Incremental Generation of LR Parsers

R. Nigel Horspool
Department of Computer Science
University of Victoria

22 December 1989

Abstract

Implementation of a new compiler usually requires making frequent adjustments to grammar definitions. An incremental technique for updating the parser tables after a minor change to the grammar could potentially save much computational effort. More importantly, debugging a grammar is made easier if the grammar is re-checked for correctness after each small change to the grammar. The basic design philosophy of an incremental parser generator, and incremental algorithms for LR(0), SLR(1) and LALR(1) parser generation are discussed in this paper. Some of these algorithms have been incorporated into an implementation of an incremental LALR(1) parser generator.

Index Terms – Compilers, Compiler Tools, Program Development Environments, LR Parsing, LALR(1), SLR(1), Incremental Algorithms, Grammar Debugging.

1 Introduction

A compiler represents a major software development effort. Simple, non-optimizing compilers for relatively small languages like Pascal may consist of several thousand lines of source code. Production compilers for large languages like Ada or that perform sophisticated optimization may consist of hundreds of thousands of lines of code. A conventional compiler is normally organized into phases. A simple compiler would have phases for lexical analysis, syntactic analysis, semantic analysis and code generation. Many tools exist to help the compiler writer develop the lexical and semantic analysis phases. There are tools based on attribute grammar formalisms which can be used to construct semantic analysis and code generation phases. Automatic techniques for developing a code generator phase from a description of the target architecture for the compiler are a subject of current research.

A collection of these tools would form a rudimentary *Compiler Writer's Workbench*. The word *rudimentary* is used because the compiler modules they create do not always fit together easily, because there is no unifying idea, and because their human interfaces may leave much to be desired. These concerns are being addressed in the *MkStar* project, whose goal is to develop a set of interactive compiler development tools for the standard compiler phases listed above. To date, only one tool in the *MkStar* project has been completed – this is an interactive lexical analyzer generator [14]. The second tool in the series will be called *MkParse* and will be an interactive syntactic analyzer generator. The remainder of this paper is concerned with the design and implementation of the LALR(1) parse table generation algorithms used in MkParse.

There are two main approaches to parsing – top-down parsing and bottom-up parsing. Top-down parsing is usually implemented by a method known as *recursive descent*, which uses a collection of mutually recursive procedures. This method has been successfully used in many compilers (for example, in the original compiler for Pascal). But recursive descent is criticized for various reasons. Here are four. First, the class of grammars it can be used with is smaller than for bottom-up methods that accept LALR(1) and LR(1) grammars. Second, recursion is not always implemented efficiently and therefore parsing speed may be adversely affected. Third, good recovery from syntactic errors is not easy in a recursive descent compiler. Fourth, semantic analysis and code generation actions are often included inside the recursive descent procedures, and that tends to spoil the modularity of the compiler. A discussion of the pros and cons of the two approaches appears in [8, section 6.11]. Since we were forced to choose one approach or the other, we picked the bottom-up approach – mostly because it is capable of being used with a larger class of grammars.

When using any existing tool for creating a syntactic analyzer, the user must first create a grammar for the language to be compiled. The grammar is processed by the tool, which we will call a *parser generator*, and it outputs a parser suitable for inclusion in the compiler (or, equivalently, it outputs tables that drive a standard parsing procedure). The form of the grammar is constrained by the class of grammars that the parser generator can accept and by the need to associate semantic actions with the production rules. Parser generators exist for various classes of grammars, including: operator precedence, LL(1), SLR(1), LALR(1), and LR(1).

The compiler writer can rarely use the grammar provided as part of the formal language description. Published grammars are usually designed for people to read and not for the implementer to use. The implementer is therefore likely to find that the grammar does not belong to the class of grammars accepted by the parser generator. Transformations on the grammar need to be performed, while being careful not to change the language that it accepts. Even after the grammar has been changed to satisfy the requirements of the parser generator, further changes are likely to be required when the implementer attempts to attach semantic actions to the production rules. The term *grammar debugging* is often applied to the activity of transforming a grammar in this way.

Inexperienced compiler implementers tend to find grammar debugging a frustrating experience. Each time the implementer wishes to make a change, he/she must edit the file containing the grammar rules, save the file, and execute the parser generator. If, as is frequently the case, the parser generator reports a problem with the grammar, the implementer is likely to receive a cryptic message describing the problem. The widely available *yacc* parser generator [15], for example, simply tells the user that there were conflicts in the parse tables. If more information is required, the compiler implementer must read through a listing of the LR(0) states that have been generated. Other parser generators are usually more helpful in that they provide an example string which would cause the parser to enter the state in which the conflict occurs. Even if the message makes the problem with the grammar clear, it is not a trivial task to formulate a correction. The less sophisticated compiler implementers often resort to a trial and error process where they *patch* the grammar – implying that they must edit the grammar file and repeat the parser generation process again. Many cycles of the debugging process may be required before the grammar is acceptable. The whole approach is reminiscent of the edit/compile/debug cycle used in conventional program development. However, a programmer usually has the benefit of a sophisticated interactive debugger for finding logic errors in his or her program. Corresponding

debugging tools are not available to assist in the development of a grammar.

If the parser generator were to be made available as an interactive program, the frustration of developing a grammar can be greatly reduced. As well as benefitting from instant feedback to erroneous input, the interactive system is capable of providing much more information to the user. Experienced compiler implementors may wish to see the LR(0) sets of items and these could be displayed. Less experienced users may wish to have the ability to enter a sample sentence and see a trace of the parsing actions. Other facilities, such as the ability to select parts of a large grammar for browsing (e.g., by asking to see all uses of a particular non-terminal symbol), could be provided to make the editing process simpler. Another reason to consider an interactive parser generator is that traditional grammar debugging is wasteful of computational effort. If only a small change has been made to a grammar, the parser generator must still process the entire grammar and repeat almost all the work that it performed in the previous stage of the debugging cycle. To reduce the wastage of computational effort and to reduce user frustration, we propose a new compiler development tool. This tool is the *incremental parser generator* (IPG). The tool would be an interactive program which allows the user to develop a grammar. After each change, the grammar is re-checked for acceptability. Instant user feedback to report problems with the grammar should eliminate much frustration and help the user see which production rules cause the difficulties. If the incremental parser generator retains the tables that it creates as part of the grammar analysis and can update, rather than re-build, these tables after a change to the grammar, it should also be possible to eliminate much wasted computational effort.

The idea of building a parser generator that accepts incremental changes to a grammar is not entirely new. However, the few publications in this area have not been widely distributed. The very first work appears to be a PhD dissertation on incremental generation of LL(1) parsers[21] dated 1974. A book the following year[13] was also concerned with top-down parsing. The first publication on incremental generation of LR parsers is Fischer's PhD dissertation at the University of Dortmund[9]. He implemented a PL/I program that starts by reading a grammar and its LR(1) parsing tables. It then reads a list of changes to the grammar (new productions to be added or productions to be deleted) and incrementally updates the tables. Fischer's implementation is a strong competitor for the work reported in this paper. Consequently, some additional comments appear in the conclusions. Six years after Fischer's work, a diploma thesis from the University of Saarbrücken[7] appeared. The thesis describes algorithms that could be used for updating LALR(1) parse tables, but it does not mention an implementation. Whether the algorithms could form the basis of a practical implementation is unclear.

Recently, a group at the Mathematics Centre in Amsterdam[11, 12] have attacked the same problem using the principle of lazy evaluation as the basis for the work. Entries in an LR(0) parse table are created at run-time only as they are needed. However, the Amsterdam system is not limited to the LR(0) class of grammars – it accepts any context free grammar. If the parser that uses these tables encounters a LR(0) conflict, it creates separate tasks that explore each of the conflicting possibilities in parallel. This approach is ideal for exploring new language designs. It does not directly help the compiler developer, however, because a complete parse table is not necessarily ever created and because the system does not report LR parser conflicts. The Amsterdam system is not capable of determining whether the grammar belongs to any particular subclass of the context free grammars. Since a compiler developer almost certainly requires a grammar that falls within one of the grammar classes accepted by a standard parser generator, the Amsterdam system would not be suitable for grammar debugging.

