SYNTACTIC ANALYSIS OF
LR(k) LANGUAGES

T. ANDERSON

Ph.D. Thesis

January 1972

University of Newcastle upon Tyne

Acknowledgements

I am very grateful for the encouragement of my supervisor, Dr. J. Eve, from whose suggestions and comments this thesis has derived much benefit.

Thanks must also go to my wife, who typed the draft, and Miss H. Bell, who typed the final version of the thesis.

Support for the research described here was received from the Science Research Council. The University of Newcastle upon Tyne aided the completion of the thesis.

Abstract

A method of syntactic analysis, termed LA(m)LR(k), is discussed theoretically. Knuth's LR(k) algorithm is included as the special case m = k. A simpler variant, SLA(m)LR(k) is also described, which in the case SLA(k)LR(0) is equivalent to the SLR(k) algorithm as defined by DeRemer. Both variants have the LR(k) property of immediate detection of syntactic errors.

The case m = 1 k = 0 is examined in detail, when the methods provide a practical parsing technique of greater generality than precedence methods in current use. A formal comparison is made with the weak precedence algorithm.

The implementation of an SLA(1)LR(0) parser (SLR) is described, involving numerous space and time optimisations. Of importance is a technique for bypassing unnecessary steps in a syntactic derivation. Direct comparisons are made, primarily with the simple precedence parser of the highly efficient Stanford AlgolW compiler, and confirm the practical feasibility of the SLR parser.

Contents

Chapter 2 Theory

Chapter 3 Precedence

Chapter 4 Practice

Appendices

Chapter 1

Introduction

In recent years, the problem of analysing the structure of computer programs written in high level programming languages has received considerable attention. Theoretical studies have been concerned mainly with methods of analysis dependent on the use of context free grammars to model the syntax of such languages. Syntactic analysis forms an important component of the process of compilation, and although context free grammars do not in general provide all of the information required for the analysis, they do give a formal specification which has proved extremely useful.

Research has been directed towards methods for constructing, from a given context free grammar, an algorithm capable of analysing any string generated by that grammar. These methods play an essential role in translator writing systems, and a number of existing techniques are discussed in the survey by Feldman and Gries (1968). Most of the methods which have been developed construct left to right analysis algorithms, and fall into two categories, referred to as top-down and bottom-up methods. This dissertation is concerned solely with methods in the latter category.

A few of the methods which have been devised, are capable of constructing analysers for any context free grammar, the best example being that of Earley (1971). For reasons of efficiency it is unfortunately necessary in practice to restrict choice to methods which are only applicable to subsets of the context free grammars.

The work of Floyd (1964) and Irons (1964) on bounded context grammars, and an algorithm due to Earley (1965) which transcended the concept of bounded context, led to the definition of LR(k) grammars by Knuth (1965). Knuth showed that the LR(k) grammars (those analysable from left to right with only k symbols of lookahead) correspond with the deterministic languages of Ginsburg and Greibach (1966). To use the LR(k) method, a value for k must be chosen, and while k = 0 does not give a sufficiently general method, the LR(1) grammars appear to be adequate for most programming languages. Restriction to LR(1) grammars also aids a human user of the syntax. Although theoretically efficient, practical LR(1) implementation founders on the magnitude of the tables needed to direct the analyser (while tables are not essential, in their absence the analyser is too slow).

Other theoretical techniques, notably the precedence group of methods, have met with practical success. Floyd (1963) describes an algorithm for operator precedence grammars. The restriction to operator grammars was removed by Wirth and Weber (1966), who defined simple precedence grammars. McKeeman (1966) considered practical ways of extending the generality of simple precedence techniques, while Gries (1968) combined the use of transition matrices with a precedence scheme for operator grammars. These methods are fast and efficient and have received extensive practical use, but they lack the generality of the LR(k) algorithm, as well as being inferior in their error detection abilities.

In Chapter 2 of this dissertation, a generalisation of LR(k) is discussed, called LA(m)LR(k), together with a simpler variant called SLA(m)LR(k). Both have the LR(k) property of immediate detection of syntactic errors, and for m = k they are identical to the LR(k) algorithm.

Methods are described for generating and minimising the size of the tables required by these techniques. Special attention is given to the LA(1)LR(0) and SLA(1)LR(0) algorithms (abbreviated LALR and SLR). Their generality, and the size of tables they require lie between those of LR(0) and LR(1).

Chapter 3 demonstrates the usefulness of the notation employed (based on that of Knuth) by making a formal comparison of SLR with precedence methods. This yields a characterisation of the precedence relations, and proves the inclusion of the weak precedence grammars within the SLR grammars, indicating to some extent the scope of the SLR algorithm.

Practical aspects of the SLR method are stressed in Chapter 4. Of importance is the incorporation, within the SLR framework of a technique for bypassing unwanted steps in a syntactic derivation. This increases the speed of analysis, as does a process of eliminating parts of an SLR table which are LR(0) in behaviour. A number of methods for compacting SLR tables are discussed, and a sequence for applying these methods is suggested. A scheme is described by which semantic routines may augment an SLR analyser, to enable the use of non SLR grammars if required. Experimental evaluations are made of the amount of storage needed for SLR tables, and a comparative sequence of timings made between an SLR analyser and the simple precedence based syntactic analyser of the highly efficient Stanford AlgolW compiler.

The results of these experiments show that the SLR, and hence also LALR, algorithms can be made to compare very favourably with current methods, both in time and space requirements; in view of their greater generality (in accommodating a much larger subset of the context free grammars) and their ability to detect syntactic errors at the earliest possible point in an input string, the SLR and LALR algorithms should substantially replace current methods.

A brief history of the development of the LA(m)LR(k) algorithm follows.

In an attempt to obtain a practical algorithm of the LR type, J. Eve devised a 'modified LR(1)' algorithm, and encoded (in PL/1 under O.S. 360) a table constructor program for both this and full LR(1). Results from the programs being encouraging, the author continued this line of investigation, making the generalisation to 'modified LR(k)' and implementing a more flexible table constructor (in AlgolW under MTS, see Appendix 2). Examination of table representations and compactions began.

In related work, Korenjak (1969) described the construction of an LR(1) type of processor by means of an ad hoc partitioning of a grammar. A much stronger connection is present in the research described by DeRemer (1969) in his doctoral thesis. He discussed SLR(k) grammars, which were virtually equivalent to our 'modified LR(k)' grammars, although from a very different approach. In addition, he defined a class of grammars which he called LALR(k), but did not specify any algorithm for these. The definition involves an exact knowledge of the possible k symbol lookahead strings at every stage of an analysis.

This exact knowledge can be obtained from an LR(k) table, which can provide lookahead information for an otherwise LR(0) algorithm. A generalisation of this procedure would be to use an LR(m) table to extend the lookahead of an otherwise LR(k) algorithm ($m \geq k$); this is the LA(m)LR(k) algorithm. If a simpler alternative is adopted, that of computing the lookahead extension directly from the grammar, then we obtain SLA(m)LR(k). The case SLA(k)LR(0) corresponds to SLR(k).

The terminology SLR(k) has the disadvantage of indicating a nonexistent connection with LR(k), the only real link being with LR(0) independently of the value of k. The same misconception is evident in the phrase 'modified LR(k)'; better would have been 'k modified LR(0)'. Similar comments apply to the term LALR(k) .

Typographical Note

Care is needed on occasion in this thesis to distinguish between a subscript numeral 1 (e.g. X_1) and a subscript letter l (e.g. X_l).

Chapter 2

Notation

Let V be a set of symbols. A string on V is a finite sequence of symbols of V ,

$$\{x_i \in V\} \quad 1 \leq i \leq n \quad n \geq 0$$

Such a string may be represented as $x_1 x_2 \ldots x_n$. If n = 0 the sequence is called the empty string, which is represented by Λ .

If $\alpha = x_1 x_2 \ldots x_n$ is a string (on V), its length n is denoted by $|\alpha|$.

Let $\alpha = x_1 x_2 \ldots x_n$, $\beta = y_1 y_2 \ldots y_m$ be strings. Their concatenation $x_1 x_2 \ldots x_n y_1 y_2 \ldots y_m$ is denoted by $\alpha\beta$. If A and B are sets of strings then their product AB is the set of strings which consist of a string in A concatenated with a string in B, i.e.

$$AB = \{\alpha \beta \mid \alpha \in A, \beta \in B\}$$

The set of all strings on V is denoted by V*. This is the smallest set which satisfies $V^* = VV^* \cup \{\Lambda\}$. (V can be regarded as a set of strings, of length 1, on itself. The lengths of the strings in V* are unbounded, but finite.) V^+ is the smallest set satisfying $V^+ = VV^+ \cup V$; hence $V^+ = V^* \setminus \{\Lambda\}$, and is the set of strings on V which have positive length.

A relation r on a set A is a subset of A x A. We write (x,y) $\in$ r as x r y. The equality relation on A will be denoted where necessary by $=_A$. If r and s are relations on A we define the relation rs by

$$x \; rs \; z \quad \text{iff} \quad x \; r \; y \text{ and } y \; s \; z \quad \text{for some } y \in A$$

The reflexive transitive completion of r is denoted by r* and is the smallest relation satisfying $r^* = rr^* \cup =_A$. The transitive completion of r is denoted by r^+, which is the smallest relation satisfying $r^+ = rr^+ \cup r$. (If r is irreflexive we have $r^+ = r^* \setminus =_A$.)

A <u>context free grammar</u> (CFG) $\mathcal{G}$ is a 4-tuple, $\mathcal{G} = (V_N, V_T, S, P)$. V_N and V_T are finite, disjoint sets of symbols; $V_N \cap V_T = \emptyset$. We now let $V = V_N \cup V_T$ which is called the <u>vocabulary</u> of $\mathcal{G}$. The elements of V_N are called <u>nonterminals</u>, those of V_T are <u>terminals</u>. S is a distinguished member of V_N called the <u>principal nonterminal</u>. P is a finite subset of $V_N \times V^*$ whose elements are called the <u>productions</u> of $\mathcal{G}$. P is used to define the relation $\rightarrow$ on V^* as follows. Given $\alpha, \beta \in V^*$, we have

$\alpha \rightarrow \beta$ iff $\sigma, \tau \in V^*$ and $(A, \varphi) \in P$ such that $\alpha = \sigma A \tau$ and $\beta = \sigma \varphi \tau$.

This definition ensures that $P \subseteq \rightarrow$, enabling us to denote a production (A, φ) by $A \rightarrow \varphi$. A is called the <u>left hand side</u> (LHS) and φ the <u>right hand side</u> (RHS) of the production $A \rightarrow \varphi$. If $\alpha, \beta \in V^*$ and $\alpha \overset{*}{\rightarrow} \beta$ then we must have that $\omega_0, \ldots, \omega_n \in V^*$, $n \geq 0$ such that

$$\omega_0 = \alpha, \ \omega_n = \beta \quad \text{and} \quad \omega_i \rightarrow \omega_{i+1} \quad 0 \leq i < n$$

The sequence $\omega_0, \ldots, \omega_n$ is called a <u>derivation</u> of β from α, and will usually be written as $\omega_0 \rightarrow \omega_1 \rightarrow \ldots \rightarrow \omega_n$. If $S \overset{*}{\rightarrow} \alpha$, then α is called a <u>sentential form</u> of $\mathcal{G}$; in particular, if $\alpha \in V_T^*$ it is called a <u>sentence</u> of $\mathcal{G}$. The <u>language</u> generated by $\mathcal{G}$ is the set of all sentences of $\mathcal{G}$, denoted by $L(\mathcal{G})$.

$$L(\mathcal{G}) = \{\alpha \in V_T^* \mid S \overset{*}{\rightarrow} \alpha\}$$

If the following condition holds, then $\mathcal{G}$ is said to be a <u>reduced</u> CFG.

$$\forall \ X \in V \ \exists \ \alpha, \beta, \omega \in V_T^* \quad \text{such that} \quad S \overset{*}{\rightarrow} \alpha X \beta \overset{*}{\rightarrow} \omega$$

This condition ensures that no member of V (nor of P) is redundant for the purpose of deriving the sentences of $\mathcal{G}$.

A CFG is said to have an endmarker if S occurs in exactly one production, which has the form $S \rightarrow \perp^m S' \perp^n \quad m,n \geq 0$. ($x^0 = \Lambda$, $x^m = xx^{m-1} \quad m > 0$.) The endmarker $\perp \in V_T$ and occurs in no other production. $S' \in V_N$ and behaves similarly to the principal nonterminal, since in this case

$$L(\mathcal{G}) = \{ \perp^m \alpha \perp^n \mid \alpha \in V_T^* , S' \overset{*}{\rightarrow} \alpha \}$$

An arbitrary CFG can be amended so as to be reduced, and can be augmented with an endmarker, so we will restrict our attention to such reduced CFGs having endmarkers. They will be referred to simply as grammars.

Consider $\omega \in L(\mathcal{G})$. Since $S \overset{*}{\rightarrow} \omega$, we know there exists a derivation of ω from S, but this is not usually unique. If we insist that each stage of a derivation, the rightmost nonterminal is replaced, then the derivation is said to be canonical; this implies that each step is of the form

$$\alpha A \beta \rightarrow \alpha \varphi \beta \quad \text{where} \quad \alpha \in V^*, A \rightarrow \varphi, \beta \in V_T^*$$

If any sentence of $\mathcal{G}$ has two distinct canonical derivations then $\mathcal{G}$ is ambiguous. We define a parse to be synonomous with a canonical derivation. The problem of determining, for any string on V_T^* whether it is a sentence of $\mathcal{G}$, and if so specifying all parses of that sentence, is called the parsing problem for $\mathcal{G}$. An algorithm which solves this problem (for $\mathcal{G}$), is a parser (for $\mathcal{G}$).

Two general techniques are available, known as top-down and bottom-up parsers respectively. The top-down approach starts with the principal nonterminal and attempts to find a sequence of productions with which to derive the given terminal string. (This does not give the canonical derivation, but this is merely a matter of definition.)

A bottom-up parser adopts the opposite strategy of examining the terminal string and trying to determine the sequence of productions which must have been used to derive it. If successful, the canonical derivation is found (in reverse order). The parsing methods described in this dissertation are all of the bottom-up variety; it is for this reason that we make the above definition of a parse.

We will use the following representation for the productions of an arbitrary grammar. Let s be the number of productions in P, then

$$P = \{A_i \to X_{i1} X_{i2} \ldots X_{in_i} \mid 0 \le i \le s\}$$

Productions can be referred to by their indices in this scheme, thus the length of the RHS of production i is n_i, and therefore $n_i \ge 0$ for $0 \le i \le s$. For convenience we assume that $A_0 = S$, the principal nonterminal.

Knuth's LR(k) Algorithm

The methods of this dissertation are based on the bottom-up parsing technique described by Knuth (1965). To facilitate comparison, this section repeats that description. The following definitions are needed.

$$H_k(\alpha) = \{\beta \in V_T{*} \mid \alpha \overset{*}{\to} \beta,\ |\beta| < k \ \text{ or } \ \alpha \overset{*}{\to} \beta\gamma,\ |\beta| = k,\ \gamma \in V{*}\}$$

$$H'_k(\alpha) = \{\beta \in H_k(\alpha) \mid \text{no step in the derivation of } \beta \text{ from } \alpha \text{ is of the form } A\omega \to \omega \text{ where the leading nonterminal is replaced by } \Lambda \}$$

The fundamental notion of a state is denoted by $[p,j;\alpha]$, where $0 \le p \le s$, $0 \le j \le n_p$, $\alpha \in V_T{*}$. At any stage of parsing a terminal string, we will be in a stateset $\mathcal{S}$, which is a set of such states. If $[p,j;\alpha] \in \mathcal{S}$, this indicates that the first j symbols of the RHS of the p^{th} production have been recognised, and that if the production is completed, it could legitimately be followed by α.

Two weak constraints are imposed on grammars. Firstly, we require that $S' \overset{+}{\to} S'\alpha$ is not possible for any $\alpha \in V{*}$, and secondly that the 0^{th} production is taken to be $S \to S' \perp^k$ ($S \to S' \perp$ if $k = 0$); this is normal when considering LR(k) techniques. Denote the string to be parsed by $x_1 \ldots x_m$, whose last k symbols are $\perp^k$. Let i indicate the first uninspected symbol of this string, and begin parsing with $i = k + 1$.

During the parse, a stack of statesets $\mathcal{S}_0 \mathcal{S}_1 \ldots \mathcal{S}_n$ is maintained. The initial stateset $\mathcal{S}_0 = \{[0,0;\Lambda]\}$. With the stack as shown, the parser is in stateset $\mathcal{S} = \mathcal{S}_n$ and proceeds inductively to stateset $\mathcal{S}_{n+1}$ as follows.

Step 1

Define the closure $\mathcal{S}'$ of $\mathcal{S}$ recursively as the smallest set satisfying,

$$\mathcal{S}' = \mathcal{S} \cup \{ [q,0;\beta] \mid \exists [p,j;\alpha] \in \mathcal{S}', j<n_p, X_{p,j+1} = A_q \text{ and } \beta \in H_k(X_{p,j+2} \ldots X_{pn_p} \alpha) \}$$

Step 2

Compute the following sets:

$$Z = \{\beta \mid \exists [p,j;\alpha] \in \mathcal{S}', j<n_p, \beta \in H'_k(X_{p,j+1} \ldots X_{pn_p} \alpha)\}$$

$$Z_p = \{\alpha \mid [p,n_p;\alpha] \in \mathcal{S}'\}, 0 \leq p \leq s$$

We inspect the string $x_{i-k} \ldots x_{i-1} = \omega$. Assuming that the above sets are mutually disjoint, ω must lie in one of them, or the input string is invalid. If $\omega \in Z$, we let $Y = x_{i-k}$, increment i by one and continue with Step 3. If $\omega \in Z_p$, we decrement n by n_p, which removes n_p statesets from the stack, and let $Y = A_p$.

Step 3

Compute the next stateset $\mathcal{S}_{n+1}$ as

$$\mathcal{S}_{n+1} = \{ [p,j+1;\alpha] \mid [p,j;\alpha] \in \mathcal{S}'_n \text{ and } X_{p,j+1} = Y \}$$

If $\mathcal{S}_{n+1} = \{ [0,1;\Lambda] \}$ the parse is complete and i should be m+1. Otherwise we have completed an inductive step on n, and the algorithm proceeds from Step 1.

(This description is equivalent to Knuth's, except as regards notation.)

The closure operation on statesets specified in Step 1 adds to a stateset states of the form $[q,0;\beta]$. These states represent those productions whose RHS we could begin to recognise when in that stateset.

The string ω is the lookahead string; symbols to the left of ω in the input are regarded as having been read in. The elements of Z are those strings which indicate that no complete applicable RHS has yet been found, and that we must read in another symbol. This is done by adding one to i, and is called a shift operation. Strings in Z_p indicate that the RHS of production p has been recognised, and that we should remove n_p statesets from the stack. The removal of these stack elements is called a reduce p operation. Notice that the use of H'_k in the definition of Z permits the recognition of RHSs with length 0.

The stateset $\mathcal{S}_{n+1}$ contains states which represent either a new symbol read in, or the LHS of a production used to reduce the stack. It can be seen that with any stateset $\mathcal{S}$ generated by Step 3, a symbol $X \in V$ can be associated, with the property that, if $[p,j;\alpha] \in \mathcal{S}$ then $X_{pj} = X$. This symbol X is called the associated symbol of $\mathcal{S}$. Step 2 of the algorithm determines Y, which is the associated symbol of $\mathcal{S}_{n+1}$. A reduce p operation removes statesets from the stack which have as their associated symbols the RHS of production p. (The only stateset used in parsing which is not generated by Step 3 is the initial stateset. If necessary, we can regard Λ as an associated symbol for $\mathcal{S}_0$.)

The presence of the k endmarkers at the end of the input string ensures that the input is not exhausted, since they can only be valid lookahead symbols in the case that we are indeed parsing a sentence, and then the algorithm terminates with stateset $\{[0,1;\Lambda]\}$.

If in Step 2 we find that ω belongs to more than one of the sets Z, Z_p $0 \leq p \leq s$ then the algorithm fails, since it cannot resolve which operation is to be applied to determine Y. If we know that for every stateset which can arise when parsing any sentence generated by $\mathcal{G}$, application of Steps 1 and 2 gives rise to sets Z, Z_p $0 \leq p \leq s$ which are mutually disjoint, then $\mathcal{G}$ is said to be an <u>LR(k) grammar</u>.
A grammar which is LR(k) must be unambiguous, since the existence of two distinct canonical derivations would ensure the algorithm's failure. An LR(k) language is a language which can be generated by some LR(k) grammar.

We now consider a very simple example grammar $\mathcal{G}_1$, with productions

0 $\quad S \rightarrow A \perp$

1 $\quad A \rightarrow B\ d$

2 $\quad A \rightarrow e\ e$

s=3 $\quad B \rightarrow e$

Observe that this specifies, the principal nonterminal as S, V_N and V_T (and hence $\mathcal{G}_1$).

$$V_N = \{S,A,B\} \quad V_T = \{d,e,\perp\} \quad L(\mathcal{G}_1) = \{ee\perp,\ ed\perp\}$$

Suppose we wish to parse $ed\perp$, using the LR(1) algorithm.
We have $m = 3$, $i = 2$, $n = 0$, $\mathcal{S}_0 = \{[0,0;\Lambda]\}$
$[0,0;\Lambda] \in \mathcal{S}_0'$, $0 < n_0 = 2$, $X_{01} = A = A_1 = A_2$, $H_1(X_{02}) = \{\perp\}$
so we add $[1,0;\perp]$ and $[2,0;\perp]$ to $\mathcal{S}_0'$
$[1,0;\perp] \in \mathcal{S}_0'$, $0 < n_1 = 2$, $X_{11} = B = A_3$, $H_1(X_{12}\perp) = \{d\}$
so we add $[3,0;d]$ to $\mathcal{S}_0'$.

Since $X_{21} = e$ and $X_{31} = e$ no further additions are made from $[2,0;\bot]$ and $[3,0;d]$, $\mathcal{S}'_0$ is completed.

$$\mathcal{S}'_0 = \{\underset{A}{[0,0;\Lambda]},\ \underset{B}{[1,0;\bot]},\ \underset{e}{[2,0;\bot]},\ \underset{e}{[3,0;d]}\}$$

where beneath $[p,j;\alpha]$ we have written $X_{p,j+1}$.

Clearly $Z = \{e\}$ and $Z_p = \emptyset \quad 0 \le p \le 3$

Since $i = 2$, $\omega = e$ and $\omega \in Z$. So let $i = 3$, $Y = e$ n becomes 1 and

$\mathcal{S}_n = \{[2,1;\bot], [3,1;d]\}$

$\mathcal{S}'_n = \mathcal{S}_n$, and now $Z = \{e\}$, $Z_3 = \{d\}$, $Z_p = \emptyset \quad 0 \le p \le 2$

$i = 3$, $\omega = d$ and $\omega \in Z_3$. n is reduced by $n_3 = 1$. So $n = 0$, $Y = A_3 = B$

n becomes 1 and $\mathcal{S}_n = \{[1,1;\bot]\}$ (computed from $\mathcal{S}'_0$)

$\mathcal{S}'_n = \mathcal{S}_n$, $Z = \{d\}$, $Z_p = \emptyset \quad 0 \le p \le 3$

$i = 3$, $\omega = d$, $\omega \in Z$. So let $i = 4$, $Y = d$

n becomes 2 $\mathcal{S}_n = \{[1,2;\bot]\}$ $\mathcal{S}'_n = \mathcal{S}_n$, $Z_1 = \{\bot\}$ $Z_0 = Z_2 = Z_3 = Z = \emptyset$

$i = 4$, $\omega = \bot$, $\omega \in Z_1$ n is reduced by $n_1 = 2$. So $n = 0$, $Y = A_1 = A$

n becomes 1 $\mathcal{S}_n = \{[0,1;\Lambda]\}$ and the parse is complete with $i = m + 1$.

The sequence of reduce operations was reduce 3, reduce 1. When taken in reverse order, and preceded by a reduce 0 (which is always the initial step in a derivation), they specify the parse

$$S \to A\bot \to Bd\bot \to ed\bot$$

The only other stateset which can arise (when parsing $ee\bot$) is

$\mathcal{S} = \{[2,2;\bot]\}$, for which $\mathcal{S}' = \mathcal{S}$, $Z_2 = \{\bot\}$, $Z_0 = Z_1 = Z_3 = Z = \emptyset$.

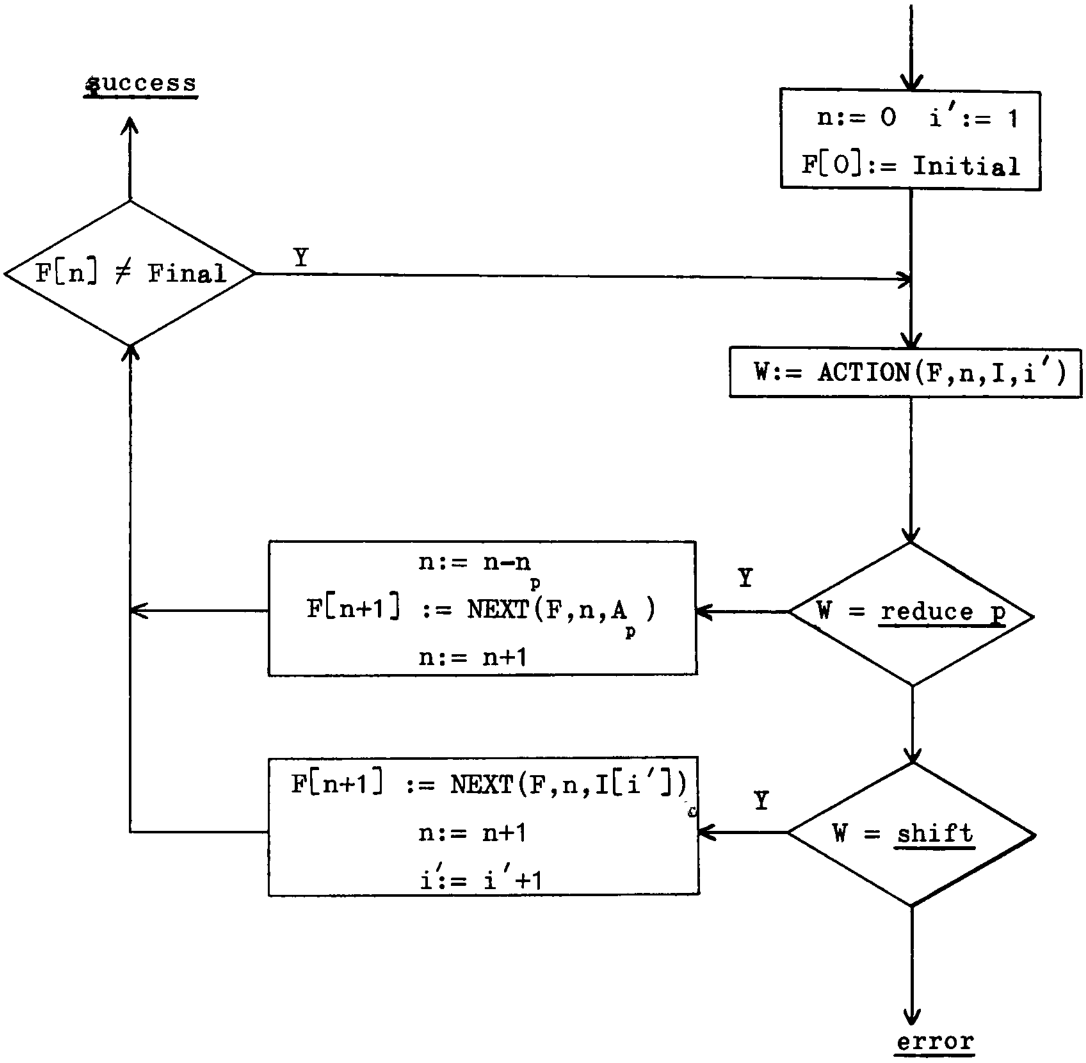
This shows that $\mathcal{G}_1$ is LR(1).

Let $\mathcal{G}'_1$ have productions $S \to A\bot$ $A \to e\ d$ $A \to e\ e$ and let $\mathcal{G}''_1$ have productions $S \to A\bot\bot$ $A \to B\ e\ d$ $A \to C\ e\ e$ $B \to \Lambda$ $C \to \Lambda$.

Then $L(\mathcal{G}_1)\bot = L(\mathcal{G}'_1)\bot = L(\mathcal{G}''_1)$, but $\mathcal{G}'_1$ is LR(0) and $\mathcal{G}''_1$ is LR(2).

Knuth showed that for any language which has an LR(k) grammar (with endmarker) we can find an LR(0) grammar generating the language.

Table Driven Parsers



The above flow diagram describes a generalised bottom-up parser which will serve as a basis for all the algorithms discussed here. F is a stack to which n is the pointer; Initial is the first element on this stack. The input string is now denoted by I, and i' indicates the first unread symbol. The procedure ACTION determines at each stage in the parse what the next operation of the parser should be; procedure NEXT computes the next element to be placed on the stack. The algorithm terminates when Final is first placed on the stack.

W is merely a work variable.

The contents of the flow diagram boxes are written in pseudo Algol, in which the algorithm may be programmed as

```
n:= 0;  i':= 1;  F[0]:= Initial;  W:= ACTION(F,0,I,1);
while  W = reduce p or W = shift  do
    begin if  W = shift  then
        begin F[n+1]:= NEXT(F,n,I[i']); n:= n+1; i':= i'+1 end
    else
        begin n:= n-$n_p$ ; F[n+1]:= NEXT(F,n,$A_p$ ); n:= n+1 end;
    W:= if  F[n] = Final then success else ACTION(F,n,I,i')
    end;
```

This algorithm will parse using precedence methods, which are to be discussed later, or it can perform LR(k) parsing if we make the following specification.

Stack elements are statesets, with Initial = S_0 and Final = $\{[0,1;\Lambda]\}$
$i' = i-k$, ACTION determines W from F[n] and I[i′] ... I[i-1]
(the lookahead string), by means of Steps 1 and 2. NEXT computes the next stateset from F[n] and either A_p or I[i′] as described in Step 3.

The restriction of the domain of ACTION to 1 stack element and k input symbols, and that of NEXT to 1 stack element and 1 symbol, in the LR(k) method, together with the fact of there only being a finite number of possible statesets and lookahead strings, enables us to consider computing a table providing the results of ACTION and NEXT for all values of their parameters.

The efficiency of the parser is greatly improved by such a table since the calculation involved in Steps 1, 2 and 3 is replaced by a simple table look-up mechanism. A parser using such techniques is said to be table driven.

LR(k) Parsing Tables

Knuth's algorithm can be modified to produce tables which will drive the above parser, and this was described in detail by Korenjak (1969). We give an alternative description.

Let $\mathcal{S}$ be a stateset (for some grammar $\mathcal{G}$). The β successor of $\mathcal{S}$ is denoted by $\mathcal{S}\beta$ and defined for all $\beta \in V^*$ as follows.

$$\mathcal{S}Y = \{[p, j+1; \alpha] \mid [p, j; \alpha] \in \mathcal{S}', \ j < n_p, \ X_{p, j+1} = Y\} \quad Y \in V$$

$$\mathcal{S}\Lambda = \mathcal{S} \quad \text{and} \quad \mathcal{S}\beta = (\mathcal{S}Y)\gamma \text{ where } \beta = Y\gamma$$

Observe that if $\mathcal{S}$ is a stateset which can arise during the parsing of a sentence of $\mathcal{G}$, then $\exists\, \alpha \in V^*$ such that $\mathcal{S} = \mathcal{S}_0\, \alpha$ (take α as the sequence of associated symbols of the statesets on the stack when $\mathcal{S}$ is first placed on the stack). So we may define the stateset table $\mathcal{T}$ (of $\mathcal{G}$) as the smallest family of statesets satisfying

$$\mathcal{T} = \{\, \mathcal{S}_0 \} \cup \{\, \mathcal{S}Y \mid \mathcal{S} \in \mathcal{T}, \ Y \in V\}$$

For any grammar, $\mathcal{S}_0$, $\{[0,1;\Lambda]\}$, $\emptyset$ are elements of $\mathcal{T}$. $\emptyset$ has the property $\emptyset Y = \emptyset \ \forall\, Y \in V$ and corresponds to the error situation of the input not being a sentence of $\mathcal{G}$.

$\mathcal{T}$ can be formed by the following iterative method. Let T be the statesets $\{\mathcal{S}_0, \ldots, \mathcal{S}_n\}$, of which j have been considered. Initially n = j = 0.

```
While $j \leq n$ perform the following,
    Compute $\mathcal{S}_j'$ and from it $\mathcal{S}_j Y$ for each $Y \in V$.
    If $\mathcal{S}_j Y \notin T$ then add $\mathcal{S}_j Y$ to T as $\mathcal{S}_{n+1}$ and increase n by one.
    Increase j by one.
```

On completion of the above we will have $T = \mathcal{T}$, and we can use the term stateset j to refer to $\mathcal{S}_j$ in the table T.

Corresponding to each stateset $\mathcal{S}$ in $\mathcal{T}$ we require a parsing-state $R(\mathcal{S})$ (which should not be confused with the states in a stateset).

