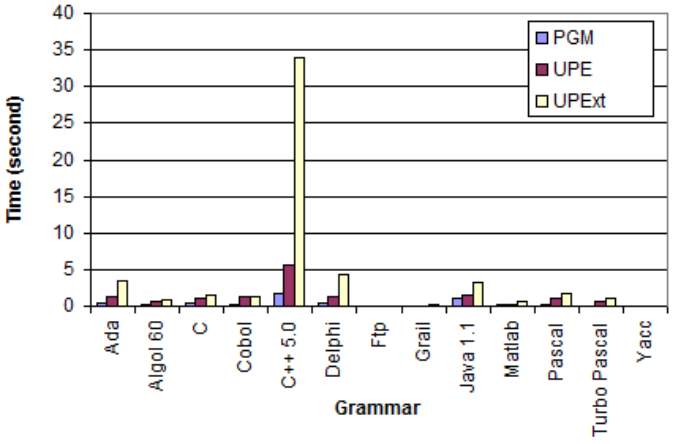

Fig. 5. Running time comparison.

4.3 Memory usage comparison

Table 4 shows the memory usage comparison, and Fig. 6 is the graphic view. When using UPE and UPExt, there is a slight increase in memory. It can also be seen that UPE and UPExt use the same amount of memory, because UPExt only works on the existing parsing table.

Table 4. Memory usage comparison

Grammar	PGM	UPE	UPExt
Ada	7.9	9.4	9.3
Algol 60	4.2	4.2	4.2
C	6.0	6.0	6.0
Cobol	6.3	6.4	6.4
C++ 5.0	23.9	30.9	30.9
Delphi	6.5	7.5	7.5
Ftp	2.8	2.9	2.9
Grail	2.9	2.9	2.9
Java 1.1	7.8	8.6	8.6
Matlab	3.9	3.9	3.9
Pascal	4.9	5.7	5.7
Turbo Pascal	4.3	4.3	4.3
Yacc	2.6	2.7	2.7

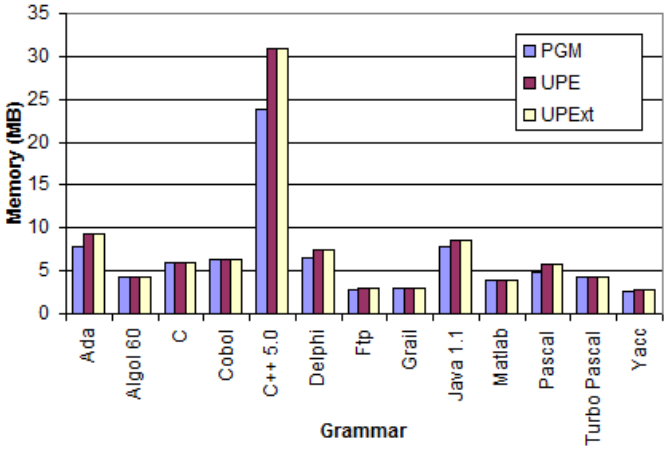

Fig. 6. Memory usage comparison

5 Conclusions

Redundant states exist in the parsing machine after applying Pager's unit production elimination algorithm. An extension is used to remove redundant states, and reduces the size of the parsing machine significantly. Measurements show that when the extension is used, parser generation takes the same amount of meory but more time, and the resulted parsing machine can be much more compact. Unit production elimination algorithm and its extension should be used on LR grammars only.

Although the extension algorithm does not require extra space to run other than needed by the unit production elimination algorithm itself, it may need much longer running time. Since parser generation is a one-time process, it should be worth such an effort.

6 References

[1] T. Anderson, J. Eve, and J. J. Horning. Efficient LR(1) parsers. Acta Informatica, 2:12–39, 1973.

[2] Alfred V. Aho and Jeffrey D. Ullman. A technique for speeding up LR(k) parsers. SIAM J. Computing, 2:2, 106-127. 1973.

[3] David Pager. On eliminating unit productions from LR(k) parsers. Technical Report PE 245, University of Hawaii, Honolulu. 1973.

[4] David Pager. On eliminating unit productions from LR(k) parsers. Automata, Languages and Programming, Lecture Notes in Computer Science, Volume 14, 242-254. 1974.

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

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

[7] Wilf R. LaLonde, On directly constructing LR(k) parsers without chain reductions, Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages, p.127-133, January 19-21, 1976, Atlanta, Georgia.

[8] M. L. Joliat. A Simple Technique for Partial Elimination of Unit Productions from LR(k) Parsers. IEEE Transactions on Computers, Volume 25 Issue 7, 763-764, July 1976, IEEE Computer Society Washington, DC, USA

[9] Demers, A. J. Elimination of single productions and merging nonterminal symbols of LR(1) grammars. Computer Languages 1:2, 105-119. 1975.

[10] Backhouse, R. C. An alternative approach to the improvement of LR(k) parsers. Acta Informatica 6:3, 277-296. 1976.

[11] Koskimies, Kai {1976} Optimization of LR(k) parsers (in Finnish). M.Sc. Thesis, University of Helsinki, Helsinki.

[12] Koskimies, Kai (1979) On a method for optimizing LR parsers. International Journal of Computer Mathematics 7(4).

[13] Eljas Soisalon-Soininen. Elimination of single productions from LR parsers in conjunction with the use of default reductions. 1977.

[14] Eljas Soisalon-Soininen. On the space optimizing effect of eliminating single productions from LR parsers. Acta Informatica. Volume 14, Number 2, 157-174. 1980.

[15] Takehiro Tokuda. Eliminating unit reductions from LR(k) parsers using minimum contexts. Acta Informatica. Volume 15, Number 4, 447-470. 1981.

[16] Stephan Heilbrunner. Practical conditions for correct elimination of chain productions from LR parsers. Length 77 pages. Hochsch. der Bundeswehr München, Fachbereich Informatik, 1983.

[17] Schmitz, Lothar (1984) On the correct elimination of chain productions from lr parsers. International Journal of Computer Mathematics 15(1-4)

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

[19] Xin Chen, David Pager. LR(1) Parser Generator Hyacc. Proceedings of International Conference on Software Engineering Research and Practice, p.471-477. WORLDCOMP'11, Las Vegas, July 18-21, 2011.

[20] Xin Chen, David Pager. Full LR(1) Parser Generator Hyacc And Study On The Performance of LR(1) Algorithms. Proceedings of The Fourth International C* Conference on Computer Science & Software Engineering, p.83-92. Montreal, Canada, May 16-18, 2011.

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