2 Designing an Incremental Parser Generator

There are two main design issues which must be decided before we can discuss the algorithms needed to implement an IPG. First, what quantum of change to the grammar should be input by the tool before the grammar is re-checked? At one extreme, we can recheck after the user adds or deletes single characters to or from the grammar specifications. At the other extreme, we can wait until the user has typed all the desired changes before re-checking. Second, what grammar class should the tool accept? By choosing a small class, such as the class of regular grammars, we would make the implementation of the tool easy but the tool would not be useful to compiler writers. By choosing a large class, such as LR(1), we might make the update algorithms too complicated to implement easily. Complicated update algorithms may also be too slow to be able to provide the user with sufficiently fast responses.

We decided that the unit of change should be a single production rule. After each addition of a new rule and after each deletion of a rule, the grammar is re-checked for acceptability. A change to a rule is considered as a deletion of the original rule followed by an insertion of the corrected rule. If the unit of change were to be made any smaller, we would be faced with the problem of handling incomplete production rules. Larger units of change would simply delay reporting possible problems to the user. But, if we permit the user to add or delete rules in any order, we must be prepared to (temporarily) accept incomplete grammars. One symptom of an incomplete grammar is that some production rules and non-terminal symbols may be inaccessible. For example, if we have entered only the following two rules

```
S  →  begin statement_list end

assignment  →  variable := expression
```

then the second rule cannot be used in any derivation that starts from (what is apparently) the goal symbol *S*. A second symptom may be that some non-terminals cannot generate sentential forms that consist only of terminal symbols. Such non-terminals are called *useless*. For example, we might add just the rule

```
statement_list  →  statement_list ; statement
```

to the grammar, above. Then, even if we temporarily consider *statement* to be a terminal symbol, the grammar is incapable of deriving sentential forms that are free from the non-terminal symbol *statement_list*. If we wish to allow rules to be entered in any order and to check the grammar after each addition, it is clear that a relaxed form of checking must be employed. Ignoring the rules for inaccessible or useless non-terminals would be an unsatisfactory approach. The user could choose to enter rules in such an order that almost the entire grammar may remain inaccessible or useless until the last rule is defined. (As a convenience to the user, the IPG should, of course, include an interactive command to ask which symbols are currently inaccessible or useless.)

Current generators for bottom-up parsers usually accept one of the LR grammar classes. We chose to implement the LALR(1) class of grammars because of its power – it contains the LL(1), LR(0) and SLR(1) classes. While it is a smaller class than the LR(1) class, the generated parser usually has far fewer states and therefore requires much less memory for its implementation. It also appears to be the case that LR(1) parsing tables require much more work to update after a small change to the grammar. Conversely, parsers for the LR(0) and SLR(1) classes of grammars require less computational effort to create than does LALR(1). A parser generator for either of

these smaller grammar classes may be more suitable in situations where the computational cost is important.

3 Handling Incomplete Grammars

It should be possible to analyze grammars which have not been completely specified. Indeed, the start symbol of the grammar may be one aspect of the grammar that remains undefined until late in the specification process. It is therefore appropriate to add a goal symbol of our own, $\hat{S}$, and to invent extra rules of the form

$$\hat{S} \rightarrow \$_N \; N \; \$_N$$

for each non-terminal symbol N in the partial grammar. The $\$_N$ symbol is a delimiter symbol of type N. It is an invented symbol that represents both a beginning of input and an end of input delimiter. Its purpose is to provide a unique context within which N can appear if it were to be used as a goal symbol. If unique delimiter symbols are not provided, our support for multiple goal symbols can cause LR conflicts in the parser construction process.

But addition of these extra rules requires us to know which symbols are non-terminals. A reasonable strategy would be to assume that every symbol encountered in the grammar so far is a terminal symbol, unless the symbol appears on the left-hand side of a rule. The alternative approach would be to require that the user must declare all symbols as terminal or non-terminal before they can be used. But, just as declarations of variables are usually not mandatory in interactive programming languages, we prefer terminal/non-terminal definitions to be optional.

The addition of the extra rules also eliminates the problem of unreachable non-terminal symbols or rules. Everything in the grammar would be reachable from the goal symbol $\hat{S}$. Of course, after the user has specified the actual start symbol and claims to have completed the grammar, we should perform checks for unreachable or useless rules and symbols. Standard algorithms exist for performing these checks [4].

While a grammar is under development, it is natural for some parts of the grammar to be incomplete. Therefore, we should not complain about useless productions until the user claims to have completed the grammar. For example, the user may have entered the rule

```
L →L , x
```

and no others with L as a left-hand side. It is clear that L is a non-terminal symbol, but it is also useless. We can circumvent this difficulty by assuming that while the grammar is in an incomplete state, it is a grammar for sentential forms – *not* a grammar for sentences in the language. For example, with the rule for L, above, the language includes the sentential forms

$\$_L$ `L` $\$_L$ $\quad$ $\$_L$ `L , x` $\$_L$ $\quad$ $\$_L$ `L , x , x` $\$_L$ $\quad$ ... etc.

where $\$_L$ is the automatically generated context delimiter symbol. If the user makes an explicit request to check the grammar for completeness or requests that the LALR(1) parse tables be output, an algorithm to check the grammar can be applied. A suitable algorithm is given in [4].

When rules are missing from a grammar, it is impossible to know for certain which symbols are nullable (can produce the empty sentence in some derivation sequence). A symbol X may appear to be non-nullable, but the addition of the rule

```
X    →   ϵ
```

(or of a rule where all symbols on the right-hand side are themselves nullable) would immediately change the status of X. It seems best to assume that symbols are non-nullable until a derivation to the empty string becomes possible using rules in the grammar. This would also avoid generation of premature error messages about ambiguities in the grammar.

4 Incremental LR Parser Construction Algorithms

When the grammar class is restricted to one of the LR(0), SLR(1) or LALR(1) classes, the computation of parsing actions for a particular grammar can be separated into two stages. The first stage is the construction of the LR(0) sets of items for the grammar. The second stage computes the lookahead sets that are associated with the LR(0) items. Computing these lookahead sets is trivial for a LR(0) parser generator and is the most expensive for LALR(1).

4.1 Terminology

A context-free grammar G is defined as a four-tuple $G = < V_T, V_N, S, P >$, where V_T is the set of terminal symbols, V_N is the set of non-terminal symbols, S is a designated start symbol, and P is the set of production rules. As explained above, the grammar that our algorithms process is not the same grammar as is entered by the user. It has been augmented by additional productions and additional symbols. We will use the name G to refer to this augmented grammar.

We will use standard conventions when discussing grammar formalisms. The symbol ϵ represents an empty string of grammar symbols. If R is a relation, then R^* represents the reflexive, transitive closure of R. Other conventions are as follows.

$$\begin{aligned} V &= V_N \cup V_T \\ S, \hat{S}, A, B, C, \ldots &\in V_N \\ X &\in V \\ a, b, c, \ldots &\in V_T \\ \alpha, \beta, \gamma, \ldots &\in V^* \end{aligned}$$

The relation $\Rightarrow$ is pronounced "directly produces" and is defined so that

$$\alpha A \gamma \;\Rightarrow\; \alpha\beta\gamma$$

for all $\alpha, \gamma \in V^*$ and $A \rightarrow \beta \in P$. The relation $\stackrel{*}{\Rightarrow}$ is pronounced as "produces" and represents the transitive closure of $\Rightarrow$.

4.2 Augmenting the Grammar

When the user adds the new production rule

$$L \rightarrow R_1\ R_2\ \ldots\ R_n \quad (n \geq 0)$$

$G = < V_T, V_N, S, P >$, a new grammar $G' = < V'_T,\ V'_N,\ P',\ S' >$ is created. The new grammar is related to the old grammar according to the following rules:

$$\begin{aligned} V'_N &= V_N \cup \{L\} \\ V'_T &= V_T \cup \{\ \$_L,\ R_1,\ R_2,\ \ldots\ R_n\ \} - V'_N \\ P' &= P \cup \{\ L \rightarrow R_1 \ldots R_n\ ,\ \hat{S} \rightarrow \$_L\ L\ \$_L\ \} \\ S' &= S \end{aligned}$$

where $\hat{S}$ is the fictitious supergoal symbol described earlier. Note: when R_i is a new occurrence of a symbol (and is not identical to the symbol L), it is considered to be a terminal. Of course, if R_i subsequently occurs on the left hand side of some rule, its class is changed from terminal to non-terminal by the rules given above.

If we begin with an initially empty grammar, $G_0 = < \phi,\ \phi,\ \hat{S},\ \phi >$, then the augmented grammar after a sequence of rule additions is completely determined.

4.3 Incremental Update of the LR(0) Sets of Items

Construction of the LR(0) sets of items is covered in texts on compiler construction [2, 3, 8, 20]. A reader who is unfamiliar with the methods and concepts of LR(0) parser construction will find the formal definitions and notation difficult to read. We therefore begin with an informal introduction.