If Z, Z_p $0 \leq p \leq s$ are as specified in Step 2 of the LR(k) algorithm, then

$$\begin{aligned} R(\mathcal{S}) = & \{(\alpha, \text{reduce } p) \mid \alpha \in Z_p\} \\ & \cup \{(\alpha, \text{shift}) \mid \alpha \in Z\} \\ & \cup \{(Y, \text{goto } i) \mid \exists\, [p,j;\alpha] \in \mathcal{S}',\ j < n_p,\ Y = X_{p,j+1},\ \mathcal{S}Y = \mathcal{S}_i\} \end{aligned}$$

Shift and reduce type pairs are used to determine ACTION, while those of the goto type determine NEXT. Clearly, the LR(k) parsing table (for $\mathcal{G}$) is given by

$$\{R(\mathcal{S}) \mid \mathcal{S} \in \mathcal{T}\}.$$

A stateset $\mathcal{S}$ is said to be adequate if the following conditions hold.

$$(\alpha, \text{shift}),\ (\beta, \text{reduce } p) \in R(\mathcal{S}) \text{ must imply } \alpha \neq \beta$$

$$(\beta, \text{reduce } p),\ (\beta, \text{reduce } q) \in R(\mathcal{S}) \text{ must imply } p = q$$

Any stateset for which one of these conditions does not hold is said to be inadequate. A grammar is LR(k) iff every stateset in $\mathcal{T}$ is adequate. If we have computed the parsing table, we need not retain information about the statesets in $\mathcal{T}$, and they may be discarded.

The LR(1) stateset table and parsing table for $\mathcal{G}_1$ are given as an example.

$\mathcal{S}_0 = \{[0,0;\Lambda]\} \quad \mathcal{S}_0\bot = \mathcal{S}_0 c = \mathcal{S}_0 d = \mathcal{S}_1 \quad \mathcal{S}_0 A = \mathcal{S}_2 \quad \mathcal{S}_0 B = \mathcal{S}_3 \quad \mathcal{S}_0 e = \mathcal{S}_4$

$\mathcal{S}_1 = \emptyset \quad \mathcal{S}_1 Y = \mathcal{S}_1 \ \forall \ Y \in V$

$\mathcal{S}_2 = \{[0,1;\Lambda]\} \quad \mathcal{S}_2\bot = \mathcal{S}_5 \quad \mathcal{S}_2 Y = \mathcal{S}_1 \ \forall \ Y \in V \setminus \{\bot\}$

$\mathcal{S}_3 = \{[1,1;\bot]\} \quad \mathcal{S}_3 d = \mathcal{S}_6 \quad \mathcal{S}_3 Y = \mathcal{S}_1 \ \forall \ Y \in V \setminus \{d\}$

$\mathcal{S}_4 = \{[2,1;\bot],[3,1;d]\} \quad \mathcal{S}_4 e = \mathcal{S}_7 \quad \mathcal{S}_4 Y = \mathcal{S}_1 \ \forall \ Y \in V \setminus \{e\}$

$\mathcal{S}_5 = \{[0,2;\Lambda]\} \quad \mathcal{S}_5 Y = \mathcal{S}_1 \ \forall \ Y \in V$

$\mathcal{S}_6 = \{[1,2;\bot]\} \quad \mathcal{S}_6 Y = \mathcal{S}_1 \ \forall \ Y \in V$

$\mathcal{S}_7 = \{[2,2;\bot]\} \quad \mathcal{S}_7 Y = \mathcal{S}_1 \ \forall \ Y \in V$

$R(\mathcal{S}_0) = \{(e,\text{shift}), (A,\text{goto } 2), (B,\text{goto } 3), (e,\text{goto } 4)\}$

$R(\mathcal{S}_1) = \emptyset$

$R(\mathcal{S}_2) = \{(\bot,\text{shift}), (\bot,\text{goto } 5)\}$

$R(\mathcal{S}_3) = \{(d,\text{shift}), (d,\text{goto } 6)\}$

$R(\mathcal{S}_4) = \{(d,\text{reduce } 3), (e,\text{shift}), (e,\text{goto } 7)\}$

$R(\mathcal{S}_5) = \{(\Lambda,\text{reduce } 0)\}$

$R(\mathcal{S}_6) = \{(\bot,\text{reduce } 1)\}$

$R(\mathcal{S}_7) = \{(\bot,\text{reduce } 2)\}$

Three of these parsing-states, $R(\mathcal{S}_1)$, $R(\mathcal{S}_2)$ and $R(\mathcal{S}_5)$ are not used by the parser. $\mathcal{S}_1$ is the error stateset and $\mathcal{S}_2$ the final stateset; on these the algorithm halts. $R(\mathcal{S}_2)$ and $R(\mathcal{S}_5)$ are not required, since they merely read the endmarker and recognise production 0 respectively.

If we had defined $\mathcal{T}$ by the equation

$$\mathcal{T} = \{\mathcal{S}_0\} \cup \{ \mathcal{S}Y \mid \mathcal{S} \neq \{[0,1;\Lambda]\}, \mathcal{S} \in \mathcal{T}, \ Y \in V\},$$

statesets having $\bot$ as their associated symbol would have been eliminated, which is convenient in practice.

Suppose (x,goto i) $\in R(\mathcal{S})$ in some LR(k) parsing table with $x \in V_T$. If $k > 0$ then

$$(x\alpha_1,\text{shift}), \ \ldots\ , (x\alpha_n,\text{shift}) \in R(\mathcal{S}) \quad \alpha_j \in V_T^*,\ |\alpha_j| = k-1, 1 \le j \le n$$
$$n \ge 1 \text{ (if } k = 1 \text{ then } n = 1)$$

If $k = 0$ we have $(\Lambda,\text{shift}) \in R(\mathcal{S})$. These entries specify that a shift operation is to be performed and that the stateset which is next to be entered is $\mathcal{S}_i$. A more convenient representation would be to combine the entries as

$$(x\alpha_1,\text{shift } i), \ \ldots\ , (x\alpha_n,\text{shift } i) \quad \text{if } k \ge 1$$
$$(x,\text{shift } i) \quad \text{if } k \le 1$$

Certainly this does not reduce the information contained in $R(\mathcal{S})$.

We can formalise this by defining

$$\begin{aligned} R'(\mathcal{S}) = {} & \{(\alpha,\text{reduce } p) \mid \alpha \in Z_p\} \\ & \cup\{(\alpha,\text{shift } i) \mid \alpha = x\gamma,\ \alpha \in Z',\mathcal{S}x = \mathcal{S}_i\} \\ & \cup\{(A,\text{goto } i) \mid \exists\,[p,j;\alpha] \in \mathcal{S}',j < n_p, A = X_{p,j+1}, A \in V_N, \mathcal{S}A = \mathcal{S}_i\} \end{aligned}$$

where

$$\begin{aligned} Z' = \{\beta \mid \exists\,[p,j;\alpha] \in \mathcal{S}',j < n_p,\ & \beta = x\gamma, x = X_{p,j+1}, x \in V_T, \\ & \text{if } k = 0 \text{ then } \gamma = \Lambda \text{ else } \gamma \in H_{k-1}(X_{p,j+2} \ldots X_{pn_p}\alpha)\} \end{aligned}$$

In appendix 1 (1.1) we show that when $k \ge 1$ we have $Z' = Z$ (defined in Step 2 of the Knuth algorithm). The definition of Z' simplifies the calculation of Z (particularly so when $k = 1$), and avoids the use of sets $H'_k(\alpha)$.

Our use of $R'(\mathbb{S})$ to provide an alternative (preferable) parsing table indicates the way in which k = 1 and k = 0 can be considered as special cases of the LR(k) algorithm. When k = 1, the (x,shift) and (x,goto i) entries in $R(\mathbb{S})$ correspond exactly; when k = 0, because the lookahead is now less than the single terminal used to determine the next stateset, (Λ,shift) corresponds to all the terminal goto entries. Moving to $R'(\mathbb{S})$ for an LR(0) parsing table gives shift entries the appearance of being LR(1) in nature (which they are not - if k = 0 and $R'(\mathbb{S})$ contains shift entries and reduce entries then it is inadequate).

<u>Table Sizes</u>

In this section we discuss the way in which stateset tables increase in size with k. The example grammar $\mathcal{G}_1$ already considered gives no indication of this since an LR(0) table for $\mathcal{G}_1$ has the same number of statesets as its LR(1) table. The grammar $\mathcal{G}_2$, which has productions

$$\begin{array}{llll} 0 & S & \rightarrow & A \perp \\ 1 & A & \rightarrow & a\ A\ b \\ s=2 & A & \rightarrow & a \end{array}$$

has LR(0) statesets

$\mathbb{S}_0 = \{[0,0]\}$ $\mathbb{S}_1 = \emptyset$ $\mathbb{S}_2 = \{[0,1]\}$ $\mathbb{S}_3 = \{[1,1],[2,1]\}$ $\mathbb{S}_4 = \{[0,2]\}$
$\mathbb{S}_5 = \{[1,2]\}$ $\mathbb{S}_6 = \{[1,3]\}$ (where we abbreviate $[p,j;\Lambda]$ by $[p,j]$),

and LR(1) statesets

$\mathbb{S}_1 = \{[0,0;\Lambda]\}$ $\mathbb{S}_1 = \emptyset$ $\mathbb{S}_2 = \{[0,1;\Lambda]\}$ $\mathbb{S}_3 = \{[1,1;\perp],[2,1;\perp]\}$
$\mathbb{S}_4 = \{[0,2;\Lambda]\}$ $\mathbb{S}_5 = \{[1,2;\perp]\}$ $\mathbb{S}_6 = \{[1,3;\perp]\}$
$\mathbb{S}_7 = \{[1,1;b],[2,1;b]\}$ $\mathbb{S}_8 = \{[1,2;b]\}$ $\mathbb{S}_9 = \{[1,3;b]\}$

The three additional statesets in the LR(1) table are due to the repetition of statesets $\mathcal{S}_3$, $\mathcal{S}_5$, $\mathcal{S}_6$ as statesets $\mathcal{S}_7$, $\mathcal{S}_8$, $\mathcal{S}_9$ with different right contextual information. The following lemma shows that this type of behaviour is always the case.

First define $H_k(\mathcal{S})$ for a stateset $\mathcal{S}$ by

$$H_k(\mathcal{S}) = \{[p,j;\alpha] \mid [p,j;\beta] \in \mathcal{S}, \quad \alpha \in H_k(\beta)\}$$

Lemma 1

Let $\mathcal{T}_m$ be an LR(m) stateset table. Then the LR(k) stateset table $\mathcal{T}_k$, for the same grammar, where $k \leq m$, is such that

$$\mathcal{T}_k = \{H_k(\mathcal{S}) \mid \mathcal{S} \in \mathcal{T}_m\}.$$

Let $\mathcal{S}_{0m} \in \mathcal{T}_m$ and $\mathcal{S}_{0k} \in \mathcal{T}_k$ be the respective initial statesets.

Certainly $H_k(\mathcal{S}_{0m}) = \mathcal{S}_{0k}$.

Suppose $\mathcal{S}_m \in \mathcal{T}_m$ and $\mathcal{S}_k \in \mathcal{T}_k$ with $H_k(\mathcal{S}_m) = \mathcal{S}_k$.

Then $H_k(\mathcal{S}'_m) = \mathcal{S}'_k$ and thus clearly $H_k(\mathcal{S}_m Y) = \mathcal{S}_k Y$, $\forall\ Y \in V$.

Now, for any $\mathcal{S} \in \mathcal{T}_m$, $\mathcal{S} = \mathcal{S}_{0m}\alpha$ for some $\alpha \in V^*$, and so

$$H_k(\mathcal{S}) = H_k(\mathcal{S}_{0m}\alpha) = \mathcal{S}_{0k}\alpha \in \mathcal{T}_k$$

$$\therefore \{H_k(\mathcal{S}) \mid \mathcal{S} \in \mathcal{T}_m\} \subseteq \mathcal{T}_k$$

Similarly, for any $\mathcal{S} \in \mathcal{T}_k$, $\mathcal{S} = \mathcal{S}_{0k}\alpha$ for some $\alpha \in V^*$, and so

$$\mathcal{S} = \mathcal{S}_{0k}\alpha = H_k(\mathcal{S}_{0m}\alpha) \text{ and } \mathcal{S}_{0m}\alpha \in \mathcal{T}_m$$

$$\therefore \mathcal{T}_k \subseteq \{H_k(\mathcal{S}) \mid \mathcal{S} \in \mathcal{T}_m\}$$

and we have our result.

The lemma indicates how an LR(k+1) table can be considered to be built up from an LR(k) table, each of the LR(k) statesets being refined, by additional right contextual information, to a number of LR(k+1) statesets. Also, we see the way in which the increase in the number of statesets can be exponential with k.

The importance of lemma 1 will become apparent in the next section, to which it is basic. A more detailed proof is given in appendix 1(1.2).

It is reasonable to ask why we should be concerned with $k > 0$ at all since, as already noted, all LR(k) languages are LR(0). The reason is to be found in the size of the LR(0) grammars required and of the transformations needed to produce them. Grammars which arise naturally as models for programming languages are not normally LR(0), and for such grammars even an LR(1) parsing table is usually prohibitively large. To put this in perspective, a run of a program to generate an LR(1) table for an Algol 60 grammar was terminated when the 10000th parsing-state entry was produced. At this point, over 1200 statesets had been generated.

Recently, Pager (1970) and Aho and Ullman (to be published) have considered ways of reducing the magnitude of LR(k) tables. Pager minimises the number of statesets required, but does not maintain the error detection capabilities of the LR(k) algorithm (which are to be discussed subsequently); Aho and Ullman are developing a formal treatment of the reduction of LR(k) tables by such techniques as eliminating unnecessary entries, and merging compatible parsing-states.

An alternative approach, adopted by Korenjak (1969) and DeRemer (1971), is to develop modifications to the original LR(k) algorithm which give rise to smaller tables. Korenjak suggests partitioning the grammar into a number of smaller parts and using an LR(k) subparser for each part; DeRemer extends the capability of an LR(0) parser to give an algorithm which he calls SLR(k).

A major drawback to the straightforward LR(k) scheme is that the parameter k performs two functions. It specifies both the amount of right context to be used in the formation of the statesets, and also, the amount of lookahead which will be permitted at parse time. Thus, for the grammar G_2, which is not LR(0), we must produce an LR(1) table having three more statesets than are necessary. The reason for the LR(0) table's inadequacy is to be found in $S_3 = \{[1,1],[2,1]\}$. Here we have

$S'_3 = \{[1,1],[2,1],[1,0],[2,0]\}$,

$R'(S_3) = \{(\Lambda, \text{reduce } 2), (a, \text{shift } 3), (A, \text{goto } 5)\}$

and cannot tell whether the reduce or shift operation is to be applied. This is rectified in the LR(1) table where

$S'_3 = \{[1,1;\perp],[2,1;\perp],[1,0;b],[2,0;b]\}$,

$R'(S_3) = \{(\perp, \text{reduce } 2), (a, \text{shift } 7), (A, \text{goto } 5)\}$

and for the duplicate state S_7,

$S'_7 = \{[1,1;b],[2,1;b],[1,0;b],[2,0;b]\}$,

$R'(S_7) = \{(b, \text{reduce } 2), (a, \text{shift } 7), (A, \text{goto } 8)\}$.

It can now be seen that replacing the (Λ,reduce 2) entry in the LR(0) table by entries (b,reduce 2) and (⊥,reduce 2), would have been sufficient to provide a parser, without incurring the overhead of statesets S_7, S_8 and S_9. These extra statesets, in fact make no contribution to the resolution of the LR(0) inadequacy, which is due entirely to the provision of one symbol lookahead for the reduce 2 operation. This observation suggests a parser having an LR(0) stateset table, but with one symbol of lookahead added subsequently to the parsing table.

In the next section we consider the generalisation of this to the calculation of LR(k) stateset tables from which can be formed m symbol lookahead parsing tables.

LA(m)LR(k) Parsing Tables

Informally, an LA(m)LR(k) parsing table is based on an LR(k) stateset table, but its m symbol lookahead may be considered to be derived from an LR(m) stateset table. Because of this, the lookahead is correct in the sense that none of the context is redundant; no better m symbol lookahead information could be used. We will normally require $m \geq k$ when considering LA(m)LR(k) techniques; the situation when $m \leq k$ is equivalent to LR(m) by virtue of lemma 1.

We define an equivalence relation $\sim$ on the members of an LR(m) stateset table by

$$\mathcal{S}_1 \sim \mathcal{S}_2 \quad \text{iff} \quad H_k(\mathcal{S}_1) = H_k(\mathcal{S}_2),$$

and denote the equivalence classes induced under $\sim$ by $B_0, \ldots, B_r$ (for consistency, let $\mathcal{S}_0 \in B_0$, and then $B_0 = \{\mathcal{S}_0\}$). Take the LA(m)LR(k) parsing states to be specified by

$$\begin{aligned} R(B_i) = {} & \{(\alpha,\text{reduce } p) \mid (\alpha,\text{reduce } p) \in R'(\mathcal{S}), \mathcal{S} \in B_i\} \\ & \cup \{(\alpha,\text{shift } j) \mid (\alpha,\text{shift } l) \in R'(\mathcal{S}), \mathcal{S} \in B_i,\ \mathcal{S}_l \in B_j\} \\ & \cup \{(A,\text{goto } j) \mid (A,\text{goto } l) \in R'(\mathcal{S}), \mathcal{S} \in B_i,\ \mathcal{S}_l \in B_j\} \end{aligned}$$

for $0 \leq i \leq r$. Equivalently, we could form an LA(m)LR(k) stateset table from an LR(m) table by

$$\mathcal{S}_i = \{[p,j;\alpha] \mid [p,j;\alpha] \in \mathcal{S}, \mathcal{S} \in B_i\} \quad 0 \leq i \leq r$$

and form a parsing table from this almost as though it were an LR(m) stateset table. The only difference is that $\mathcal{S}Y$ is no longer necessarily a stateset in the table. There will certainly exist $\mathcal{S}_j$ with $\mathcal{S}_j \sim \mathcal{S}Y$, and we can show (using the same argument as lemma 2 below) that it is unique. Hence this $\mathcal{S}_j$ may be taken as the Y successor of $\mathcal{S}$.

Suppose $\mathcal{S}_1 \sim \mathcal{S}_2$. This means that $H_k(\mathcal{S}_1) = H_k(\mathcal{S}_2) = \mathcal{S}$ where $\mathcal{S}$ is a member of the LR(k) stateset table. We know that $H_k(\mathcal{S}_1 Y) = \mathcal{S}Y$ (see lemma 1) and similarly $H_k(\mathcal{S}_2 Y) = \mathcal{S}Y$. So $H_k(\mathcal{S}_1 Y) = H_k(\mathcal{S}_2 Y)$ and therefore $\mathcal{S}_1 Y \sim \mathcal{S}_2 Y$. This will be needed in the proof of the following lemma.

Lemma 2

If (xα,shift i) and (xβ,shift j) are entries in the same LA(m)LR(k) parsing-state, then i = j. If (A,goto i) and (A,goto j) are entries in the same LA(m)LR(k) parsing-state, then i = j.

First, let (xα,shift i), (xβ,shift j) $\in R(B_l)$

then (xα,shift m) $\in R'(\mathcal{S}_1)$, $\mathcal{S}_1 \in B_l$, $\mathcal{S}_m \in B_i$

and (xβ,shift n) $\in R'(\mathcal{S}_2)$, $\mathcal{S}_2 \in B_l$, $\mathcal{S}_n \in B_j$

We can deduce $\mathcal{S}_1 \sim \mathcal{S}_2 \therefore \mathcal{S}_m = \mathcal{S}_1 x \sim \mathcal{S}_2 x = \mathcal{S}_n$

and $\mathcal{S}_m \sim \mathcal{S}_n$ implies i = j.

Similarly, if (A,goto i), (A,goto j) $\in R(B_l)$

then (A,goto m) $\in R'(\mathcal{S}_1)$, $\mathcal{S}_1 \in B_l$, $\mathcal{S}_m \in B_i$

and (A,goto n) $\in R'(\mathcal{S}_2)$, $\mathcal{S}_2 \in B_l$, $\mathcal{S}_n \in B_j$

We can deduce $\mathcal{S}_1 \sim \mathcal{S}_2 \therefore \mathcal{S}_m = \mathcal{S}_1 A \sim \mathcal{S}_2 A = \mathcal{S}_n$

and $\mathcal{S}_m \sim \mathcal{S}_n$ implies i = j.

Thus the goto and shift entries in the LA(m)LR(k) parsing table are as they should be, the next parsing-state for any symbol in V being unique.
The underlying reason for lemma 2 is, of course, that the equivalence classes correspond (under H_k) precisely to the statesets of the LR(k) table.

If the conditions

$$(\alpha,\text{shift } j),\ (\beta,\text{reduce } p) \in R(B_i) \text{ must imply } \alpha \neq \beta$$

$$(\beta,\text{reduce } p),\ (\beta,\text{reduce } q) \in R(B_i) \text{ must imply } p = q$$

hold for $0 \leq i \leq r$, then the grammar in question is said to be <u>LA(m)LR(k)</u>.
Clearly, if a grammar is LR(k) it will also be LA(m)LR(k) for any $m \geq k$.
We have that

$$\text{LA(k)LR(k)} = \text{LR(k)} \subseteq \text{LA(m)LR(k)} \subseteq \text{LR(m)} = \text{LA(m)LR(m)}$$

where the inclusions are strict if m > k (we use LA(m)LR(k) here as an abbreviation for the class of grammars which are LA(m)LR(k)).

Despite the above definition, it is not necessary to form an LR(m) table as a first step in computing an LA(m)LR(k) parsing table. We now give two algorithms of a more practical nature.

The first of these behaves initially like an LR(m) stateset algorithm, except that a newly generated stateset is only added to the table if no equivalent (under ~) stateset is already in the table. If there is such an equivalent stateset in the table and it does not contain the new stateset, then the two are merged, to form their union, which replaces the equivalent stateset. When all statesets have been considered we return to those members of the table which were merged. Their successor statesets are recomputed and if necessary merged with their original versions. This is continued until no merged stateset has not subsequently had its successor statesets recomputed. We can specify this more precisely as:

```
Let T be initialised as $\{\mathcal{S}_0\}$, with $\mathcal{S}_0$ marked, and let $n = 0$.
Repeat the following until no stateset in T remains marked.
    Set j to 0.
    While $j \leq n$ perform the following.
      If $\mathcal{S}_j$ is marked, first remove the mark, then compute $\mathcal{S}'_j$
      (as an LR(m) closure), and from $\mathcal{S}'_j$ compute $\mathcal{S}_j Y \;\forall\; Y \in V$.
      If $\exists\, \mathcal{S}_i \in T$ with $\mathcal{S}_i \sim \mathcal{S}_j Y$ and $\mathcal{S}_i \not\supseteq \mathcal{S}_j Y$, then replace $\mathcal{S}_i$
      by $\mathcal{S}_i \cup \mathcal{S}_j Y$ and mark this new $\mathcal{S}_i$.
      If $\nexists\, \mathcal{S}_i \in T$ with $\mathcal{S}_i \sim \mathcal{S}_j Y$, then add $\mathcal{S}_j Y$ to T as $\mathcal{S}_{n+1}$,
      mark $\mathcal{S}_{n+1}$, and increase n by one.
      Increase j by one.
```

On completion, T is the LA(m)LR(k) stateset table.

A mark on a stateset indicates that further computation is required for that stateset. The first run of j from 0 to n corresponds to the initial formation of the table; only during this stage can we have $\nexists\, \mathcal{S}_i \in T$ with $\mathcal{S}_i \sim \mathcal{S}_j Y$. Termination is assured since merging enlarges a stateset, and there are only a finite number of states (and thus a finite number of statesets).

The first stage of this LA(1)LR(0) algorithm applied to $\mathcal{G}_2$ yields

$\mathcal{S}_0 = \{[0,0;\Lambda]\}$ $\mathcal{S}_1 = \emptyset$ $\mathcal{S}_2 = \{[0,1;\Lambda]\}$ $\mathcal{S}_3 = \{[1,1;\perp],[2,1;\perp],[1,1;b],[2,1;b]\}$
$\mathcal{S}_4 = \{[0,2;\Lambda]\}$ $\mathcal{S}_5 = \{[1,2;\perp]$ $\mathcal{S}_6 = \{[1,3;\perp]\}$

with $\mathcal{S}_3$ marked due to $\{[1,1;b],[2,1;b]\}$ having been merged in.
Recomputing successor statesets for $\mathcal{S}_3$ merges (and marks) $\mathcal{S}_5$ to $\{[1,2;\perp],[1,2;b]\}$.
Recomputing successor statesets for $\mathcal{S}_5$ merges $\mathcal{S}_6$ to $\{[1,3;\perp],[1,3;b]\}$.
The only successor stateset to $\mathcal{S}_6$ is $\emptyset$ so we are done.

The parsing table is

$R'(\mathcal{S}_0) = \{(a,\text{shift } 3),\ (A,\text{goto } 2)\}$
$R'(\mathcal{S}_3) = \{(\perp,\text{reduce } 2),\ (b,\text{reduce } 2),\ (a,\text{shift } 3),\ (A,\text{goto } 5)\}$
$R'(\mathcal{S}_5) = \{(b,\text{shift } 6)\}$
$R'(\mathcal{S}_6) = \{(\perp,\text{reduce } 1),(b,\text{reduce } 1)\}$
($\mathcal{S}_1$, $\mathcal{S}_2$ and $\mathcal{S}_4$ being irrelevant to this table).

The second method begins with an LR(k) stateset table and extends the lookahead string of each state to m symbol strings. Suppose $[p,j;\alpha] \in \mathcal{S}_1'$ where $\mathcal{S}_1$ is a member of an LR(k) stateset table. We wish to determine those strings of length m, which begin with α and can validly follow the occurrences of production p to which $[p,j;\alpha]$ refers. Denote this set of strings by $R_m([p,j;\alpha],\mathcal{S}_1)$. Let $\mathcal{S}_2$ be any stateset for which $\mathcal{S}_2 Y = \mathcal{S}_1$ where Y is the associated symbol of $\mathcal{S}_1$; such statesets are called <u>predecessors</u> of $\mathcal{S}_1$. If $j > 0$ then clearly

$R_m([p,j;\alpha],\mathcal{S}_1) = \{\alpha' \in R_m([p,j-1;\alpha],\mathcal{S}_2) \mid \mathcal{S}_2$ is a predecessor of $\mathcal{S}_1\}$. If $j = 0$ then we are interested in those members of $\mathcal{S}'_1$ which caused us to begin the recognition of production p.

$$R_m([p,0;\alpha],\mathcal{S}_1) = \{\alpha' | [q,l;\beta] \in \mathcal{S}'_1,\ X_{q,l+1} = A_p, \alpha' \in H_m(X_{q,l+2} \dots X_{qn_q} \beta'),$$
$$\beta' \in R_m([q,l;\beta],\mathcal{S}_1),\ \alpha' = \alpha\gamma \text{ for some } \gamma \in V_T^*\}$$

$R_m([0,0;\Lambda],\mathcal{S}_0)$ is taken to be $\{\Lambda\}$ as a special case (instead of $\emptyset$).

These equations admit the possibility of circularity, i.e. the evaluation of $R_m([p,j;\alpha],\mathcal{S})$ invoking its own re-evaluation. In these circumstances, to obtain a finite algorithm, we instead re-evaluate $R_{m-1}([p,j;\alpha],\mathcal{S})$ (taking $R_0([p,j;\alpha],\ \mathcal{S}) = \{\Lambda\}$), and delete any strings of length less than m which remain in the final version of $R_m([p,j;\alpha],\ \mathcal{S})$. The problem is eliminated for the purposes of definition, by saying that these sets are the smallest such that the equations hold.

An <u>nth predecessor</u> of a stateset $\mathcal{S}$ is any predecessor of an n-1th predecessor of $\mathcal{S}$ ($n \geq 1$) and the 0th predecessor of $\mathcal{S}$ is $\mathcal{S}$ itself. We can now combine the above equations and define $R_m([p,j;\alpha],\ \mathcal{S}_1)$ as the smallest set satisfying

$$R_m([p,j;\alpha],\mathcal{S}_1) = \{\alpha' | [q,l;\beta] \in \mathcal{S}'_2,\ X_{q,l+1} = A_p, \alpha' \in H_m(X_{q,l+2} \dots X_{qn_q} \beta'),$$
$$\beta' \in R_m([q,l;\beta],\ \mathcal{S}_2),\ \alpha' = \alpha\gamma \text{ some } \gamma \in V_T^*,$$
$$\mathcal{S}_2 \text{ is a } j^{th} \text{ predecessor of } \mathcal{S}_1 \quad \}$$

If we replace $[p,j;\alpha] \in \mathcal{S}$ by the members of

$$\{[p,j;\alpha'] | \alpha' \in R_m([p,j;\alpha],\ \mathcal{S})\}$$

for every state in an LR(k) stateset table then we have an LA(m)LR(k) table from which the parsing table can be formed.

As an example of this method consider $\mathcal{G}_2$ with an extra endmarker (0^{th} production $S \rightarrow A \perp \perp$). This adds an irrelevant stateset $\mathfrak{S}_7 = \{[0,3]\}$ to the LR(0) stateset table. We can now compute $R_2([1,3],\mathfrak{S}_6)$.

$\mathfrak{S}_5$ is the only predecessor of $\mathfrak{S}_6$

$\mathfrak{S}_3$ is the only predecessor of $\mathfrak{S}_5$

$\mathfrak{S}_0$ and $\mathfrak{S}_3$ are the predecessors of $\mathfrak{S}_3$

So the 3^{rd} predecessors of $\mathfrak{S}_6$ are $\mathfrak{S}_0$ and $\mathfrak{S}_3$

$R_2([1,3],\mathfrak{S}_6) = R_2([1,0],\mathfrak{S}_0) \cup R_2([1,0],\mathfrak{S}_3)$

$[0,0] \in \mathfrak{S}_0$ and $X_{01} = A = A_1$ so we need $H_2(X_{02}X_{03}\beta)$, $\beta \in R_2([0,0],\mathfrak{S}_0) = \{\Lambda\}$

thus $H_2(X_{02}X_{03}\beta) = \{\perp\perp\} = R_2([1,0],\mathfrak{S}_0)$.

$[1,1] \in \mathfrak{S}_3$ and $X_{12} = A = A_1$ so we need $H_2(X_{13}\beta), \beta \in R_2([1,1],\mathfrak{S}_3)$

$R_2([1,1],\mathfrak{S}_3) = R_2([1,0],\mathfrak{S}_0) \cup R_2([1,0],\mathfrak{S}_3)$

$R_2([1,0],\mathfrak{S}_0) = \{\perp\perp\}$ already computed.

$R_2([1,0],\mathfrak{S}_3)$ is a re-evaluation, we compute $R_1([1,0],\mathfrak{S}_3) = H_1(X_{13}\beta)$

$\beta \in R_1([1,1],\mathfrak{S}_3) = R_1([1,0],\mathfrak{S}_0) \cup R_1([1,0],\mathfrak{S}_3)$

$R_1([1,0],\mathfrak{S}_0) = \{\perp\}$

$R_1([1,0],\mathfrak{S}_3)$ is a re-evaluation, it is taken to be $\{\Lambda\}$.

This yields approximations to $R_1([1,1],\mathfrak{S}_3)$ as $\{\perp,\Lambda\}$

$R_1([1,0],\mathfrak{S}_3)$ as $\{b\}$

$R_2([1,1],\mathfrak{S}_3)$ as $\{\perp\perp,b\}$

Finally $R_2([1,0],\mathfrak{S}_3) = \{b\perp,bb\}$

and $R_2([1,3],\mathfrak{S}_6) = \{\perp\perp,b\perp,bb\}$.

(This example is complicated by the method having to cater for the possibility of $X_{13} \overset{*}{\rightarrow} \Lambda$.)

A 2 symbol lookahead parsing-state for $\mathfrak{S}_6$ would thus be

$\{(\perp\perp,\text{reduce } 1), (b\perp,\text{reduce } 1), (bb,\text{reduce } 1)\}$

The calculation of an LA(m)LR(k) parsing table by either of these methods requires less work than a full LR(m) calculation, but both are more complicated than the original algorithm. The second is the more complex of the two, since in one sense, everything is worked out backwards. This may be alleviated to some extent by processing statesets in the order in which they are generated. The advantage of the second method is indicated in the next section where a simpler means of deriving m symbol lookahead is described.

SLA(m)LR(k) Parsing Tables

An LR(k) stateset table is again used as a basis for the method. For each $[p,j;\alpha] \in \mathcal{S}$ in this table, we compute a set of m symbol lookahead strings, which contains $R_m([p,j;\alpha],\mathcal{S})$ of the previous section. The additional strings, which are invalid and not required, do not necessarily prevent the resolution of inadequacies in the LR(k) table.

Define $F_m(Y)$, the m terminal follow set of $Y \in V$ by

$$F_m(Y) = \{\beta \in V_T^* \mid |\beta| = m,\ S \overset{*}{\Rightarrow} \alpha\, Y\, \beta\, \omega \text{ for some } \alpha,\ \omega \in V^*\}$$

An equivalent formulation is that $F_m(Y)$ is the smallest set satisfying

$$F_m(Y) = \{\beta \mid \beta \in H_m(X_{p,j+1} \ldots X_{pn_p}\alpha),\ Y = X_{pj},\ \alpha \in F_m(A_p)$$

$F_m(S)$ is taken to be $\{\Lambda\}$ as a special case (instead of $\emptyset$).

To obtain a finite algorithm from this equation, if any $F_m(Y)$ is required to be re-evaluated, instead compute $F_{m-1}(Y)$, taking $F_0(Y) = \{\Lambda\}$ and delete any strings of length less than m which remain in the final version of $F_m(Y)$.

