Parsing Extended LR(k) Grammars

by

Paul Walton Purdom, Jr.

and

Cynthia A. Brown

Computer Science Department

Indiana University

Bloomington, Indiana 47405

TECHNICAL REPORT NO. 87

PARSING EXTENDED LR(k) GRAMMARS

PAUL WALTON PURDOM, JR.

AND

CYNTHIA A. BROWN

DECEMBER 1979

Research reported herein was supported in part by the National Science Foundation under grant number MCS 79 06110.

Parsing Extended LR(k) Grammars *

Paul Walton Purdom, Jr.

and

Cynthia A. Brown

Abstract. An extended LR(k) (ELR(k)) grammar is a context free grammar in which the right sides of the productions are regular expressions and which can be parsed from left to right with k symbol lookahead. We present a practical algorithm for producing small fast parsers directly from certain ELR(k) grammars, and an algorithm for converting the remaining ELR(k) grammars into a form that can be processed by the first algorithm. This method, when combined with previously developed methods for improving the efficiency of LR(k) parsers, usually produces parsers that are significantly smaller and faster than those produced by previous LR(k) and ELR(k) algorithms.

---

* Research reported herein was supported in part by the National Science Foundation under grant number MCS 79 06110.

1. Introduction

An extended context free grammar is a context free grammar in which the right sides of productions are regular expressions over the terminal and nonterminal symbols of the grammar [1]. (We represent the regular expressions by deterministic finite state machines.) Extended context free grammars have many advantages over ordinary context free grammars for representing the syntax of programming languages: the specifications are shorter, easier to construct, and easier to understand. An early example of the use of this technique is in the specification of the programming language Pascal [2, 3].

An extended context free grammar is an ELR(k) grammar if $S \overset{+}{\Longrightarrow} S$ is impossible and if $S \overset{*}{\Longrightarrow} \alpha Az \Longrightarrow \alpha\beta z$, $S \overset{*}{\Longrightarrow} \gamma Bx \Longrightarrow \alpha\beta y$, and first $k(z)$ = first $k(y)$ implies $A = B$, $\alpha = \gamma$, and $x = y$, where all derivations are rightmost [4]. Figures 1 and 2 show ELR(k) grammars; Figure 3 shows a non-ELR(k) grammar. Two approaches have been suggested for parsing ELR(k) grammars. Madsen and Kristensen [1] and Heilbrunner [4] have studied transformations that convert ELR(k) grammars to LR(k) grammars; Heilbrunner shows how to do this for any ELR(k) grammar. Although such transformations are of theoretical importance, they are unsuitable for direct production of efficient parsers. Parsing the input with the LR(k) grammar that results from the transformation requires that the parser find the complete structure of the input under the new grammar, even though much of this structure may be unnecessary for reconstructing the parse for the original ELR(k) grammar.

The second approach, used by La Londe [5, 6] and also by Madsen and Kristensen [1], is to build the parser directly from the ELR(k) grammar.

Most aspects of Knuth's original LR(k) parsing algorithms [7] have obvious extensions to ELR(k) grammars. (These extensions are particularly natural when an extension of Earley's dot notation [8] is used to specify the items for the parser states, as we shall see.) How to pop the stack, however, is less evident. As part of each reduce action, Knuth's algorithm pops the stack by an amount equal to the length of the right side of the production being reduced. A regular right side can generate strings of varying lengths. Previous investigators handled this problem with special rules for recognizing the strings generated by the right side after they have been put on the stack [5, 6], or by putting extra symbols on the stack to help find the beginnings of such strings [1]. These methods do not handle all ELR(k) grammars; they also add complexity and inefficiency to the parser.

Our approach is to stack a state only when the symbol being processed indicates the beginning of a new right side (as a special case, this includes states where the empty string is being reduced). When the end of the production is found, exactly one entry is popped from the stack. This approach does not work for every ELR(k) (or even every LR(k)) grammar because for some (state, symbol) pairs it may not be possible to tell whether the parser is starting a new right side without seeing more of the input. In this situation our first algorithm produces a parser with a stacking conflict.