4.3.1 Informal Introduction to LR(0) Concepts

The LR(0) construction method is based on the concept of an *item*. An item is simply a production rule with a marker (frequently called a *dot*) inserted anywhere in the right-hand side of the rule. The marker indicates how many symbols of the right-hand side have been recognized at some point in a parse. That is, all symbols to the left of the marker have been recognized and symbols to its right have not yet been used. An LR parser is implemented as a finite state automaton, where each of its states corresponds to some set of items. Each such set of items represents parsing possibilities – showing which rules are eligible to be matched if appropriate symbols are read by the parser. The sets of items which give rise to the LR recognizer are constructed by a process of closure.

There are some initial items, formed by inserting the marker at the beginnings of the right-hand sides of all goal rules in the grammar. There are two closure operations which we will call *Item Closure* and *State Closure*. Given a set of items, the *Item Closure* operation adds extra items to the set; these extra items are usually called *completion items*. If any item in the set has the marker immediately to the left of a non-terminal symbol N, then this item generates completion items. These completion items are formed by taking each rule which has N on its left-hand side and placing the marker at the beginning of the right-hand side. These completion items may, themselves, require the addition of more completion items – which is why a closure process is required.

The *Kernel* operation is applied to a grammar symbol and a set of items that has been completed (using *Item Closure*), to yield a new set of items. This new set of items plus completion items added by the *Item Closure* operation, corresponds to the next state in the LR recognizer as reached by a transition on the grammar symbol. The items for a new state before *Item Closure* is applied, are called the *kernel* items of the new states. (Some texts call these the *core* items.)

The LR(0) collection of sets of items is formed by a process of applying Item Closure to the initial items, and applying *State Closure* to this set to yield the remaining sets of items. *State Closure* can be implemented using an iterative algorithm based on a work list. The algorithm is sketched out below. In this algorithm, *Start* is the set of items for the start state of the recognizer; W is the work list. On termination, K is a set that contains all the recognizer's states.

State Closure:

```
Start := { "the initial items" };
W := { Start };
K := { };
while W is not empty do
    remove state S from W;
    S := ItemClosure(S);
    K := K ∪ { S };
    for each grammar symbol X do
        compute the kernel items, S' = Kernel(S,X);
        if S' is a new state then
            W := W ∪ { S' };
```

Since states are uniquely determined by their sets of kernel items, the test to see if G is a new state does not require that the *Item Closure* function be applied to G first.

The operation of *Item Closure* can also be implemented by an iterative algorithm based on a work list. In this case, the work list holds individual items. The algorithm has a similar structure to the one given above and therefore we omit giving its details.

4.3.2 Formal Introduction to LR(0) Concepts

Our definitions and notation for LR recognizers are derived from the notation used by DeRemer and Pennello [6]. There are two major differences. First, DeRemer and Penello introduced a mapping function F which performs a one-to-one mapping from states in the LR automaton to sets of items (and they used the inverse function F^{-1} for the opposite mapping). We have dropped this distinction between states and sets of items. A second difference is that the *Reduce* function has been replaced by a function LA which explicitly associates lookahead sets with LR(0) items.

The LR automaton for G is a sixtuple,

$$LRA(G) = < K,\ V,\ P,\ Start,\ NextState,\ LA >$$

where K is a set of sets of LR(0) items (or, equivalently, a set of states), $V = V_N \cup V_T$, P is as in G, *Start* is the start state for the automaton, *NextState* is a transition function between states, and LA is a function which associates lookahead sets with the LR(0) items computed for each state. The LA function will not be defined further until the following section.

There is a one-to-one correspondence between states of the LR automaton and sets of LR(0) items, where an LR(0) item is defined as follows. Each LR(0) item is a triple $< A, \alpha, \beta >$, where $A \rightarrow \alpha\beta \in P$, and is conventionally written as $[A \rightarrow \alpha \bullet \beta]$.

The sets of items for the *Start* state and for successor states (reached via the *NextState* function) are created as follows.

$$Start = ItemClosure(\ \{\ [\ \hat{S} \rightarrow \bullet\ \$_X\ \ X\ \$_X\ ]\ \mid\ X \in V_N\ \}\ )$$

$$\forall q, X\ (q \in K,\ X \in V):$$
$$NextState(q, X)\ =\ ItemClosure(Kernel(q, X))$$

where $ItemClosure(IS)$ is the smallest set CS such that $IS\ \subseteq\ CS$ and if there is an item of the form $[B \to \alpha \bullet A\beta]$ in CS then

$$\{\ [A \to \bullet\ \omega]\ \mid\ A \to \omega\ \in\ P\ \}\ \subseteq\ CS$$

and where

$$Kernel(IS, X)\ =\ \{\ [A \to \alpha X \bullet \beta]\ \mid\ [A \to \alpha \bullet X\beta]\ \in\ IS\ \}$$

The function $ItemClosure(IS)$ yields the result of adding completion items to a kernel set of items, IS. The function $Kernel(IS, X)$ yields the kernel items for a state reached by a transition on symbol X from the state IS. The *ItemClosure* operation can be implemented by an iterative algorithm using a work list of items, as mentioned previously.

Another way of considering the problem of constructing the LR(0) recognizer is as another kind of closure – a closure over the set of states. We can say:

$$K\ \ =\ \ StateClosure(\ \{\ Start\ \}\ )$$

where $StateClosure(SS)$ is the smallest set of states CS such that $SS \subseteq CS$ and

$$\forall\ ST,\ X\ (ST \in CS,\ X \in V):\ NextState(ST, X)\ \in\ CS$$

The *StateClosure* operation is conveniently implemented by an iterative algorithm using a work list of states, as outlined previously.

As a minor detail, we should point out that the *StateClosure* operation will create an empty set of items as one of the LR recognizer's states. This state would be an error state which is entered when the LR parser is applied to an invalid sequence of terminal symbols. In practice, an explicit error state is usually omitted from LR recognizers.

4.3.3 Incremental Update of the LR(0) Sets of Items

After the user has added the new production $L \to R_1\ R_2\ \ldots\ R_n$ to the grammar, we need to construct the corresponding LR automaton $LRA(G')$. It will usually be the case that the sets of items for states in the new automaton correspond closely to sets of items for states in the old automaton. Some sets of items will be unchanged, while other sets will have increased in size due to the addition of new members. Unfortunately, it is not always possible to determine what items (if any) need to be added to an item set by inspection of the current members of the set and, perhaps, of the grammar rules too. Here is an example to illustrate the difficulty. Suppose that the user has entered the following (incomplete) grammar.

$$\begin{array}{lcl} S & \to & C\ \mid\ D\ \mid\ f\ D\ f \\ C & \to & a\ A\ b \\ D & \to & a\ e\ c \\ B & \to & e\ c \end{array}$$

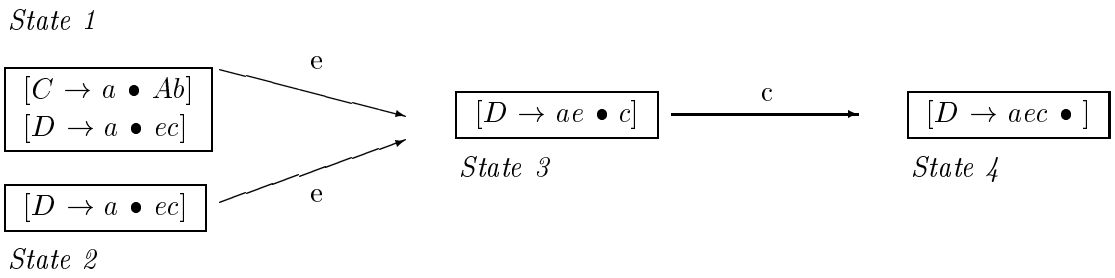


Figure 1: LR(0) States Before Rule Addition

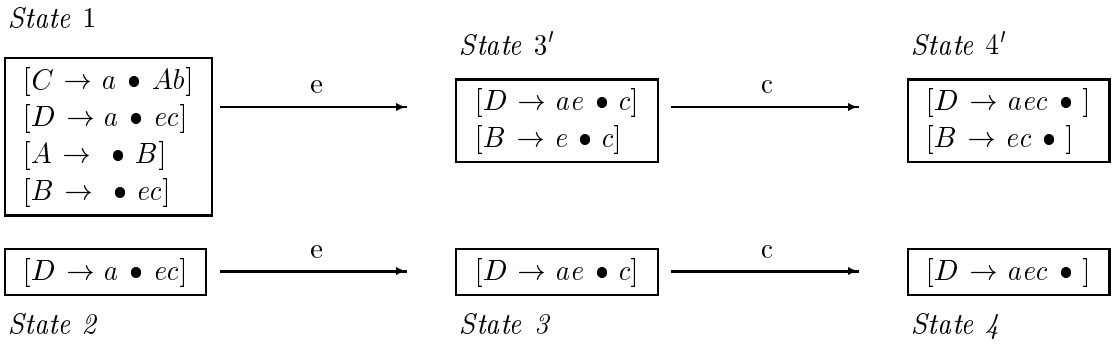


Figure 2: LR(0) States After Rule Addition

Construction of the LR recognizer will generate three sets of items (amongst about 20 other sets) that are related as shown in Figure 1. But after we add the extra rule $A \rightarrow B$, the corresponding states in the new LR automaton have sets of items as shown in Figure 2.