If we replace $[p,j;\alpha]$ by the members of

$$\{[p,j;\alpha\beta] \mid \alpha\,\beta \in F_m(A_p)\}$$

for every state in an LR(k) stateset table, the result is an SLA(m)LR(k) stateset table from which the corresponding parsing table can be calculated. If for every stateset $\mathcal{S}$ in the SLA(m)LR(k) table, the conditions

$$(\alpha,\text{shift } j),\ (\beta,\text{reduce } p) \in R'(\mathcal{S}) \quad \text{must imply } \alpha \neq \beta$$

$$(\beta,\text{reduce } p),\ (\beta,\text{reduce } q) \in R'(\mathcal{S}) \quad \text{must imply } p = q$$

hold, then the grammar in question is said to be <u>SLA(m)LR(k)</u>.

Our example is again $\mathcal{G}_2$ with an extra endmarker.

$F_2(S) = \{\Lambda\}$

$F_2(A) = H_2(X_{02}X_{03}\alpha) \cup H_2(X_{13}\beta)$ with $\alpha \in F_2(S)$, $\beta \in F_2(A)$

$F_2(A)$ is a re-evaluation, we compute $F_1(A)$

$F_1(A) = H_1(X_{02}X_{03}\alpha) \cup H_1(X_{13}\ \beta)$, $\alpha \in F_1(S)$, $\beta \in F_1(A)$

$F_1(A)$ is a re-evaluation, and is taken to be $\{\Lambda\}$.

This yields an approximation to $F_1(A) = \{\perp, b\}$

from which we deduce $F_2(A) = \{\perp\perp, b\perp, bb\}$.

The simple 2 symbol lookahead extensions of $\mathcal{S}_3$, $\mathcal{S}_5$, $\mathcal{S}_6$ are

$$\mathcal{S}_3 = \{[1,1;\ \{\perp\perp, b\perp, bb\}], [2,1;\ \{\perp\perp, b\perp, bb\}]\}$$

$$\mathcal{S}_5 = \{[1,2;\ \{\perp\perp, b\perp, bb\}]\} \qquad \mathcal{S}_6 = \{[1,3;\ \{\perp\perp, b\perp, bb\}]\}$$

where $[p,j;\ \{\alpha_1, \ldots, \alpha_n\}]$ abbreviates $[p,j;\alpha_1], \ldots, [p,j;\alpha_n]$.

It should be clear that the calculation of an SLA(m)LR(k) table requires less work than that of an LA(m)LR(k) table by either of the methods of the previous section. An LR([illegible]) calculation is a part of all three, but the provision of m symbol lookahead in the SLA(m)LR(k) case is independent of any stateset table; it depends directly on the grammar, a major simplification. The LA(m)LR(k) table may be regarded either as a coarse version of an LR(m) table, or as a refined LR(k) table. The SLA(m)LR(k) table is merely a conveniently computed approximation to the LA(m)LR(k) table, and for this reason we regard the LA(m)LR(k) table as the more funadmental of the two.

We have that

$$\mathrm{SLA}(k)\mathrm{LR}(k) = \mathrm{LR}(k) \subseteq \mathrm{SLA}(m)\mathrm{LR}(k) \subseteq \mathrm{LA}(m)\mathrm{LR}(k)$$

where the inclusions are strict if $m > k$.

The SLA(m)LR(k) and LA(m)LR(k) techniques may be combined, as is now described. We first compute the LR(k) stateset and parsing tables. If any stateset is inadequate, its lookahead is extended to m symbols using the SLA(m)LR(k) method, and its parsing-state recalculated. If the stateset is still inadequate, then the lookahead is refined using the second LA(m)LR(k) method, which can be applied to individual statesets. Hopefully this will resolve the inadequacy (of this, and possibly other statesets). The combination of these two techniques gives a method which yields a table having no inadequate statesets for any LA(m)LR(k) grammar, with the possibility of a large economy of effort over a full LA(m)LR(k) computation.

Referring to the LR(0) table for $\mathcal{G}_2$, only SLA(1)LR(0) need be applied to $\mathcal{S}_3$ to produce a useable parsing table, with

$$R'(\mathcal{S}_3) = \{(\bot,\text{reduce } 2),\ (b,\text{reduce } 2),\ (a,\text{shift } 3),\ (A,\text{goto } 3)\}.$$

Minimal LR(k) Parsing Tables

To determine a method of constructing LR(k) tables which are minimal in some sense, we compare the original LR(k) table constructor with the first of the two LA(m)LR(k) techniques in the case LA(k)LR(0). By either of these methods, statesets $\mathcal{S}_j Y$ are constructed, and must be dealt with. Full LR(k) adds $\mathcal{S}_j Y$ as a new stateset unless it already exists as $\mathcal{S}_i$. LA(k)LR(0) only requires that a stateset $\mathcal{S}_i$ with $\mathcal{S}_i \sim \mathcal{S}_j Y$

$(H_0(\mathcal{S}_i) = H_0(\mathcal{S}_j Y))$ exist, and if so, replaces $\mathcal{S}_i$ by $\mathcal{S}_i \cup \mathcal{S}_j Y$. It is this merging of the statesets $\mathcal{S}_i$ and $\mathcal{S}_j Y$ which provides an LR(0) sized stateset table, but which also can create an inadequate stateset (if the grammar is not LA(k)LR(0)). The methods may be considered as two extremes; LR(k) which never merges, and LA(k)LR(0) which always does so. An ideal technique would be to merge whenever the merged stateset will not subsequently result in the production of an inadequate stateset.

If we have $\mathcal{S}_i \sim \mathcal{S}_j Y$ and either $\mathcal{S}_i \subseteq \mathcal{S}_j Y$ or $\mathcal{S}_i \supseteq \mathcal{S}_j Y$, clearly merging does not introduce a new stateset and should therefore take place. Conversely, if $\mathcal{S}_i \cup \mathcal{S}_j Y$ is itself inadequate, then the merge should be avoided. If neither of these conditions apply, it would theoretically be possible to test $\mathcal{S}_i \cup \mathcal{S}_j Y$ by continuing as for full LR(k), and if no inadequate statesets were produced then the merge could be made. A better solution (though still expensive computationally for a non LA(k)LR(0) grammar), would be a backtrack algorithm, which merges if $\mathcal{S}_i \sim \mathcal{S}_j Y$, but is able to back up and split the statesets if their merge resulted in an inadequate stateset being generated. An outline of such an algorithm is given in appendix 1(1.3).

The minimisation resulting from the above technique is concerned with the use of right contextual information in the creation of the statesets in the stateset table. It can be viewed as producing an LA(k)LR(m) table over which m varies between 0 and k, taking the least value consistent with the non-production of inadequate statesets. The table produced is not optimal since the order in which statesets are merged can affect the final size of the table. The simplest situation in which this can be seen is as follows:

$$\mathcal{S}_1 = \{[p,n_p;x],[q,n_q;z]\} \quad \mathcal{S}_2 = \{[p,n_p;y],[q,n_q;x]\}$$
$$\mathcal{S}_3 = \{[p,n_p;x],[q,n_q;y]\} \quad \mathcal{S}_4 = \{[p,n_p;y],[q,n_q;z]\}$$

Here $\mathcal{S}_1 \sim \mathcal{S}_2 \sim \mathcal{S}_3 \sim \mathcal{S}_4$, but $\mathcal{S}_1 \cup \mathcal{S}_2$, $\mathcal{S}_2 \cup \mathcal{S}_3$ and $\mathcal{S}_3 \cup \mathcal{S}_4$ are inadequate. If we merge $\mathcal{S}_1$ and $\mathcal{S}_4$ no further merging is possible, leaving three statesets, but if we merge $\mathcal{S}_1$ and $\mathcal{S}_3$ we may also merge $\mathcal{S}_2$ and $\mathcal{S}_4$ leaving only the two statesets,

$$\mathcal{S}_1 \cup \mathcal{S}_3 = \{[p,n_p;x],[q,n_q;\ \{y,z\}]\}$$
$$\mathcal{S}_2 \cup \mathcal{S}_4 = \{[p,n_p;y],[q,n_q;\ \{x,z\}]\}.$$

An algorithm could, in principle, be devised which tried all sequences of attempted merges, and produced an optimal LR(k) stateset table.

A more elementary minimisation can be applied to the lookahead strings in the parsing-states produced by any of the preceding methods. Suppose $\mathcal{S}$ is an adequate stateset; we can minimise the lookahead strings in $R'(\mathcal{S})$ independently of any other parsing-state as follows.
Let $(\gamma,\text{reduce } p) \in R'(\mathcal{S})$. Replace this by $(\alpha,\text{reduce } p)$ where

$$\gamma = \alpha\beta,\ (\alpha\beta',\text{reduce } q) \in R'(\mathcal{S}) \text{ implies } p = q,\ \nexists\ (\alpha\beta',\text{shift } i) \in R'(\mathcal{S}).$$

Similarly $(x\gamma,\text{shift } i)$ can be replaced by $(x\alpha,\text{shift } i)$ where

$$\gamma = \alpha\beta,\ \nexists\ (x\alpha\beta',\text{reduce } p) \in R'(\mathcal{S}). \qquad (\beta,\beta' \in V_T^*)$$

When all such replacements have been made, further minimisation is still possible if we are prepared to order the entries of $R'(\mathcal{S})$. Thus, for example, if when parsing we inspect all the reduce type of lookahead strings first, the shift entries need only be of the form (x,shift i) (which corresponds neatly with the goto entries). Alternatively, a set of reduce p entries can all be replaced by $(\Lambda,\text{reduce } p)$ if this is regarded as the last entry to be inspected.

This type of minimisation is concerned with the second use of right contextual information, that of providing the lookahead strings on which parsing decisions are made. It produces a parsing table for which the length of the lookahead strings varies from parsing-state to parsing-state, and for example, can convert an LA(m)LR(k) table to an LR(k) table if the grammar is in fact LR(k).

As a straightforward example, in the LA(1)LR(0) table for $\mathcal{G}_2$, the parsing-state $R'(\mathcal{S}_6)$ can be minimised to $\{(\Lambda, \text{reduce } 1)\}$.

The SLR and LALR Algorithms

Some comments from a practical viewpoint are now appropriate. It has already been mentioned that for programming language grammars, any value of k other than 0 results in LR(k) parsing tables of excessive size, and that such grammars are rarely LR(0). Also ruled out are the LA(m)LR(k) and SLA(m)LR(k) tables except as the special cases LA(k)LR(0) and SLA(k)LR(0). DeRemer (1969) independently defined these particular versions as LALR(k) and SLR(k) respectively, and discussed an algorithm for constructing SLR(k) parsers. (To be precise, an SLR(k) parser as defined by DeRemer is equivalent to an SLA(k)LR(0) parsing table for which the lookahead strings have been minimised.) His approach is based on a finite state machine which can be derived from Knuth's first method for testing whether a grammar is LR(k) (with k = 0). This dissertation parallels Knuth's second method, described earlier in this chapter, and should clarify the understanding of these algorithms and their connection with LR(k).

DeRemer also discussed an algorithm termed L(m)R(k) which utilises m symbols of the left context of a stateset together with k symbols of lookahead for the resolution of inadequacies. It would be possible to define a similar generalisation of this, as LB(l)LA(m)LR(k) which would consist of an LR(k) stateset table with some statesets split on the basis of the l symbols of left context, and the lookahead of m symbols computed for each stateset. L(m)R(k) would, in fact, correspond to LB(m)SLA(k)LR(0).

We further wish to restrict attention to a lookahead of only one symbol. This is done partly because of the benefits of simplification which result. Also, a one symbol lookahead is normally sufficient for programming language grammars. When this is not the case, minor grammatical changes or utilisation of a lexical pre-scan will usually remove the problem.

This leaves us the algorithms LR(0), SLA(1)LR(0), LA(1)LR(0) and minimal LR(1). LR(0) is insufficiently general for our purposes, and minimal LR(1) can only be useful on non LA(1)LR(0) grammars, for which it will be computationally expensive and produce tables larger than LA(1)LR(0). As candidates for current practical use, we are left with only SLA(1)LR(0) and LA(1)LR(0). For convenience, and following DeRemer, we abbreviate these as *SLR* and *LALR* respectively. We have

$$\mathrm{LR}(0) \subset \mathrm{SLR} \subset \mathrm{LALR} \subset \mathrm{LR}(1).$$

To exhibit these inclusions, we consider the grammars

$\mathcal{G}'_1$		$\mathcal{G}_1$		$\mathcal{G}_3$	
0	$S \rightarrow A \perp$	0	$S \rightarrow A \perp$	0	$S \rightarrow A \perp$
1	$A \rightarrow e\ d$	1	$A \rightarrow B\ d$	1	$A \rightarrow B\ d$
s=2	$A \rightarrow e\ e$	2	$A \rightarrow e\ e$	2	$A \rightarrow e\ B\ e$
		s=3	$B \rightarrow e$	s=3	$B \rightarrow e$

Their LR(0) stateset and parsing tables will be specified by printing for each stateset $\mathbb{S}$, its number, associated symbol and members, a comma, the additional members of its closure and the entries in the corresponding parsing-state $R'(\mathbb{S})$. The error stateset is omitted in each case, as are the two irrelevant parsing-states.

First for $\mathcal{G}_1'$.

0	Λ	$\underset{A}{[0,0]}$, $\underset{e}{[1,0]}$ $\underset{e}{[2,0]}$	(e,shift 3) (A,goto 1)
1	A	$\underset{\perp}{[0,1]}$	
2	⊥	$[0,2]$	
3	e	$\underset{d}{[1,1]}$ $\underset{e}{[2,1]}$	(d,shift 4) (e, shift 5)
4	d	$[1,2]$	(Λ,reduce 1)
5	e	$[2,2]$	(Λ,reduce 2)

Although after the parser has read the first e it is unaware of which production RHS is actually present, no reduction is called for and it can continue to read either the d or the e which determines the production uniquely. $\mathcal{G}_1'$ is LR(0).

0	Λ	$\underset{A}{[0,0]}$, $\underset{B}{[1,0]}$ $\underset{e}{[2,0]}$ $\underset{e}{[3,0]}$	(e,shift 3) (A,goto 1) (B,goto 4)
1	A	$\underset{\perp}{[0,1]}$	
2	⊥	$[0,2]$	
3	e	$\underset{e}{[2,1]}$ $[3,1]$	(Λ,reduce 3) (e,shift 6)
4	B	$\underset{d}{[1,1]}$	(d,shift 5)
5	d	$[1,2]$	(Λ,reduce 1)
6	e	$[2,2]$	(Λ,reduce 2)

With $\mathcal{G}_1$, when the parser has read the first e it must decide whether a reduce 3 is called for, and cannot determine this from 0 symbol lookahead. Since $F_1(B) = \{d\}$ we can add 1 symbol lookahead to the parsing-state of $\mathcal{S}_3$, indicating that the reduce 3 should only be performed if the next symbol is d (and hence we are recognising production 1). The new parsing-state is $\{(d,\text{reduce } 3), (e,\text{shift } 6)\}$, $\mathcal{G}_1$ is SLR.

0	Λ	$[0,0]_A$, $[1,0]_B$ $[2,0]_e$ $[3,0]_e$	(e,shift 3) (A,goto 1) (B,goto 4)
1	A	$[0,1]_\perp$	
2	$\perp$	$[0,2]$	
3	e	$[2,1]_B$ $[3,1]$, $[3,0]_e$	(Λ,reduce 3) (e,shift 7) (B,goto 6)
4	B	$[1,1]_d$	(d,shift 5)
5	d	$[1,2]$	(Λ,reduce 1)
6	B	$[2,2]_e$	(e,shift 9)
7	e	$[3,1]$	(Λ,reduce 3)
8	e	$[2,3]$	(Λ,reduce 2)

The SLR technique fails with $\mathcal{G}_3$ because $e \in F_1(B) = \{d,e\}$ due to the use of B in production 2. $R'(\mathcal{S}_3)$ becomes

$$\{(d,\text{reduce } 3),\ (e,\text{reduce } 3),\ (e,\text{shift } 7),\ (B,\text{goto } 6)\}$$

To remove the spurious (e,reduce 3) it is necessary to compute $R_1([3,1],\mathcal{S}_3) = \{d\}$, and we have that $\mathcal{G}_3$ is LALR.

For completeness we construct a grammar $\mathcal{G}_4$ which is LR(1) but not LALR. This requires that at least two LR(1) statesets combine to form an inadequate stateset at the LR(0) level. Denote two such LR(1) statesets by $\mathcal{S}_1$ and $\mathcal{S}_2$. Suppose that after the application of LALR, the inadequacy in $\mathcal{S}_1 \cup \mathcal{S}_2$ is (e,reduce p), (e,reduce q). If this is removed by LR(1) we will have, say $[p,n_p;e] \in \mathcal{S}_1$ and $[q,n_q;e] \in \mathcal{S}_2$. We have assumed $\mathcal{S}_1 \sim \mathcal{S}_2$ so we need $[p,n_p;d] \in \mathcal{S}_2$ and $[q,n_q;c] \in \mathcal{S}_1$. Further, we know that $X_{pn_p} = X_{qn_q}$ (associated symbol of $\mathcal{S}_1$ and $\mathcal{S}_2$), $d,e \in F_1(A_p)$ and $c,e \in F_1(A_q)$. If we take $A_p = B$, $A_q = C$, $n_p = n_q = 1$, $X_{p1} = e$ then $\mathcal{G}_4$ could have the form

0 $S \rightarrow A \perp$

1 $A \rightarrow \alpha B e$

2 $A \rightarrow \beta B d$

3 $A \rightarrow \gamma C e$

4 $A \rightarrow \delta C c$

5 $B \rightarrow e$

s=6 $C \rightarrow e$

Then $\alpha \neq \gamma$ to avoid ambiguity; $\alpha = \delta$ since these will be the associated symbols of statesets on the stack when we are in $\mathcal{S}_1$, similarly $\beta = \gamma$. So we take $\alpha = \delta = c$ and $\beta = \gamma = d$ to give $\mathcal{G}_4$ as

0 $S \rightarrow A \perp$

1 $A \rightarrow c B e$

2 $A \rightarrow d B d$

3 $A \rightarrow d C e$

4 $A \rightarrow c C c$

5 $B \rightarrow e$

s=6 $C \rightarrow e$

The LR(0) tables for $\mathcal{G}_4$ are,

0	Λ	$\underset{A}{[0,0]}, \underset{c}{[1,0]}\ \underset{d}{[2,0]}\ \underset{d}{[3,0]}\ \underset{c}{[4,0]}$	(c,shift 3) (d,shift 9) (A,goto 1)
1	A	$\underset{\perp}{[0,1]}$	
2	$\perp$	$[0,2]$	
3	c	$\underset{B}{[1,1]}\ \underset{C}{[4,1]}, \underset{e}{[5,0]}\ \underset{e}{[6,0]}$	(e,shift 8) (B,goto 4) (C,goto 6)
4	B	$\underset{e}{[1,2]}$	(e,shift 5)
5	e	$[1,3]$	(Λ,reduce 1)
6	C	$\underset{c}{[4,2]}$	(c,shift 7)
7	c	$[4,3]$	(Λ,reduce 4)
8	e	$[5,1]\ [6,1]$	(Λ,reduce 5) (Λ,reduce 6)
9	d	$\underset{B}{[2,1]}\ \underset{C}{[3,1]}, \underset{e}{[5,0]}\ \underset{e}{[6,0]}$	(e,shift 8) (B,goto 10) (C,goto 12)
10	B	$\underset{d}{[2,2]}$	(d,shift 11)
11	d	$[2,3]$	(Λ,reduce 2)
12	C	$\underset{e}{[3,2]}$	(e,shift 13)
13	e	$[3,3]$	(Λ,reduce 3)

We see that $\mathcal{S}_8$ is inadequate. Since $F_1(B) = R_1([5,1],\mathcal{S}_8) = \{d,e\}$ and $F_1(C) = R_1([6,1],\mathcal{S}_8) = \{e,c\}$, both SLR and LALR yield

$$\{(d,\text{reduce } 5),\ (e,\text{reduce } 5),\ (e,\text{reduce } 6),\ (c,\text{reduce } 6)\}$$

as parsing-state for $\mathcal{S}_8$, which remains inadequate. Full LR(1) analysis gives two versions of $\mathcal{S}_8$, i.e.

$$\mathcal{S}_8 = \{[5,1;e],[6,1;c]\} \qquad \mathcal{S}_{14} = \{[5,1;d],[6,1;e]\}$$

$R'(\mathcal{S}_8) = \{(e,\text{reduce } 5),\ (c,\text{reduce } 6)\}$ $R'(\mathcal{S}_{14}) = \{(d,\text{reduce } 5),\ (e,\text{reduce } 6)\}$
The entry (e,shift 8) in $R'(\mathcal{S}_9)$ is replaced by (e,shift 14).

The only lookahead strings affected by moving upwards from LR(0) in these examples, have been those in reduce entries. This must always be the case and is due to the mechanism of the shift operation. Since a single symbol is read and inspected during a shift (even by LR(0)), provision of one symbol lookahead leaves these entries unchanged. The way in which reduce entries are modified is now described.

Consider $(\Lambda,\text{reduce } p) \in R'(\mathcal{S})$ in an LR(0) table. This is equivalent to $\{(x,\text{reduce } p) \mid x \in V_T\}$ although the symbols are not examined. It is replaced by $\{(x,\text{reduce } p) \mid x \in F_1(A_p)\}$ under SLR, which is in turn replaced by $\{(x,\text{reduce } p) \mid x \in R_1([p,n_p],\mathcal{S})\}$ under LALR. Application of LR(1) may split $\mathcal{S}$ into a number of statesets, each having (x,reduce p) entries, but with x a member of some subset of $R_1([p,n_p],\mathcal{S})$. The union of these subsets over all the versions of $\mathcal{S}$ will be $R_1([p,n_p],\mathcal{S})$. Observe the successive refinement of the lookahead as evidenced by

$$V_T \supseteq F_1(A_p) \supseteq R_1([p,n_p],\mathcal{S}) \supseteq \text{subset of } R_1([p,n_p],\mathcal{S}).$$

The proximity of the SLR and LALR methods to LR(1) indicated by these comments may explain why they are so successful in handling grammars which require one symbol of lookahead.

Error Detection

By our definition, a parser for $\mathcal{G}$ must be able to determine if a string β is not a sentence of $\mathcal{G}$, since this is equivalent to saying β has no canonical derivations. An LR(k) parser has the additional capability of being able to locate the first symbol of β which is in error, i.e. the earliest point at which the symbols to the left do not begin any sentence of $\mathcal{G}$.

If $\beta = \alpha x \omega$ and $\exists\, \omega'$ such that $\alpha\omega' \in L(\mathcal{G})$ and $\nexists\, \omega''$ such that $\alpha x \omega'' \in L(\mathcal{G})$ then x is the first erroneous symbol of β. As soon as the LR(k) parser inspects x, the error is detected. Thus, when $k \geq 1$, a shift operation is performed and x is detected as the new k^{th} symbol of lookahead (we assume the first k symbols of the input are valid). There are then exactly k terminal symbols up to and including x which have not yet been read by the parser. LR(0) parsers perform no lookahead and because of this, errors can only be detected during shift operations. In this case x is detected as being invalid as soon as it is read.

The LR(k) parser's ability to perform this error location is inherent in the provision of the lookahead strings. In any stateset the lookahead strings constitute exactly those k symbol strings which can validly be encountered next in the input. An LR(0) parser, having no lookahead, must rely on the knowledge of which terminal symbols are $X_{p,j+1}$ for some $[p,j]$ in its current stateset.

Now consider an LA(m)LR(k) parser. When in a stateset $\mathcal{S}$, the parser has m symbol lookahead strings, say $\alpha_1, \ldots, \alpha_n$ with which it compares the actual lookahead $\beta = x_1 \ldots x_m$. If $\beta = \alpha_i$ for some $1 \leq i \leq n$, then an operation is determined, which the parser performs. It is possible, however, that an LR(m) parser would have detected an error, since that parser would be in a refinement of $\mathcal{S}$ which does not necessarily have α_i as a lookahead string. If $\beta \neq \alpha_i$, $1 \leq i \leq n$ then the LA(m)LR(k) parser does detect an error, and if x_r is the first symbol of β which causes $\beta \neq \alpha_i$, $1 \leq i \leq n$ then $k \leq r \leq m$ ($1 \leq r \leq m$ if $k = 0$) is ensured by the LR(k) error detection capabilities of the parser. Unfortunately, this does not necessarily locate the first invalid symbol, which could be any of $x_k, \ldots, x_r$ ($x_1, \ldots, x_r$ if $k = 0$). If $r > k$ ($r > 1$ if $k = 0$) and we require the parser to locate the error, it must continue parsing by performing any operation which has an r-1 symbol lookahead equal to $x_1 \ldots x_{r-1}$. Eventually an error will be detected with $r = k$ which locates the first invalid symbol.

An SLA(m)LR(k) parser behaves similarly, and will detect errors no earlier than LA(m)LR(k) and no later than LR(k). Notice that SLR and LALR do locate the error when it is detected, but this may not be as soon as LR(1). This is again due to the special behaviour of LR(0). Minimisation of the lookahead strings of any parser can degrade its error detection, possibly down to that of an LR(0) parser.

It should be clear that all the parsing algorithms discussed in this chapter have at least the error detection capability of an LR(0) parser, which ensures that they all detect an erroneous symbol, at the latest, when it is read. This feature is of practical importance when a parser is used in a compiler for a programming language; early detection is an aid to good error recovery, and the location of the first incorrect symbol is of obvious value to the programmer.

The grammar $\mathcal{G}_5$,

0 $S \rightarrow A \perp \perp$

1 $A \rightarrow d\ B\ e\ e$

2 $A \rightarrow e\ B\ d\ d$

3 $B \rightarrow d\ C$

s=4 $C \rightarrow d$

gives an example of an LA(2)LR(0) parser's inability to locate a detected error.

A part of the LA(2)LR(0) tables for $\mathcal{G}_5$ is,

0	Λ	$\underset{A}{[0,0;\Lambda]}$, $\underset{d}{[1,0;\perp\perp]}$ $\underset{e}{[2,0;\perp\perp]}$	(dd,shift 1) (ed,shift 4) (A,goto 5)
1	d	$\underset{B}{[1,1;\perp\perp]}$, $\underset{d}{[3,0;ee]}$	(dd,shift 2) (B,goto 6)
2	d	$\underset{C}{[3,1;ee]}$ $\underset{C}{[3,1;dd]}$,$\underset{d}{[4,0;ee]}$$\underset{d}{[4,0;dd]}$	(ee,shift 3) (dd,shift 3) (C,goto 7)
3	d	$[4,1;ee]$ $[4,1;dd]$	(dd,reduce 4) (ee,reduce 4)
4	e	$\underset{B}{[2,1;\perp\perp]}$, $\underset{d}{[3,0;dd]}$	(dd,shift 2) (B,goto 8)

$L(\mathcal{G}_5) = \{dddee\perp\perp,\ edddd\perp\perp\}$

Take as input to be parsed the string ddddee⊥⊥, the first invalid symbol of which is the last d. The parser will detect an error when in S_3 with de as lookahead and cannot as yet determine that it is the d which is invalid.

Chapter 3

Preliminary

Before embarking on the main purpose of this chapter, which is a comparison of precedence methods with the SLR parsing algorithm, we digress to state (and prove) a necessary condition for a grammar to be LR(1). The condition is somewhat elementary, but provides a demonstration of the utility of the $[p,j;\alpha]$ notation for obtaining formal results in the area of LR(k) methods.

We repeat the conditions which must be satisfied by an LR(1) grammar. For each stateset $\mathcal{S}$ in the LR(1) stateset table of the grammar we require that

$$(x,\text{reduce } p),\ (y,\text{shift } i) \in R'(\mathcal{S}) \quad \text{must imply} \quad x \neq y$$
$$(x,\text{reduce } p),\ (x,\text{reduce } q) \in R'(\mathcal{S}) \quad \text{must imply} \quad p = q$$

Directly applied to $\mathcal{S}$, the conditions may be restated as

$$[p,n_p;x],[q,l;\alpha] \in \mathcal{S}',\ l < n_q,\ X_{q,l+1} = y \quad \text{must imply} \quad x \neq y$$
$$[p,n_p;x],[q,n_q;x] \in \mathcal{S}' \quad \text{must imply} \quad p = q$$

and these are of course equivalent to

$$Z \cap Z_p = \emptyset \quad \text{and} \quad Z_p \cap Z_q = \emptyset \quad \text{if} \quad p \neq q$$

where the sets Z and Z_p are defined for each stateset $\mathcal{S}$ as described in the previous chapter.

The disadvantage of the above conditions is that they apply to the (closure of the) statesets, and are not directly in terms of the grammar. There are two other criteria by which a grammar may be said to be LR(1).

The first is the intuitive definition applied to the sentences of the grammar; that they can all be parsed by scanning once from left to right, only looking one symbol ahead of the RHS to be reduced at any point in the parse. Thus, if the input string has been reduced to the sentential form $X_1 \dots X_n x\omega$ with $x\omega \in V_T^*$, and the correct parsing action is a reduce p on the symbols $X_{n-n_p+1} \dots X_n$, then this must be the case for any sentential form $X_1 \dots X_n x\omega'$ with $\omega' \in V_T^*$. The second is Knuth's first method for testing a grammar, already mentioned in connection with DeRemer's work. This test involves the construction of an extended right regular grammar from the original grammar, which is closely related to the LR(1) stateset table and can be checked in a similar fashion (a grammar is <u>extended right regular</u> if $j < n_p$ implies that $X_{pj} \in V_T$).

The provision of necessary, and if possible sufficient, conditions for LR(1) expressed in terms of the grammar would aid in the writing (and perhaps the testing) of LR(1) grammars. We now establish a necessary condition; if a grammar is LR(1), then for each $A \in V_N$ such that $A \overset{+}{\to} \Lambda$, we must have $H_1(A) \cap F_1(A) = \emptyset$.

For, suppose $A \overset{+}{\to} \Lambda$ and $x \in H_1(A) \cap F_1(A)$.

Since $x \in F_1(A)$ we can find a stateset $\mathcal{S}$ with,

$[p,j;\alpha] \in \mathcal{S}'$, $X_{p,j+1} = A$ and $H_1(X_{p,j+2} \dots X_{pn_p} \alpha) = x$

Since $x \in H_1(A)$, $\exists\, [q,0;\beta] \in \mathcal{S}'$ with $X_{q1} = x$,

and so $(x, \text{shift } i) \in R'(\mathcal{S})$.

Also $A \overset{+}{\to} \Lambda$ and we can find A_r such that

$A \overset{*}{\to} A_r \to \Lambda$ and $[r,0;x] \in \mathcal{S}'$.

Since $n_r = 0$, $(x, \text{reduce } r) \in R'(\mathcal{S})$.

Hence $\mathcal{S}$ is inadequate, the grammar is not LR(1) and we have our result.

The above is a formalisation of the following argument.

$x \in F_1(A)$ implies $S \xrightarrow{*} \alpha\ Ax\beta$, $x \in H_1(A)$ implies $A \xrightarrow{*} x\ \gamma$.

If also $A \xrightarrow{+} \Lambda$ then $S \xrightarrow{*} \alpha\ x\ \gamma\ x\ \beta$ and $S \xrightarrow{*} \alpha x\beta$

By examining only αx, a parser cannot determine whether x should be read, or a reduction made to αAx.

<u>Precedence Parsers</u>

The precedence methods with which we shall be mostly concerned are known as simple precedence and weak precedence. The operation of these parsers is determined by <u>precedence relations</u>, which are defined to be relations on V, specified by

$$\doteq\ = \{(X,Y) \mid \exists\ A \rightarrow \alpha XY\omega \in P,\ \alpha,\omega \in V^*\}$$

$$<\cdot\ = \{(X,Y) \mid \exists\ A \rightarrow \alpha XB\omega \in P,\ B \xrightarrow{+} Y\beta,\ \alpha,\beta,\omega \in V^*\}$$

$$\cdot>\ = \{(X,Y) \mid \exists\ A \rightarrow \alpha BC\omega \in P,\ B \xrightarrow{+} \beta X,\ C \xrightarrow{*} Y\gamma,\ \alpha,\beta,\gamma,\omega \in V^*\}$$

$$\leq\cdot\ = \{(X,Y) \mid \exists\ A \rightarrow \alpha XB\omega \in P,\ B \xrightarrow{*} Y\beta,\ \alpha,\beta,\omega \in V^*\}$$

The sets first(X) and last(X) are defined for $X \in V$ by

$$\text{first}(X) = \{Y \in V \mid X \xrightarrow{+} Y\alpha,\ \alpha \in V^*\}$$

$$\text{last}(X) = \{Y \in V \mid X \xrightarrow{+} \alpha Y,\ \alpha \in V^*\}$$

(clearly we have first(x) = last(x) = $\emptyset$ if $x \in V_T$). As immediate consequences of these definitions, we can state,

$X \doteq Y$ iff $\exists\ A \rightarrow \alpha XY\omega$

$X <\cdot Y$ iff $\exists\ A \rightarrow \alpha XB\omega$, $Y \in$ first(B)

$X \cdot> Y$ iff $\exists\ A \rightarrow \alpha BC\omega$, $X \in$ last(B), and either $Y = C$ or $Y \in$ first(C)

$X \leq\cdot Y$ iff $\exists\ A \rightarrow \alpha XB\omega$, and either $Y = B$ or $Y \in$ first(B)

iff $X \doteq Y$ or $X <\cdot Y$

A grammar is said to be Λ-free if $\nexists$ $A \rightarrow \Lambda$ in P i.e. $n_p > 0$, $0 \leq p \leq s$.

A grammar is said to be a simple precedence grammar if

(i) it is Λ-free

(ii) $\leq\cdot$ and $\cdot>$ are disjoint

(iii) $<\cdot$ and $\doteq$ are disjoint

(iv) $A_p \rightarrow \alpha$ and $A_q \rightarrow \alpha$ implies $p = q$

Simple precedence grammars, and a solution to the parsing problem for them, were first described by Wirth and Weber (1966).