We also present an algorithm for transforming any ELR(k) grammar whose parser has stacking conflicts into one whose parser has none. Each step of the transformation splits a production into two parts in such a way that it is trivial to reconstruct the parse with the original grammar from the parse with the transformed grammar. Usually only a few steps of the transformation are needed, so the efficiency of the parser is maintained. The transformation

permits us to parse any ELR(k) grammar.

At first sight it might appear that our algorithm could handle only LL(k) grammars, because the parsers it produces recognize when they are starting each right side. Our parsers, however, do not need to recognize which right side they are starting. The algorithm (using the transformation) can build a parser for any ELR(k) grammar.

The inspiration for the basic algorithm came from a comparison of Earley's [8] and Knuth's [7] parsing algorithms. In Earley's algorithm, the number of the current state is recorded at the start of each production and is then copied along with the production as each symbol is recognized. The stack is organized to permit parallel investigation of several possible parses. Since the LR(k) algorithm uses a single stack, we can dispense with the copying of the stack top. The resulting parser has many similarities to the parser that De Remer [9] produces by stacking only those states that must be stacked.

If in some state the same symbol occurs at the beginning of one production and midway in another, a stacking conflict results. The transformation breaks the second production into two parts, so that the beginning of the second part is in the state that had the conflict. Since both items now suggest stacking the state, the conflict is removed. This process requires introducing a new nonterminal into the second production. Heilbrunner [4] indicates how to do this without introducing stacking conflicts.

2. Notation and Basic Algorithm

An extended context free grammar has a starting symbol, a set of

nonterminal symbols that contains the starting symbol, a set of terminal symbols that is disjoint from the nonterminal symbols, and a set of productions. Each production has a left side consisting of a nonterminal symbol and a right side consisting of a deterministic finite state machine with an initial state and one or more accepting states. The finite state machine can have transitions under both terminal and nonterminal symbols. Each nonterminal is used as the left side of one or more productions. It is convenient to have an initial production with no left side and a two state right side. This right side has a transition from its initial state to an accepting state under the starting symbol of the grammar. (The first item in the parser in Figure 4 shows an initial production)

A terminal state is an accepting state with no outgoing transitions. An extended right linear grammar is a grammar where every transition under a nonterminal symbol is to a terminal state. Extended right linear grammars share many properties with right linear grammars.

A path through states $i_0, \ldots, i_m$ spells α iff $\alpha = X_1 \ldots X_n$, where there is a transition from i_j to i_{j+1} under X_j for $1 \leq j \leq n-1$. State i directly generates α iff some path from i to an accepting state spells α. Production p directly generates α iff the initial state of the right side of p directly generates α. State i generates α iff state i directly generates some string β such that α can be obtained from β in zero or more steps, where at each step a nonterminal is replaced by a string that is directly generated by a production for that nonterminal. Production p generates string α iff the initial state of the right side of p generates α. A sentential form is a string generated by the initial production.

A position in a grammar is a pair [i,j] where i is a production and

j is a marked state in the machine for production i . A position is represented in dot notation by giving the left side of production i and the right side of i with a dot in state j . A follow string for production i is a string of k terminal symbols that immediately follows the left side of i in some sentential form. An item is a triple [i,j,γ] , where [i,j] is a position in the grammar and γ is a follow string for i . The initial item is [initial production, initial state, $\dashv^k$]. The item column of Figure 3 shows some items in dot notation.

A lookahead string for item [i,j,γ] is the first k symbols of a string of terminal symbols formed by concatenating a string of terminal symbols generated by state j and the string γ . Item [i,j,γ] directly derives item [a,b,δ] iff b is the initial state of the right side of a , state j has a transition under the left side of a to some state c , and δ is a lookahead string for [i,c,γ] . The derives relation is the nonreflexive transitive closure of the directly derives relation.

The basic parser building algorithm constructs the states of a controller for a pushdown automaton. These states are called parser states to distinguish them from states in the finite state machines used to specify the grammar. Each parser state consists of an item set, an action set, and a go to set. The item set has two parts: a set of main items and a set of derived items. The main items are those that are present in the state initially (see below). Each parser state has a unique set of main items. The derived items are the items that can be derived from the main items; in some states a particular item may be both a main item and a derived item. A parser state for which only the main items have been constructed is called an incomplete state. The item set is used to construct the parser, but it

can be discarded once the parser is built.