States 3 and 4 in the first LR recognizer have split to become pairs of states in the second recognizer. Unfortunately, there does not appear to be a simple test for determining when such state splitting is required – or, indeed, for determining what extra elements must be added to a set of items.

We therefore adopt a heuristic approach to the problem. We assume that almost all states in the original LR recognizer, $LRA(G)$, also exist in the new recognizer, $LRA(G')$, and with the same kernel items. It is easy to add extra completion items to states in $LRA(G)$ to convert them into (plausible) states for $LRA(G')$. This conversion is accomplished by the *map* function, defined below. As before, the new rule added to the grammar is $L \rightarrow R_1 \ldots R_n$. We map the set of items for the *Start* state as follows:

$$map(Start) \;=\; \begin{cases} Start & \text{if } L \in V_N \\ Start \;\cup\; \{\, [\, L \rightarrow \bullet \;\; \$_L L \$_L \,] \,\} & \text{otherwise} \end{cases}$$

and map the item sets for all other states as follows:

$$map(IS) \;=\; \begin{cases} IS \;\cup ItemClosure(\, \{\, [\, L \rightarrow \bullet\, R_1 \ldots R_n \,] \,\}\,) \\ \qquad\qquad \text{if } \exists\, I\ (I \in IS): \text{ I has the form } [\, Y \rightarrow \alpha \bullet L\beta \,] \\ IS \qquad\qquad \text{otherwise} \end{cases}$$

If we modify the sets of items in this way, we may be creating some states that should not be present in $LRA(G')$ and we will certainly not have created all the states that do occur in $LRA(G')$. However, if the assumption that *most* states in $LRA(G)$ occur with the same kernel items in $LRA(G')$ is reasonably accurate in practice, we have a good heuristic basis for an incremental algorithm. Experimental data, given below in the *Practical Experience* section, does indicate that the assumption is valid.

To implement the heuristic, all we need do is map the item sets, as explained above, then use the *StateClosure* function to add any missing states, and finally remove any superfluous states. The principle of this approach can be captured in the following equation which creates the new set of LR states K' from the previous set K.

$$K' \;=\; Reachable(\; StateClosure(\; \{\; map(ST) \mid ST \;\in\; K \;\}\;)\;)$$

where *Reachable* acts like a garbage collection algorithm, yielding only those states that are reachable from the start state. It may be defined as:

Reachable(SS) = the smallest set of states RS such that:

$RS \subseteq SS$, and

$Start \in RS$, and

$\forall\ ST,\ X\ (ST \in RS,\ X \in V):\ NextState(ST,\ X)\ \in\ RS$

The correctness of the procedure is easy to see. As long as the set of items for the start state is updated correctly, the *StateClosure* function is guaranteed to construct all the necessary states – and this is true whatever the *map* function does to other states.

The process of creating the $LRA(G')$ states can be implemented a little more efficiently than is suggested by the above procedure. It is possible that too many states are contained in the set passed as an argument to the *StateClosure* function. During the closure operation, the unnecessary states may give rise to yet more unreachable states – thus creating additional work for the *Reachable* function. A more efficient approach is to combine the reachability analysis with the state closure operation, only tracing transitions out of states that we have already determined to be reachable from the start state.

Another way of improving efficiency, and the method actually used in our implementation, is to leave unreachable states in the recognizer. As long as the unreachable states do not cause LR parsing conflicts or do not become too numerous, they should not interfere with our algorithms. Our implementation in *ilalr* therefore defers performing a proper reachability analysis on the states until a LR parsing conflict is encountered or until the user requests that the parser be output to a file.

4.4 Incremental Update of Lookahead Sets

The LR automaton can be used to recognize the language generated by G. While recognizing a particular string in the language, it performs *shift* or *reduce* actions as determined by the current state of the automaton and by the current input symbol. A *shift* action is a state transition which has the side-effect of causing a new input symbol to be read. Therefore the shift actions correspond to the *NextState* function which has already been defined. A *reduce* action corresponds to the successful recognition of the right-hand side of some production rule $A \rightarrow \alpha$. The action causes the recognizer to be halted and then restarted in a different state. This new state is the

state that would have been reached if a transition on symbol A had been followed instead of the transitions followed for the symbols in α. More detailed descriptions of the operation of an LR parser may be found in texts on compiler construction [3, 8, 20].

The reduce actions that may be performed in a state q can be determined from its set of items and from the lookahead function $LA : K \times I \rightarrow V^*$. K is the set of states for the recognizer and I represents the domain of all items that can be constructed from the grammar G. That is, LA maps each item in a state to its lookahead set.

An item where no symbols appear to the right of the marker (the *dot*) is called a *reduce item*. Any other item, i.e. a non-reduce item, will be called a *shift item*.

There are different methods of constructing the LA function depending on whether we wish to restrict the class of grammars to the LR(0), SLR(1) or LALR(1) class. These construction methods will be described below.

If the reduce item $I = [A \rightarrow \alpha \bullet]$ is a member of the state q, then a reduce action for the rule $A \rightarrow \alpha$ is performed in state q whenever the input symbol is a member of the set $LA(q, I)$.

It is a requirement of parsing methods in the LR(k) family that the parsing actions be uniquely defined. It is said that state q has a *shift-reduce* conflict if there is a symbol t such that $NextState(q, t) \neq \emptyset$ and q contains a reduce item I where $t \in LA(q, I)$. Similarly, a state q has a *reduce-reduce* conflict if there is a symbol t such that q has two reduce items I_1 and I_2 where $t \in LA(q, I_1) \cap LA(q, I_2)$.

The parser generator is unable to construct deterministic parsing tables when the LR recognizer has conflicts. There are three possible approaches to handling a conflict. The first and simplest approach is simply to detect and report the conflict as an error. A user-friendly parser generator should include an example sequence of symbols which illustrates the parsing conflict in the error report. A second approach is for the parser generator to make an assumption as to which of the conflicting actions is desired by the user. The *yacc* parser generator [2, 15] adopts this approach because it resolves conflicts based on the order in which production rules have been entered. Finally, the third and most sophisticated approach is to attempt to analyze the conflict, tracing paths through the parser states to discover the origin of the conflict. If the conflict occurs because the grammar is LR(1) and not LALR(1), it is possible to split states and remove the conflict. This approach is the basis of some LR(1) parser generation methods [18, 19].

4.4.1 LR(0) Lookahead Sets

The lookahead sets are trivially constructed in the LR(0) case. For an item $I = [A \rightarrow \alpha \bullet]$ and a state q where $I \in q$, we simply use

$$LA(q, I) = V_T$$

Checking for conflicts is simplified if we extend the domain of LA to include shift items. For an item $I = [A \rightarrow \alpha \bullet X\beta]$ where $I \in q$, we define

$$LA(q, I) = \{X\}$$

Any state containing two reduce items necessarily has a reduce-reduce conflict. Any state q containing a reduce item I_1 and a non-reduce item I_2 has a shift-reduce conflict if $LA(q, I_1) \cap LA(q, I_2) \neq \emptyset$.

4.4.2 SLR(1) Lookahead Sets

The SLR(1) method uses a slightly more sophisticated definition for the lookahead sets. The sets are computed using a function called *FOLLOW*. But computation of *FOLLOW* is facilitated if the set of all nullable symbols, *NULL*, and a function called *START* are also computed.

The set of nullable symbols is formally defined as

$$NULL \;=\; \{\, X \mid X \;\stackrel{*}{\Rightarrow}\; \epsilon \,\}$$

Using *NULL*, we can determine the nullability of any sentential form.