A grammar is said to be a weak precedence grammar if

(i) it is Λ-free

(ii) $\leq\cdot$ and $\cdot>$ are disjoint

(iii)$'$ $A \rightarrow \alpha XY\beta$ and $B \rightarrow Y\beta$ implies $X \not\leq\cdot B$

(iv) $A_p \rightarrow \alpha$ and $A_q \rightarrow \alpha$ implies $p = q$

Notice that if $<\cdot \cap \doteq\ = \emptyset$, $A \rightarrow \alpha XY\beta$, $B \rightarrow Y\beta$ then we have $X \doteq Y$, which implies $X \not<\cdot Y$, which implies $X \not\leq\cdot B$. So (iii)$'$ can be deduced from (iii), showing that any simple precedence grammar is also a weak precedence grammar. Further, if $\doteq \cap \cdot> = \emptyset$, $A \rightarrow \alpha XY\beta$, $B \rightarrow Y\beta$ then we have $X \doteq Y$, which implies $X \cdot\!\not> Y$, which implies $X \cdot\!\not> B$. The original definition by Ichbiah and Morse (1970) required that

$$A \rightarrow \alpha XY\beta \text{ and } B \rightarrow Y\beta \text{ implies } X \cdot\!\not> B$$

holds for a weak precedence grammar. We have shown that this can be deduced from (ii) and thus the apparently less restrictive definition given here coincides with that of Ichbiah and Morse.

When discussing precedence grammars, we usually take as 0^{th} production, $S \rightarrow \perp S' \perp$, and let this override the LR(1) convention if both apply.

The general bottom-up algorithm described in chapter 2 can perform weak precedence parsing with the following specification.

Stack elements are symbols of V, with Initial = Final = $\perp$.

ACTION(F,n,I,i′) yields,

<u>shift</u> - <u>if</u> $F[n] \lessdot I[i'] $ <u>and</u> $(I[i'] \neq \perp$ <u>or</u> $n = 1)$

<u>reduce p</u> - <u>if</u> $F[n] \gtrdot I[i']$ <u>and</u> $F[n-n_p+1] \ldots F[n] = X_{p1} \ldots X_{pn_p}$ <u>and</u> $F[n-n_p] \leq\!\cdot\, A_p$

<u>error</u> - otherwise

NEXT(F,n,X) = X

This parser will also parse sentences of simple precedence grammars, and for such a grammar, the determination that ACTION should yield <u>reduce p</u> can be made more efficient by observing that $F[n-n_p] \leq\!\cdot\, A_p$ above implies $F[n-n_p] \lessdot F[n-n_p+1]$, which for a simple precedence grammar implies $F[n-n_p] \not\doteq F[n-n_p+1]$. By scanning down F for the first $\lessdot$ relation, the value of n_p can be evaluated. The test for n = 1 when we have $F[n] \leq\!\cdot \perp$ amounts to a check for $\perp S'$ on the stack. A practical implementation of this algorithm would be driven by a precomputed table of the precedence relations.

It can now be seen that (ii) ensures that the parser can decide whether to read the next symbol or perform a reduction, and that (iv) and (iii)′ (or (iii)) ensure that the production to be used for a reduction can be determined by inspection of the parser's stack. Condition (i) is required because the precedence relations are defined on V. It is possible to relax this condition, but then the relations must be defined on $V \setminus \{A \mid A \to \Lambda\}$, and are slightly more complex. This will not be pursued here, but is discussed further by Gray and Harrison (1969).

Comparison of SLR and Weak Precedence

A bottom-up parser performs two distinct functions; the location of the right most symbol of the RHS which must be reduced next, and the determination of which production should be used for that reduction. The SLR method uses the information inherent in the current stateset $\mathcal{S}$ to achieve this. The symbols to be read are those terminals which are $X_{p,j+1}$ for some state $[p,j;\alpha]$ in $\mathcal{S}'$, while terminals x with $[q,n_q;x] \in \mathcal{S}'$ indicate that a reduction is necessary, and in fact identify the production to be used as the q^{th}. The stateset $\mathcal{S}$ corresponds to a set of occurrences of its associated symbol on the RHSs of the productions of the grammar.

The only contextual information available to the weak precedence parser is the current symbol, X say; in SLR terms this corresponds to <u>all</u> occurrences of X in the production RHSs. Terminals in $F_1(X)$ are split into two disjoint sets by $\leq\cdot$ and $\cdot>$, those which must be read, and those which signal a reduction respectively. The production to be used in a reduction must be determined by inspecting the parser's stack. The conditions for a weak precedence grammar ensure that the longest RHS which matches the top of the stack may be used, since no other valid match can then be found.

Error detection by a weak precedence parser is inferior to that of SLR, since invalid symbols can be read, and the error only discovered later when no production RHS matches the stack. This can be seen by considering a weak precedence parser for $\mathcal{G}_1$ (with extra leading endmarker) applied to the string $\perp eeee \perp$. The error remains undetected until a reduction is attempted, then since $e \not\leq\cdot A$, none can be made.

Grammars which describe programming languages usually require substantial modification before the weak precedence conditions are satisfied, and although this can often be done without altering the language, the phrase structure imposed by the original grammar is invariably corrupted.

These disadvantages also apply to the simple precedence algorithm, even more modifications being needed to comply with the stricter conditions in this case. Despite these drawbacks, the simple precedence method has been utilised very successfully, for example, in the PL360 and Algol W compilers.

We next wish to show that any weak precedence grammar is also an SLR grammar, the proof of which is roughly based on the above discussion. First, since $\leq\cdot$ corresponds to the shift entries in the SLR parsing-states, and $\cdot\!>$ to the reduce entries, the absence of shift - reduce clashes can be argued from $\leq\cdot \cap \cdot\!> = \emptyset$. Secondly, if weak precedence can determine from the stack which production to use in a reduction, the SLR method (knowing which completed productions are on the stack) must also be able to deduce the correct reduction. This argument will now be presented more rigorously.

The following lemma establishes formally the connection between the precedence relations and the states in the SLR statesets.

Lemma 3

Let $\mathcal{S}$ be a stateset, with associated symbol Y, and suppose $[p,j;\alpha] \in \mathcal{S}'$. a) If $0 < j < n_p$ then $Y \doteq X_{p,j+1}$

b) If $0 = j < n_p$ then $Y \lessdot A_p$

c) If $j = n_p$, $\alpha = x\beta$ and the grammar is Λ-free then $Y \gtrdot x$

These results are now established.

a) If $0 < j < n_p$, we have $A_p \rightarrow X_{p1} \ldots X_{p,j-1} Y X_{p,j+1} \ldots X_{pn_p}$ and so $Y \doteq X_{p,j+1}$

b) If $0 = j < n_p$, we can find $[q,l;\gamma] \in \mathcal{S}$ with $0 < l < n_q$ and a sequence of productions

$$A_{q_i} \rightarrow X_{q_i,1}\alpha_i \quad n_{q_i} > 0 \quad 1 \leq i \leq r \quad r \geq 0$$

with $X_{q,l+1} = A_{q_1}$ $\quad X_{q_i 1} = A_{q_{i+1}}$ $\quad 1 \leq i \leq r$ and $A_{q_{r+1}} = A_p$

(These productions correspond to the sequence of states $[q_i,0;\gamma_i] \in \mathcal{S}'$ $1 \leq i \leq r$ which is responsible for the inclusion of $[p,0;\alpha]$ in $\mathcal{S}'$)

Then we have $A_q \rightarrow X_{q1} \ldots X_{q,l-1} Y X_{q,l+1} \ldots X_{qn_q}$ and $X_{q,l+1} \overset{*}{\rightarrow} A_p \alpha_r \ldots \alpha_1$ and so $Y \lessdot A_p$

c) If $j = n_p$, $\alpha = x\beta$ and the grammar is Λ-free, we can find a stateset $\mathcal{S}_l$ with $[q,l;\gamma] \in \mathcal{S}'_l$ $0 \le l < n_q - 1$, and a sequence of productions

$$A_{q_i} \rightarrow \alpha_i X_{q_i n_{q_i}} \qquad n_{q_i} > 0 \qquad 1 \le i \le r \qquad r \ge 0$$

with $X_{q,l+1} = A_{ql}$ $\quad x \in H_1(X_{q,l+2} \ldots X_{qn_q}\gamma)$

$$X_{q_i n_{q_i}} = A_{q_{i+1}} \qquad 1 \le i \le r \quad \text{and } A_{q_{r+1}} = A_p$$

(These productions correspond to the sequence of states and statesets $[q_i,0;x\beta] \in \mathcal{S}_i'$, $1 \le i \le r+1$ where we have $[q_i, n_{q_i} - 1;x\beta] \in \mathcal{S}'_{i+1}$ and $\mathcal{S}_i$ is an $n_{q_i} - 1^{\text{th}}$ predecessor of $\mathcal{S}_{i+1}$. $[q_{r+1},0;x\beta] = [p,0;x\beta] \in \mathcal{S}'_{r+1}$ and $\mathcal{S}_{r+1}$ is an $n_p{}^{\text{th}}$ predecessor of $\mathcal{S}$. This sequence is one which ensures $[p,n_p;x\beta] \in \mathcal{S}$)

Since the grammar is Λ-free, $n_p > 0$ and $x \in H_1(X_{q,l+2})$.

Then we have $A_q \rightarrow X_{q1} \ldots X_{q,l+1} X_{q,l+2} \ldots X_{qn_q}$,

$X_{q,l+1} \overset{+}{\rightarrow} \alpha_1 \ldots \alpha_r X_{p1} \ldots X_{p,n_p-1} Y$ and $X_{q,l+2} \overset{*}{\rightarrow} x\delta$ and so $Y \cdot\!> x$

<u>Corollary</u>

Let $[p,j;\alpha] \in \mathcal{S}'$, whose associated symbol is Y.

If $0 = j < n_p$ then $Y <\!\cdot X_{p1}$, and thus if $0 \le j < n_p$ then $Y \le\!\cdot X_{p,j+1}$.

Follows immediately from b) and a) above.

The following statements also hold for any stateset table.

If $Y \doteq X$ then $\exists$ a stateset $\mathcal{S}$ with associated symbol Y and $[p,j;\alpha] \in \mathcal{S}$, $0 < j < n_p$, $X_{p,j+1} = X$.

If $Y <\!\cdot X$ then $\exists$ a stateset $\mathcal{S}$ with associated symbol Y and $[p,0;\alpha] \in \mathcal{S}'$, $X_{p1} = X$.

If $Y \cdot\!> x \in V_T$ then $\exists$ a stateset $\mathcal{S}$ with associated symbol Y and $[p,n_p;x\beta] \in \mathcal{S}$ (if $\mathcal{S}$ is not LR(0)).

These can all be deduced from the definitions of the precedence relations.

Now suppose that $[p,j;\alpha]$, $[q,l;\beta]$ are members of the closure of some stateset $\mathcal{S}'$, with $0 \leq j \leq l$. By considering the associated symbols of the 0^{th}, 1^{st}, ..., $j-1^{th}$ predecessors of $\mathcal{S}$, we can see that $X_{pj} = X_{ql}$, $X_{p,j-1} = X_{q,l-1}$, ..., $X_{p1} = X_{q,l-j+1}$. When the parser is in stateset $\mathcal{S}$, $X_{p1} \ldots X_{pj} = X_{q,l-j+1} \ldots X_{ql}$ will be the associated symbols of the statesets on the top of the stack. This result may be regarded as an extension of the associated symbol concept, and in conjunction with lemma 3 enables us to prove the following theorem.

Theorem

Weak precedence grammars are SLR grammars.

Let $\mathcal{G}$ be any grammar which is not SLR. We wish to show that $\mathcal{G}$ is not a weak precedence grammar. Consider the SLR stateset table for $\mathcal{G}$. This contains at least one inadequate stateset. Let $\mathcal{S}$ be such a stateset. The inadequacy may be

a) shift - reduce p or b) reduce p - reduce q $p \neq q$

a) Suppose (x, shift i) and (x, reduce p) $\in R'(\mathcal{S})$.

Then $\exists\ [p,n_p;x] \in \mathcal{S}'$. We must take $\mathcal{G}$ to be Λ-free, so $n_p > 0$, X_{pn_p} is the associated symbol of $\mathcal{S}$ and by lemma 3,

$$X_{pn_p} \cdot\!> x$$

Also $\exists\ [q,j;\alpha] \in \mathcal{S}'$, $j < n_q$ and $X_{q,j+1} = x$. Then by the corollary to lemma 3, $X_{pn_p} \leq\cdot\ x$

Hence $\mathcal{G}$ is not a weak precedence grammar.

b) Suppose (x,reduce p) and (x,reduce q) $\in R'(\mathcal{S})$, $p \neq q$. Then $\exists\, [p,n_p;x], [q,n_q;x] \in \mathcal{S}'$, and we must take $n_p, n_q > 0$. If $n_p = n_q$ then $X_{p1} \ldots X_{pn_p} = X_{q1} \ldots X_{qn_q}$ and therefore $\mathcal{G}$ is not a weak precedence grammar.

If $n_p \neq n_q$ then take $0 < n_p < n_q$. We may write

$A_q \rightarrow X_{q1} \ldots X_{q,n_q-n_p} X_{p1} \ldots X_{pn_p}$ and $A_p \rightarrow X_{p1} \ldots X_{pn_p}$

Let $\mathcal{S}_1$ be an n_p th predecessor of $\mathcal{S}$. Then $[p,0;x] \in \mathcal{S}_1'$ and $[q,n_q - n_p;x] \in \mathcal{S}_1$. Since $n_q - n_p > 0$, X_{q,n_q-n_p} is the associated symbol of $\mathcal{S}_1$. By lemma 3, $X_{q,n_q-n_p} \leq\!\cdot\, A_p$ and therefore $\mathcal{G}$ is not a weak precedence grammar.

In both cases we have deduced that $\mathcal{G}$ is not a weak precedence grammar, which gives us our result.

The proof of case b) of the theorem shows that for a weak precedence grammar a parsing-state cannot contain two distinct reduce entries, since the information that both reductions were on the same terminal was not used.

For simple precedence grammars a stronger result of a similar nature can be obtained. Let $\mathcal{S}$ be a stateset with $[p,j;\alpha],[q,l;\beta] \in \mathcal{S}$ and $j \neq l$. Then $\mathcal{S} \neq \mathcal{S}_0$ and we may take $0 < j < l$. Since $X_{p1} \ldots X_{pj} = X_{q,l-j+1} \ldots X_{ql}$ we have $A_q \rightarrow X_{q1} \ldots X_{q,l-j} X_{p1} \ldots X_{pj} X_{q,l+1} \ldots X_{qn_q}$. Thus $X_{q,l-j} \doteq X_{p1}$. If $\mathcal{S}_1$ is a j^{th} predecessor of $\mathcal{S}$, then $[q,l-j;\beta] \in \mathcal{S}_1$ and $[p,0;\alpha] \in \mathcal{S}_1'$. The associated symbol of $\mathcal{S}_1$ is therefore $X_{q,l-j}$ and so by the corollary to lemma 3, $X_{q,l-j} \lessdot X_{p1}$.

We have shown that, if for a grammar the precedence relations $\lessdot$ and $\doteq$ are disjoint (condition (iii) for simple precedence), then for any stateset $\mathcal{S}$ with $[p,j;\alpha]$, $[q,l;\beta] \in \mathcal{S}$ we must have $j = l$. The presence of two reduce entries in a parsing-state for such a grammar would imply that the respective productions had identical RHSs.

Weaker Precedence

This section describes a modification of the weak precedence method which removes two anomalies in the description given here. First, since the parser checks $F[n-n_p] \lessdot A_p$ for a reduce p operation, it is possible to change condition (iv) to

$$\text{(iv)}' \quad A_p \rightarrow \alpha,\ A_q \rightarrow \alpha \text{ implies either } p = q \text{ or } \{X \in V \mid X \lessdot A_p \text{ and } X \lessdot A_q\} = \emptyset$$

Second, it can be seen that the weak precedence algorithm, after inspecting a terminal symbol which indicates that a reduce is required, proceeds to ignore that symbol for the purpose of deciding which production to use. A more uniform criterion for ACTION to yield a reduce p operation would be,

$$F[n] \gtrdot I[i'] \ \underline{\text{and}}\ F[n-n_p+1] \ldots F[n] = X_{p1} \ldots X_{pn_p} \ \underline{\text{and}}\ F[n-n_p] \lessdot A_p \ \underline{\text{and}}\ (A_p \gtrdot I[i'] \ \underline{\text{or}}\ A_p \lessdot I[i'])$$

This gives a more powerful parser, and permits a further weakening of the conditions imposed on a grammar, which we define to be a weaker precedence grammar if

(i) It is Λ-free

(ii) $\lessdot$ and $\gtrdot$ are disjoint

(iii)″ $A \rightarrow \alpha XY\beta,\ B \rightarrow Y\beta$ implies either $X \not\lessdot B$ or $F_1(A) \cap F_1(B) = \emptyset$

(iv)″ $A_p \rightarrow \alpha,\ A_q \rightarrow \alpha$ implies either $p = q$ or $F_1(A_p) \cap F_1(A_q) = \emptyset$ or $\{X \in V \mid X \lessdot A_p \text{ and } X \lessdot A_q\} = \emptyset$

Conditions (iii)″ and (iv)″ have been simplified by the use of $\{x \in V_T \mid A \lessdot x \text{ or } A \gtrdot x\} = F_1(A)$, which holds for Λ-free grammars.

The proof of case b) of the theorem may be modified to show that the weaker precedence grammars are included in the SLR grammars. Two changes are required in case b), namely the observation that $x \in F_1(A_p) \cap F_1(A_q)$ for both parts, and when $n_p = n_q$, the consideration of any n_p^{th} predecessor of $\mathcal{S}$; if $\mathcal{S}_1$ is such a predecessor with associated symbol X, then since $[p,0;x]$, $[q,0;x] \in \mathcal{S}_1'$ we may deduce by lemma 3 that $X \leq\cdot A_p$ and $X \leq\cdot A_q$. Examples to show that the inclusion is strict are easily found.

Weaker precedence grammars are also included in the mixed strategy precedence of degree (1,1;1,1) grammars as defined by McKeeman, Horning and Wortman (1970).

An SLR parsing table can be used to construct a table which drives a precedence type of parser, by combining the entries of parsing-states whose corresponding statesets have the same associated symbol. The shift and goto entries must be altered to refer to the merged parsing-states, i.e. for (A,goto i), i must designate the unique merged parsing-stage corresponding to A. The effect of this merging is to remove from the parsing table any contextual information other than the current symbol.

By merging LR(0) statesets on associated symbol as they are produced by an LR(0) stateset computation, and using $F_1(A_p)$ to add lookahead to $[p,n_p]$ states, the same table can be constructed more efficiently. Notice that merging an LR(1) table on associated symbol would result in an identical table to the above.

The resulting parser does not have the error detection capability of an LR(0) parser, and when a <u>reduce p</u> operation is called for, it must verify that the RHS of production p is represented by the top of the stack. In view of this verification, we permit the resolution of (x,reduce p), (x,reduce q) inadequacies on the same basis as weaker precedence; for production p to be used we require $X_{p1} \ldots X_{pn_p}$ represented on the stack, and the associated symbol of $F[n-n_p]$ must be $\le\cdot\ A_p$.

The class of grammars which can be parsed by the above, informally described, precedence-SLR method may be termed PLR. In appendix 1 (1.4) we show that a Λ-free grammar is PLR iff it is weaker precedence. Essentially this is because the technique used for reductions is defined to be equivalent, and $\le\cdot \cap \cdot> = \emptyset$ iff there are no shift-reduce inadequacies in the PLR table (in fact the PLR table corresponds to a tabulation of $\le\cdot$ and $\cdot>$). As a consequence of this result, it should be clear that the PLR method is merely a more costly version of weaker precedence, and should not be considered for practical use.

The preceding discussion of simple, weak and weaker precedence within the framework of the SLR parsing algorithm is intended to indicate the ways in which SLR can be regarded as an extension of the precedence methods. The inclusion results and the characterisation of $\le\cdot$ and $\cdot>$ in terms of the SLR stateset table provide an aid to the understanding of the SLR parser.

Other parsing methods which may be considered with advantage from an SLR viewpoint are described by Hext and Roberts (1970) (Domolki's algorithm) and Lynch (1968) (ICOR grammars). Connections with the augmented operator techniques of Gries (1968) are discussed by Anderson, Eve and Horning (1971). Finally, the SLR algorithm may itself be considered as a special case of Korenjak's method, by constructing LR(1) subparsers for all nonterminals in a grammar.

Chapter 4

Practical Considerations

Chapter 4 is concerned with the practical implementation of the SLR parsing algorithm, and in this first section we attempt to justify our concentration on LR techniques in general and the SLR method in particular.

Techniques are available for the construction, from an arbitrary grammar, of a parser for the sentences of that grammar. However, Earley (1970) reports that to parse an input string of length n, his method may require time proportional to n^3, and space proportional to n^2 (the time bound is reduced to n^2 for unambiguous grammars). Since these are currently the best results for the parsing of arbitrary grammars, we restrict our attention to less general methods having time and space requirements that are linearly proportional to the length of the input.

Because of the reduced generality of such methods, a grammar may need substantial modification before it is acceptable. This is the case with the predictive top-down analysers, which only work for the LL(k) grammars as defined by Lewis and Stearns (1968). The precedence methods also require the transformation of most grammars, although McKeeman, Horning and Wortman claim that this can often be done for the mixed strategy precedence method of degree (2,1;1,1) without destroying the grammar's usefulness as a syntactic reference. The inferior error detection capability of precedence methods is an added disadvantage.

In contrast, we claim that a programming language grammar, which has been made unambiguous, will usually be LR(1), and further, that few if any changes will be needed for it to be LALR or even SLR. All the LR methods can locate the first erroneous symbol of an invalid string, and they utilise time and space at most in linear proportion to the length of the input. For programming languages this space requirement

is somewhat misleading, since it is normally possible to ensure that the space needed for the stack (which is variable), is very small compared with the space occupied by the parser itself (with its tables).
We consider only the LALR and SLR methods because of the magnitude of LR(1) tables. The selection of SLR for discussion and implementation is merely for convenience, in description and computation. In any case, much of what is said can be seen to apply equally to the LALR algorithm.

While the above does not present a conclusive argument, the points which are made serve as motivation (if one is needed) for what follows.

SLR Parsing

We give a compact description of the formation of SLR tables, omitting some of the complexity which was required in Chapter 2.
The notation used here (for convenience) conflicts slightly with that chapter, but the description is essentially equivalent.

For SLR, a state is an ordered pair [p,j], which indicates that j symbols of production p have been recognised (recall $A_p \to X_{p1} \ldots X_{pn_p}$).
At any stage during the parse of a sentence, the status of the parser can be represented by a set $\mathcal{S}$ of such states, called a stateset. The closure of $\mathcal{S}$, denoted by $\mathcal{S}'$, is formed by adding to $\mathcal{S}$ states indicating those productions which at this stage in the parse we could begin to recognise, and is the smallest set satisfying,

$$\mathcal{S}' = \mathcal{S} \cup \{[q,0] \mid \exists\, [p,j] \in \mathcal{S}',\ j < n_p,\ X_{p,j+1} = A_q\}$$

If the next symbol encountered is $Y \in V$, the new status of the parser is represented by $\mathcal{S}Y$, the Y successor of $\mathcal{S}$.

$$\mathcal{S}Y = \{[p,j+1] \mid [p,j] \in \mathcal{S}',\ j < n_p,\ X_{p,j+1} = Y\}$$

Initially the status of the parser is represented by $\mathcal{S}_0 = \{[0,0]\}$. The stateset table $\mathcal{T}$, containing all statesets which can occur during parsing is given by

$$\mathcal{T} = \{\mathcal{S}_0\} \cup \{\mathcal{S}Y \mid \mathcal{S} \in \mathcal{T},\ \mathcal{S} \neq \{[0,1]\},\ Y \in V,\ \mathcal{S}Y \neq \emptyset\}$$

whose members are indexed, i.e. we refer to the i^{th} member of $\mathcal{T}$ as $\mathcal{S}_i$. Corresponding to the i^{th} member of $\mathcal{T}$, a parsing-state $R(\mathcal{S}_i)$ specifying the parser's operation is computed by

$$\begin{aligned} R(\mathcal{S}_i) = {} & \{(x,\text{reduce } p) \mid [p,n_p] \in \mathcal{S}_i',\ x \in F_1(A_p)\} \\ & \cup \{(Y,\text{shift } l) \mid [p,j] \in \mathcal{S}_i',\ j < n_p, X_{p,j+1} = Y, \mathcal{S}_l = \mathcal{S}_i Y\} \end{aligned}$$

Finally, the parsing table is $\{R(\mathcal{S}) \mid \mathcal{S} \in \mathcal{T}\}$

Throughout this chapter, we will use the grammar $\mathcal{G}_6$ as an example, the productions of which are

0 $S \rightarrow D \perp$
1 $D \rightarrow A$
2 $D \rightarrow C$
3 $A \rightarrow \underline{id} := E$
4 $E \rightarrow T$
5 $E \rightarrow E + T$
6 $T \rightarrow P$
7 $T \rightarrow T * P$
8 $P \rightarrow (E)$
9 $P \rightarrow \underline{id}$
10 $C \rightarrow \underline{if}\ B\ \underline{then}\ A\ L$
11 $B \rightarrow B\ \underline{or}\ \underline{id}$
12 $B \rightarrow \underline{id}$
13 $L \rightarrow \underline{else}\ D$
s=14 $L \rightarrow \Lambda$

In the SLR parsing table for $\mathcal{G}_6$, we represent an entry (Y,shift l) by l in the column headed Y, and an (x,reduce p) entry by -p in the column headed x.

i	S_i		id	+	(	)	*	if	then	or	else	:=	⊥	D	A	C	E	T	P	B	L
0	[0,0]		4					5						1	2	3					
1	[0,1]																				
2	[1,1]												-1								
3	[2,1]												-2								
4	[3,1]											6									
5	[10,1]		8																	7	
6	[3,2]		13		12												9	10	11		
7	[10,2]	[11,1]							14	15											
8	[12,1]								-12	-12											
9	[3,3]	[5,1]		16							-3		-3								
10	[4,1]	[7,1]		-4		-4	17				-4		-4								
11	[6,1]			-6		-6	-6				-6		-6								
12	[8,1]		13		12												18	10	11		
13	[9,1]			-9		-9	-9				-9		-9								
14	[10,3]		4											19							
15	[11,2]		20																		
16	[5,2]		13		12													21	11		
17	[7,2]		13		12														22		
18	[5,1]	[8,2]		16		23															
19	[10,4]										25		-14								24
20	[11,3]								-11	-11											
21	[5,3]	[7,1]		-5		-5	17				-5		-5								
22	[7,3]			-7		-7	-7				-7		-7								
23	[8,3]			-8		-8	-8				-8		-8								
24	[10,5]												-10								
25	[13,1]		4					5						26	2	3					
26	[13,2]												-13								

Stateset and parsing tables for $\mathcal{G}_6$

The operation of the SLR parser is described by the following flow diagram and equivalent pseudo Algol, in which:

F is a stack, to which n is the pointer.

I denotes the input string, and i indicates the first unread symbol.

The stack elements are integers which index the stateset table (0 and 1 corresponding to the initial and final statesets).

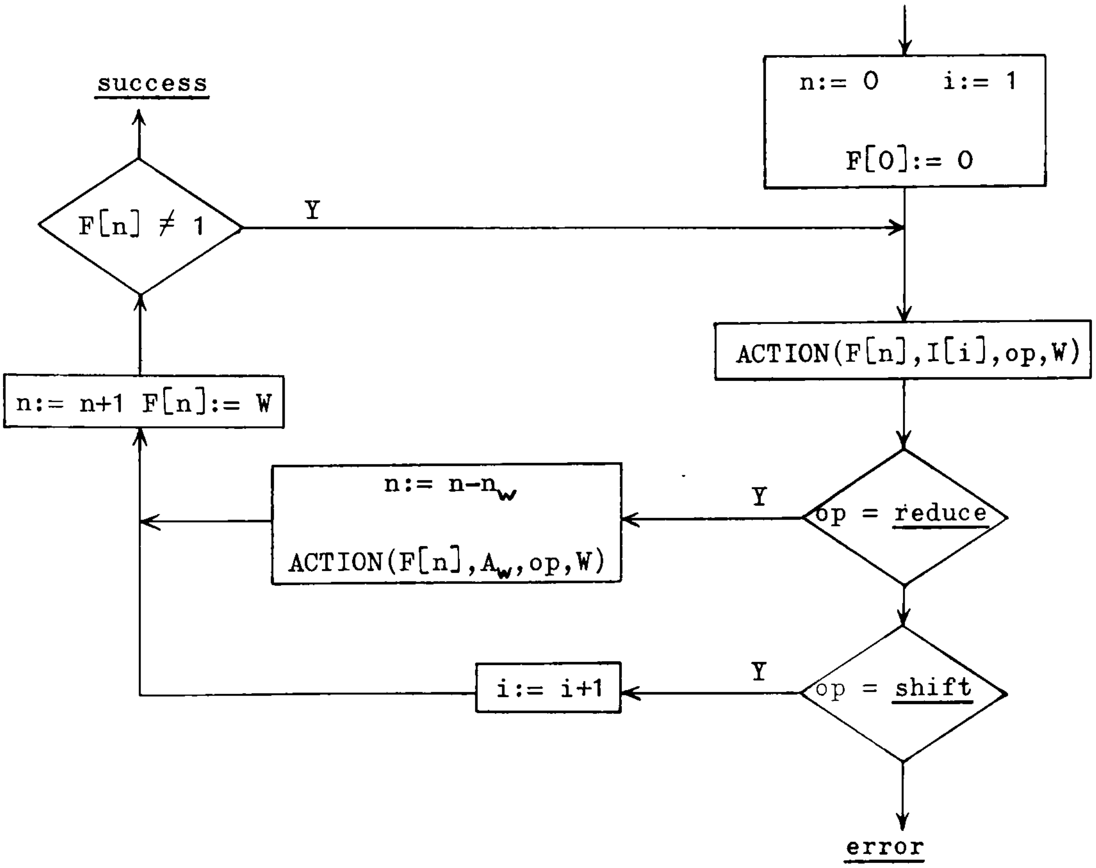
ACTION(l,X,op,W) inspects the parsing-state $R(S_l)$ to assign values to op and W.

If (X,shift j) $\in R(S_l)$ then op:= <u>shift</u> W:= j.

If (X,reduce p) $\in R(S_l)$ then op:= <u>reduce</u> W:= p.

Otherwise op:= <u>error</u>.

The variables op and W are work variables; op indicates the type of operation the parser must perform next, while W specifies either the index of a new stateset, or a production number.



```
F[0]:= n:= 0;  i:= 1; ACTION(F[0],I[1],op,W);
while F[n] ≠ 1 and op ≠ error do
    begin if op = shift then i:= i+1 else
        begin n:= n-$n_w$; ACTION(F[n],$A_w$,op,W) end;
    n:= n+1;
    F[n] := W; ACTION(F[n],I[i],op,W)
    end;
```

Differences between this section and Chapter 2 include the use of an LR(0) stateset table to produce an SLR parsing table(using the sets $F_1(A_p)$ for reduce p entries), the omission of the error stateset $\emptyset$ and statesets having $\perp$ as associated symbol. The procedure ACTION now incorporates NEXT, and is simplified by the replacement of (A,goto j) by (A,shift j). For notational convenience we omit the $'$ from $R'(\mathcal{S})$ and i'.

Chain Productions

Grammars for programming languages often include productions which have no semantic significance for compilation. By this we mean that a reduction involving such a production invokes no special semantic routine, and need not affect the output of the parser. (We have ignored this aspect of parsing, and in the main will continue to do so.) These productions commonly occur in connection with the generation of arithmetic expressions, and are mainly of the form $A_p \rightarrow A_q$. When a production of no semantic significance has a RHS of length one, it is possible for the parser to omit any reduction involving that production, since a reduction of length one does not change the size of the stack. The omission of all such reductions can result in a large increase in parsing speed. This can be achieved within the SLR framework, as is now described.

We say that the p^{th} production is a chain production iff it has no semantic significance and $n_p = 1$, and write $A_p \,\underline{c}\, X_{p1}$ (analagously to $A_p \rightarrow X_{p1}$). In $\mathcal{G}_6$ we let productions 1,2,4 and 6 be chain productions. The chain $E \,\underline{c}\, T \,\underline{c}\, P$ so formed is typical of the situation in programming languages, as for example Algol W where a variable can be parsed as an expression only by means of 12 chain productions.

To modify the SLR table constructor so that the parser can bypass chain productions, it is necessary to replace $\mathcal{S}Y$ by the chained Y successor of $\mathcal{S}$, denoted by $\mathcal{S}_Y$.

$$\mathcal{S}_Y = \{[p,j+1] \mid [p,j] \in \mathcal{S}',\ j < n_p,\ X_{p,j+1} \,\underline{c}^*\, Y, \\ p^{th} \text{ production not a chain production}\}$$

This definition is a generalisation of Y successor, since in the absence of chain productions we have $\mathcal{S}Y = \mathcal{S}_Y$. The tables which result from the use of this generalisation will be termed SLRC tables.

An alternative means of computing $\mathcal{S}_Y$ is provided by the following equations.