Each member of the action set of a parser state consists of a lookahead and an associated action. For each main item $[i,j,\gamma]$ such that j is an accepting state, the parser state contains the action reduce i associated with lookahead γ. (The reduce action for the initial item is also called accept.) For each derived item $[i,j,\gamma]$ such that j is an accepting state (as well as an initial state), the parser state contains the action stack-reduce i associated with lookahead γ. For each item $[i,j,\gamma]$ such that state j has a transition to state b under terminal symbol a, the parser state contains the action shift associated with each lookahead that consists of a concatenated with the first $k-1$ symbols of some lookahead for item $[i,b,\gamma]$. A parser has no action conflict iff it has no more than one action associated with each lookahead string.

The Y-shifted item for item $[i,j,\gamma]$ is the item $[i,j',\gamma]$ such that state j has a transition under symbol Y to state j', if such a state j' exists; otherwise there is no Y-shifted item for $[i,j,\gamma]$. The set of Y-shifted items for a set of items S is the set of items obtained by Y-shifting each item in S.

Each element of a go to set consists of a stacktop symbol, a next state, and a stacking mark. Symbol Y is a stacktop symbol for a parser state iff there is at least one item $[i,j,\gamma]$ in the parser state such that state j has a transition under Y. Each such element is associated with stacktop symbol Y. The next state entry for stacktop symbol Y is the number of the parser state whose main items are the Y-shifted items for the items in the current state. The stacking mark for Y is stack

if all its associated items are derived items, don't stack if all its associated items are main items, and stacking conflict if its associated items include both main and derived items. A parser is deterministic iff it has no action conflicts and no stacking conflicts.

Algorithm 1 (Basic ELR(k) Parser Building Algorithm). This algorithm takes as input a reduced extended context free grammar and produces a (possibly nondeterministic) ELR(k) parser for it.

Step 1 [Initialize]. Form the incomplete initial parser state with the initial item for its main item set.

Step 2 [Complete States]. If there are no incomplete states, stop: the parser is finished. Otherwise choose an incomplete state x and complete it as follows. Add the derived item set, the action set, and the go to set as described above. The next state entry for stacktop symbol Y is the parser state y whose main items are the Y-shifted items of x. If the parser does not already have such a parser state (either completed or incomplete), then create incomplete state y. After state x is completed, repeat step 2.

Figure 4 shows the results of applying Algorithm 1 (with modifications similar to those discussed by Pager [10] for using minimum lookahead at each state) to the grammar in Figure 1. The operation of the parser is the same as that of a traditional LR(k) parser, except for the procedures it uses for manipulating the stack. The parser pushes the current state onto the stack when the action is stack-reduce or when it follows a go to transition associated with a stack mark of stack. The parser pops an entry from the stack (transferring to that parser state) when the action is reduce or stack-reduce. (The stack-reduce action is the same as transferring back

to the current state without changing the stack: it is used when reducing the null string.) Algorithm 1 produces small fast parsers for many ELR(k) grammars.

When Algorithm 1 is applied to an ELR(k) grammar, the resulting parser has no action conflicts. It may, however, be nondeterministic as a result of stacking conflicts.

3. Removing Stacking Conflicts

Figure 5 shows a parser state from the parser Algorithm 1 produces for the grammar in Figure 2. The parser state in Figure 5 has a stacking conflict. A conflict transition in production p of grammar G is a transition from some state i of p under a symbol Y such that there is a parser state in the parser for G having $[p,i,\gamma]$ as a main item for some γ and such that the stacking mark associated with stacktop symbol Y is conflict. For the grammar in Figure 2, the transition from state 1 under a is a conflict transition.

The following grammar transformation removes stacking conflicts. Repeated application of the transformation results in a grammar with no conflict transitions. If the original grammar is ELR(k), so is the transformed grammar; Algorithm 1 produces a deterministic parser when applied to the transformed grammar. The transformed grammar is related to the original grammar in such a way that it is trivial to reconstruct a parse for the original grammar from a parse for the transformed grammar.