The set of nullable symbols after a rule addition is closely related to that set beforehand. Continuing with the convention that primes refer to the grammar G', the relationship is:

$$NULL' \;=\; \begin{cases} NULL \;\cup\; \{B | B \stackrel{*}{\Rightarrow} L\} & \text{if } \forall\, i\ (1 \le i \le n) \;:\; R_i \in NULL \\ NULL & \text{otherwise} \end{cases}$$

where the rule $L \rightarrow R_1 \ldots R_n$ is added to the grammar G to create G'. A simple iterative algorithm based on a worklist of non-terminal symbols that need to be re-checked for nullability can be used to obtain $NULL'$ from *NULL* efficiently.

The *START* function yields the set of starter symbols for a grammar symbol. It is formally defined as

$$START(X) \;=\; \{\, Y \mid X \;\stackrel{*}{\Rightarrow}\; Y\ \alpha \,\}$$

The *FOLLOW* function yields the set of symbols that may legally follow a grammar symbol in a sentential form. It is defined as

$$FOLLOW(X) \;=\; \{\, Y \mid S \;\stackrel{*}{\Rightarrow}\; \alpha X Y \beta \,\}$$

Methods for computing the *START* and *FOLLOW* functions can be found in most books on compiler construction. The approach given here is based on [10]. The *START* function can be found by constructing the *Immediate Starters* relation for the grammar and then forming the transitive closure of that relation. If we choose to represent that relation by a matrix *IS*, where $IS[X, Y] \;=\; true$ means that symbol X has Y as one of its immediate starters, then the matrix is defined as follows. An entry $IS[X, Y]$ has the value *true* iff the grammar contains a rule $X \rightarrow Z_1 Z_2 \ldots Z_k Y \beta$ such that $\forall i\ (1 \le i \le k) :\; Z_i \in NULL$. All other entries have the value *false*.

The transitive closure of IS is written as IS^* and is defined by

$$IS^*[X, Z] \;=\; IS[X, Z] \textbf{ or } (\; \exists Y_1, Y_2, ...Y_n : IS[X, Y_1] \& IS[Y_1, Y_2] \& ... \& IS[Y_n, Z] \;)$$

There are well-known algorithms for computing the transitive closure of a relation [1, 22]. Efficient incremental algorithms for transitive closure also exist [23] and would be particularly suitable for use here. If we form the transitive closure IS^* using any of the standard methods, we have

$$START(X) \;=\; \{\, Y \mid IS^*[X, Y] \;=\; true \text{ and } Y \in V_T \,\}$$

Similarly, the *FOLLOW* function can be computed by first constructing an *Immediate Followers* relation and then taking its transitive closure. We define the *IF* matrix so that $IF[X, Y]$ has the value *true* iff either (1) the grammar contains a rule $Z \rightarrow \alpha Y Z_1 Z_2 \ldots Z_k W \beta$ and $\forall i\ (1 \le i \le k) :\; Z_i \in NULL$ and $X \in START(W)$, or (2) the grammar contains a rule $X \rightarrow \alpha Y Z_1 Z_2 \ldots Z_k$ and $\forall i\ (1 \le i \le k) :\; Z_i \in NULL$.

The transitive closure of IF can be computed, again using standard techniques. We can then obtain $FOLLOW$ from IF^* as follows.

$$FOLLOW(X) \;=\; \{\; Y \mid IF^*[X,Y] \;=\; true \text{ and } Y \in V_T \;\}$$

The formulation of $START$ and $FOLLOW$ in terms of IS^* and IF^* demonstrates the monotonic nature of the problem. When a new rule $L \rightarrow R_1 R_2 \ldots R_n$ is added to the grammar, we must change entries from *false* to *true* in the IS and IF matrices. Changes in the reverse direction cannot occur. For example, if $n > 0$, the entry $IS[L, R_1]$ would be set to *true*. In turn, this implies changing entries from *false* to *true* in transitive closure, IS^*, and thus we would have computed $START'$. Having computed $START'$, we can update IF. For example, when we add the rule $L \rightarrow R_1 R_2 \ldots R_n$ and where $n > 1$, the entries $IF[R_1, x]$ for $x \in START'(R_2)$ would be set to *true*.

An algorithm which works well is to update IS and IF as suggested, and then use an iterative, worklist-based, approach for updating the transitive closures.

Once we have computed the IF^* relation (and thus the $FOLLOW$ function), we can determine the lookahead sets. According to the SLR(1) approach, we use:

$$LA(q, [A \rightarrow \alpha \bullet\,]) \;=\; FOLLOW(A)$$

And, if we define LA for non-reduce items in the same way as for LR(0) parsers, conflict checking can also be performed in the same way.

4.5 LALR(1) Lookahead Sets

Several methods for computing LALR(1) lookahead sets have been published. Of these, an iterative algorithm due to Aho and Ullman, described in [3, algorithm 4.13] and [8, figure 6.25], appears to be best suited for conversion to use in an incremental setting. We give a modified version of this algorithm below.

If the existence of some item, I_1, in some state implies the existence of another item, I_2, either in the same state (through the addition of completion items) or in some other state (through the state completion process), then the lookahead function applied to I_2 yields a set which may contain symbols determined by I_1. This is called *spontaneous* generation of lookahead symbols. In addition, it is possible that the set of lookahead symbols for I_2 must include the entire set of lookahead symbols for I_1. In this case, the symbols are said to *propagate* from I_1 to I_2.

The rules for spontaneous generation of symbols and propagation of symbols in the two possible settings are as follows.

Case 1 – Completion Items
Suppose that state q contains an item I_1 where the marker appears to the left of a non-terminal symbol. That is, I_1 has the form $[A \rightarrow \alpha \bullet X\beta]$. The state must also contain one or more completion items with the form $[X \rightarrow \bullet\, \gamma]$. Let I_2 be one such item.

The symbols which can follow the righthand-side of I_2 must include the symbols which follow X in item I_1, and the symbols which can follow X in that item must include $FIRST(\beta)$, where $FIRST$ is defined below. In other words, $a \in FIRST(\beta)$ implies $a \in LA(q, I_2)$. In the terminology of [3], the symbol a is spontantaneously generated (by I_1) and must appear in the lookahead set of I_2.

In addition, if $\beta \stackrel{*}{\Rightarrow} \epsilon$ and $b \in LA(q, I_1)$ then $b \in LA(q, I_2)$ must hold. This would be a case of symbol b propagating from I_1 to I_2.

Case 2 – Kernel Items
Suppose that state q_1 contains an item I_1 with the form $[A \rightarrow \alpha \bullet X\beta]$. There must necessarily be another state q_2 reached by a transition on symbol X from q_1, where q_2 contains a kernel item with the form $[A \rightarrow \alpha X \bullet \beta]$. In such a case, if $a \in LA(q_1, I_1)$ then $a \in LA(q_2, I_2)$. This is another example of propagation, where symbol a propagates from I_1 to I_2.

The *FIRST* function is a simple extension of *START* to the domain of sentential forms.

$$FIRST(\alpha) \;=\; \{\; x \mid \alpha \stackrel{*}{\Rightarrow} x\beta,\; x \in V_T \;\}$$

An alternative definition which shows how to derive *FIRST* from *START* is

$$\begin{aligned} FIRST(X_1 \ldots X_k) &= \begin{cases} START(X_1) \cup FIRST(X_2 \ldots X_k) & \text{if } X_1 \in NULL \\ START(X_1) & \text{otherwise} \end{cases} \\ FIRST(\epsilon) &= \emptyset \end{aligned}$$

A simple algorithm to determine the lookahead sets can start by initializing all lookahead sets to empty. Then it can make repeated passes over all items in all states adding spontaneously generated symbols and propagated symbols to the sets. This iterative procedure can halt when a pass fails to add any new symbols to any set. (A faster version, and the method used in *ilalr*, would use a worklist so that only items whose lookahead sets have changed participate in the next pass. Entries in the worklist consist of $< state, item >$ pairs.)

The algorithm is appealing because it appears that we can use it incrementally. Updating the LR(0) recognizer may have added a few items to some states and created a few states containing new items. Subsequently, we can initialize the lookahead sets of the new items to empty and leave the lookahead sets of the others alone. Then we can execute the Aho and Ullman algorithm to determine the lookahead sets for the new items and to propagate lookahead symbols from the new items to other items. This process will converge much more quickly than performing a full re-computation of all lookahead sets.