Let $W_Y = \mathcal{S}Y \cup \{[p,j] \in \mathcal{S}X \mid [q,1] \in W_Y, A_q = X, \\ q^{th} \text{ production is a chain production}\}$

then $W_Y = \{[p,j+1] \mid [p,j] \in \mathcal{S}',\ j < n_p,\ X_{p,j+1} \,\underline{c}^*\, Y\}$

so $\mathcal{S}_Y = \{[p,j] \in W_Y \mid p^{th} \text{ production is not a chain production}\}$

Consider stateset $\mathcal{S} = \mathcal{S}_6 = \{[3,2]\}$ for grammar $\mathcal{G}_6$. We have

Y	$\mathcal{S}Y$	W_Y	$\mathcal{S}_Y$
id	{[9,1]}	{[9,1]}	{[9,1]}
(	{[8,1]}	{[8,1]}	{[8,1]}
E	{[3,3] [5,1]}	{[3,3] [5,1]}	{[3,3] [5,1]}
T	{[4,1] [7,1]}	{[4,1] [7,1] [3,3] [5,1]}	{[7,1] [3,3] [5,1]}
P	{[6,1]}	{[6,1] [4,1] [7,1] [3,3] [5,1]}	{[7,1] [3,3] [5,1]}

S_T and S_P are of interest here. In W_T we include the members of $\mathcal{S}E$ because $[4,1] \in \mathcal{S}T$. In W_P we include the members of $\mathcal{S}T$ because $[6,1] \in \mathcal{S}P$. Then, since $[4,1] \in W_P$, we also include the members of $\mathcal{S}E$. States $[4,1]$ and $[6,1]$ are deleted to form $\mathcal{S}_T$ and $\mathcal{S}_P$, since productions 4 and 6 are chain productions.

We discuss the differences between the SLRC and SLR tables for $\mathcal{G}_6$, in an attempt to indicate the effects of utilising chained successor statesets. To aid in this comparison, the statesets in the SLRC table have been numbered in correspondence with their SLR counterparts, with the result that this numbering is not consecutive.

First consider $R(\mathcal{S}_0)$. The SLR entries (D,shift 1), (A,shift 2), (C,shift 3) occur in the SLRC parsing table as (D,shift 1), (A,shift 1), (C,shift 1). This results from the elimination of statesets 2 and 3, which in the SLR table contained reduce entries for productions 1 and 2 respectively. Since these are chain productions, the SLRC parser can ignore the reductions, and when in $R(\mathcal{S}_0)$ will regard A and C as being implicitly reduced to D. The parsing-state entries correspond to this implicit reduction having been made.

$R(\mathcal{S}_{25})$ exhibits the same behaviour, for the same reasons, (A,shift 2), (C,shift 3) being replaced by (A,shift 26), (C,shift 26).

The elimination of $\mathcal{S}_{11}$, which like $\mathcal{S}_2$ and $\mathcal{S}_3$ only performs a chain reduction (on production 6), affects the entries on P in $\mathcal{S}_6$, $\mathcal{S}_{12}$, $\mathcal{S}_{16}$. These are changed to correspond to the respective entries for T.

SLR statesets $\mathcal{S}_8$, $\mathcal{S}_9$ and $\mathcal{S}_{10}$ are involved in a more complicated transformation. Because some, but not all, entries in $\mathcal{S}_{10}$ specify chain reductions, $\mathcal{S}_{10}$ cannot be eliminated, and instead is amended as follows. The chain reduce 4 entries are replaced by the corresponding entries from $\mathcal{S}_9$, since production 4 reduces a T to an E, and $\mathcal{S}_8 E = \mathcal{S}_9$. In particular, the ( ),reduce 4) entry is deleted.

i	S_i	id	+	(	)	*	if	then	or	else	:=	⊥	D	A	C	E	T	P	B	L
0	[0,0]	4					5						1	1	1					
1	[0,1]																			
4	[3,1]										6									
5	[10,1]	8																	7	
6	[3,2]	13		12												9	10	10		
7	[10,2] [11,1]							14	15											
8	[12,1]							-12	-12											
9	[3,3] [5,1]		16							-3		-3								
10	[7,1] [3,3] [5,1]		16			17				-3		-3								
12	[8,1]	13		12												18	10′	10′		
13	[9,1]		-9		-9	-9				-9		-9								
14	[10,3]	4												19						
15	[11,2]	20																		
16	[5,2]	13		12													21	21		
17	[7,2]	13		12														22		
18	[5,1] [8,2]		16		23															
10′	[7,1] [5,1] [8,2]		16		23	17														
19	[10,4]									25		-14								24
20	[11,3]							-11	-11											
21	[5,3] [7,1]		-5		-5	17				-5		-5								
22	[7,3]		-7		-7	-7				-7		-7								
23	[8,3]		-8		-8	-8				-8		-8								
24	[10,5]											-10								
25	[13,1]	4					5						26	26	26					
26	[13,2]											-13								

SLRC tables for G_6

Similar comments apply when we consider statesets $\mathcal{S}_{12}$, $\mathcal{S}_{18}$ and $\mathcal{S}_{10}$. Since $\mathcal{S}_{12}E = \mathcal{S}_{18}$, chain reduce 4 entries in $\mathcal{S}_{10}$ must be replaced by the corresponding entries from $\mathcal{S}_{18}$. These amendments to $\mathcal{S}_{10}$ are incompatible with those described previously, since in this case the (else,reduce 4) and ($\perp$,reduce 4) entries are deleted. In consequence, a new stateset appears in the SLRC tables, and is referred to as $\mathcal{S}_{10'}$.

For a programming language having a grammar which defines arithmetic expressions in the usual way, the replication of statesets, in the way $\mathcal{S}_{10}$ is duplicated, is a prominent feature of the SLRC tables for the grammar. The elimination of statesets occurs less often, so that in a practical situation an SLRC table usually contains many more statesets than the SLR table for the same grammar. However, this is more than offset by the increased speed of the SLRC parser.

If all replications of the same stateset are merged back to a single stateset, chain reductions are still avoided, but at the expense of the parser's LR(0) error detection capability. The stack must then be checked whenever a reduce operation is performed.

Alternatively, chain reductions may be partially avoided, by using a modification of $\mathcal{S}_Y$. This modified $\mathcal{S}_Y$ is computed (in Q) by

```
Q := $\mathcal{S}Y$;
while $\forall$ [q,l] $\in$ Q, q is a chain production do
Q := $\{[p,j] \in \mathcal{S}X \mid [q,l] \in Q, X = A_q\}$
```

The tables which result from using this modification will be termed SLRPC tables. Statesets are eliminated from an SLR table, but none are replicated. (It is therefore possible to produce SLRPC tables directly from SLR tables.) The parser's LR(0) error detection capability is maintained but some chain reductions will still be performed.

A number of complications arise in both the SLRC and SLRPC methods, and are now discussed.

More than one final stateset can be created by these algorithms. If no chain production has X_{01} as its LHS then this anomaly does not occur.

In the SLRC tables for $\mathcal{G}_6$, the entries in $\mathcal{S}_0$ and $\mathcal{S}_{25}$ for D are never used, and can be deleted. In general, if all productions of which a nonterminal A is the LHS are chain productions, then all entries for A may be deleted from an SLRC table. Any statesets which are only accessed from these entries can also be deleted.

There is a slight degradation of error detection in going from SLR to SLRPC to SLRC parsers, similar to that resulting from lookahead minimisation; namely that error detection may be deferred until after the stack is reduced, but before the next input symbol is read. This means, as already noted, that the LR(0) detection capability is maintained.

The concept of an associated symbol for SLRC and SLRPC statesets is less simple than for SLR statesets. A generalisation of the original definition must be made. Let $\mathcal{S}$ be any member of a stateset table, other than $\mathcal{S}_0$. Then the set

$$\{Y \in V \mid \forall\ [p,j] \in \mathcal{S} \quad X_{pj}\ \underline{c}^*\ Y\}$$

is not empty, and may be regarded as an associated symbol set for $\mathcal{S}$.

An important point, which must be made, is that the SLRC and SLRPC methods are not strictly parsing algorithms within our definition, although we will continue to refer to them as such. This is because they do not determine the canonical derivation for a sentence; rather they determine what Gray and Harrison call a <u>sparse parse</u> i.e. a specification of a parse from which reductions by a subset of the productions have been omitted (in our case, the chain productions).

One consequence is that the sentences of some grammars which are not SLR can be 'parsed' by means of the SLRC or SLRPC algorithms. Since only a sparse parse is to be determined, in certain circumstances additional right context may be examined before an essential reduction is called for. Grammar $\mathcal{G}_4$ (constructed in Chapter 2), which is LR(1) but not LALR, is an SLRC grammar if productions 5 and 6 are chain productions. Perhaps more surprising is the existence of SLRC grammars which are not LR(1), or even unambiguous. Consider $\mathcal{G}_7$ which has productions

0 $S \rightarrow A \perp$
1 $A \rightarrow c$
2 $A \rightarrow B$
s=3 $B \rightarrow c$

Although clearly ambiguous, if productions 1,2 and 3 are chain productions, then $\mathcal{G}_7$ is SLRC. The sentence $c\perp$ has a unique (trivial) sparse parse.

In appendix 1 (1.5) we establish theorem A, which states that a Λ-free SLR grammar is an SLRC grammar, for any set of chain productions. The Λ-free premise of theorem A cannot be removed, as is shown by $\mathcal{G}_8$, with productions

0 $S \rightarrow A \perp$
1 $A \rightarrow B\ L$
2 $A \rightarrow C\ d$
3 $A \rightarrow L\ d$
4 $B \rightarrow e$
5 $C \rightarrow e$
s=6 $L \rightarrow \Lambda$

$\mathcal{G}_8$ is SLR, but not SLRC if productions 4 and 5 are chain productions.

(Stateset and parsing tables for $\mathcal{G}_7$ and $\mathcal{G}_8$ are exhibited in appendix 2.)

The difficulty arises from the combination of an empty RHS with the inaccuracy of the SLR method's lookahead calculation. If any SLR grammar is not SLRC, the inadequacies must involve the reduction of empty RHSs. A rudimentary refinement of the lookahead for such a reduction, local to the stateset in question, will be sufficient to remove the inadequacy. The proof of theorem A indicates that the corresponding result for LALRC grammars (defined analagously to SLRC), is not complicated by Λ; that an LALR grammar is LALRC.

LR(0) Statesets

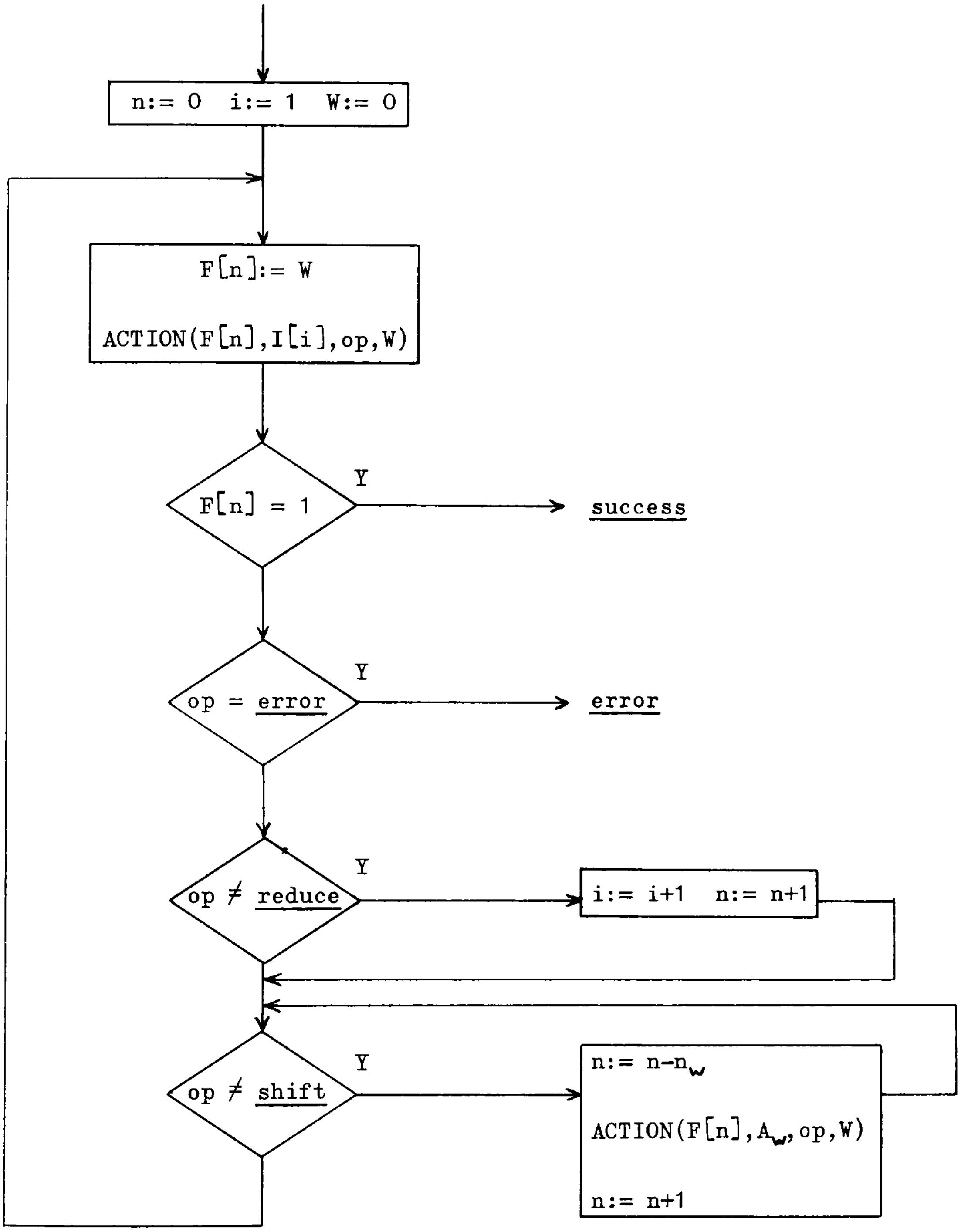
Technically, an SLR stateset is said to be LR(0) if it is adequate without any provision of lookahead, in which case either all the corresponding parsing-state entries are shifts, or all are reduce entries for the same production. If all are shifts, the input must still be inspected to determine the next stateset, and for this reason we shall regard an SLR stateset as having the LR(0) property only if all entries are reduce p for some unique p. In such circumstances the parser's next operation may be considered independent of the lookahead symbol. Statesets which are LR(0) by this definition can be eliminated from the SLR stateset table at the cost of introducing a new type of parsing-state entry. Their elimination increases the speed of the parser as well as reducing the size of the stateset table.

Suppose an SLR stateset $\mathcal{S}_i$ is LR(0). Then $\mathcal{S}_i = \{[p,n_p]\}$ for some p, and $R(\mathcal{S}_i) = \{(x,\text{reduce } p) \mid x \in F_1(A_p)\}$. Minimisation of the lookahead converts these to the single entry (Λ,reduce p). Now consider a stateset $\mathcal{S}$, whose parsing-state contains an entry (X,shift i). Since the parser's action in $\mathcal{S}_i$ is known to be unique and independent of the lookahead, this entry could be altered to (X,'shift i, and then reduce p'). The shift portion of this operation involves stacking i.

Since n_p must be positive, the reduce p will immediately remove i, and so the actual value stacked is immaterial. If $X \in V_N$ the entry can be just (X,reduce p), if reduce operations on nonterminals increment the stack pointer. The new type of entry is needed when $X \in V_T$ and is written (X,scan reduce p). The scan indicates that X must be read from the input, and the stack pointer incremented, before the reduction takes place.

ACTION is modified to yield op = scanreduce and W = p for such an entry. We again specify an SLR (and SLRC, SLRPC) parsing algorithm by means of a pseudo Algol description and corresponding flow diagram. In the absence of scan reduce and nonterminal reduce entries this parser is in fact equivalent to that given earlier in this chapter.

```
F[0]:= n:= 0; i:= 1; ACTION(F[0],I[1],op,W);
while F[n] ≠ 1 and op ≠ error do
    begin if op ≠ reduce then
        begin i:= i+1; n:= n+1 end;
    while op ≠ shift do
        begin n:= n-n_W;
        ACTION(F[n],A_W,op,W); n:= n+1
        end;
    F[n]:= W; ACTION(F[n],I[i],op,W)
    end;
```



Elimination of LR(0) statesets affects the parser's error detection in the same way as the elimination of statesets by SLRPC techniques. The number of statesets which are LR(0) is in practice much greater than the number eliminated by SLRPC.

The SLRC tables for $\mathcal{G}_6$, with LR(0) statesets eliminated, provide an example. Scan reduce p entries in the parsing table are represented by *p in the appropriate columns. The original (SLR) stateset numbering is retained.

i	S_i	<u>id</u>	+	(	)	*	<u>if</u>	<u>then</u>	<u>or</u>	<u>else</u>	:=	⊥	D	A	C	E	T	P	B	L
0	[0,0]	4					5						1	1	1					
1	[0,1]																			
4	[3,1]										6									
5	[10,1]	*12																	7	
6	[3,2]	*9		12												9	10	10		
7	[10,2] [11,1]							14	15											
9	[3,3] [5,1]		16							-3		-3								
10	[7,1] [3,3] [5,1]		16			17				-3		-3								
12	[8,1]	*9		12												18	10′	10′		
14	[10,3]	4												19						
15	[11,2]	*11																		
16	[5,2]	*9		12													21	21		
17	[7,2]	*9		12														-7		
18	[5,1] [8,2]		16		*8															
10′	[7,1] [5,1] [8,2]		16		*8	17														
19	[10,4]									25		-14								-10
21	[5,3] [7,1]		-5		-5	17				-5		-5								
25	[13,1]	4					5						-13	-13	-13					

Table Compaction

For the purpose of driving the parser, it is only necessary to retain the parsing table; the stateset table may be discarded. To represent economically the information contained in a parsing table, a number of compaction techniques are available. Initially we consider these techniques independently of each other.

Conditions may be postulated under which parsing-states can be merged, by combining their entries and changing references to the component parsing-states to refer to their combination. Sufficient conditions for two parsing-states to be merged, without detriment to the parser's error detection are now given.

1) Nonterminal entries must not conflict.
2) The terminal shift (and scan reduce) entries must be identical.
3) Terminal reduce entries must not conflict.
4) The set of productions used in terminal reductions must be identical.

(The entries of two parsing-states conflict if for some symbol in V they contain distinct entries.)

All syntactic errors are detected by the parser on terminal symbols, so parsing-states are only accessed with valid nonterminals. Thus the addition of nonconflicting nonterminal entries does not alter the parser's behaviour. Similarly we may add an (x,reduce p) entry to a parsing-state if there is already in the parsing-state a (y,reduce p) entry for some y ($x,y \in V_T$), and no entry for x (compare lookahead minimisation).

Two parsing-states satisfying the above conditions can therefore have entries added to make them identical, when clearly they can be merged. Such a merge would eliminate a parsing-state and economise on any common entries. Unfortunately, for parsing tables derived from LR(0) stateset tables, if $R(S_i)$ and $R(S_j)$ satisfy these conditions, we must have $i = j$; hence no merging is allowed in the tables with which we are here concerned. Substantiation of this remark is deferred to appendix 1 (1.6).

Parsing tables can conveniently be regarded as transition matrices, with rows indexed by parsing-state number and columns indexed by symbol. An element of such a matrix may be blank, corresponding to an error, or else indicates a type of operation (shift, scan reduce, reduce) and the number of either a parsing-state or a production. It may be possible to reduce the number of bits required to encode an element of the matrix. If within each row (or each column) the range of the entries is small then an economy can be made by the following means. Let the entries in the row (column) be a_i for $1 \leq i \leq n$ say. We evaluate $c = \min_i(a_i) -1$, and then the entries can be stored as $a_i - c$. The value of c for each row (column) must also be recorded.

To decrease the range of entries in a row (column) we may reorder parsing-states and productions. Indeed, for SLR, by reordering parsing-states, we can ensure that the shift entries in a column form a sequence of consecutive integers; a consequence of the fact that all references to a particular parsing-state must lie in one column of the matrix, namely, the column indexed by the parsing-state's associated symbol. Different constants can be kept for the different types of entry, e.g. one constant could apply to reduce (and scan reduce) entries and another to shift entries. Reduce entries should then be handled by row, since the number of different production numbers used in a parsing-state is likely to be small.

Applied to the SLRC parsing table (with LR(0) statesets eliminated) for $\mathcal{G}_6$, this technique can be used to reduce the size of the matrix entries from 7 bits to 4 bits (including 2 bits which indicate the type of operation) and thus reduce an overall requirement of 2394 bits to 1535 bits.

To represent a matrix in computer memory it must be linearised, and we can take advantage of the indirection involved in such a linearisation to overlay rows (parsing-states) of the matrix. The conditions for such overlaying are as for the merging of parsing-states, but now, by partitioning the matrix into a terminal section and a nonterminal section, conditions 2) to 4) may be applied to the terminal submatrix, and (independently) condition 1) to the nonterminal submatrix. For SLR, conditions 2) to 4) imply that the two parsing-states have identical terminal entries, which means that determination of the optimum overlay is trivial for the terminal submatrix. The problem of determining the optimum solution for the nonterminals is combinatorial in nature, as is indicated by an example.

	A	B	C
1	(shift,a)		
2	(shift,b)	(shift,c)	
3		(shift,d)	(shift,e)
4			(shift,f)

We can overlay 1 with 4 and are done, but better would be 1 with 3 and 2 with 4. (This example is similar to that given in Chapter 2 on the merging of statesets.) In general, more overlaying of nonterminal rows than terminal rows can take place, a consequence of the less stringent condition imposed.

After overlaying the rows of the SLRC parsing table (with LR(0) statesets eliminated) for $\mathcal{G}_6$, we obtain the following terminal and nonterminal submatrices, and associated indirection table.

Terminal Submatrix

	<u>id</u>	+	(	)	*	<u>if</u>	<u>then</u>	<u>or</u>	<u>else</u>	:=	⊥
1	4					5					
2										6	
3	*12										
4	*9		12								
5							14	15			
6		16							-3		-3
7		16			17				-3		-3
8	4										
9	*11										
10		16		*8							
11		16		*8	17						
12									25		-14
13		-5		-5	17				-5		-5

Nonterminal Submatrix

	D	A	C	E	T	P	B	L
1	1	1	1	9	10	10	7	-10
2	-13	-13	-13	18	10′	10′		
3		19			21	21		
4						-7		

Indirection Table

Parsing-state	0	1	4	5	6	7	9	10	12	14	15	16	17	18	10′	19	21	25
Terminal row	1	0	2	3	4	5	6	7	4	8	9	4	4	10	11	12	13	1
Nonterminal row	1	0	0	1	1	0	0	0	2	3	0	3	4	0	0	1	0	2

An entry of 0 in the indirection table is used to indicate that the corresponding row does not contain any entries. By renumbering the parsing-states it is possible to eliminate these entries from the nonterminal portion of the indirection table. If overlaying is not performed, then this renumbering makes a large saving on the nonterminal submatrix.

Direct representation of the parsing table as a matrix is very costly in space. Since the matrix is usually sparse, an obvious saving can be achieved with the following scheme.

The parsing table entries are stored in two vectors named SYM and ACT. Elements of SYM are symbols and those of ACT are their corresponding parsing operations, thus (SYM(i),ACT(i)) forms a parsing table entry. The entries of parsing-state j are located by accessing a vector called STATE with j, to obtain an index and a length. The index specifies the location, in SYM and ACT, of the first entry in parsing-state j, while the length gives the number of entries in the parsing-state. All entries of a parsing-state are placed consecutively in SYM,ACT , following the first. This representation of the parsing table will be referred to as a list representation, as opposed to a direct matrix representation.

To aid in further compactions, we partition SYM and therefore ACT, into a terminal and a nonterminal part, accessed via two vectors called TSTATE and NSTATE. With a terminal lookahead symbol, in parsing-state j, we obtain from TSTATE(j) an index to TSYM and TACT, the terminal parts of SYM and ACT; similarly, for a nonterminal symbol we obtain from NSTATE(j) an index to NSYM and NACT.

The constraints we impose on these vectors, as a consequence of the above construction, are that,

if TSTATE(j) = (ptr,len) then

(TSYM(ptr),TACT(ptr)), ... , (TSYM(ptr+len-1),TACT(ptr+len-1))

must be the terminal entries of R(S_j),

and if NSTATE(j) = (ptr′,len′) then

(NSYM(ptr′),NACT(ptr′)), ... , (NSYM(ptr′+len′-1),NACT(ptr′+len′-1))

must be the nonterminal entries of R(S_j).

The possibility then arises of overlapping rows of the parsing table having common subsets of entries. This will reduce the lengths of TSYM,TACT and NSYM,NACT. The problem of performing this overlapping optimally, subject to the above constraints is combinatorial and nontrivial. (The constraints can be weakened to correspond with conditions 1) to 4) for mergeability.) Special cases such as rows having only one entry, identical entries or included entries assist in obtaining worthwhile compactions by heuristic methods.

If overlapping is not performed, then the length entries in TSTATE and NSTATE are not essential; they can be deduced from the index entries.

The list representation of the SLRC parsing table (with LR(0) statesets eliminated) for G_6 is now given.

Parsing-state	0	1	4	5	6	7	9	10	12	14	15	16	17	18	10′	19	21	25	
Index	1		3	4	5	7	9	12	16	18	19	20	22	24	26	29	31	36	TSTATE
Length	2	0	1	1	2	2	3	4	2	1	1	2	2	2	3	2	5	2	
Index	1			4	5				8	11		12	14			15		16	NSTATE
Length	3	0	0	1	3	0	0	0	3	1	0	2	1	0	0	1	0	3	

TSYM and TACT

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
symbol	id	if	:=	id	id	(	then	or	+	else	⊥	+	*	else	⊥	id	(	id
operation	4	5	6	*12	*9	12	14	15	16	-3	-3	16	17	-3	-3	*9	12	4

	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37
symbol	id	id	(	id	(	+	)	+	)	*	else	⊥	+	)	*	else	⊥	id	if
operation	*11	*9	12	*9	12	16	*8	16	*8	17	25	-14	-5	-5	17	-5	-5	4	5

NSYM and NACT

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
symbol	D	A	C	B	E	T	P	E	T	P	A	T	P	P	L	D	A	C
operation	1	1	1	7	9	10	10	18	10′	10′	19	21	21	-7	-10	-13	-13	-13

Overlapping of rows leaves NSYM,NACT unaffected in this case (since all nonterminal entries are distinct) but reduces TSYM,TACT by almost 50%.

TSTATE

Parsing-state	0	1	4	5	6	7	9	10	12	14	15	16	17	18	10′	19	21	25
Index	1		3	4	5	7	11	11	5	1	19	5	5	10	9	20	14	1
Length	2	0	1	1	2	2	3	4	2	1	1	2	2	2	3	2	5	2

TSYM and TACT

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
symbol	id	if	:=	id	id	(	then	or	*	)	+	else	⊥	*	+	)	else	⊥
operation	4	5	6	*12	*9	12	14	15	17	*8	16	-3	-3	17	-5	-5	-5	-5

	19	20	21
symbol	id	else	⊥
operation	*11	25	-14

In the list representation economies can be made by lookahead minimisation. If a parsing-state contains a terminal reduce p entry, then all terminal reduce p entries in the parsing-state may be replaced by a single (Λ,reduce p) entry, which is regarded as a default entry for the parsing-state, to be used only as a last resort (recall that (Λ,reduce p) is equivalent to (x,reduce p) $\forall$ x $\in V_T$). To ensure this, we position the (Λ,reduce p) entry last in the sequence of entries in TSYM,TACT corresponding to the parsing-state.

Since we can add nonconflicting nonterminal entries to a parsing-state, we can have a default nonterminal entry for the most frequently occurring operation on a nonterminal in the parsing-state. This is only relevant to SLRC and SLRPC tables, since for an SLR parsing-state, all operations on nonterminals are distinct. An alternative technique is to associate with each nonterminal symbol a default operation (a default on columns as opposed to rows of the matrix). It is then possible to delete from NSYM,NACT all entries consisting of a nonterminal and its default operation.

If the parser does not find an entry for a nonterminal symbol, this can only have occurred due to that entry having been deleted, so the parser can take the default operation for the nonterminal symbol.

Notice that the use of default entries necessitates the adjustment of the length components in TSTATE and NSTATE.

When there are fewer entries, on average, in a column than in a row of the parsing table, it may be advantageous to use an inverted list representation in which the roles of STATE and SYM are interchanged. SYM is then accessed with the current symbol to give an index and length, the index referring to STATE and ACT. ACT(i) specifies the parsing operation to be performed if the parser is in parsing-state STATE(i). Thus the parsing table can be represented by columns instead of rows.

The method of reducing the size of the individual parsing table entries described earlier also applies to the list representation, but the savings are not so great.

Use of a list representation, while yielding a large economy of storage, has serious implications for the speed of the parser.
The parsing table is examined by the procedure ACTION, and if a matrix representation is used, ACTION inspects only the single entry of the matrix located by the current symbol and parsing-state number.
With the list representation, ACTION must search the entries for the current parsing-state until either the current symbol is found, or the sequence of entries is exhausted. Clearly this search will result in some deterioration of the speed of the parser. The time taken for the search can be made more acceptable on modern computers by the use of hardware implemented searching instructions, such as are available on the IBM 360, PDP 10 and Univac 1108.

Reducing the size of the parsing table entries also results in a degradation of the parser's speed, because it is then necessary to compute the operation required after an entry has been accessed.

Inadequate Statesets

Programming languages are not necessarily strictly context free, and because of this, a CFG constructed to represent the syntax of such a language will often be ambiguous. Context sensitivity is then restored by means of semantic routines which remove the ambiguities inherent in the grammar.

This provides an example of a situation for which modifications to a grammar in order to obtain a version which is SLR may not always be the best solution to the problem of parsing sentences in the language. It may be more expedient to produce an inadequate SLR parsing table for an otherwise convenient grammar, and use semantic routines to resolve inadequacies as they arise.

We now describe a scheme for incorporating this technique into an SLR parser, given in terms of the list representation discussed in the previous section.

If a parsing-state is inadequate, then for some symbol $x \in V_T$, the parsing-state contains more than one entry. All the entries on x (in the parsing-state) are replaced in TSYM,TACT by a single entry (x,multiple z). The value of z must specify, by some means, the set of parsing operations which were replaced. One method is for z to prescribe an index and a length, the length being the number of operations, and the index locating the first of them in a supplementary vector which we will refer to as SUPTACT. The operations, of which at most one can be a shift (or scan reduce), are stored sequentially in SUPTACT, with any shift (or scan reduce) operation being placed last. Among the reduce operations the ordering is irrelevant. Here we are assuming that semantic routines are only called when reductions are performed, and that at most one reduction can be applicable at any point in a parse.

For an entry (x,multiple z), the procedure ACTION uses z to compute the appropriate index and length, and records these in two variables zp and zc. It then assigns to op and W from SUPTACT(zp), and in fact we must have op:= reduce.

Before applying any reduce operation, the parser calls a semantic routine to determine whether the reduction is semantically valid, and only if this is the case is the reduction performed. If the reduction is not valid, the parser calls ACTION in an endeavour to obtain an alternative operation, indicating that an operation from SUPTACT is required by giving a negative parsing-state number as a parameter.

In these circumstances, ACTION inspects the value of zc.

If zc = 1 then op is returned as error.

If zc > 1 then the counter zc is decremented by one,
the pointer zp is incremented by one,
op and W are returned from SUPTACT(zp).

Because of the possibility of an adequate reduce operation being rejected on semantic grounds, ACTION sets zc to 1 whenever it returns an adequate reduce or scan reduce operation (from TACT).

This SLR parser is described by the following pseudo Algol. The specification of the algorithm has been simplified by regarding zp and zc as own variables of the procedure ACTION.

```
F[0]:= n:= 0; i:= 1; ACTION(F[0],I[1],op,W);
while F[n] ≠ 1 and op ≠ error do
    begin if op ≠ reduce then
        begin i:= i+1; n:= n+1 end;
    while op ≠ shift and op ≠ error do
    if reduce W is semantically invalid then
        begin ACTION (-1,I[i],op,W);
        if op ≠ reduce and op ≠ error then
            begin i:= i+1; n:= n+1 end
        end
    else begin n:= n-$n_W$;
        ACTION(F[n],$A_W$,op,W); n:= n+1
        end;
    if op ≠ error then
        begin F[n]:= W; ACTION(F[n],I[i],op,W) end
    end;
```

If SUPTACT contains no shift or scan reduce operations, the conditional statement following ACTION(-1,I[i],op,W) may be omitted.

It can be important to compact inadequate parsing-states, since they may contain a large number of entries, as for example, in connection with the association of type information with identifiers in Algol like programming languages.

For this reason we would normally overlap the elements of the vector SUPTACT. The compaction techniques described in the previous section apply to both adequate and inadequate parsing-states. In particular, a multiple entry will often be the default operation for an inadequate parsing-state.

A useful modification can be made to the criteria for parsing-state merging. Conditions 1) to 4) given in the previous section do not seem appropriate for inadequate parsing-states, which already contain conflicting terminal entries. We therefore suggest the following conditions for two inadequate parsing-states to be mergeable.

1) Nonterminal entries must not conflict.
2) The terminal shift (and scan reduce) entries must be identical.
3) The adequate terminal reduce entries must be identical.
4) The multiple entries must be on the same set of terminals.
5) The inadequate entries in one parsing-state must be a subset of those in the other.

These conditions do permit merging of inadequate parsing-states in SLR tables, and ensure that a set of reductions require resolution by semantic means after merging only if their resolution was necessary before merging.

Parsing Tables for PL360 AlgolW and XPL

Programs were written to generate list representations of SLR, SLRPC and SLRC parsing tables, and applied to grammars for the programming languages PL360, AlgolW and XPL. These languages are described by Wirth (1968), Wirth and Hoare (1966) and McKeeman, Horning and Wortman (1970) respectively.