To prepare for the transformation, give each state in the grammar a unique number. Associate with each state i a new nonterminal N_i,

called a secondary nonterminal. (The N_i for initial states that have no incoming transitions are not used and may be discarded.) Each N_i has one production. Initially the production consists of a nonaccepting initial state (with a unique number) that has no outgoing transitions, a copy of state i, and a copy of all states that can be reached from state i. If there is a transition from state a to state b under symbol Y, and there is a copy a' of state a in the machine for N_i, then there is a transition under Y from a' to b' (the copy of state b in the machine for N_i). State a' in the machine for N_i is an accepting state iff state a was. The copied state has the same number as the original state. At this stage N_i generates no strings. Each application of the transformation removes one transition from the grammar and adds a transition from the initial state of some N_i to the copy of state i. For purposes of description it is convenient to give copied states the same number as the original state. Any state can be identified uniquely by noting which parser state it is in - associating the machine and the state number.

Conflict Removing Transformation

Obtain grammar G' from grammar G by copying G and making the following modifications. If G has no conflict transitions, do nothing. Otherwise, let k be a production that has a conflict transition from state i to state j under symbol Y.

1. If i is an initial state, modify production k by making i a noninitial state (i must have incoming transitions because it is associated with a conflict and is therefore the marked state in a main item) and by adding a new initial state with a unique number. For each transition from state i to some state a under some symbol X, add a transition from the new initial state to a under X. The new initial state has no incoming transitions.

2. In production k replace the transition from i to j under Y by a transition under N_j from i to the terminal accepting state of k. (if necessary, add such a state to k, with a unique number). In the production for N_j add a transition under Y from the initial state to the copy of state j. In ~~state~~ k delete any states that are no longer accessible.

This transformation replaces a production by a right linear subgrammar for which the terminals and nonterminals of the original grammar are terminals and the secondary nonterminals are nonterminals. The subgrammar generates the same strings as the original production. Figure 6 shows the results of applying the transformation to the grammar of Figure 2, removing the conflict transition from state 1. Inaccessible states and productions have been omitted from Figure 6. The grammar of Figure 6 has no conflict transitions, and Algorithm 1 produces an eight state parser for it.

The procedure for applying the transformation is summarized in the following algorithm.

Algorithm 2

1. Add the initial versions of the productions for the secondary nonterminals, as described above.

2. Repeatedly apply the Conflict Removing Transformation until a grammar with no conflict transitions is obtained.

3. Reduce the grammar.

In the next section we show that Algorithm 2 always terminates (thereby producing a grammar with no conflict transitions), and that

when it is applied to an ELR(k) grammar it produces an ELR(k) grammar. Therefore if we apply Algorithms 1 and 2 to an ELR(k) grammar, we obtain a deterministic parser.

Algorithm 1 does not produce a parser with action conflicts for every non ELR(k) grammar. Figure 3 shows a grammar for which Algorithm 1 produces a parser with a stacking conflict (under stacktop b) but with no action conflicts. After Algorithm 2 is applied to this grammar, Algorithm 1 produces a parser with an action conflict but no stacking conflicts. Only for ELR(k) grammars does Algorithm 2 followed by Algorithm 1 produce a deterministic parser, since only an ELR(k) grammar can have a left-to-right deterministic parser that uses k character lookahead.

4. Proofs

Let G_0 be the original grammar and G_i be the grammar obtained after i applications of the conflict removing transformation.

Lemma 1. Transitions in G_i from initial states that have no incoming transitions are not conflict transitions.

A conflict is always associated with a main item where the dot has moved through one or more states.

Lemma 2. Each transition in G_i from an initial state is under a terminal or original nonterminal symbol (but not under a secondary nonterminal).

This is ensured by step 1 of the transformation.

Lemma 3. The transitions in G_i under secondary nonterminals are not conflict transitions.

This follows from Lemma 2.

Theorem 1. Algorithm 2 terminates.

Proof. Let T_i be the number of transitions in grammar G_i, not counting transitions under secondary nonterminals and not counting transitions from initial states that have no incoming transitions. By Lemma 1 and 3, T_i is an upper limit on the number of conflict transitions in G_i. Each application of the Conflict Removing Transformation reduces T_i by at least one. Therefore, the number of transformations required is no more than T_0.

The proof of Theorem 1 shows that the number of applications of the Conflict Removing Transformation is never more than st, where s is the number of states in G and t is the number of transitions. It is an open question whether there is a better upper limit. In practice only a small number of transformations are usually needed.

Lemma 4. For G_i every transition under a secondary nonterminal is to a terminal accepting state.