Such an incremental approach would be correct only if lookahead sets cannot lose any symbols when a new rule is added to the grammar. That is, it would be correct only if lookahead sets grow monotonically. Unfortunately, the example in Figures 1 and 2 provides a demonstration that the LALR(1) lookahead sets do not necessarily grow monotonically. In the picture before the addition of a new rule (Figure 1), the lookahead set for the sole item in state 4 is obtained by propagating lookahead symbols from the relevant items in states 1 and 2. The lookahead set for this item, $[D \rightarrow aec \bullet\ ]$, is $\{\$_S, \$_D, f\}$, where the symbols $\$_S$ and $\$_D$ are the context delimiter symbols introduced by our incremental algorithms. But in the picture for after the rule addition (Figure 2), the lookahead sets must be separated. The correct lookahead set for the item in state 4 is now $\{f\}$ and the lookahead set for the same item in state $4'$ is $\{\$_S, \$_D\}$.

Although the direct method of augmenting the lookahead sets does not work, there are ways to correct the problem. One way takes advantage of the fact that the iterative algorithm will yield the correct results if the lookahead sets are initially assigned values which are subsets of their final values. This can be achieved if we assign empty lookahead sets to all new items created during the incremental computation of the LR(0) sets of items and reset the lookahead sets of items in states which split to empty.

An alternative strategy, and the approach actually used in *ilalr*, is to ignore the problem as long as no conflicts are detected. If we assign empty lookahead sets to new items created during the incremental update of the LR(0) sets of items and if we simply apply the iterative algorithm to the LR recognizer, some lookahead sets will contain too many symbols. (We note that lookahead sets computed by this approach will never contain symbols that do not appear in the SLR(1) lookahead sets.) Occasionally, the over-large sets will cause spurious conflicts. But when any conflict is detected, we simply clear all lookahead sets to empty and re-compute their correct values with the iterative algorithm. If the conflict still exists after the re-computation, it is reported to the user as a LALR(1) conflict. If the conflict no longer exists, we can just carry on as normal. This approach is sensible only if spurious conflicts have a low frequency of occurrence. Fortunately, experience with *ilalr* indicates that spurious conflicts are rare in practice.

It can be proved that the method of computing lookahead sets used in *ilalr* cannot miss any potential conflicts. One such a proof would begin with the observation that the lookahead set computation satisfies the requirements of a *monotone dataflow framework* [16]. Given a particular LR(0) collection of states and a particular vocabulary of terminal symbols V_T, an element in the lattice corresponds to an ordered collection of lookahead sets associated with the LR(0) items. The bottom element, $\perp$, in the lattice corresponds to the case when every lookahead set is empty, and the top element, $\top$, to the case where every lookahead set is equal to V_T. The lattice relation, $\leq$, corresponds to a pairwise conjunction of $\subseteq$ tests on the lookahead sets for corresponding LR(0) items.

We assume that the iterative algorithm of Aho and Ullman is already known to be correct. If the lattice element corresponding to the correct LALR(1) collection of lookahead sets is denoted by L_{LALR}, then we can write

$$L_{LALR} \quad = \quad f_{LA}(\perp)$$

where f_{LA} is a function that represents the effect of applying the iterative algorithm to the initial situation where every lookahead set is empty.

Next, we must establish that f_{LA} is a monotonic function. That is, $a \leq b$ implies $f_{LA}(a) \leq f_{LA}(b)$. This property can be proved in two steps. The first step is to observe that the iterative algorithm only ever adds elements to the lookahead sets. The second step is to show that if some element is added to a lookahead set in one application of f_{LA} then the same element must already be present or must be added in the other application of f_{LA}.

Once monotonicity has been established, we can see that given any starting configuration of the lookahead sets, X, it follows from the fact $\perp \leq X$ that $f_{LA}(\perp) \leq f_{LA}(X)$ and, therefore, that $L_{LALR} \leq f_{LA}(X)$. In other words, we are guaranteed that applying the iterative algorithm to X yields lookahead sets that must be identical to or must be supersets of the correct LALR(1) lookahead sets. Thus, it is impossible to miss a conflict. The worst that can happen is that we discover a conflict that would not be present with the true LALR(1) lookahead sets. The strategy of recomputing correct lookahead sets after a conflict has been detected is therefore necessary and correct.

The above argument shows that *ilalr* can begin with lookahead sets that are incorrect and, yet, still be able to detect conflicts correctly. However, the argument has not taken account of one other feature of the implementation – that the LALR(1) recognizer may contain extra, unreachable, states. It is necessary to argue that the extra states can only cause additional elements to be present in the lookahead sets that correspond to items in reachable states. In other words, the monotonicity property must still hold. The consequence is, again, that spurious

conflicts may be discovered but no conflicts can be missed. After a conflict has been found, it is therefore necessary to remove the unreachable states as well as to recompute the lookahead sets.

5 Decremental Algorithms

Finding reasonably efficient, practical, algorithms to handle deletion of a production rule remains an area for further research. The incremental parser generator that was implemented does support rule deletion, but the operation usually takes a little more time than the corresponding rule addition. The implementation has been oriented towards optimizing the addition of rules. When a new element is added to a set (such as the set of items in an LR(0) state), no record of why and when the element has been added is retained. Thus, if we later wish to undo the work, a considerable amount of re-computation is needed to decide whether this element should be removed.

When the user decides to delete a rule from the grammar, the change has the following consequences. First some symbols may have to be removed from *NULL*. Second, items must be removed from some item sets in the LR(0) construction and the LR(0) recognizer will lose some states. Third, many lookahead sets are also likely to lose members.

We will briefly review what needs to be done. To start with, updating the augmented grammar to remove a production rule is relatively easy. If the rule being deleted is $L \rightarrow R_1 \ldots R_n$ and this rule is the only rule where L appears on the left-hand side, we would also have to remove the production rule $\hat{S} \rightarrow \$_L\ L\ \$_L$ from G and change the status of L from non-terminal to terminal.

When updating the LR(0) recognizer, we can use a technique similar to that employed when adding a rule to the grammar. This involves finding items based on the deleted production rule and removing them from the sets of items. However, these items may have caused completion items to be included and these completion items must be removed too. If $L \rightarrow R_1 \ldots R_n$ is the rule to be deleted, we can precompute the set of all items based on the deleted rule:

$$DeleteItems \ = \ \{ L \rightarrow R_1 \ldots R_j \bullet R_{j+1} \ldots R_n \mid 0 \leq j \leq n \ \}$$

And we can also precompute its closure:

$$AllDeleteItems \ = \ ItemClosure(DeleteItems)$$

This second set includes all items which must be removed from the item sets plus all other items which might have to be removed.

We now define a mapping function for item sets of the form

$$DelMap(IS) \ = \ ItemClosure(IS - AllDeleteItems)$$

where '$-$' denotes set difference. Removing all the items in the *AllDeleteItems* set may remove too many completion items and therefore we have to return some completion items to the set. The mapping process is, in general, a many-to-one mapping and therefore it will normally reduce the number of states in the LR(0) recognizer. In spite of the mapping reducing the number of states, it may still leave unreachable states in the recognizer. These should be removed by applying the reachability analysis technique described earlier. In addition, we may have to delete states that have kernel items based on the rule $\hat{S} \rightarrow \$_L\ L\ \$_L$.

Construction of the lookahead sets is best accomplished by performing a total recomputation. That is, all lookahead sets are cleared to empty sets and the iterative algorithm is applied.

However, a complete recomputation can be safely deferred until there is an apparent conflict or until the final version of the parse tables must be output. The correctness arguments given in the preceding section apply here too and prove that no conflicts can be missed by this approach.

6 Worst Case Complexity

A natural question to ask is "how long can it take to update the parser tables after the addition or deletion of one production rule using the algorithms described in this paper?" We will demonstrate that the problem has exponential worst-case time complexity and therefore no algorithm that is efficient in the worst-case can exist. The demonstration is based on the following grammar[1] which contains $2n + 2$ productions.

$$
\begin{array}{rcl}
S_0 & \rightarrow & a\ S_0 \mid b\ S_0 \\
S_0 & \rightarrow & c\ S_1 \\
 & & \\
S_1 & \rightarrow & a\ S_2 \mid b\ S_2 \\
S_2 & \rightarrow & a\ S_3 \mid b\ S_3 \\
S_3 & \rightarrow & a\ S_4 \mid b\ S_4 \\
 & \cdots & \\
S_{n-1} & \rightarrow & a\ S_n \mid b\ S_n \\
 & & \\
S_n & \rightarrow & d
\end{array}
$$

The LR(0) construction algorithm applied to this grammar generates a recognizer with $4n + 6$ states.