In each case, the syntax employed was a version intended for use in a compiler for the language. For PL360 and AlgolW this was available from the source listings of the two compilers (which are written in PL360). In all, three grammars for AlgolW were analysed, and are here designated AlgolW1, AlgolW2 and AlgolW3. AlgolW2 represents the current compiler syntax, and was used to implement a replacement SLR parser for the compiler, as is described in the next section. AlgolW1 is an older version of the syntax, which in conjunction with the PL360 grammar, was the basis for most of the decisions presented here. AlgolW3 is an extension and modification of the basic AlgolW syntax; its ambiguity and some eccentricities of a simple precedence nature have been removed, and a number of language extensions incorporated. Tables were constructed for XPL to enable a comparison to be made with the results obtained by Lalonde (1971) from his LALR parsing table constructor.

In addition, sufficient information was available to determine, for each syntax, those productions which should be considered as chain productions (one semantic routine of XPL was ignored to complete the chain for arithmetic expressions).

Uncompacted lists were produced initially, and measurements made to indicate the effectiveness of the various compaction techniques. From this empirical evidence, a sequence of compactions was chosen which should be capable of producing an economic list representation. Before describing this selection of compaction methods, we discuss informally the interactions which take place between the different methods.

For a matrix representation, the only interaction is between overlaying of rows and reducing the number of bits needed to represent an element of the matrix. If both techniques are to be employed, the reduction can with advantage be performed first, when the resulting smaller entries may assist in obtaining additional overlaying.
If overlaying is not performed, the renumbering of rows to economise on empty nonterminal rows becomes very important; this renumbering may be incompatible with that required to reduce the size of elements of the matrix.

In a list representation, we have the possibility of incompatability between the use of default entries and overlapping, where an instance of one may prevent an instance of the other (since a default entry must occur last). If a nonterminal default is taken by symbol (i.e. column) then incompatability can only arise if overlapping is performed under the nonconfliction criterion. Again, reduction of the size of entries should be done before overlapping, which may be aided by the reduction.

The three algorithms, SLR, SLRPC and SLRC were regarded as requiring individual evaluation, and were applied to each grammar. The decision to use a list representation stems from both the sparseness (5% to 15%) and the magnitude (up to 70000 elements) of the matrices required for a full matrix representation. Other techniques are of course available for representing sparse matrices, the list representation we have described was chosen as being relatively economic and convenient. The inverted form of list representation was not used, since columns contained more entries than rows.

The merging of inadequate statesets was done in an ad hoc fashion; any two mergeable statesets were merged, and this was continued until no further merging was possible. Inadequate entries were then replaced by multiple entries, the inadequate entries corresponding to a given multiple entry being placed in SUPTACT. Overlapping in SUPTACT was performed by an elementary sorting technique, which produced excellent economies, and is described in more detail in Appendix 2.

The techniques of LR(0) stateset elimination and terminal default entries are closely related (both depend on lookahead minimisation). They produce major savings of terminal entries (between 50% and 70%) and increase the speed of parsing by removing parsing-states and reducing the length of rows contained in TSYM,TACT. It was decided to apply these techniques fully before attempting any other compactions. In view of this decision, it would be permissable to modify the constructor algorithm so as to suppress the production of LR(0) statesets.

Having utilised terminal defaults we consider default entries for the nonterminals. Taking a default action for each nonterminal symbol reduced the total number of nonterminal entries by about 65% for SLRC tables, and about 90% for SLR tables; sufficient to justify full use of the technique. Overlapping of nonterminal entries was not attempted since it was determined that few entries could be saved from NSYM,NACT (overlapping by nonconfliction yields no additional benefits in a list representation).

The method of reducing the size of entries was not applied, since only a slight economy could be achieved, and only at the expense of convenience of access. This omission allowed parsing-states to be renumbered to eliminate elements of NSTATE with length components equal to zero i.e. those corresponding to parsing-states having only default entries for nonterminals.

All the compaction techniques applied so far were fairly simple to implement. The final compaction, a heuristic for overlapping terminal entries was more complicated.

Considerable economy was still possible by overlapping in TSYM,TACT , and much of this could be obtained simply by overlaying identical terminal rows. As noted earlier, the determination of the optimum overlap is distinctly nontrivial, particularly in the presence of default terminal entries, which must occur last in a row of entries.

A first attempt at a heuristic method, which overlapped pairs of parsing-states having most elements in common, and performed well for SLR parsing tables, was found to be unacceptable when applied to SLRC tables. This was a consequence of the replication of statesets in SLRC tables, which results in sequences of parsing-states whose sets of terminal entries form a sequence of nested inclusions. Clearly if $A \subset B \subset \ldots \subset D$ we need only store the members of D, in some order. To cater for this situation, the following algorithm was used.

We initially exclude terminal rows consisting of a single entry, and all but one of a set of identical terminal rows.

i) Overlap inclusion sequences of length greater than two (largest first).

ii) Overlap pairwise from the remaining parsing-states (largest first).

iii) Overlap (optimally) those rows excluded initially.

A more detailed description of the algorithm is given in appendix 2.

The grammars for which measurements were made, and parsing tables produced, are characterised in the following table.

Grammar	$\mathcal{G}_6$	PL360	AlgolW1	AlgolW2	AlgolW3	XPL
Terminals	11	62	58	62	71	42
Nonterminals	9	62	84	72	61	49
Productions	15	151	203	191	178	109
Chain productions	4	14	18	18	13	12
Average length of production RHS	2.00	1.93	2.24	2.30	2.46	2.07

The effects of the various compactions as applied to the SLR, SLRPC and SLRC parsing tables for these grammars are recorded in the following tables.

Grammar	$\mathcal{G}_6$	PL360	AlgolW1	AlgolW2	AlgolW3	XPL
SLR	151	1179	2108	2161	2145	1107
SLRPC	160	1269	2375	2434	2229	1182
SLRC	171	1624	5191	5523	5344	1980

Storage requirements (in bytes) of the compacted tables
(for the IBM 360, the encoding is described
in the next section)

Grammar	$\mathcal{G}_6$	PL360	AlgolW1	AlgolW2	AlgolW3	XPL
Statesets	27	224	341	330	328	183
Terminal entries	64	1230	4397	4513	4507	1178
Nonterminal entries	18	280	1493	1552	1202	395
Inadequate statesets	0	3	7	7	0	0
Inadequate entries	-	148	993	810	-	-
Multiple entries	-	68	179	204	-	-
Parsing-states removed by merging	-	0	4	4	-	-
Entries saved	-	-	117	133	-	-
LR(0) statesets removed	10	136	170	166	146	84
Entries saved	28	763	2183	2445	2382	636
Default entries:						
Terminal entries saved	7	96	392	332	682	179
Nonterminal entries removed	12	235	1389	1424	1082	339
NSTATE entries saved by renumbering	11	57	111	100	121	60
Overlapped entries:						
Terminal entries saved	10	117	581	674	1109	210
Inadequate entries saved	-	142	967	792	-	-

Results for SLR parsing tables

Grammar	$\mathcal{G}_6$	PL360	AlgolW1	AlgolW2	AlgolW3	XPL
Statesets	24	219	330	319	323	175
Terminal entries	57	1218	4199	4316	4418	1095
Nonterminal entries	18	280	1493	1552	1202	395
Inadequate statesets	0	3	7	7	0	0
Inadequate entries	-	148	993	810	-	-
Multiple entries	-	68	179	204	-	-
Parsing-states removed by merging	-	0	4	4	-	-
Entries saved	-	-	117	133	-	-
LR(0) statesets removed	7	131	159	155	141	76
Entries saved	21	751	1985	2248	2293	553
Default entries:						
Terminal entries saved	7	96	392	332	682	179
Nonterminal entries removed	9	226	1300	1338	1059	325
NSTATE entries saved by renumbering	11	57	111	100	121	60
Overlapped entries:						
Terminal entries saved	10	96	581	669	1104	199
Inadequate entries saved	-	142	967	792	-	-

Results for SLRPC parsing tables

Grammar	$\mathcal{G}_6$	PL360	AlgolW1	AlgolW2	AlgolW3	XPL
Statesets	25	237	471	469	519	223
Terminal entries	59	1411	6699	7483	7907	1570
Nonterminal entries	18	320	1707	1566	1208	437
Inadequate statesets	0	3	7	7	0	0
Inadequate entries	-	148	993	810	-	-
Multiple entries	-	68	179	204	-	-
Parsing-states removed by merging	-	0	4	4	-	-
Entries saved	-	-	117	133	-	-
LR(0) statesets removed	7	131	159	155	141	76
Entries saved	21	751	1985	2248	2293	553
Default entries:						
Terminal entries saved	5	158	1141	1430	1644	226
Nonterminal entries removed	7	247	1177	1022	781	273
NSTATE entries saved by renumbering	12	76	251	249	316	108
Overlapped entries:						
Terminal entries saved	13	139	1825	2129	3008	487
Inadequate entries saved	-	142	967	792	-	-

Results for SLRC parsing tables

Grammar	G_8	PL360	AlgolW1	AlgolW2	AlgolW3	XPL	
TSTATE	17	88	167	160	182	99	
NSTATE	6	31	56	60	61	39	
TSYM,TACT	19	174	310	323	334	153	SLR
NSYM,NACT	6	45	104	128	120	56	
SUPTACT	-	6	26	18	-	-	
TSTATE	17	88	167	160	182	99	
NSTATE	6	31	56	60	61	39	
TSYM,TACT	19	195	310	328	339	164	SLRPC
NSYM,NACT	9	54	193	214	143	70	
SUPTACT	-	6	26	18	-	-	
TSTATE	18	106	308	310	378	147	
NSTATE	6	30	57	61	62	39	
TSYM,TACT	20	283	817	937	962	304	SLRC
NSYM,NACT	11	73	530	534	427	164	
SUPTACT	-	6	26	18	-	-	
NDEF	8	61	83	71	60	48	
LHS,RHS	14	150	202	190	177	108	

Number of elements in the compacted list representation data structures.

NDEF(i) specifies the default operation for the nonterminal i.

LHS(i) specifies the nonterminal on the LHS of production i.

RHS(i) specifies the length of the RHS of production i.

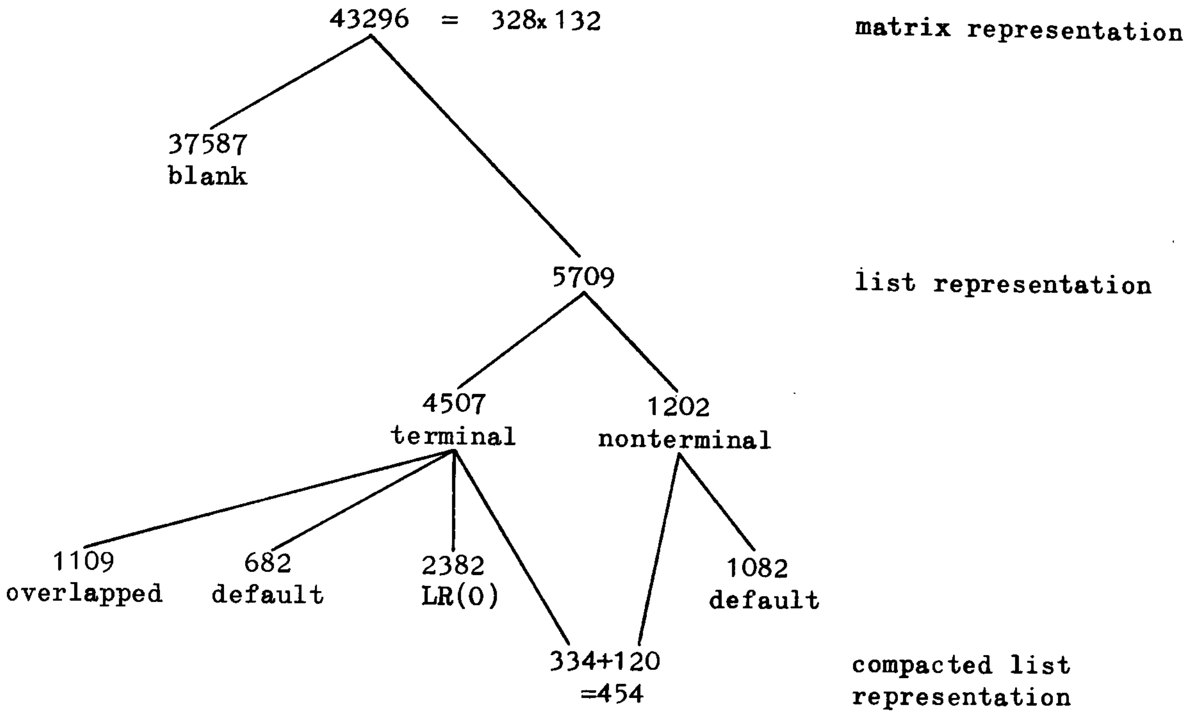
(A_0 and n_0, the LHS and length of the RHS of production 0 are never used by the parser.)

We draw attention to the high proportion of parsing-states (over 80% for SLR and over 85% for SLRC) which for nonterminals have either no entries or only default entries. A contributory factor is the proportion of LR(0) statesets; over 50% for SLR and over 35% for SLRC.

The results for SLRPC data structures are very similar to those for SLR. Indeed, just after the elimination of LR(0) statesets the SLRPC tables contain exactly the same number of entries as the equivalent SLR tables. Fewer nonterminals have default operations in the SLRPC case which results in NSYM,NACT showing an increase. Changes in TSYM,TACT are due to differing overlapping, and because the RHS of only one chain production in AlgolW is a terminal, the effects for that language are slight.

The proximity between the figures for AlgolW2 and AlgolW3 is coincidental; language extensions and the elimination of ambiguities required an additional 600 bytes of storage. This excess was more than regained by the removal of productions which were only introduced to avoid precedence conflicts, and by the convenient use of an empty RHS. Since the AlgolW3 grammar is the only grammar not specifically designed for use by a precedence parser, this improvement was very welcome.

Although the selection of compaction methods was made mainly from results obtained for PL360 and AlgolW1, and cannot be an optimal technique, it proved extremely effective in producing economic representations of parsing tables. We illustrate this with the SLR results for the AlgolW3 grammar, for which a list representation of 5700 entries (equivalent to a matrix representation of over 43000 entries) is reduced to merely 454 entries.



We can compare our results with the number of entries in the matrix used by a simple precedence parser, for each of the grammars.

Grammar	$\mathcal{G}_6$	PL360	AlgolW1	AlgolW2	AlgolW3	XPL
Precedence matrix	220	7688	8236	8308	9372	3822

The comparison is superficial, since these values are for an uncompacted precedence matrix, such entries can be encoded in 2 bits, and furthermore, an encoding of the grammar must also be stored. As an indication of what has proved acceptable in practice for storing parsing tables we quote the storage requirements of the simple precedence parser for AlgolW2 (6730 bytes) and of the mixed strategy precedence parser for XPL (2962 bytes).

Lalonde describes a parsing table constructor based on DeRemer's work, for LA(k)LR(0) grammars. When applied to XPL (with k = 1), he reports that LALR tables requiring 1250 bytes were generated. Because of the different theoretical approach adopted by DeRemer, the data structures utilised differ from those described here, and require a more complex parsing routine.

Timing the SLR Parser

To enable timings of the SLR parser to be made, an encoding of the compacted list representation, and of the SLR parser was designed for the IBM 360.

The Stanford written AlgolW compiler was chosen as the vehicle for these timings. It is a fast, three pass compiler, producing efficient object code for a language which is a substantial extension of Algol 60. The second pass utilises a simple precedence parser embodying a mechanism for bypassing chain reductions. Designed and implemented by S. Graham, no description of the mechanism is as yet published. Since the compiler is written in PL360, it was relatively easy to replace the existing parser and its tables by an SLR parser. Because of the ambiguity of the AlgolW2 grammar used by the compiler, it was necessary to program (in PL360) the modified SLR parser given on page 84, which utilises multiple entries (as well as scan reduce entries and reduce entries on nonterminals).

The programs which construct the compacted tables were modified to output their results as initialised PL360 arrays in a format which we now describe. (The word length of the IBM 360 is 32 (4 bytes each of 8 bits).)

TSTATE and NSTATE are recorded as half word arrays. Each half word contains two fields, one of 10 bits for the index, and one of 5 bits for the length. The high order bit is not needed, and can be set to zero for ease of accessing.

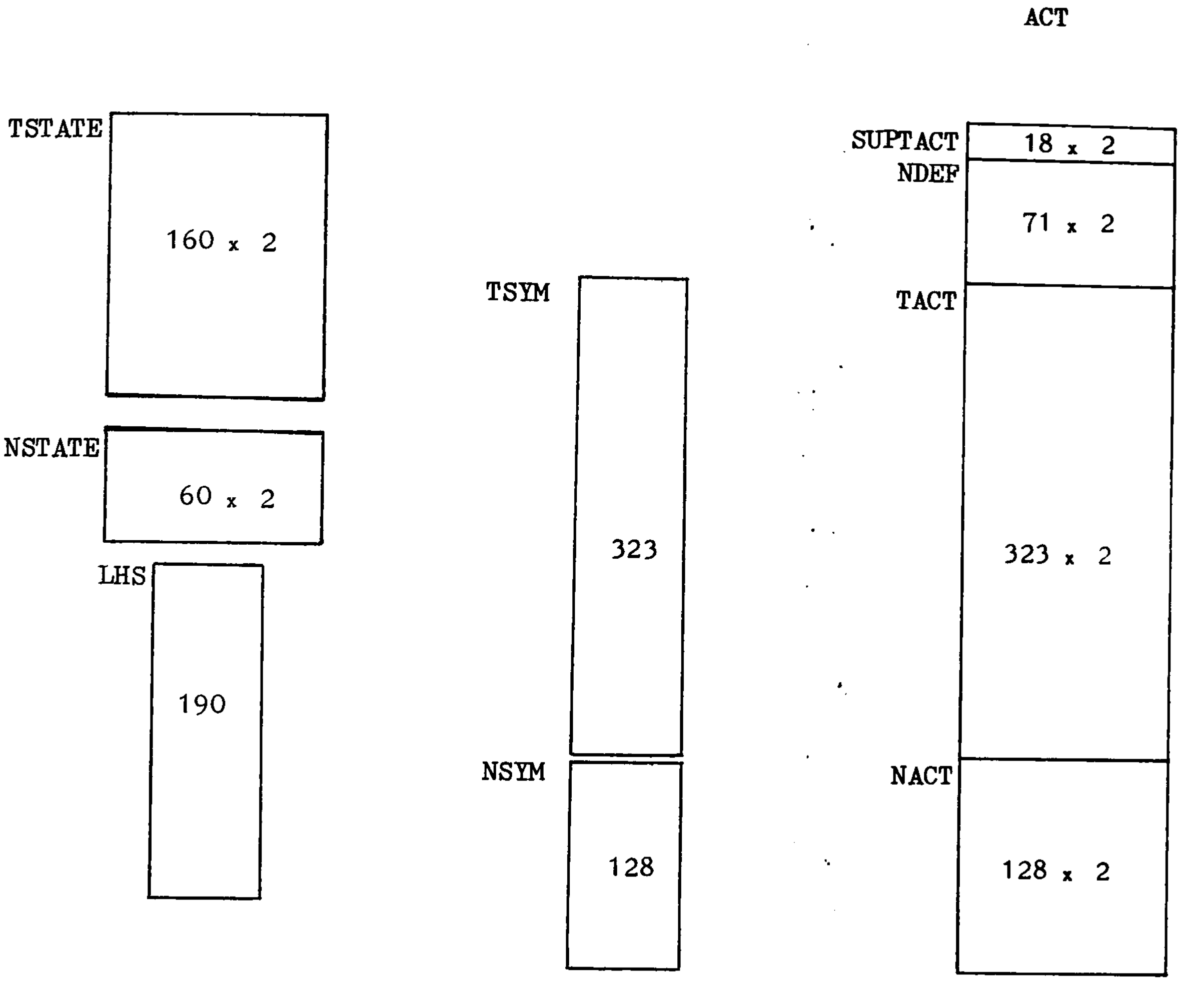
TSYM and NSYM contain byte entries which represent symbols of the grammar.

Because of addressing considerations, SUPTACT, NDEF, TACT and NACT are combined into a single half word array, ACT. Entries in ACT represent one of four types of parsing operation, the type being specified by the high order bits.

1. Multiple entry. Bit 15 = 1 (can only occur in TACT)
   Comprises two fields, one of 10 bits for the index, and one of 5 bits for the length.
2. Reduce entry. Bit 15 = 0, bit 14, 13 = 1
   Comprises two fields, one of 8 bits specifying a production, and one of 5 bits specifying the length of the production RHS.
3. Scan reduce entry. Bits 15, 14 = 0, bit 13 = 1
   (can only occur in TACT and SUPTACT - for AlgolW2 only occurs in TACT)
   Except for bit 14, the format is identical to that of a reduce entry.
4. Shift entry. Bits 15, 14, 13 = 0
   One 13 bit field specifying a parsing-state.

A byte array LHS contains the LHS of each production. We have avoided using a separate array to indicate the length of production RHSs by including this information in each reduce and scan reduce entry.
For each production, a byte array SEMANTIC is used to specify a semantic rule number to be applied whenever a reduction by that production is made. A rule number of zero indicates that no semantic action is required.

This encoding, designed for the 360, with AlgolW2 in mind, imposes various restrictions e.g. the number of productions in a grammar must be less than 256. Clearly these restrictions can be eased at the expense of additional storage. All are met by the SLR, SLRPC and SLRC data structures for AlgolW2.



360 data structures for AlgolW2, SLR - 2161 bytes

These data structures are interrogated by the procedure ACTION, which is in fact written into the parser as inline code. To search a subvector of TSYM or NSYM for the current symbol, the translate and test instruction (TRT) of the 360 is used. By means of this instruction it is possible to search a vector for any of a set of values marked in a second vector. The first vector can be the subvector of TSYM or NSYM, while the second represents the symbols of the grammar, and in which the current symbol is marked.

Since the principal nonterminal A_0 does not occur in NSYM, the entry of the second vector corresponding to A_0 can remain permanently marked. Terminal default entries in TSYM,TACT can then have their symbol component (in TSYM) encoded as A_0. Since A_0 will only occur last in a vector of terminals being searched, default entries will have the required property of only being selected if the current symbol is not found.

To obtain the required timings, it was only necessary to interface the SLR parser with the first pass (lexical analysis) of the AlgolW compiler. However, it proved possible to interface with both the error recovery routines of the syntax phase, and the third pass (code generation), and thus produce a fully working compiler.

Timings were made of the syntax phase of the compiler, using several AlgolW programs as input, for each of the following six parsers.

SLR	The AlgolW2 SLR parser.
SLRPC	The AlgolW2 SLRPC parser.
SLRC	The AlgolW2 SLRC parser.
SLRC′	An SLRC parser for AlgolW2 with one less chain production (a semantic routine was stipulated which splits the 12 step derivation for an expression).
SP	The AlgolW2 simple precedence parser with the elimination of chain derivations suppressed.
SPC	The existing AlgolW2 simple precedence parser.

These timings were obtained on an IBM 360/67, under the Michigan Terminal System (MTS). Over all the tested programs, the performance ranking of the parsers was substantially uniform. Results for five of these programs are now given; times are quoted in seconds, and are not more accurate than ± 0.01 seconds.

Program	Symbols	SLR	SP	SLRPC	SLRC'	SLRC	SPC
1	793	0.43	0.41	0.35	0.31	0.28	0.27
2	2004	1.12	1.09	0.94	0.83	0.76	0.75
3	3740	1.64	1.59	1.34	1.21	1.09	1.10
4	4034	2.40	2.32	2.02	1.78	1.61	1.60
5	6253	4.07	3.92	3.42	3.02	2.76	2.75

Differences between the simple precedence and SLR algorithms in the above table are sufficiently small for us to regard them as being roughly equivalent in terms of parsing speed (the SP parser is approximately 3% faster than the SLR parser, while the SPC parser is only 1% faster than the SLRC parser). However, timing results given by Lalonde (1971) show that his LALR parser runs about 40% faster than the mixed strategy precedence parser with which he makes comparison (MSP). Since this implementation of LALR searches vectors by means of software, the apparent discrepancy deserves comment.

An important distinction between the MSP and SP parsers is to be found in the methods by which they determine the production to use when a reduction is required. MSP must search through the productions whose final symbol matches the stack top, while SP uses a hashing function based on production length as well as final symbol. This is known to be very efficient, and largely negates a potential advantage of the SLR method, namely the a priori knowledge of which production to use whenever a reduce operation is specified.

The precedence matrix employed by MSP has 2 bit entries which must be unpacked, a penalty avoided in SP by the use of one byte for each entry in the matrix (an additional cost of 3800 bytes). Also, the use of the TRT instruction for searching vectors, although advantageous, is far from ideal. Overheads incurred tend to nullify the benefits when the vectors to be searched are short.

Time - space trading is evident in the figures for the four variants of the SLR algorithm (the tables for the SLRC′ parser occupy 4412 bytes). In particular, SLRPC gives a substantial improvement in speed over the ordinary SLR parser, for only a modest space premium. The SLRC′ and SLRC versions are less economical of storage, but yield valuable increases in parsing speed.

To evaluate more precisely the effects of chain elimination, measurements were taken of the number of reduce operations performed by the SLR parsers for each of the test programs. On average, for every 1000 input symbols, the following number of reductions were made.

	SLR	SLRPC	SLRC′	SLRC
Chain reductions	2843	1049	273	0
Other reductions	1201	1201	1201	1201

The preponderance of chain reductions performed by the SLR parser reflects the frequency with which expressions occur in the AlgolW language. The need for bypassing such reductions when parsing is clearly indicated.

Chapter 5

Conclusion

The definition of LA(m)LR(k) grammars is instructive from a theoretical viewpoint, since it points out the dual role of the parameter k in the LR(k) method. In the LA(m)LR(k) formulation we see that k controls the amount of right contextual information which is to be retained directly in the statesets (and therefore affects the number of statesets), while m specifies the amount of lookahead that the parser inspects to determine its next parsing operation. To accommodate every grammar whose sentences can all be parsed using only k symbols of lookahead, both parameters must equal k, and this gives LA(k)LR(k) which is identical to LR(k).

Separation of the two functions of the lookahead parameter adds flexibility to the LR(k) method, and permits an algorithm intermediate to LR(0) and LR(1). This algorithm, LA(1)LR(0) or LALR, has an LR(0) stateset table, but utilises 1 symbol lookahead when parsing. If the complex (order of LR(1)) calculation of this lookahead is simplified by computing an approximate version directly from the grammar, then we obtain SLA(1)LR(0) or SLR. The classes of grammars accepted satisfy the inclusions

$$\mathrm{LR}(0) \subset \mathrm{SLR} \subset \mathrm{LALR} \subset \mathrm{LR}(1).$$

The $[p,j;\alpha]$ notation for a state, as used by Knuth (1965), provided a useful formalism which aided in deriving theoretical results, i.e. the development of LA(m)LR(k) and of SLRC, and the formal comparison and inclusion result for weak precedence and SLR.

Parsing table compaction techniques were very successful in decreasing the storage requirements of the SLR parser, achieving 80-90% reductions on the full list representation. The sequence of compactions suggested is not optimal, and some bias may have been introduced in that most of the grammars involved were designed for use by precedence parsers. However, the results for AlgolW3 indicate that removal of simple precedence features from a grammar may improve storage requirements.

Timing figures for SLR show that, despite the overhead of searching vectors (incurred as a consequence of the list representation of the tables), the algorithm remains comparable in speed with an efficient simple precedence implementation. Also demonstrated is the benefit available from bypassing chain reductions. These comprise around 70% of all reductions made in parsing AlgolW2 programs; their elimination increases parsing speed by almost 50%. Since the syntax phase of the AlgolW compiler occupies about 40% of the total compilation time, this represents an increase of 15% in the speed of compilation.

In summary, our examination of the SLR algorithm confirms that its storage requirements can be made acceptable for practical implementation, and that its generality is adequate to encompass most programming language grammars. Unlike the widely used precedence methods, very little modification of an unambiguous grammar is needed for the SLR algorithm. SLR retains the LR property of immediate detection of syntactic errors; a further advantage over precedence, which is of potential benefit for the provision of error diagnostics and error recovery routines. A further consequence of this property is that the SLR parser's stack always represents the prefix of a sentential form (if α is the string represented by the stack, then $\exists\ \beta \in V_T{}^*$ such that $\alpha\beta$ is a string in some canonical derivation). The integrity of the stack may simplify the design of related sections of a compiler.

These comments also apply to the LALR algorithm, which has greater generality and requires less storage than SLR, but involves a more complex calculation to produce its parsing tables.

To give an indication of the CPU time required for the production of parsing tables, we give the figures for AlgolW2. On an IBM 360/67, SLR tables required 1½ minutes and SLRC tables 3½ minutes. Since the table constructor is written in AlgolW, and was designed for flexibility rather than speed, a recoding could be expected to reduce these times by a factor of 2 or more (c.f. DeRemer (1971) - SLR tables for Algol 60 required 1 minute on a 360/40).

Syntactic problems have been considered here virtually in isolation from considerations of semantics or lexical analysis. The LR methods are amenable to the incorporation of these aspects of compilation. Issues raised by translation are discussed by DeRemer (1969) where the simple Polish transduction grammars of Lewis and Stearns (1968) feature prominently.

Further optimisations of parsing speed are possible. By using a mixed matrix-list representation, a renumbering of the parsing-states could ensure that parsing-states having many entries were represented in a matrix, while those with few entries were stored as vectors. The nonterminal tables for the SLRC method tend to be suitable for such a scheme. Another possibility is the sorting of elements of vectors in the list representation by some estimate of their frequency of occurrence (during parsing). Most commonly encountered entries would then be examined first when vectors are searched. Sophisticated strategies would be required for the overlapping of ordered vectors. Sorting could be done after overlapping, but to much less advantage.

Other avenues for further investigation include

1. Modification of Theorem A to prove LALR $\subset$ LALRC (major difficulty is notational).
2. Consideration of LR(1) storage requirements, in view of the success of the compaction techniques (stateset merging is then possible, and overlapping would yield increased savings).
3. Evaluation in practical terms of the increased generality of LALR and LALRC over SLR and SLRC (requires an environment having a translator writing or similar system; J. Horning at Toronto has found that while a majority of practical grammars are SLR, a few do require LALR).
4. Structure preserving grammatical transformations which yield SLR or LALR grammars (Graham (1970) gives these for simple precedence and LR(1) grammars).

APPENDICES

There are two appendices, the first of which consists mainly of proofs omitted from the main text. The second provides some details of the author's implementation of an SLR parsing table constructor, and also gives examples of the output from that implementation.

Appendix 1

1.1 The sets Z and Z' are defined for an LR(k) stateset $\mathcal{S}$ by,

$$Z = \{\beta \mid \exists[p,j;\alpha] \in \mathcal{S}',\ j < n_p,\ \beta \in H'_k(X_{p,j+1} \ldots X_{pn_p}\alpha)\}$$

$$Z' = \{\beta \mid \exists[p,j;\alpha] \in \mathcal{S}',\ j < n_p,\ \beta = x\gamma,\ x = X_{p,j+1} \in V_T,\ \gamma \in H_{k-1}(X_{p,j+2} \ldots X_{pn_p}\alpha)\}$$

We show that $Z = Z'$ for $k \geq 1$. The reason is to be found in the calculation of $\mathcal{S}'$ from $\mathcal{S}$, which adds to $\mathcal{S}'$ states $[q,0;\beta]$ corresponding to any $[p,j;\alpha] \in \mathcal{S}'$ with $A_q = X_{p,j+1} \in V_N$.

Clearly $Z' \subseteq Z$, we need only prove $Z \subseteq Z'$.

Let $\omega \in Z$. Then $\exists[p,j;\alpha] \in \mathcal{S}'$ with $j < n_p, \omega \in H'_k(X_{p,j+1} \ldots X_{pn_p}\alpha)$. $|\omega| = k \geq 1$ and so $\omega = x\delta$, $x \in V_T$, $|\delta| = k-1$.

If $X_{p,j+1} \in V_T$, then $x = X_{p,j+1}$ and so $\omega \in Z'$.

Otherwise, because $x\delta \in H'_k(X_{p,j+1} \ldots X_{pn_p}\alpha)$, we can find a sequence of productions,

$$A_{q_i} \to X_{q_i,1}\alpha_i \qquad n_{q_i} > 0 \qquad 1 \leq i \leq r \qquad r \geq 1$$

with $X_{p,j+1} = A_{q_1}$ $\quad X_{q_i,1} = A_{q_{i+1}}$ $\quad 1 \leq i < r$ and $X_{q_r,1} = x$

and such that $\delta \in H_{k-1}(\alpha_r \ldots \alpha_1 X_{p,j+2} \ldots X_{pn_p}\alpha)$.

By virtue of this sequence,

$$[q_i,0;\beta] \in \mathcal{S}' \ \forall\ \beta \in H_k(\alpha_{i-1} \ldots \alpha_1 X_{p,j+2} \ldots X_{pn_p}\alpha) \qquad 1 \leq i \leq r$$

Since $[q_r,0;\beta] \in \mathcal{S}'$ $\forall$ $\beta \in H_k(\alpha_{r-1} \ldots \alpha_1 X_{p,j+2} \ldots X_{pn_p}\alpha)$

and $0 < n_{q_r}$, $X_{q_r,1} \in V_T$ we have that

$$X_{q_r,1}\gamma \in Z' \ \forall\ \gamma \in H_{k-1}(\alpha_r \ldots \alpha_1 X_{p,j+2} \ldots X_{pn_p}\alpha),$$

and so $x\delta = \omega \in Z'$.