After Algorithm 2 each original production has been replaced by a (possibly trivial) extended right linear subgrammar.

Theorem 2. If G_0 is ELR(k), then each G_i is ELR(k).

Proof. Use the proof of Theorem 2 of Heilbrunner [4], replacing right linear with extended right linear and G^H with G_i. After such notational changes it becomes a proof of our Theorem 2.

Theorems 1 and 2 show that applying Algorithm 2 followed by Algorithm 1 produces an ELR(k) parser for any ELR(k) grammar.

5. Practical Considerations

Up to this point we have presented the algorithms in a form that is easy to understand. In this section we discuss some aspects of an efficient implementation of them.

There are many transformations other than the one presented in this paper that can be used to remove conflict transitions. They differ in the time required to complete the transformation, the size of the resulting parser, and the amount of stack space the parser uses. An interesting open question is which transformation is (in some sense) the best one. Until this question is settled a practical system should probably use one transformation consistently, but it should make the results of the transformation known to the user so that he can provide an alternative subgrammar if he wishes.

Another practical consideration is the notation that should be used for describing the finite state machines to the parser building program. One good method is to use right linear subgrammars and have the program reconstruct the machines. The program can automatically determine which nonterminals are used to represent machines, provided the original grammar has no nonterminals used only in a right linear way and without any associated semantic actions. (If the original grammar had such nonterminals, it is useful to optimize them out.)

For efficient operation the ~~initial machines~~ secondary nonterminals and their associated machines should be constructed only for ~~the secondary nonterminals whose associated~~ that states are involved in conflict transitions, ~~as these~~ this should be done as the conflict transitions are discovered. If the accepting states in the grammar are labelled, then all the productions for a single nonterminal can be combined into one production,

with the labels indicating which original production has been recognized.

Algorithm 2 implicitly uses Algorithm 1 for finding conflict transitions. An efficient implementation would build the parser for a transformed grammar by modifying the parser for the original grammar. This process is straightforward, though tedious to describe.

Several techniques for producing efficient LR(k) parsers, such as variable lookahead methods [10] and unit production elimination [11, 12], have been developed. These methods can and should be applied to our algorithm, as suggested by Figure 3.

In a parser it is necessary to stack semantic information as well as the numbers of parsing states. One way to do this is to associate a semantic item with each parser state the parser goes through. This technique can easily be adapted for use with our algorithm. A single stack that contains both the state number (if it is stacked) and then the semantic item is used. Separate alphabets or marking bits can be used to distinguish the two types of entries. For a reduce action the stack is popped back to the previous state entry.

6. Conclusions

Algorithm 1 has been used by one of the authors' students to produce a parser for a subset of Pascal, using hand generated transformations to remove conflicts. Very few transformations were needed and a small fast parser was produced.

We believe this method produces the best available parsers for most ELR(k) languages. It does not require an analysis of stacked entries, nor does it require transforming the grammar to an LR(k) form.

The method should also produce better parsers than LR(k) methods, once it has had the various techniques for optimizing LR(k) parsers added to it, because ELR(k) grammars give a more compact representation of languages than LR(k) grammars do. As shown in [13], the size of a parser is usually closely related to the size of the grammar (even though the best worst case result gives an exponential bound). And, of course, language specifications are easier to produce and understand using extended grammars.

Acknowledgements

We would like to thank Ching-min Jimmy Lo for producing the parser for the student compiler mentioned in the text, and Mitchell Wand for continually stressing the importance of invariants, which led us to the proof of Theorem 1. We wish to thank the referees for their careful reading of the manuscript.

References

1. O.L. Madsen and B.B. Kristensen, "LR-Parsing of Extended Context Free Grammars", Acta Informatica, 7 (1976), pp. 61-73.

2. N. Wirth, "The programming language Pascal", Acta Informatica, 1 (1971), pp. 35-63.

3. N. Wirth, Systematic programming: an introduction, Prentice-Hall, Englewood Cliffs, New Jersey (1973).

4. Stephan Heilbrunner, "On the definition of ELR(k) and ELL(k) grammars", Acta Informatica, 11 (1979), pp. 169-176.

5. Wilf R. La Londe, "Constructing LR parsers for regular right part grammars", Acta Informatica, 11 (1979), pp. 177-193.