Suppose, now, that the production rule

$$S_0 \rightarrow a\ S_1$$

is added to the grammar. The extra rule causes the number of states in the LR(0) recognizer to be increased to $2^n + 4n + 6$ states.

Therefore, the unfortunate implication is that any algorithm for incrementally updating the LR parse tables must be prepared to create an exponential number of states when processing a single rule addition. Therefore both the time and space cost are exponential in the size of the grammar. (The amended grammar also illustrates that even a non-incremental LR(0) parser generator has exponential worst-case time and space complexity.)

The example grammar is constructed in such a way that it illustrates another interesting property. Suppose that we add yet one more production

$$S_0 \rightarrow b\ S_1$$

Surprisingly, this addition causes the number of states in the LR(0) recognizer to be reduced to $6n + 7$. Now, consider what happens if this last rule is deleted. We would observe the number of LR(0) states to increase from $6n+7$ to 2^n+4n+6. In other words, this example demonstrates that an incremental algorithm for rule deletion must also have exponential time and space complexity.

[1] The grammar was constructed by Alan Demers.

	GRAMMAR					
	PL/0	Pascal	XPL	C	Oberon	Ada
Total CPU time	0.3	2.0	2.7	10.9	2.1	23.6
Average time per rule	0.005	0.009	0.02	0.04	0.008	0.05
Maximum time for one rule	0.02	0.05	0.16	3.72	0.04	0.8
Total CPU time used by *yacc*	0.3	1.2	1.0	3.8	1.4	11.5

Note: All times are in seconds and were measured on a SUN SparcStation I workstation.

Table 1: Timing Measurements

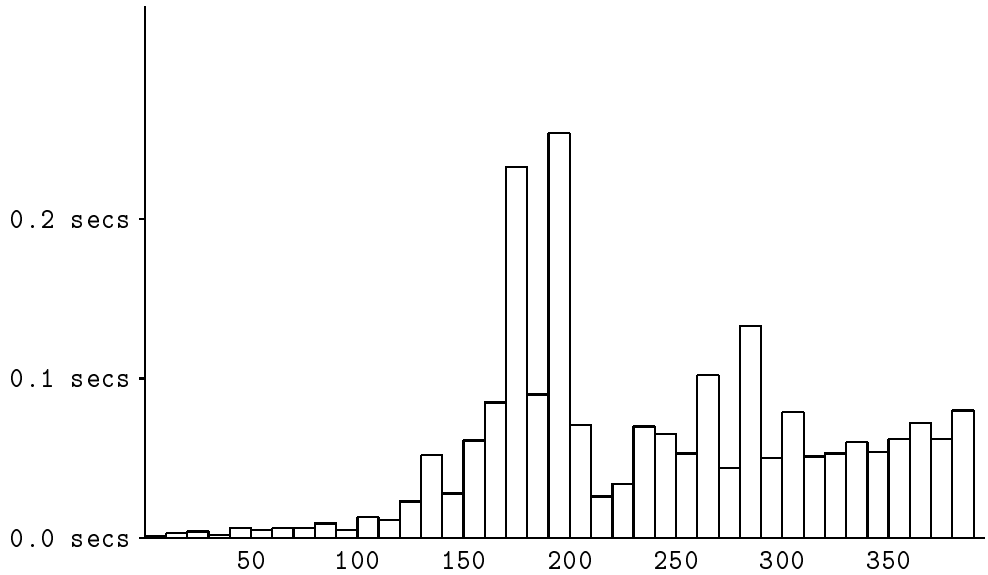
7 Practical Experience

The techniques described in this paper have been used in an experimental implementation called *ilalr*. It is a fully interactive incremental generator of LALR(1) parsers. Because the human interface was not the main focus of the research effort, input commands have a simple line-oriented structure. *Ilalr* has been designed to create parsers that link with lexical analyzers generated by another interactive tool, *MkScan* [14], as well as by *lex* [17]. There should be little difficulty in interfacing with lexical analyzers from other sources.

As the timing measurements in Table 1 show, the speed of execution of *ilalr* appears to be adequate for interactive use. For example, a grammar for the C language containing 219 production rules requires about 11 seconds of CPU time to process on a SUN Sparcstation I workstation – or an average of 0.04 CPU seconds per production. In comparison, the *yacc* parser generator [15] requires a little under 4 CPU seconds for the identical grammar. Considering that *ilalr* actually creates 219 different LALR(1) recognizers in the course of its execution and checks that each one is free from conflicts, the three to one speed ratio is surprisingly good. (The performance ratio for the other sample grammars was less.) In any event, the implementation easily keeps up with the rate at which new production rules can be entered by a human. The alternative approach of re-running a non-incremental parser generator like *yacc* after each rule addition would not be competitive.

The actual amount of processing time needed for processing one rule addition varies quite widely. The third row in table 1 shows the worst case for each grammar – the actual time for the rule that required the most processing. These maximum times depend strongly on the order in which the production rules are presented to *ilalr*. In the case of the C grammar, the maximum time was observed for the last rule in the group of rules that describe the structure of expressions. The expression operators of C, and the corresponding production rules, are organized into 15 precedence levels. Changes to the grammar of expressions for the lowest precedence level cause an avalanche of changes to the recognizer states and to lookahead sets across all the precedence levels.

Although there is considerable variation in the processing time requirements for successive rule additions, there is a general trend toward increased times as the grammar grows larger. Figure 3 shows the growth trend for the largest grammar in the collection, the Ada grammar. To make the data easier to read, the times are reported as a bar graph where each column represents the average processing time (in CPU seconds) over 10 consecutive rule additions. That is, the first



Note: The times were measured on a SUN SparcStation I workstation.

Figure 3: Time for Rule Addition versus Grammar Size

	GRAMMAR					
Number of	PL/0	Pascal	XPL	C	Oberon	Ada
Productions	40	164	109	219	181	399
Accessible states	129	518	345	624	588	1140
Inaccessible states	90	111	80	140	46	302
LR(0) states	84	296	187	338	306	811
States after optimization	55	161	99	168	171	480

Table 2: Statistics for Some Sample Grammars

column corresponds to the average time over productions 1 through 10, the second column over productions 11 through 20, and so on.

Two heuristics were used to improve the speed of the implementation. One heuristic was to avoid running the garbage collection algorithm unless absolutely necessary. In fact it is only executed in *ilalr* when a (possibly spurious) conflict is detected and, also, immediately prior to construction of the final form of the parser tables. This heuristic could lose more CPU time than it gains if the proportion of unreachable states becomes too high. However, the data in Table 2 indicate that the proportion of unreachable states grows to only 21% for the largest grammar tested. (A more complete description of the numbers appearing in Table 2 is given below.) The alternative strategy of performing a garbage collection after every rule addition causes an approximate doubling of the CPU time requirements for these grammars.

A second heuristic was used to reduce the computational expense of computing LALR(1) lookahead sets. *Ilalr* normally computes sets that may contain extra symbols – only performing the proper computation after a conflict is discovered in some state (and also immediately before

Percentage of	GRAMMAR PL/0	Pascal	XPL	C	Oberon	Ada
Modified states	8.80	1.61	6.83	1.56	1.19	1.34
Missing states	4.23	0.87	1.41	0.72	0.69	0.45
Superfluous states	1.33	0.19	0.39	0.18	0.09	0.11

Table 3: Efficacy of State Updating Heuristic

constructing the final form of the parser tables). If the frequency of spurious conflicts is too high, the heuristic could cause a net loss of CPU time. However, it turns out that none of the sample grammars causes a spurious conflict, even though five of the six grammars are not SLR(1). In fact, we have only been able to trigger a spurious conflict and its concomitant recomputation of lookahead sets by using specially constructed grammars.

Table 2 shows four different numbers for the number of states generated in the LR(0) recognizer. The first number is the total number of reachable states in the recognizer for the augmented grammar (where the start symbol is the super-goal $\hat{S}$). The second number is the total number of unreachable states that are also created as a result of our approximate algorithm for determining the set of LR(0) states. The third number is the number of states in the LALR(1) parser that should be output by *ilalr* after the user has specified the true start symbol. The fourth number is the number of states in an optimized LR parser that is actually output by *ilalr*. A simple optimization of eliminating states that contain only a unique reduce action and combining the reduce action with the shift action in the predecessor state is performed. This optimization is easy to perform, it eliminates a high proportion of states, and (unlike most other parse table compaction techniques) does not reduce the execution efficiency of the generated parser[5].