Thus $Z \subseteq Z'$.

1.2 In the proof of lemma 1, we claim that if $\mathcal{S}_m$ and $\mathcal{S}_k$ are respectively LR(m) and LR(k) statesets (for the same grammar), with $H_k(\mathcal{S}_m) = \mathcal{S}_k$, then $H_k(\mathcal{S}'_m) = \mathcal{S}'_k$. This will now be justified.

Suppose $[q,0;\gamma] \in \mathcal{S}'_k$ (and $[q,0;\gamma] \notin \mathcal{S}_k$). Then we can find $[p,j;\beta] \in \mathcal{S}_k$ with $0 \le j < n_p$ and a sequence of productions

$$A_{q_i} \to X_{q_i,1}\alpha_i \quad n_{q_i} > 0 \quad 1 \le i \le r \quad r \ge 0$$
$$\text{with } X_{p,j+1} = A_{q_1} \quad X_{q_i,1} = A_{q_{i+1}} \quad 1 \le i \le r \text{ and } A_{q_{r+1}} = A_q$$
$$\text{and such that } \gamma \in H_k(\alpha_r \dots \alpha_1 X_{p,j+2} \dots X_{pn_p} \beta).$$

Also $\exists\, [p,j;\alpha] \in \mathcal{S}_m$ with $\beta \in H_k(\alpha)$,

$$\text{hence } \gamma \in H_k(\alpha_r \dots \alpha_1 X_{p,j+2} \dots X_{pn_p} \alpha).$$

Because of the above sequence of productions,

$$[q,0;\sigma] \in \mathcal{S}'_m \ \forall\, \sigma \in H_m(\alpha_r \dots \alpha_1 X_{p,j+2} \dots X_{pn_p} \alpha)$$

So we can find $[q,0;\delta] \in \mathcal{S}'_m$ with $\gamma \in H_k(\delta)$.

Thus $[q,0;\gamma] \in H_k(\mathcal{S}'_m)$ and we have $\mathcal{S}'_k \subseteq H_k(\mathcal{S}'_m)$.

Conversely, suppose $[q,0;\gamma] \in H_k(\mathcal{S}'_m)$ (and $[q,0;\gamma] \notin \mathcal{S}_k$).

Then $\exists\, [q,0;\delta] \in \mathcal{S}'_m$ with $\gamma \in H_k(\delta)$. We can find $[p,j;\alpha] \in \mathcal{S}_m$ with $0 \le j < n_p$ and a sequence of productions,

$$A_{q_i} \to X_{q_i,1}\alpha_i \quad n_{q_i} > 0 \quad 1 \le i \le r \quad r \ge 0$$
$$\text{with } X_{p,j+1} = A_{q_1} \quad X_{q_i,1} = A_{q_{i+1}} \quad 1 \le i \le r \text{ and } A_{q_{r+1}} = A_q$$
$$\text{and such that } \delta \in H_m(\alpha_r \dots \alpha_1 X_{p,j+2} \dots X_{pn_p} \alpha).$$

Also $\exists\, [p,j;\beta] \in \mathcal{S}_k$ with $\beta \in H_k(\alpha)$.

Because of the above sequence of productions,

$$\begin{aligned}[q,0;\sigma] \in \mathcal{S}'_k \ \forall\, \sigma \in\ & H_k(\alpha_r \dots \alpha_1 X_{p,j+2} \dots X_{pn_p} \beta) \\ =\ & H_k(\alpha_r \dots \alpha_1 X_{p,j+2} \dots X_{pn_p} \alpha)\end{aligned}$$

Thus $[q,0;\gamma] \in \mathcal{S}'_k$ and we have $H_k(\mathcal{S}'_m) \subseteq \mathcal{S}'_k$.

Hence $H_k(\mathcal{S}'_m) = \mathcal{S}'_k$.

1.3 Next we give an outline of a minimal LR(k) backtracking algorithm. This has not been implemented and should only be regarded as indicating that such a technique is theoretically possible. The method used is a modification of the first LA(k)LR(0) algorithm, where if two statesets are combined and generate an inadequate stateset, sufficient information is retained to permit their separation. M is a matrix, whose elements are statesets, which are all initially set to $\emptyset$. We denote by S_i the contents of M[i,j] where j is a maximum such that M[i,j] $\neq \emptyset$ (if no such j then $S_i = \emptyset$). All non void entries in the i^{th} row of M will be different k symbol lookahead refinements of the same LR(0) stateset, S_i being the most recent version. At any stage in the computation a variable t indicates how many rows of M are in use, and $\{S_i \mid 1 \leq i \leq t\}$ is the current table of statesets. An attempt is made to combine a generated stateset SY with each S_i such that $S_i \sim SY$ (under H_0). Only if no attempt succeeds is t incremented and SY inserted as a new stateset S_t. TEST is a recursive <u>boolean</u> <u>procedure</u> with three parameters, a <u>stateset</u> S, and <u>integers</u> n and l. TEST returns <u>false</u> if S generates any inadequate statesets, but otherwise returns <u>true</u> and updates M with all statesets generated by S. n specifies a row, and l a column of M (l indicates the depth of recursion). In the following pseudo Algol description, efficiency is sacrificed in the hope of reducing obscurity.

```
boolean procedure TEST ($\mathcal{S}$,n,l); value $\mathcal{S}$,n,l; stateset $\mathcal{S}$; integer n.l;
if $\mathcal{S}$ is inadequate then TEST:= false else
  begin M[n,l] := $\mathcal{S}$;
  for each Y $\in$ V do if $\nexists\ \mathcal{S}_i$ with $\mathcal{S}_i \sim \mathcal{S}Y$ and $\mathcal{S}_i \supseteq \mathcal{S}Y$ then
    begin i:= 1; while i $\leq$ t do
    if $\mathcal{S}_i \sim \mathcal{S}Y$ and TEST($\mathcal{S}_i \cup \mathcal{S}Y$,i,l+1) then i:= t+2 else i:= i+1;
    if i = t+1 then
      begin t:= t+1; if not TEST($\mathcal{S}Y$,t,l+1) then
        begin for i:= 1 step 1 until t do M[i,l] := $\emptyset$;
        t:= t-1; TEST:= false; goto EXIT
        end
      end
    end;
  for i:= 1 step 1 until t do if M[i,l] $\neq \emptyset$ then
    begin M[i,l-1]:= M[i,l]; M[i,l] := $\emptyset$ end;
  TEST:= true;
EXIT : end TEST;
t:= 1; if TEST({[0,0;$\Lambda$]},1,2) then
  comment minimal LR(k) machine is in M[1,1] ... M[1,t];;
```

1.4 To prove that a Λ-free grammar is PLR iff it is weaker precedence, we show that a grammar which is not PLR is not weaker precedence, and that a Λ-free grammar which is not weaker precedence is not PLR.

First suppose $\mathcal{G}$ is not PLR. Then we can find a stateset with either a) a shift - reduce p inadequacy or b) a reduce p - reduce q inadequacy which cannot be resolved by examining the stack.

a) If the stateset's associated symbol is Y, and the inadequate lookahead symbol is x we can deduce both $Y \lessdot x$ and $Y \gtrdot x$ by applying lemma 3 to the SLR statesets which contributed the inadequacy causing states.

Thus $\mathcal{G}$ is not weaker precedence.

b) If the inadequate lookahead symbol is x, then $x \in F_1(A_p) \cap F_1(A_q)$. If $n_p = n_q$ and the inadequacy cannot always be resolved, then we must have $A_p \to \alpha$, $A_q \to \alpha$ and X such that $X \lessdot A_p$, $X \lessdot A_q$.

Then $\mathcal{G}$ is not weaker precedence.

If $n_p \neq n_q$, say $n_p < n_q$, and the inadequacy cannot always be resolved, we must have $A_q \to \alpha XY\beta$, $A_p \to Y\beta$ and $X \lessdot A_p$.

Then $\mathcal{G}$ is not weaker precedence.

Now suppose $\mathcal{G}$ is Λ-free, but not weaker precedence. One of conditions (ii), (iii)″, (iv)″ does not hold.

If $Y \leq\cdot X$ and $Y \cdot> X$ then since $X \overset{*}{\to} x\gamma$ with $x \in V_T$ we have $Y \leq\cdot x$ and $Y \cdot> x$. Consider the PLR stateset $\mathcal{S}$ with associated symbol Y. Then $[p,j;\alpha] \in \mathcal{S}$ $\quad 0 \leq j < n_p \quad X_{p,j+1} = x$ and $[q,n_q;x] \in \mathcal{S}$;
ie. shift - reduce q inadequacy,

so $\mathcal{G}$ is not PLR.

If $A_p \to \alpha XY\beta$, $A_q \to Y\beta$, $X \leq\cdot A_q$ and $x \in F_1(A_p) \cap F_1(A_q)$, then consider the stateset $\mathcal{S}$ with associated symbol the final symbol of $Y\beta$. $[p,n_p;x]$, $[q,n_q;x] \in \mathcal{S}$, and if the stack top represents $\alpha XY\beta$ the inadequacy cannot be resolved.

So $\mathcal{G}$ is not PLR.

If $A_p \to \alpha$, $A_q \to \alpha$, $x \in F_1(A_p) \cap F_1(A_q)$ and $X \leq\cdot A_p$, $X \leq\cdot A_q$ then consider the stateset $\mathcal{S}$ with associated symbol the final symbol of α. $[p,n_p;x]$, $[q,n_q;x] \in \mathcal{S}$ and if the stack top represents $X\alpha$, the inadequacy cannot be resolved.

So $\mathcal{G}$ is not PLR.

This completes the proof.

1.5 In this section we establish, as theorem A, the result that any SLR grammar which is Λ-free is an SLRC grammar (for any set of chain productions). The following lemma is central to the proof of this theorem.

Lemma A

Let $\mathcal{G}$ be SLR, and $\mathcal{S}$ an SLR stateset.

If $[p,j-1]$, $[q,l-1] \in \mathcal{S}'$, and $X_{pj} = B_m \rightarrow \ldots \rightarrow B_1 = Y$
$Y_{ql} = C_n \rightarrow \ldots \rightarrow C_1 = Y$ and $A_q \neq B_i \quad m \geq i \geq 1 \quad A_p \neq C_i$
$n \geq i \geq 1$ and $F_1(X_{pj}) \cap F_1(X_{ql}) \neq \emptyset$, then
$X_{pj} = X_{ql}$.

Let $y \in F_1(X_{pj}) \cap F_1(X_{ql})$. Then $y \in F_1(B_i) \quad m \geq i \geq 1$, $y \in F_1(C_i) \quad n \geq i \geq 1$. Take $n \geq m$. Determine r such that $\forall\, i < r$ we have $B_i = C_i$ and either $B_r \neq C_r$ or $r = m$.

If $B_r \neq C_r$, then $r > 1$ so let $B_r = A_{p'} \rightarrow X_{p'1} = B_{r-1}$
$C_r = A_{q'} \rightarrow X_{q'1} = C_{r-1}$.
We have $[p',0]$, $[q',0] \in \mathcal{S}'$, so $[p'1]$, $[q',1] \in \mathcal{S}\, B_{r-1}$.
Since $p' \neq q'$, $n_{p'} = n_{q'} = 1$, $y \in F_1(A_{p'}) \cap F_1(A_{q'})$ the stateset $\mathcal{S}\, B_{r-1}$ is inadequate - a contradiction.

So $B_r = C_r$ and $r = m$. If $n > m$, let $C_{m+1} = A_{q'} \rightarrow X_{q'1} = C_m = B_m$.
We have $[q',0] \in \mathcal{S}'$, so $[q',1]$, $[p,j] \in \mathcal{S}\, B_m$.
Since $n_{q'} = 1$, $y \in F_1(A_{q'})$ we have a (y,reduce q') entry.
Also, $y \in F_1(X_{pj})$ so

if $j < n_p$ we either have a (y,shift) entry or a (y,reduce p') entry with $n_{p'} = 0$, so $p' \neq q'$.

if $j = n_p$ we have a (y,reduce p) entry, and $p \neq q'$ since $A_p \neq C_{m+1}$.

In either case $\mathcal{S}\, B_m$ is inadequate - a contradication.

Thus $r = m = n$, and hence $X_{pj} = X_{ql}$.

Theorem A

Let $\mathcal{G}$ be an SLR, Λ-free grammar. Then $\mathcal{G}$ is SLRC.

For suppose $\exists$ an inadequate SLRC stateset $\mathcal{S}$ say. Extending the chained successor notation, we have $\mathcal{S} = \mathcal{S}_{0Y_1 \dots Y_m}$ (with $m \geq 1$ since $\mathcal{G}$ is SLR). Let $\mathcal{S}_i$ denote $\mathcal{S}_{0Y_1 \dots Y_i}$ $0 \leq i \leq m$.

We can find two sequences of states,

$[p_i, j_i]$, $[q_i, l_i]$ $0 \leq i \leq m$ with the following properties.

1) $[p_i, j_i], [q_i, l_i] \in \mathcal{S}_i$ $0 \leq i \leq m$

2) For $0 \leq i < m$, $j_i < n_{p_i}$ and

either $[p_i, j_i + 1] = [p_{i+1}, j_{i+1}]$

or $[p_{i+1}, 0] \in \mathcal{S}_i'$, $j_{i+1} = 1$ as a result

of $X_{p_i, j_i+1} \xrightarrow{*} A_{p_{i+1}} \alpha_i$

Thus $[p_{i+1}, j_{i+1} - 1] \in \mathcal{S}_i'$.

$l_i < n_{q_i}$ and

similarly $[q_{i+1}, l_{i+1} - 1] \in \mathcal{S}_i'$

3) $[p_m, j_m]$ introduces (or is equal to) a state

$[p, n_p] \in \mathcal{S}_m'$, $x \in F_1(A_p)$

$[q_m, l_m]$ introduces (or is equal to) a state

$[q, l] \in \mathcal{S}_m'$

with either $l < n_q$, $x = X_{q, l+1}$

or $l = n_q$, $x \in F_1(A_q)$, $p \neq q$.

Since $X_{p_i,j_i+1} \overset{*}{\rightarrow} Y_{i+1}\alpha$ and $X_{q_i,l_i+1} \overset{*}{\rightarrow} Y_{i+1}\beta$ for some $\alpha,\beta \in V^*$, we have
$F_1(X_{p_i j_i}) \cap F_1(X_{q_i l_i}) \neq \emptyset \quad 1 \le i < m$

Since $\mathcal{G}$ is Λ-free, $n_p > 0$ and we have $[p,n_p] \in \mathcal{S}_m$, so
$[p_m,j_m] = [p,n_p]$.

$\therefore$ $x \in F_1(A_{p_m})$ and so $x \in F_1(X_{p_m j_m})$.

If $l = n_q$ we similarly deduce $x \in F_1(X_{q_m l_m})$.

If $l < n_q$ then $X_{q_m,l_m+1} \overset{*}{\rightarrow} X_{q,l+1}\beta = x\beta$ such that $x \in F_1(X_{q_m l_m})$.

So $F_1(X_{p_m j_m}) \cap F_1(X_{q_m l_m}) \neq \emptyset$.

Thus, $F_1(X_{p_i j_i}) \cap F_1(X_{q_i l_i}) \neq \emptyset$, $X_{p_i j_i} \underline{c}^* Y_i$, $X_{q_i l_i} \underline{c}^* Y_i$, and A_{p_i}, A_{q_i} cannot occur in these chains since (by virtue of $[p_i,j_i]$, $[q_i,l_i] \in \mathcal{S}_i$) p_i and q_i do not designate chain productions, all for $1 \le i \le m$.

By induction on i we can show

$[p_m,j_m]$, $[q_m,l_m] \in \mathcal{S}_0 X_{p_1 j_1} \ldots X_{p_m j_m}$

Clearly $[p_0,j_0]$, $[q_0,l_0] = [0,0] \in \mathcal{S}_0$.

Assume, with $1 \le i \le m$, that

$[p_{i-1},j_{i-1}],[q_{i-1},l_{i-1}] \in \mathcal{S}_0 X_{p_1 j_1} \ldots X_{p_{i-1} j_{i-1}}$

This is an SLR stateset.

Also $[p_i,j_i-1],[q_i,l_i-1] \in (\mathcal{S}_0 X_{p_1 j_1} \ldots X_{p_{i-1} j_{i-1}})'$

The conditions of lemma A are all satisfied, and so $X_{p_i j_i} = X_{q_i l_i}$.

Then $[p_i,j_i]$, $[q_i,l_i] \in \mathcal{S}_0 X_{p_1 j_1} \ldots X_{p_i j_i}$, completing an inductive step.

From $[p_m,j_m]$, $[q_m,l_m] \in \mathcal{S}_0 X_{p_1 j_1} \ldots X_{p_m j_m}$ we deduce that $[p,n_p]$, $[q,l] \in (\mathcal{S}_0 X_{p_1 j_1} \ldots X_{p_m j_m})'$ which contradicts $\mathcal{G}$ being SLR, since $\mathcal{S}_0 X_{p_1 j_1} \ldots X_{p_m j_m}$ would be inadequate. This gives us our result.

1.6 The final result of this appendix shows that SLR parsing-states cannot be merged under the criteria given in Chapter 4.

Let $R(\mathcal{S}_l)$ and $R(\mathcal{S}_r)$ be SLR (or LALR) parsing-states such that

(1) their nonterminal entries do not conflict,

(2) their terminal shift entries are identical,

(3) the productions used for reductions in each are the same.

Then we can show $l = r$.

For suppose $[p,j] \in \mathcal{S}_l$.

If $j = n_p$ then we will have reduce p entries in $R(\mathcal{S}_l)$ and so (by (3)) also in $R(\mathcal{S}_r)$. Hence $[p,n_p] \in \mathcal{S}_r$.

If $j < n_p$ then consider $X_{p,j+1}$.

If $X_{p,j+1} = x \in V_T$ then $(x,\text{shift } t) \in R(\mathcal{S}_l)$ for some t. By (2), $(x,\text{shift } t) \in R(\mathcal{S}_r)$ also. Since $[p,j+1] \in \mathcal{S}_t$ we deduce $[p,j] \in \mathcal{S}_r$.

Otherwise $X_{p,j+1} \in V_N$. We can find a sequence of productions

$$A_{q_i} \to X_{q_i,1}\alpha_i \qquad n_{q_i} > 0 \qquad 1 \le i < m \qquad m \ge 1$$

$$\text{with } X_{p,j+1} = A_{q_1} \qquad X_{q_i,1} = A_{q_{i+1}} \qquad 1 \le i < m$$

$$[q_i,0] \in \mathcal{S}'_l \qquad 1 \le i \le m$$

and either $A_{q_m} \to \Lambda$ or $A_{q_m} \to x\alpha_m$, $x \in V_T$.

If $A_{q_m} \to \Lambda$ then we will have reduce q_m entries in $R(\mathcal{S}_l)$ and so (by (3)) also in $R(\mathcal{S}_r)$. Hence $[q_m,0] \in \mathcal{S}'_r$.

If $A_{q_m} \to x\alpha_m$ then $(x,\text{shift } t) \in R(\mathcal{S}_l)$ for some t. By (2), $(x,\text{shift } t) \in R(\mathcal{S}_r)$ also. Since $[q_m,1] \in \mathcal{S}_t$ we deduce $[q_m,0] \in \mathcal{S}'_r$.

Now suppose $[q_i,0] \in \mathcal{S}'_r$ for some i with $1 < i \le m$. Since $(A_{q_i},\text{shift } t) \in R(\mathcal{S}_l)$ for some t_i, by (1) the entry on A_{q_i} in $R(\mathcal{S}_r)$ must also be $(A_{q_i},\text{shift } t_i)$, and since $[q_{i-1},1] \in \mathcal{S}_{t_i}$ and $A_{q_i} = X_{q_{i-1},1}$ we deduce $[q_{i-1},0] \in \mathcal{S}'_r$.

By induction we deduce $[q_1,0] \in \mathcal{S}_r{}'$ and then consider $(A_{q_1},\text{shift } t) \in R(\mathcal{S}_l)$ for some t. By (1) the entry on A_{q_1} in $R(\mathcal{S}_r)$ must also be $(A_{q_1},\text{shift } t)$, and since $[p,j+1] \in \mathcal{S}_t$ and $A_{q_1} = X_{p,j+1}$ we deduce $[p,j] \in \mathcal{S}_r$.

Thus in all cases $[p,j] \in \mathcal{S}_r$. Therefore $\mathcal{S}_l \subseteq \mathcal{S}_r$ and by symmetry $\mathcal{S}_l = \mathcal{S}_r$. Hence $l = r$.

Appendix 2

A parsing table constructor was implemented as a suite of four programs, written in AlgolW, and run under the Michigan Terminal System on the IBM 360/67 at Newcastle. These programs, named SLRIN, SLR, SLRED and SLROUT, accomplish the following tasks (in addition to serving as the test programs 2-5 used for timings in Chapter 4).

SLRIN : Accepts a CFG, in a free format BNF style of notation, and from this constructs an internal representation of the grammar. Checks are made on the validity of the input, to ensure that it represents a CFG in reduced form. The internal representation is augmented by a zeroth production with endmarker.

SLR : Computes from the grammar the sets $F_1(A)$ for each $A \in V_N$, and then constructs an uncompacted list representation of the SLR (or SLRPC or SLRC) parsing table.

SLRED : Applies the various compactions, in order -
merging of inadequate statesets, elimination of LR(0) statesets, terminal defaults, nonterminal defaults, renumbering of parsing-states, overlapping in SUPTACT.

SLROUT : Overlaps terminal entries, and formats the list representation into initialisation for PL360 declarations.

CPU time requirements, in seconds, for each of these programs operating on the AlgolW2 grammar are given in the following table.

Program	SLRIN	SLR	SLRED	SLROUT	Parsing Table
	5	76	9	7	SLR
	5	130	9	7	SLRPC
	5	147	23	40	SLRC

The computation of $F_1(A)$ in the program SLR is based on the equation

$$F_1(A) = \{a \in V_T \mid C \rightarrow \alpha X B_1 \ldots B_n Y \omega \in P \text{ with } B_1 \ldots B_n \overset{*}{\rightarrow} \Lambda, \quad X \overset{*}{\rightarrow} \beta A,\ Y \overset{*}{\rightarrow} a\gamma,\ \alpha,\ \beta,\ \gamma,\ \omega \in V^* \}$$

This result is stated by Knuth (1971), who uses the name 'follow' for F_1. Before the equation can be used, we must determine which nonterminals can generate the empty string. They may be computed recursively, using

$$A \overset{*}{\rightarrow} \Lambda \text{ iff } A \rightarrow B_1 \ldots B_n,\ B_i \overset{*}{\rightarrow} \Lambda,\ 1 \leq i \leq n,\ n \geq 0$$

Also required, for every $A \in V_N$, are

$$\{a \in V_T \mid A \overset{*}{\rightarrow} a\alpha,\ \alpha \in V^*\} = \text{first}(A) \cap V_T$$

$$\text{and} \quad \{B \in V_N \mid A \overset{*}{\rightarrow} \alpha B,\ \alpha \in V^*\} = \{A\} \cup \text{last}(A) \cap V_N$$

which may be determined using

$$X \in \text{first}(A) \quad \text{iff} \quad A \rightarrow B_1 \ldots B_n Y \alpha,\ B_1 \ldots B_n \overset{*}{\rightarrow} \Lambda,\ n \geq 0,$$
$$X = Y \quad \text{or } X \in \text{first}(Y)$$

$$X \in \text{last}(A) \quad \text{iff} \quad A \rightarrow \alpha Y B_1 \ldots B_n,\ B_1 \ldots B_n \overset{*}{\rightarrow} \Lambda,\ n \geq 0,$$
$$X = Y \quad \text{or } X \in \text{last}(Y)$$

Iterative routines can be programmed to perform these calculations.

In an attempt to simplify storage allocation in the program SLR, an upper bound was derived for the number of statesets in a stateset table. If s is the number of productions and t the number of terminals of a grammar then 2^m, where $m = t^k . \sum_{i=0}^{s} n_i$, is quoted as an upper bound for the LR(k) stateset table by Korenjak (1969). Since every stateset has an associated symbol, it can be shown that an improved upper bound is $\sum_{X \in V} 2^{f(X)}$ where f(X) denotes the product of the number of occurrences of X on the RHSs of productions with $\max_{A \in V_N}$ (number of elements in $F_k(A)$). For the XPL grammar, with k = 0, we have $2^m = 2^{228}$ and $\sum_{X \in V} 2^{f(X)} \div 2^{16}$, compared with an actual value of less than 2^8. Despite the improvement, the new upper bound was not used.

The strategies employed by SLRED and SLROUT for overlapping, can be described in terms of the following model. The elements of the sets $D_i = \{\lambda_{i1}, \ldots, \lambda_{im_i}\}$ $1 \leq i \leq n$ are to be stored as a vector E, subject to the constraint,

$$\forall i \;\; \exists j \text{ such that } D_i = \{E(j), \ldots, E(j+m_i-1)\}$$

Clearly one solution is $E = \lambda_{11} \ldots \lambda_{1m_1} \ldots \lambda_{n1} \ldots \lambda_{nm_n}$; the objective of an overlapping strategy is to reduce the length of E.

The method used by SLRED for overlapping in SUPTACT is very simple. Each set D_i is in turn either overlapped on E if there already exists a j with $D_i = \{E(j), \ldots, E(j+m_i-1)\}$ or concatenated onto the elements already in E(the process is in fact further simplified by imposing an ordering on the λ_{ij}, storing each D_i as an ordered vector, and requiring $D_i(1) \ldots D_i(m_i) = E(j) \ldots E(j+m_i-1)$ for overlapping to take place).

A more complex technique was needed in SLROUT for overlapping entries of TSYM,TACT, and is now described. Two types of flag are employed, referred to as marks and ticks respectively; initially the D_i are not marked or ticked. Any D_i with $m_i = 1$, or for which $D_i = D_j$ (for some D_j not yet excluded), is excluded from Stages 1 and 2. Repeat Stage 1 until all the D_i are marked.

Stage 1

Select the largest unmarked set, say D_i. Determine the sequence $C_1, \ldots, C_m$ where $C_1 = D_i$, C_{j+1} is the largest unmarked set such that $C_{j+1} \subset C_j$ $1 \leq j < m$, and no unmarked set is contained in C_m. If $m \leq 2$ then tick and mark D_i, otherwise concatenate D_i onto E, and mark $C_1, \ldots, C_m$.

Repeat Stage 2 while 2 or more D_i remain ticked.

Stage 2

Select $i \neq j$ such that D_i, D_j are ticked and $D_i \cap D_j$ is largest. Concatenate $D_i \cup D_j$ onto E, and remove ticks from D_i and D_j.

Stage 3

Any remaining ticked D_i is concatenated onto E. Those D_i excluded from Stages 1 and 2 are optimally overlapped or concatenated onto E.

The elements of sets concatenated onto E must be ordered to ensure that the constraint on E is satisfied. This is complicated by the presence of default operations, and in Stage 2, if D_i and D_j both contain default operations, then $D_i \cap D_j$ must be regarded as $\emptyset$.

The four programs produce the following information in a readable format.

SLRIN : Tables of the terminals, nonterminals and productions of the grammar are printed, together with the integers which identify these items within the programs. The principal nonterminal and endmarker are represented by ++++ and _|_ respectively. Productions $A \rightarrow \alpha_1 , \ldots , A \rightarrow \alpha_n$ are printed as $A ::= \alpha_1 \mid \ldots \mid \alpha_n$. Some statistics of the grammar are also given.

SLR : Nonterminals A such that $A \overset{*}{\rightarrow} \Lambda$ are listed, and the sets F_1 tabulated for every nonterminal. If $F_1(A) = \{a_1 , \ldots , a_n\}$ this is printed as $A \mid a_1 \ldots a_n$ under the heading 'TABLE OF FOLLOW'. Stateset and parsing tables (SLR or SLRPC or SLRC as required) are output together.

SLRED : Prints statistics of the parsing table and of the compactions obtained, followed by a specification of the stateset renumbering resulting from the compactions. The renumbered compacted parsing table is printed, with terminal default entries indicated by ++++ (no confusion can arise from the use of this symbol). Then the table of nonterminal defaults is given.

SLROUT : Specifies the saving from terminal overlapping, and the amount of storage required for the tables. Initialised PL360 declarations for LHS, TSTATE, NSTATE, TSYM, NSYM and ACT are printed. The integers NDEF, TACT and NACT denote the locations of the first member of their respective vectors within ACT.

The output from each program is terminated (and hence delimited) by the CPU time used in the program's execution. We present examples of output for the grammars $\mathcal{G}_2$, $\mathcal{G}_6$, $\mathcal{G}_7$ and $\mathcal{G}_8$ (introduced on pages 20, 60 and 69).