6. Wilf R. La Londe, "Regular right part grammars and their parsers", Comm. ACM, 20 (1977), pp. 731-741.

7. Donald E. Knuth, "On the translation of languages from left to right", Information and Control, 8 (1965), pp. 607-639.

8. Jay Earley, "An efficient context-free parsing algorithm", Comm. ACM, 13 (1970), pp. 94-102.

9. Franklin L. De Remer, Practical translators for LR(k) languages, Massachusetts Institute of Technology, Ph.D. thesis (1969).

10. David Pager, "The lane-tracing algorithm for constructing LR(k) parsers and ways of enhancing its efficiency", Information Sciences 12 (1977), pp. 19-42.

11. A.V. Aho and J.D. Ullman, "A technique for speeding up LR(k) parsers", SIAM J. Comput. 2 (1973), pp. 106-127.

12. David Pager, "Eliminating unit productions from LR parsers", Acta Informatica, 9 (1977), pp. 31-59.

13. Paul Purdom, "The size of LALR(1) parsers", BIT 14 (1974), pp. 326-337.

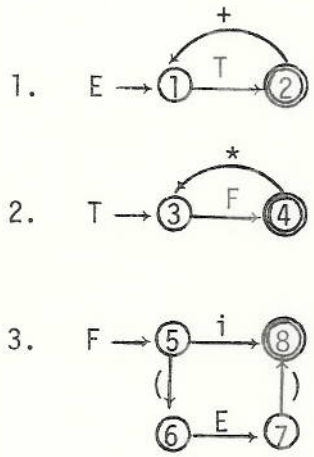


Figure 1

An extended context free grammar for arithmetic expressions

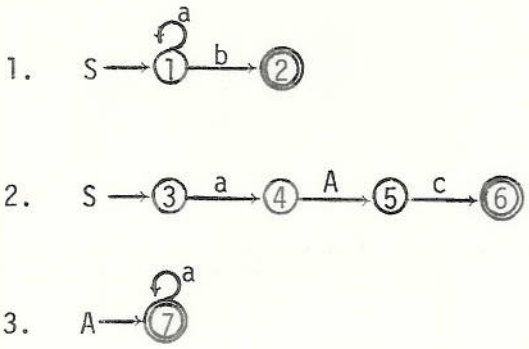


Figure 2

A grammar of La Londe [6].

The transition from state 1 under a is a conflict transition.

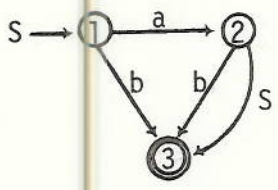
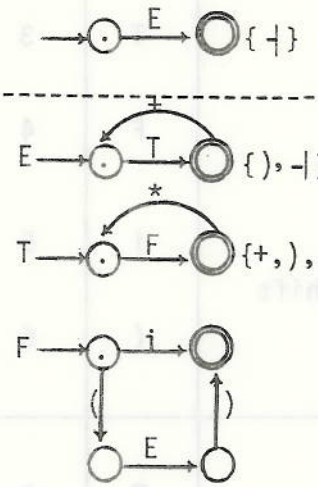
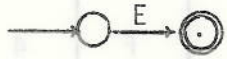
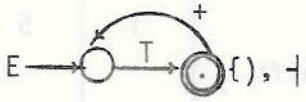
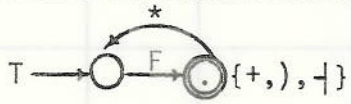
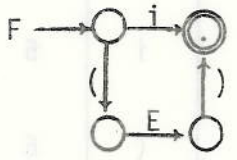
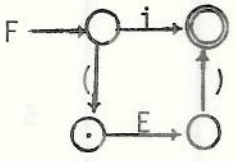


Figure 3

This grammar is not ELR(k) . There are two ways to parse the string ab .

state number	main items --- derived items	lookahead	action	stack top	next state	stacking mark
1			shift	E T F i (	2 3 4 5 6	 stack stack stack stack
2			accept			
3		),⊣ +	reduce 1 shift	+ 	7 	
4		+,),⊣ *	reduce 2 shift	* 	8 	
5			reduce 3			
6				E	9	