The algorithm for updating the LR(0) sets of items after the addition of a new production was based on a heuristic. This heuristic performs a simple mapping process to change each set in the original LR(0) recognizer into a new set that is likely to be needed in the new LR(0) recognizer. Some data to support the efficacy of the heuristic appears in Table 3. The first row shows the percentage of states whose sets of items are actually modified by the mapping process. The second row shows the percentage of new states added by the subsequent *State Closure* process. The third row shows the percentage of superfluous states after the mapping process (states which are created by the mapping algorithm but are not actually needed). The proportion of superfluous states ranges from a miniscule 0.1% to 1.3%, depending on the grammar. Another way of looking at the statistic is to compare the third row of the table with the first row. This shows that between 6% and 15% of the states modified by the mapping process are not needed.

8 Conclusions and Further Work

Practical algorithms for incremental analysis of grammars and for incremental generation of LR(0), SLR(1) and LALR(1) recognizers have been presented. Compiler construction tools based on these algorithms would help compiler writers develop suitable grammars and might also permit the incremental construction of compilers.

Although the worst-case execution times of the algorithms are poor, practical experience shows

that they work well. A possibility that might reduce expected execution time requirements even further would be to incorporate an efficient incremental transitive closure algorithm[23]. There is, of course, no hope for such an algorithm improving the worst-case time complexity of the problem. The transitive closure algorithms are efficient only for the case when a single edge is added or deleted from a graph. As section 6 in this paper demonstrates, the number of edges that must be added or deleted from the state graph of the LR(0) recognizer can be exponential in the size of the grammar.

It would be nice to give a proper comparison between the speeds of our implementation and Fischer's implementation[9], but only a meaningless comparison seems possible. Using our implementation, the time needed to process the 109 productions of the XPL grammar is 2.7 CPU seconds on a SUN SparcStation I workstation, for an average of 0.024 CPU seconds per production. Fischer's implementation applied to the same grammar used an average of 1.8 CPU seconds per production on a Siemens 7.748 computer (which he describes as being nearly half as fast as an IBM 370/158). We cannot conclude whose algorithms and techniques are faster from this data. A person who uses an incremental parser generator to help design a new grammar would, however, notice a major difference in use between our tool and Fischer's. Our tool automatically performs the maximum possible checking and the maximum possible table generation on an incomplete grammar. However, Fischer's techniques only used production rules when they became usable in a derivation that starts from the goal symbol. If the grammar contains ambiguities, our tool will always report this fact at the earliest possible moment, whereas this would not be true of Fischer's parser generator.

We note that incremental generation of LL(1) parsing tables can be performed relatively easily because these tables are constructed using *FIRST* and *FOLLOW* functions. The incremental construction of these functions has been described under the SLR(1) parsing method.

More work remains to be done before an interactive parser generator is included in our Compiler Writer's Workbench. Most importantly, the current, line-oriented, interface of *ilalr* should be replaced by a more sophisticated, full-screen interface. The final product, which will be called *MkParse*, will also have facilities to exchange information automatically with the other compiler development tools. Such exchanges of information can help eliminate interfacing errors between the compiler phases and help eliminate any need for the user to re-enter information.

Acknowledgements

The students of the CSc471 course in Compiler Construction at the University of Victoria deserve thanks for acting as guinea pigs. Their comments and the unpredictable inputs they gave to *ilalr* proved very helpful in improving the error diagnostic messages and in finding obscure errors. Funding for this research was gratefully received from the Natural Sciences and Engineering Research Council of Canada.

References

[1] A.V. Aho, J. Hopcroft and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] Aho, A.V., Johnson, S.C. LR Parsing. *ACM Computing Surveys,* vol. 6, no. 2, pp. 99-124, 1974.

[3] Aho, A.V., Sethi, R., Ullman, J.D. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, MA (1986).

[4] Aho, A.V., Ullman, J.D. *The Theory of Parsing, Translation and Compiling; vol. 1, Parsing.* Prentice Hall, Englewood Cliffs, NJ (1972).

[5] Dencker, P., Dürre, K., Heuft, J. Optimization of Parser Tables for Portable Compilers. *ACM Trans. on Prog. Lang. and Sys.,* **6,4**, 546-572 (1984).

[6] DeRemer, F., Pennello, T. Efficient Computation of LALR(1) Look-Ahead Sets. *ACM Trans. on Prog. Lang. and Sys.,* **4,4**, 615-649 (1982).

[7] Ehre, R. Die Generierung eines Multiple-Entry Parsers und ein inkrementeller LALR(1)-Parsergenerator. Diploma thesis, Department of Mathematics and Computer Science, University of Saarbrücken, Federal Republic of Germany (1986).

[8] Fischer, C.N., LeBlanc Jr., R.J. *Crafting a Compiler.* Benjamin/Cummings, Menlo Park, CA (1988).

[9] Fischer, G. Incremental LR(1) Parser Construction as an Aid to Syntactical Extensibility. PhD Dissertation, Tech. Report 102, Department of Computer Science, University of Dortmund, Federal Republic of Germany (1980).

[10] Griffiths, M. LL(1) Grammars and Analysers **in** *Compiler Construction: An Advanced Course – second edition,* (Lecture Notes in Computer Science vol. 21), F.L. Bauer and J. Eickel, Eds., Springer-Verlag, Berlin (1976).

[11] Heering, J., Klint, P., Rekers, J. Incremental Generation of Parsers. Report CS-R8822, Centre for Mathematics and Computer Science, Amsterdam (1988).

[12] Heering, J., Klint, P., Rekers, J. Incremental Generation of Parsers. Proceedings of Sigplan '89 Conference on Programming Language Design and Implementation. *ACM Sigplan Notices* **24, 7**, 179-191 (1989).

[13] Heindel, L.E., Roberto, J.T. *LANGPAK – An Interactive Language Design System.* American Elsevier, New York, NY (1979).

[14] Horspool, R.N., Levy, M.R. MkScan – An Interactive Scanner Generator. *Software – Pract. & Exper.* **17, 6**, 369-378 (1987).

[15] Johnson, S.C. YACC – Yet Another Compiler-Compiler. Bell Laboratories, Murray Hill, NJ, Rep. CSTR 32 (1974).

[16] Kam, J.B., Ullman, J.D. Monotone Data Flow Analysis Frameworks. *Acta Informatica* **7, 3**, 305-318 (1977).

[17] Lesk, M.E., Schmidt, E. LEX – A Lexical Analyzer Generator. Bell Laboratories, Murray Hill, NJ, Rep. CSTR 39 (1975).

[18] Pager, D. A Practical General Method for Constructing LR(k) Parsers. *Acta Informatica* **9** 249-268 (1977).

[19] Spector, D. Efficient Full LR(1) Parser Generation. *ACM Sigplan Notices* **23, 12** 143-150 (1988).

[20] Tremblay, J.-P., Sorenson, P.G. *The Theory and Practice of Compiler Writing.* McGraw-Hill, New York, NY (1985).

[21] Vidart, J. Extensions Syntactiques dans une Contexte LL(1). Doctoral dissertation, University of Grenoble (1974).

[22] Warshall, S. A Theorem on Boolean Matrices. *J. ACM* **9,1**, 11-12 (1962).

[23] Yellin, D. A Dynamic Transitive Closure Algorithm. IBM T.J. Watson Research Center, Yorktown Heights, NY, report RC 13535 (1988).

Incremental Generation of LR Parsers

R. Nigel Horspool

Summary

Developers of software systems have support tools which allow the system to be efficiently rebuilt after an addition or a change to the system source code. The analogous tools for compiler developers do not yet exist.

One tool which a compiler development environment will need is an incremental generator of parsers. Given a small change to a grammar specification, the tool will incrementally re-build the parser tables and report any problems with the modified grammar. If the tool can process changes sufficiently fast, it would be usable as an interactive program for both debugging and developing grammars. Existing, non-incremental, parser generators are too slow to permit interactive editing of a grammar specification.

Although the worst-case time complexity of processing a change to a grammar is poor, it is possible to develop algorithms that work well in practice. The algorithms described in this paper are based on two basic techniques. One is the use of worklists to drive iterative algorithms. The other is an application of lazy evaluation – deferring work until it becomes necessary. An interactive, incremental LALR(1) parser generator that uses the algorithms has been developed and is shown to give highly acceptable performance.