$\mathcal{G}_2$ is a trivial SLR grammar which is not LR(0). The tables presented are the output of the programs SLRIN and SLR.

```
TERMINALS
  1  a    b     _|_

NONTERMINALS
  4  ++++ A

PRODUCTIONS
  0  ++++ ::= A     _|_
  1  A    ::= a     A     b     | a


THERE ARE 3 TERMINALS AND 2 NONTERMINALS
THERE ARE 3 PRODUCTIONS WITH AVERAGE LENGTH OF RHS 2.00


000.11 SECONDS IN EXECUTION


TABLE OF FOLLOW

A    |   b     _|_
```

```
GRAMMAR  IS  SLR


SLR STATESET AND PARSING TABLE


STATESET  0
(0,0)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS
PARSING-STATE  0
                        (a    ,SHIFT    2)       (A    ,GOTO    1)
 0  REDUCE ENTRIES       1  SHIFT ENTRY           1  GOTO ENTRY

STATESET  1
(0,1)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  A
PARSING-STATE  1
 0  REDUCE ENTRIES       0  SHIFT ENTRIES         0  GOTO ENTRIES

STATESET  2
(1,1)  (2,1)
NUMBER OF STATES = 2   ASSOCIATED SYMBOL IS  a
PARSING-STATE  2
(b    ,REDUCE   2)      (a    ,SHIFT    2)       (A    ,GOTO    3)
(_|_  ,REDUCE   2)
 2  REDUCE ENTRIES       1  SHIFT ENTRY           1  GOTO ENTRY

STATESET  3
(1,2)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  A
PARSING-STATE  3
                        (b    ,SHIFT    4)
 0  REDUCE ENTRIES       1  SHIFT ENTRY           0  GOTO ENTRIES

STATESET  4
(1,3)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  b
PARSING-STATE  4
(b    ,REDUCE   1)
(_|_  ,REDUCE   1)
 2  REDUCE ENTRIES       0  SHIFT ENTRIES         0  GOTO ENTRIES


000.25 SECONDS IN EXECUTION
```

G_8 is a small grammar of a programming language type, incorporating statements and simple arithmetic expressions. It includes a production with an empty RHS. The tables presented are the output from all four programs (with chain elimination) and from SLR (without chain elimination).

```
TERMINALS
  1  id    +     (     )     *     if    then or    else :=    _|_

NONTERMINALS
 12  ++++ D     A     E     T     P     C     B     L

PRODUCTIONS
  0  ++++ ::= D     _|_
  1  D    ::= A     | C
  3  A    ::= id    :=    E
  4  E    ::= T     | E    +    T
  6  T    ::= P     | T    *    P
  8  P    ::= (     E     )    | id
 10  C    ::= if    B     then A    L
 11  B    ::= B     or    id   | id
 13  L    ::= else  D     |

THERE ARE 11 TERMINALS AND 9 NONTERMINALS
THERE ARE 15 PRODUCTIONS WITH AVERAGE LENGTH OF RHS 2.00
PRODUCTIONS 1 2 4 6 ARE CHAIN PRODUCTIONS


000.30 SECONDS IN EXECUTION


L     GENERATES THE EMPTY STRING


TABLE OF FOLLOW

D    |   _|_
A    |   else _|_
E    |   +    )    else _|_
T    |   +    )    *    else _|_
P    |   +    )    *    else _|_
C    |   _|_
B    |   then or
L    |   _|_
```

```
GRAMMAR  IS  SLRC


SLRC STATESET AND PARSING TABLE


STATESET  0
(0,0)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS
PARSING-STATE  0
                       (id  ,SHIFT   2)      (D   ,GOTO    1)
                       (if  ,SHIFT   3)      (A   ,GOTO    1)
                                             (C   ,GOTO    1)
 0  REDUCE ENTRIES      2  SHIFT ENTRIES      3  GOTO ENTRIES

STATESET  1
(0,1)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  D
PARSING-STATE  1
 0  REDUCE ENTRIES      0  SHIFT ENTRIES      0  GOTO ENTRIES

STATESET  2
(3,1)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  id
PARSING-STATE  2
                       (:=  ,SHIFT   4)
 0  REDUCE ENTRIES      1  SHIFT ENTRY        0  GOTO ENTRIES

STATESET  3
(10,1)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  if
PARSING-STATE  3
                       (id  ,SHIFT   6)      (B   ,GOTO    5)
 0  REDUCE ENTRIES      1  SHIFT ENTRY        1  GOTO ENTRY

STATESET  4
(3,2)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  :=
PARSING-STATE  4
                       (id  ,SHIFT  10)      (E   ,GOTO    7)
                       ((   ,SHIFT   9)      (T   ,GOTO    8)
                                             (P   ,GOTO    8)
 0  REDUCE ENTRIES      2  SHIFT ENTRIES      3  GOTO ENTRIES

STATESET  5
(10,2)   (11,1)
NUMBER OF STATES = 2   ASSOCIATED SYMBOL IS  B
PARSING-STATE  5
                       (then,SHIFT  11)
                       (or  ,SHIFT  12)
 0  REDUCE ENTRIES      2  SHIFT ENTRIES      0  GOTO ENTRIES
```

```
STATESET   6
(12,1)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  id
PARSING-STATE   6
(then,REDUCE   12)
(or  ,REDUCE   12)
 2  REDUCE ENTRIES       0  SHIFT ENTRIES        0  GOTO ENTRIES

STATESET   7
(3,3)   (5,1)
NUMBER OF STATES = 2   ASSOCIATED SYMBOL IS  E
PARSING-STATE  7
(else,REDUCE    3)     (+    ,SHIFT  13)
(_|_ ,REDUCE    3)
 2  REDUCE ENTRIES       1  SHIFT ENTRY          0  GOTO ENTRIES

STATESET  8
(3,3)   (5,1)   (7,1)
NUMBER OF STATES = 3   ASSOCIATED SYMBOL IS  T
PARSING-STATE  8
(else,REDUCE    3)     (+    ,SHIFT  13)
(_|_ ,REDUCE    3)     (*    ,SHIFT  14)
 2  REDUCE ENTRIES      2  SHIFT ENTRIES         0  GOTO ENTRIES

STATESET   9
(8,1)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS   (
PARSING-STATE   9
                       (id   ,SHIFT  10)        (E    ,GOTO  15)
                       ((    ,SHIFT   9)        (T    ,GOTO  16)
                                                (P    ,GOTO  16)
 0  REDUCE ENTRIES      2  SHIFT ENTRIES        3  GOTO ENTRIES

STATESET   10
(9,1)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  id
PARSING-STATE   10
(+    ,REDUCE    9)
()    ,REDUCE    9)
(*    ,REDUCE    9)
(else,REDUCE     9)
(_|_ ,REDUCE     9)
 5  REDUCE ENTRIES      0  SHIFT ENTRIES         0  GOTO ENTRIES

STATESET   11
(10,3)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  then
PARSING-STATE   11
                       (id  ,SHIFT   2)         (A    ,GOTO  17)
 0  REDUCE ENTRIES      1  SHIFT ENTRY          1  GOTO ENTRY

STATESET   12
(11,2)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  or
PARSING-STATE  12
                       (id   ,SHIFT  18)
 0  REDUCE ENTRIES      1  SHIFT ENTRY          0  GOTO ENTRIES
```

```
STATESET  13
(5,2)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  +
PARSING-STATE  13
                       (id  ,SHIFT  10)      (T   ,GOTO  19)
                       ((   ,SHIFT   9)      (P   ,GOTO  19)
 0  REDUCE ENTRIES      2  SHIFT ENTRIES      2  GOTO ENTRIES

STATESET  14
(7,2)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  *
PARSING-STATE  14
                       (id  ,SHIFT  10)      (P   ,GOTO  20)
                       ((   ,SHIFT   9)
 0  REDUCE ENTRIES      2  SHIFT ENTRIES      1  GOTO ENTRY

STATESET  15
(5,1)  (8,2)
NUMBER OF STATES = 2   ASSOCIATED SYMBOL IS  E
PARSING-STATE  15
                       (+   ,SHIFT  13)
                       ()   ,SHIFT  21)
 0  REDUCE ENTRIES      2  SHIFT ENTRIES      0  GOTO ENTRIES

STATESET  16
(5,1)  (7,1)  (8,2)
NUMBER OF STATES = 3   ASSOCIATED SYMBOL IS  T
PARSING-STATE  16
                       (+   ,SHIFT  13)
                       ()   ,SHIFT  21)
                       (*   ,SHIFT  14)
 0  REDUCE ENTRIES      3  SHIFT ENTRIES      0  GOTO ENTRIES

STATESET  17
(10,4)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  A
PARSING-STATE  17
(_|_ ,REDUCE  14)      (else,SHIFT  23)      (L   ,GOTO  22)
 1  REDUCE ENTRY        1  SHIFT ENTRY        1  GOTO ENTRY

STATESET  18
(11,3)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  id
PARSING-STATE  18
(then,REDUCE  11)
(or  ,REDUCE  11)
 2  REDUCE ENTRIES      0  SHIFT ENTRIES      0  GOTO ENTRIES

STATESET  19
(5,3)  (7,1)
NUMBER OF STATES = 2   ASSOCIATED SYMBOL IS  T
PARSING-STATE  19
(+   ,REDUCE   5)      (*   ,SHIFT  14)
()   ,REDUCE   5)
(else,REDUCE   5)
(_|_ ,REDUCE   5)
 4  REDUCE ENTRIES      1  SHIFT ENTRY        0  GOTO ENTRIES
```

```
STATESET  20
(7,3)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  P
PARSING-STATE  20
(+    ,REDUCE   7)
()    ,REDUCE   7)
(*    ,REDUCE   7)
(else,REDUCE    7)
(_|_ ,REDUCE    7)
 5  REDUCE ENTRIES      0  SHIFT ENTRIES      0  GOTO ENTRIES

STATESET  21
(8,3)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  )
PARSING-STATE  21
(+    ,REDUCE   8)
()    ,REDUCE   8)
(*    ,REDUCE   8)
(else,REDUCE    8)
(_|_ ,REDUCE    8)
 5  REDUCE ENTRIES      0  SHIFT ENTRIES      0  GOTO ENTRIES

STATESET  22
(10,5)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  L
PARSING-STATE  22
(_|_ ,REDUCE  10)
 1  REDUCE ENTRY        0  SHIFT ENTRIES      0  GOTO ENTRIES

STATESET  23
(13,1)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  else
PARSING-STATE  23
                       (id  ,SHIFT   2)       (D   ,GOTO  24)
                       (if  ,SHIFT   3)       (A   ,GOTO  24)
                                              (C   ,GOTO  24)
 0  REDUCE ENTRIES      2  SHIFT ENTRIES      3  GOTO ENTRIES

STATESET  24
(13,2)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  D
PARSING-STATE  24
(_|_ ,REDUCE  13)
 1  REDUCE ENTRY        0  SHIFT ENTRIES      0  GOTO ENTRIES


001.36 SECONDS IN EXECUTION
```

```
THERE ARE A TOTAL OF 25 PARSING-STATES
THEY CONTAIN 59 TERMINAL AND 18 NONTERMINAL ENTRIES
0 STATESETS ARE INADEQUATE
7 STATESETS ARE LR(0); THEIR REMOVAL SAVES 21 TERMINAL ENTRIES
TERMINAL DEFAULTS SAVE 5 TERMINAL ENTRIES
NONTERMINAL DEFAULTS REMOVE 7 NONTERMINAL ENTRIES
RENUMBERING PARSING-STATES SAVES 12 ENTRIES FROM NSTATE


STATESET     0   RENUMBERED AS STATESET    0
STATESET     1   RENUMBERED AS STATESET    6
STATESET     2   RENUMBERED AS STATESET    7
STATESET     3   RENUMBERED AS STATESET    8
STATESET     4   RENUMBERED AS STATESET    9
STATESET     5   RENUMBERED AS STATESET   10
STATESET     6   LR(0); ELIMINATED AS REDUCE    12
STATESET     7   RENUMBERED AS STATESET   11
STATESET     8   RENUMBERED AS STATESET   12
STATESET     9   RENUMBERED AS STATESET    1
STATESET    10   LR(0); ELIMINATED AS REDUCE     9
STATESET    11   RENUMBERED AS STATESET    2
STATESET    12   RENUMBERED AS STATESET   13
STATESET    13   RENUMBERED AS STATESET    3
STATESET    14   RENUMBERED AS STATESET    4
STATESET    15   RENUMBERED AS STATESET   14
STATESET    16   RENUMBERED AS STATESET   15
STATESET    17   RENUMBERED AS STATESET   16
STATESET    18   LR(0); ELIMINATED AS REDUCE    11
STATESET    19   RENUMBERED AS STATESET   17
STATESET    20   LR(0); ELIMINATED AS REDUCE     7
STATESET    21   LR(0); ELIMINATED AS REDUCE     8
STATESET    22   LR(0); ELIMINATED AS REDUCE    10
STATESET    23   RENUMBERED AS STATESET    5
STATESET    24   LR(0); ELIMINATED AS REDUCE    13


COMPACTED  PARSING  TABLE


PARSING-STATE  0
(id  ,SHIFT          7)      (D   ,SHIFT     6)
(if  ,SHIFT          8)
 2  TERMINAL ENTRIES     1  NONTERMINAL ENTRY
WAS NUMBERED   0    ASSOCIATED SYMBOL IS

PARSING-STATE  1
(id  ,SCANREDUCE    9)      (E   ,SHIFT   14)
((   ,SHIFT         1)      (T   ,SHIFT   15)
                            (P   ,SHIFT   15)
 2  TERMINAL ENTRIES     3  NONTERMINAL ENTRIES
WAS NUMBERED   9    ASSOCIATED SYMBOL IS  (
SAME TERMINAL ENTRIES AS PARSING-STATE 9

PARSING-STATE  2
(id  ,SHIFT          7)      (A   ,SHIFT   16)
 1  TERMINAL ENTRY       1  NONTERMINAL ENTRY
WAS NUMBERED  11    ASSOCIATED SYMBOL IS  then
```

```
PARSING-STATE  3
(id  ,SCANREDUCE   9)     (T   ,SHIFT   17)
((   ,SHIFT        1)     (P   ,SHIFT   17)
 2  TERMINAL ENTRIES    2  NONTERMINAL ENTRIES
WAS NUMBERED  13    ASSOCIATED SYMBOL IS  +
SAME TERMINAL ENTRIES AS PARSING-STATE 9

PARSING-STATE  4
(id  ,SCANREDUCE   9)     (P   ,REDUCE   7)
((   ,SHIFT        1)
 2  TERMINAL ENTRIES    1  NONTERMINAL ENTRY
WAS NUMBERED  14    ASSOCIATED SYMBOL IS  *
SAME TERMINAL ENTRIES AS PARSING-STATE 9

PARSING-STATE  5
(id  ,SHIFT        7)     (D   ,REDUCE  13)
(if  ,SHIFT        8)     (A   ,REDUCE  13)
                          (C   ,REDUCE  13)
 2  TERMINAL ENTRIES    3  NONTERMINAL ENTRIES
WAS NUMBERED  23    ASSOCIATED SYMBOL IS  else
SAME TERMINAL ENTRIES AS PARSING-STATE 0

PARSING-STATE  6
NO ENTRIES        'FINAL' PARSING-STATE
WAS NUMBERED   1    ASSOCIATED SYMBOL IS  D

PARSING-STATE  7
(:=  ,SHIFT        9)
 1  TERMINAL ENTRY
WAS NUMBERED   2    ASSOCIATED SYMBOL IS  id

PARSING-STATE  8
(id  ,SCANREDUCE  12)
 1  TERMINAL ENTRY
WAS NUMBERED   3    ASSOCIATED SYMBOL IS  if

PARSING-STATE  9
(id  ,SCANREDUCE   9)
((   ,SHIFT        1)
 2  TERMINAL ENTRIES
WAS NUMBERED   4    ASSOCIATED SYMBOL IS  :=

PARSING-STATE  10
(then,SHIFT        2)
(or  ,SHIFT       13)
 2  TERMINAL ENTRIES
WAS NUMBERED   5    ASSOCIATED SYMBOL IS  B

PARSING-STATE  11
(+   ,SHIFT        3)
(++++,REDUCE       3)
 2  TERMINAL ENTRIES
WAS NUMBERED   7    ASSOCIATED SYMBOL IS  E
```

```
PARSING-STATE  12
(+   ,SHIFT        3)
(*   ,SHIFT        4)
(++++,REDUCE       3)
 3  TERMINAL ENTRIES
WAS NUMBERED   8     ASSOCIATED SYMBOL IS  T

PARSING-STATE  13
(id  ,SCANREDUCE  11)
 1  TERMINAL ENTRY
WAS NUMBERED  12     ASSOCIATED SYMBOL IS  or

PARSING-STATE  14
(+   ,SHIFT        3)
()   ,SCANREDUCE   8)
 2  TERMINAL ENTRIES
WAS NUMBERED  15     ASSOCIATED SYMBOL IS  E

PARSING-STATE  15
(+   ,SHIFT        3)
()   ,SCANREDUCE   8)
(*   ,SHIFT        4)
 3  TERMINAL ENTRIES
WAS NUMBERED  16     ASSOCIATED SYMBOL IS  T

PARSING-STATE  16
(else,SHIFT        5)
(_|_ ,REDUCE      14)
 2  TERMINAL ENTRIES
WAS NUMBERED  17     ASSOCIATED SYMBOL IS  A

PARSING-STATE  17
(*   ,SHIFT        4)
(++++,REDUCE       5)
 2  TERMINAL ENTRIES
WAS NUMBERED  19     ASSOCIATED SYMBOL IS  T


NONTERMINAL DEFAULT ENTRIES

(D   ,SHIFT    6)
(A   ,SHIFT    6)
(E   ,SHIFT   11)
(T   ,SHIFT   12)
(P   ,SHIFT   12)
(C   ,SHIFT    6)
(B   ,SHIFT   10)
(L   ,REDUCE  10)


000.78 SECONDS IN EXECUTION
```

```
13 TERMINAL ENTRIES HAVE BEEN SAVED BY OVERLAPPING.

PARSING TABLES REQUIRE 171 BYTES

INTEGER NDEF= 0,TACT= 16,NACT= 56;

ARRAY 14 BYTE LHS=
(13,13,14,15,15,16,16,17,17,18,19,19,20,20);

ARRAY 18 SHORT INTEGER TSTATE=
(193,257,192,257,257,193,512,544,576,257,321,33,2,608,129,98,
385,449);

ARRAY 6 SHORT INTEGER NSTATE=
(0,34,128,161,224,258);

ARRAY 20 BYTE TSYM=
(5,2,12,5,4,2,1,6,1,3,7,8,9,11,5,12,11,10,1,1);

ARRAY 11 BYTE NSYM=
(13,15,16,17,14,16,17,17,13,14,18);

ARRAY 39 SHORT INTEGER ACT=
(6,6,11,12,12,6,10,17674,4,3,17155,4,8968,3,7,8,8457,1,2,13,5,
16398,4,17157,0,9,8460,8971,6,14,15,15,16,17,17,17159,16909,
16909,16909);

SHORT INTEGER SUPTACT SYN ACT;


000.28 SECONDS IN EXECUTION





GRAMMAR  IS  SLR


SLR STATESET AND PARSING TABLE


STATESET  0
(0,0)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS
PARSING-STATE  0
                        (id   ,SHIFT    4)       (D    ,GCTC    1)
                        (if   ,SHIFT    5)       (A    ,GOTO    2)
                                                 (C    ,GCTC    3)
 0  REDUCE ENTRIES        2  SHIFT ENTRIES        3  GOTO ENTRIES
```

```
STATESET   1
(0,1)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  D
PARSING-STATE  1
 0  REDUCE ENTRIES       0  SHIFT ENTRIES      0  GOTO ENTRIES

STATESET  2
(1,1)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  A
PARSING-STATE  2
(_|_ ,REDUCE   1)
 1  REDUCE ENTRY         0  SHIFT ENTRIES      0  GOTO ENTRIES

STATESET  3
(2,1)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  C
PARSING-STATE  3
(_|_ ,REDUCE   2)
 1  REDUCE ENTRY         0  SHIFT ENTRIES      0  GOTO ENTRIES

STATESET  4
(3,1)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  id
PARSING-STATE  4
                       (:=  ,SHIFT   6)
 0  REDUCE ENTRIES      1  SHIFT ENTRY         0  GOTO ENTRIES

STATESET  5
(10,1)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  if
PARSING-STATE  5
                       (id  ,SHIFT   8)      (B   ,GOTO   7)
 0  REDUCE ENTRIES      1  SHIFT ENTRY         1  GOTO ENTRY

STATESET  6
(3,2)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  :=
PARSING-STATE  6
                       (id  ,SHIFT  13)      (E   ,GOTO   9)
                       ((   ,SHIFT  12)      (T   ,GOTO  10)
                                             (P   ,GOTO  11)
 0  REDUCE ENTRIES      2  SHIFT ENTRIES       3  GOTO ENTRIES

STATESET  7
(10,2)  (11,1)
NUMBER OF STATES = 2   ASSOCIATED SYMBOL IS  B
PARSING-STATE  7
                       (then,SHIFT  14)
                       (or  ,SHIFT  15)
 0  REDUCE ENTRIES      2  SHIFT ENTRIES       0  GOTO ENTRIES
```

```
STATESET   8
(12,1)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  id
PARSING-STATE  8
(then,REDUCE  12)
(or  ,REDUCE  12)
 2  REDUCE ENTRIES       0  SHIFT ENTRIES      0  GOTO ENTRIES

STATESET   9
(3,3)   (5,1)
NUMBER OF STATES = 2   ASSOCIATED SYMBOL IS  E
PARSING-STATE  9
(else,REDUCE   3)      (+   ,SHIFT  16)
(_|_ ,REDUCE   3)
 2  REDUCE ENTRIES       1  SHIFT ENTRY        0  GOTO ENTRIES

STATESET   10
(4,1)   (7,1)
NUMBER OF STATES = 2   ASSOCIATED SYMBOL IS  T
PARSING-STATE  10
(+   ,REDUCE   4)      (*   ,SHIFT  17)
()   ,REDUCE   4)
(else,REDUCE   4)
(_|_ ,REDUCE   4)
 4  REDUCE ENTRIES       1  SHIFT ENTRY        0  GOTO ENTRIES

STATESET   11
(6,1)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  P
PARSING-STATE  11
(+   ,REDUCE   6)
()   ,REDUCE   6)
(*   ,REDUCE   6)
(else,REDUCE   6)
(_|_ ,REDUCE   6)
 5  REDUCE ENTRIES       0  SHIFT ENTRIES      0  GOTO ENTRIES

STATESET   12
(8,1)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  (
PARSING-STATE  12
                       (id  ,SHIFT  13)      (E   ,GOTO  18)
                       ((   ,SHIFT  12)      (T   ,GOTO  10)
                                             (P   ,GOTO  11)
 0  REDUCE ENTRIES       2  SHIFT ENTRIES      3  GOTO ENTRIES

STATESET   13
(9,1)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  id
PARSING-STATE  13
(+   ,REDUCE   9)
()   ,REDUCE   9)
(*   ,REDUCE   9)
(else,REDUCE   9)
(_|_ ,REDUCE   9)
 5  REDUCE ENTRIES       0  SHIFT ENTRIES      0  GOTO ENTRIES
```

```
STATESET   14
(10,3)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  then
PARSING-STATE   14
                       (id  ,SHIFT   4)       (A    ,GCTC  19)
 0  REDUCE ENTRIES      1  SHIFT ENTRY         1  GOTO ENTRY

STATESET   15
(11,2)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  or
PARSING-STATE   15
                       (id  ,SHIFT  20)
 0  REDUCE ENTRIES      1  SHIFT ENTRY         0  GOTO ENTRIES

STATESET   16
(5,2)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  +
PARSING-STATE   16
                       (id  ,SHIFT   13)      (T    ,GCTC  21)
                       ((   ,SHIFT   12)      (P    ,GOTO   11)
 0  REDUCE ENTRIES      2  SHIFT ENTRIES       2  GOTC ENTRIES

STATESET   17
(7,2)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  *
PARSING-STATE   17
                       (id   ,SHIFT   13)     (P    ,GOTO   22)
                       ((    ,SHIFT   12)
 0  REDUCE ENTRIES      2  SHIFT ENTRIES       1  GOTO ENTRY

STATESET   18
(5,1)   (8,2)
NUMBER OF STATES = 2   ASSOCIATED SYMBOL IS  E
PARSING-STATE   18
                       (+    ,SHIFT   16)
                       ()    ,SHIFT   23)
 0  REDUCE ENTRIES      2  SHIFT ENTRIES       0  GOTC ENTRIES

STATESET   19
(10,4)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  A
PARSING-STATE   19
(_|_ ,REDUCE   14)     (else,SHIFT  25)       (L    ,GCTC  24)
 1  REDUCE ENTRY        1  SHIFT ENTRY         1  GOTO ENTRY

STATESET   20
(11,3)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  id
PARSING-STATE   20
(then,REDUCE   11)
(or   ,REDUCE   11)
 2  REDUCE ENTRIES      0  SHIFT ENTRIES       0  GOTC ENTRIES
```

```
STATESET  21
(5,3)   (7,1)
NUMBER OF STATES = 2   ASSOCIATED SYMBOL IS  T
PARSING-STATE  21
(+   ,REDUCE   5)       (*   ,SHIFT  17)
()   ,REDUCE   5)
(else,REDUCE   5)
(_|_ ,REDUCE   5)
 4  REDUCE ENTRIES       1  SHIFT ENTRY          0  GOTO ENTRIES

STATESET  22
(7,3)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  P
PARSING-STATE  22
(+   ,REDUCE   7)
()   ,REDUCE   7)
(*   ,REDUCE   7)
(else,REDUCE   7)
(_|_ ,REDUCE   7)
 5  REDUCE ENTRIES       0  SHIFT ENTRIES        0  GOTO ENTRIES

STATESET  23
(8,3)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  )
PARSING-STATE  23
(+   ,REDUCE   8)
()   ,REDUCE   8)
(*   ,REDUCE   8)
(else,REDUCE   8)
(_|_ ,REDUCE   8)
 5  REDUCE ENTRIES       0  SHIFT ENTRIES        0  GOTO ENTRIES

STATESET  24
(10,5)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  L
PARSING-STATE  24
(_|_ ,REDUCE  10)
 1  REDUCE ENTRY         0  SHIFT ENTRIES        0  GOTC ENTRIES

STATESET  25
(13,1)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  else
PARSING-STATE  25
                        (id  ,SHIFT   4)      (D   ,GOTO  26)
                        (if  ,SHIFT   5)      (A   ,GOTC   2)
                                              (C   ,GOTO   3)
 0  REDUCE ENTRIES       2  SHIFT ENTRIES        3  GOTO ENTRIES

STATESET  26
(13,2)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  D
PARSING-STATE  26
(_|_ ,REDUCE  13)
 1  REDUCE ENTRY         0  SHIFT ENTRIES        0  GOTC ENTRIES


001.44 SECONDS IN EXECUTION
```

G_7 is an ambiguous grammar (and hence not SLR), but which is SLRC. The tables presented are the output of the programs SLRIN and SLR (with and without chain elimination).

```
TERMINALS
  1  c      _|_

NONTERMINALS
  3  ++++ A      B

PRODUCTIONS
  0  ++++  ::= A     _|_
  1  A     ::= c     | B
  3  B     ::= c


THERE ARE 2 TERMINALS AND 3 NONTERMINALS
THERE ARE 4 PRODUCTIONS WITH AVERAGE LENGTH OF RHS 1.25
PRODUCTIONS 1 2 3 ARE CHAIN PRODUCTIONS


000.13 SECONDS IN EXECUTION



TABLE OF FOLLOW

A     |   _|_
B     |   _|_


GRAMMAR  IS  SLRC


SLRC STATESET AND PARSING TABLE


STATESET  0
(0,0)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS
PARSING-STATE  0
                       (c    ,SHIFT    1)       (A    ,GOTO    1)
                                                (B    ,GOTO    1)
 0  REDUCE ENTRIES       1  SHIFT ENTRY          2  GOTO ENTRIES

STATESET  1
(0, 1)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  A
PARSING-STATE  1
 0  REDUCE ENTRIES      0  SHIFT ENTRIES        0  GOTO ENTRIES


000.21 SECONDS IN EXECUTION
```

```
GRAMMAR  IS  NOT  SLR


SLR STATESET AND PARSING TABLE


STATESET  0
(0,0)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS
PARSING-STATE  0
                        (c    ,SHIFT    2)       (A    ,GOTO    1)
                                                 (B    ,GCTC    3)
 0  REDUCE ENTRIES       1  SHIFT  ENTRY          2  GOTO ENTRIES

STATESET  1
(0,1)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  A
PARSING-STATE  1
 0  REDUCE ENTRIES       0  SHIFT ENTRIES        0  GOTC ENTRIES

STATESET  2
(1,1)   (3,1)
             **********    INADEQUATE    **********
NUMBER OF STATES = 2   ASSOCIATED SYMBOL IS  c
PARSING-STATE  2
(_|_ ,REDUCE    1)
(_|_ ,REDUCE    3)
 2  REDUCE ENTRIES       0  SHIFT ENTRIES        0  GOTC ENTRIES

STATESET  3
(2,1)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  B
PARSING-STATE  3
(_|_ ,REDUCE    2)
 1  REDUCE ENTRY         0  SHIFT ENTRIES        0  GOTC ENTRIES


000.23 SECONDS IN EXECUTION
```

$\mathcal{G}_8$ is an SLR grammar which is not Λ-free. It is not SLRC or even SLRPC. The tables presented are the output of the programs SLRIN and SLR (with and without partial chain elimination).

```
TERMINALS
  1  d     e     _|_

NONTERMINALS
  4  ++++ A     B     C     L

PRODUCTIONS
  0  ++++ ::= A     _|_
  1  A    ::= B     L     | C     d     | L     d
  4  B    ::= e
  5  C    ::= e
  6  L    ::=

THERE ARE 3 TERMINALS AND 5 NONTERMINALS
THERE ARE 7 PRODUCTIONS WITH AVERAGE LENGTH OF RHS 1.43
PRODUCTIONS 4 5 ARE CHAIN PRODUCTIONS


000.16 SECONDS IN EXECUTION


L     GENERATES THE EMPTY STRING


TABLE OF FOLLOW

A       |   _|_
B       |   _|_
C       |   d
L       |   d     _|_
```

```
GRAMMAR  IS  NOT  SLRPC


SLRPC STATESET AND PARSING TABLE


STATESET  0
(0,0)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS
PARSING-STATE  0
(d    ,REDUCE    6)     (e    ,SHIFT    5)       (A    ,GOTO    1)
(_|_  ,REDUCE    6)                              (B    ,GOTO    2)
                                                 (C    ,GOTO    3)
                                                 (L    ,GOTO    4)
 2  REDUCE ENTRIES       1  SHIFT ENTRY           4  GOTO ENTRIES

STATESET  1
(0,1)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  A
PARSING-STATE  1
 0  REDUCE ENTRIES       0  SHIFT ENTRIES         0  GOTO ENTRIES

STATESET  2
(1,1)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  B
PARSING-STATE  2
(d    ,REDUCE    6)                              (L    ,GOTO    6)
(_|_  ,REDUCE    6)
 2  REDUCE ENTRIES       0  SHIFT ENTRIES         1  GOTO ENTRY

STATESET  3
(2,1)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  C
PARSING-STATE  3
                        (d    ,SHIFT    7)
 0  REDUCE ENTRIES       1  SHIFT ENTRY           0  GOTO ENTRIES

STATESET  4
(3,1)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  L
PARSING-STATE  4
                        (d    ,SHIFT    8)
 0  REDUCE ENTRIES       1  SHIFT ENTRY           0  GOTO ENTRIES

STATESET  5
(1,1)   (2,1)
           **********     INADEQUATE    **********
NUMBER OF STATES = 2   ASSOCIATED SYMBOL IS  e
PARSING-STATE  5
(d    ,REDUCE    6)     (d    ,SHIFT    7)       (L    ,GOTO    6)
(_|_  ,REDUCE    6)
 2  REDUCE ENTRIES       1  SHIFT ENTRY           1  GOTO ENTRY
```

```
STATESET   6
(1,2)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  L
PARSING-STATE  6
(_|_ ,REDUCE    1)
 1  REDUCE ENTRY          0  SHIFT ENTRIES        0  GOTO ENTRIES

STATESET   7
(2,2)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  d
PARSING-STATE  7
(_|_ ,REDUCE    2)
 1  REDUCE ENTRY          0  SHIFT ENTRIES        0  GOTO ENTRIES

STATESET   8
(3,2)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  d
PARSING-STATE  8
(_|_ ,REDUCE    3)
 1  REDUCE ENTRY          0  SHIFT ENTRIES        0  GOTO ENTRIES


000.48 SECONDS IN EXECUTION




GRAMMAR  IS  SLR


SLR STATESET AND PARSING TABLE


STATESET   0
(0,0)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS
PARSING-STATE  0
(d    ,REDUCE    6)     (e    ,SHIFT    5)       (A    ,GOTO    1)
(_|_  ,REDUCE    6)                              (B    ,GOTO    2)
                                                 (C    ,GOTO    3)
                                                 (L    ,GOTO    4)
 2  REDUCE ENTRIES       1  SHIFT ENTRY          4  GOTO ENTRIES

STATESET   1
(0,1)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  A
PARSING-STATE  1
 0  REDUCE ENTRIES       0  SHIFT ENTRIES        0  GOTO ENTRIES
```

```
STATESET  2
(1,1)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  B
PARSING-STATE  2
(d   ,REDUCE    6)                                (L   ,GOTO   6)
(_|_ ,REDUCE    6)
 2  REDUCE ENTRIES       0  SHIFT ENTRIES        1  GOTO ENTRY

STATESET  3
(2,1)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  C
PARSING-STATE  3
                        (d   ,SHIFT   7)
 0  REDUCE ENTRIES       1  SHIFT ENTRY          0  GOTO ENTRIES

STATESET  4
(3,1)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  L
PARSING-STATE  4
                        (d   ,SHIFT   8)
 0  REDUCE ENTRIES       1  SHIFT ENTRY          0  GOTO ENTRIES

STATESET  5
(4,1)   (5,1)
NUMBER OF STATES = 2   ASSOCIATED SYMBOL IS  e
PARSING-STATE  5
(d   ,REDUCE    5)
(_|_ ,REDUCE    4)
 2  REDUCE ENTRIES       0  SHIFT ENTRIES        0  GOTO ENTRIES

STATESET  6
(1,2)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  L
PARSING-STATE  6
(_|_ ,REDUCE    1)
 1  REDUCE ENTRY         0  SHIFT ENTRIES        0  GOTO ENTRIES

STATESET  7
(2,2)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  d
PARSING-STATE  7
(_|_ ,REDUCE    2)
 1  REDUCE ENTRY         0  SHIFT ENTRIES        0  GOTO ENTRIES

STATESET  8
(3,2)
NUMBER OF STATES = 1   ASSOCIATED SYMBOL IS  d
PARSING-STATE  8
(_|_ ,REDUCE    3)
 1  REDUCE ENTRY         0  SHIFT ENTRIES        0  GOTO ENTRIES


000.43 SECONDS IN EXECUTION
```

References

Aho A.V. and Ullman J.D. (to be published)

'Techniques for Parser Optimisation' (Chapter 6)

Anderson T., Eve J. and Horning J.J. (1971)

'Efficient LR(1) parsers'

Technical Report 24, Computing Laboratory,

University of Newcastle upon Tyne

DeRemer F.L. (1969)

'Practical Translators for LR(k) Languages'

Ph.D. Thesis, Department of Electrical Engineering,

M.I.T.

DeRemer F.L. (1971)

Simple LR(k) Grammars

CACM 14 p.453

Earley J. (1965)

'Generating a recognizer for a BNF grammar'

Computation Centre Report,

Carnegie Institute of Technology, Pittsburgh

Earley J. (1970)

An Efficient Context-Free Parsing Algorithm

CACM 13 p.94

Feldman J. and Gries D. (1968)

Translator Writing Systems

CACM 11 p.77

Floyd R.W. (1963)

Syntactic Analysis and Operator Precedence

JACM 10 p.316

Floyd R.W. (1964)

Bounded Context Syntactic Analysis

CACM 7 p.62

Ginsburg S. and Greibach S.A. (1966)

Deterministic Context Free Languages

Information and Control 9 p.620

Graham S.L. (1970)

Extended Precedence Languages, Bounded Right Context Languages, and Deterministic Languages

IEEE Conference Record of the Eleventh Annual Symposium on Switching and Automata Theory p. 175

Gray J.N. and Harrison M.A. (1969)

Single Pass Precedence Analysis

IEEE Conference Record of the Tenth Annual Symposium on Switching and Automata Theory p. 106

Gries D. (1968)

Use of Transition Matrices in Compiling

CACM 11 p.26

Hext J.B. and Roberts P.S. (1970)

Syntax analysis by Domolki's algorithm

Computer Journal 13 p.263

Ichbiah J.D. and Morse S.P. (1970)

A Technique for Generating Almost Optimal Floyd-Evans Productions for Precedence Grammars

CACM 13 p.501

Irons E.T. (1964)

"Structural Connections" in Formal Languages

CACM 7 p.67

Knuth D.E. (1965)

On the Translation of Languages from Left to Right

Information and Control 8 p.607

Knuth D.E. (1971)

Top-Down Syntax Analysis

Acta Informatica 1 p.79

Korenjak A.J. (1969)

A Practical Method for Constructing LR(k) Processors

CACM 12 p.613

Lalonde W.R. (1971)

'An Efficient LALR Parser Generator'

Technical Report CSRG-2, University of Toronto

Lewis II P.M. and Stearns R.E. (1968)

Syntax-Directed Transduction

JACM 15 p.465

Lynch W.C. (1968)

'A high-speed parsing algorithm for ICOR grammars'

Report 1097, Andrew Jennings Computing Centre,

Case Western Reserve University

McKeeman W.M. (1966)

'An Approach to Computer Language Design'

Technical Report CS48, Stanford University

McKeeman W.M., Horning J.J. and Wortman D.B. (1970)

'A Compiler Generator'

Prentice-Hall

Pager D. (1970)

A Solution to an Open Problem by Knuth

Information and Control 17 p.462

Wirth N. (1968)

PL360, A Programming Language for the 360 Computers

JACM 15 p.37

Wirth N. and Hoare C.A.R. (1966)

A Contribution to the Development of ALGOL

CACM 9 p.413

Wirth N. and Weber H. (1966)

EULER: A Generalisation of ALGOL, and its Formal Definition

CACM 9 p.13