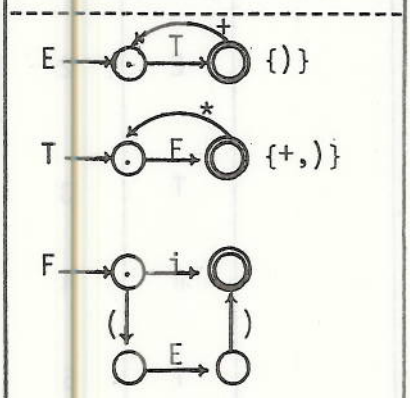
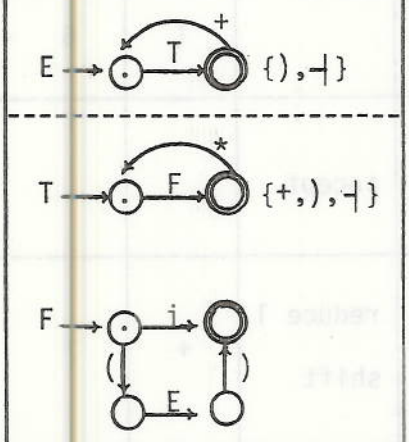
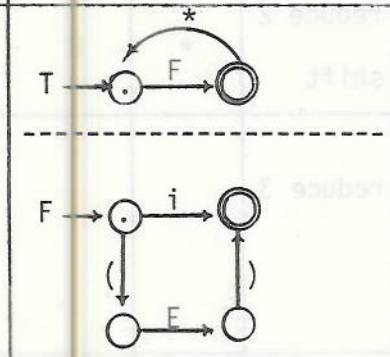
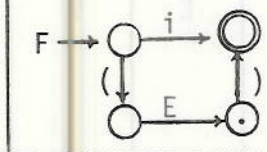
state number	main items --- derived items	lookahead	action	stack top	next state	stacking mark
[6]			shift	T F i (	3 4 5 6	stack stack stack stack
7			shift	T F i (	3 4 5 6	 stack stack stack
8			shift	F i (	4 5 6	 stack stack
9			shift	)	5	

Figure 4

An ELR(1) parser for the grammar in Figure 1.

the methods of Pager [10] were used to reduce the number of ~~states in the~~ parser states.

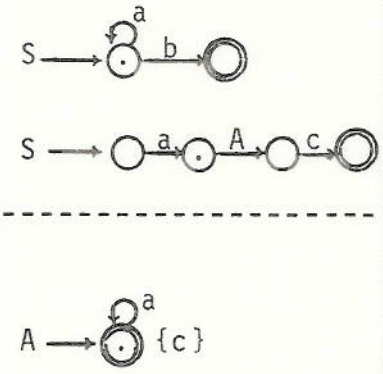
state number	main items --- derived items	lookahead	action	stack top	next state	stacking mark
⋮						
3		a , b a c	shift stack reduce 3	b a A	4 5 6	conflict
⋮						

Figure 5

Parser state 3 of an ELR(1) parser for the grammar in Figure 2.

Notice the stacking conflict that occurs when reading an a .

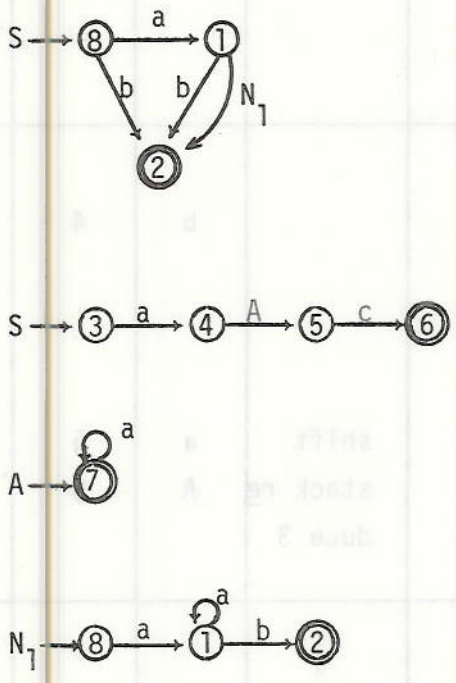


Figure 6

The results of applying Algorithm 2 to the grammar in Figure 2.

Algorithm 1 produces an 8 state deterministic parser for this grammar.