ON PARSING CONTEXT FREE LANGUAGES IN PARALLEL ENVIRONMENTS+

Charles N. Fischer

TR 75-237

April 1975

Department of Computer Science
Cornell University
Ithaca, New York 14853

+This research was supported in part by National Science Foundation under Grant GJ-42512.

ON PARSING CONTEXT FREE LANGUAGES
IN PARALLEL ENVIRONMENTS

Charles Nicholas Fischer, Ph.D.
Cornell University, 1975

ABSTRACT

Non-canonical generalizations of several bottom-up parsing methods, including Simple Precedence, Simple Mixed Strategy Precedence, LR(k), SLR(k), and LALR(k) are considered. It is seen that these methods can readily be made to generate many concurrent reductions and thus can be used to advantage in parallel environments. It is suggested that such methods could be used to produce practical parsers for such parallel computers as the CDC Star-100. Further, the grammar classes defined by these parallel methods are studied and compared with the grammar classes defined by a number of serial parsing techniques.

BIOGRAPHICAL SKETCH

Charles Nicholas Fischer was born on August 14, 1948 in Buffalo, New York, the son of Mr. and Mrs. Fillmore Fischer. He graduated from the State University of New York at Buffalo in 1970 with a B.A. in Physics awarded summa cum laude. He entered the graduate school at Cornell University in September of 1970 and obtained an M.S. in Computer Science in 1973.

Mr. Fischer is a member of Phi Beta Kappa and the Association for Computing Machinery.

To my Parents

ACKNOWLEDGEMENTS

I wish to thank Professor John H. Williams, my thesis advisor, for his help and guidance during the course of my research. I am also indebted to Professor Robert Constable and Professor Richard Conway for serving on my Special Committee and to Professor John Hopcroft for many helpful conversations. Finally, I must acknowledge the generous aid and support of my fellow graduate students including, but by no means limited to, Mr. Larry Brenner, Mr. Robert Fisher, Mr. Narain Gehani, Mr. Morris Siegel and Mr. Dennis Ting.

TABLE OF CONTENTS

1 INTRODUCTION

Surely one of the most salient themes in computer design today is parallelism. Such diverse architectures as the Illiac IV [BB 68], CDC Star-100 [Co 73], and C.mmp network [WB 72] all rely extensively on concurrent operation and thus may justifiably be termed parallel computers.

Quite naturally, the problem of designing programming languages for parallel computers has received much attention (Zwankenberg [Zw 72], Millstein [Mi 73]), as has the problem of generating efficient object code for them (Lamport [La 74]). Rather surprisingly however, virtually the only work done on the collateral problem of developing efficient compilers to run on parallel machines is that of Zozel [Zo 73]. We believe that additional work in this area is very much in order.

Theoretically, such work should provide additional insight into the nature of the compilation process. In particular, is compilation an inherently serial process or is it, to some degree, amenable to concurrency?

Practically, if parallel compilation techniques are not developed, then, as parallel machines are installed, compilation will either be done on separate, serial, machines (as is done with the Illiac IV) or, with great inefficiency, on the parallel machines themselves. Clearly, neither possibility is as attractive or flexible as an efficient parallel compiler.

Let us, as is commonly done, divide the compilation process into three constituent parts:

(1) Lexical analysis (or scanning)

(2) Code generation and semantic analysis

(3) Parsing (or syntatic analysis)

The problem of performing lexical analysis in parallel environments has been considered by both Zozel and Lincoln [Li 70]. Although by no means exhastively studied, the problem is fairly well understood and adequate solutions exist.

We briefly touch upon the problem of performing, in parallel, code generation and semantic analysis in Chapter 7. For the most part however, it remains open. We believe it to be a fruitful area for future research.

The subject matter of this dissertation is the problem of parsing context free languages in parallel environments. It is our thesis that a number of well known, serial, bottom-up parsing techniques can be generalized to perform many concurrent reductions. Further, such techniques yield reasonably simple and compact parsers. Inasmuch as such parsers are inherently non-canonical (that is, not strictly left to right), the work of Colmerauer [Co 70], Williams [Wi 69], and Szymanski [Sz 73] on non-canonical parsing techniques is both complemented and extended.

In Chapter 2 we review briefly some of the fundamentals of context free grammars and parsing. In Chapter 3 Simple

Precedence Grammars are considered. It is seen that the Simple Precedence conditions can be strengthened to yield the Strict Precedence Grammars and weakened to yield Colmerauer's Total Precedence Grammars. Simple Mixed Strategy Precedence Grammars are considered in Chapter 4. As was the case with Simple Precedence, grammar requirements may be strengthened to yield the Strict Mixed Strategy Precedence Grammars or weakened to yield the Total Mixed Strategy Precedence Grammars. In Chapters 5 and 6 the well known LR parsing techniques (LR(k), SLR(k), LALR(k)) are abstracted to a common form. It is proposed that, rather than using a single serial parser, a number of Synchronous Parsing Machines (SPM's) be run along the entire length of an input string. Finally, in Chapter 7, a special parsing technique for operator infix expressions is investigated. This technique, it is seen, can be extended to yield object code (in the form of quadruples) directly from the input, bypassing conventional parsing completely.

2 PRELIMINARIES

2.1 Grammars, Languages, and Phrases

In this section we review basic terminology and exhibit our notation. We assume that the reader is familiar with such concepts as strings, formal grammars, formal languages, finite state automata (FSA), push down automata (PDA), regular expressions, decidability and undecidability.

We denote a context free grammar (CFG) by a quadruple (V,V_T,P,S) where

V is the vocabulary of the grammar

$V_T \subseteq V$ is the terminal vocabulary

$P \subseteq (V-V_T) \times V^*$ is a finite set of productions

$S \in V-V_T$ is the start symbol

Individual productions of a grammar are written $A_i \rightarrow \gamma_i$ where A_i designates whatever non-terminal is the left side of the i-th production and γ_i designates that string of V^* which is the right side of the i-th production. We allow empty right sides and refer to such productions as ε-rules. We also use the symbol $\rightarrow$ to denote the application of a production to a string. Thus if $\phi=wA_iy$ and $\psi=w\gamma_iy$, we will write $\phi\rightarrow\psi$. An asterisk (*) written above a relation denotes the reflexive and transitive completion of that relation and a plus sign (+) the transitive completion.

We designate by SF(G) the set of sentential forms derivable from S according to the grammar G, that is,

$$SF(G) = \{w \in V^* \mid S \overset{*}{\rightarrow} w\}.$$

The *language* of G, written L(G), is the set of terminal strings derivable by G, that is

$L(G) = SF(G) \cap V_T^*$.

A derivation $S \to w_1 \to w_2 \to \ldots \to w_n$ is said to be a *rightmost* (or *canonical*) *derivation* if at each step the production being applied is applied to the rightmost nonterminal in the current sentential form w_i. We symbolize a rightmost derivation step by $\to_{rt}$.

A CFG is said to be *unambiguous* if $x \in L(G)$ implies x has a unique canonical derivation.

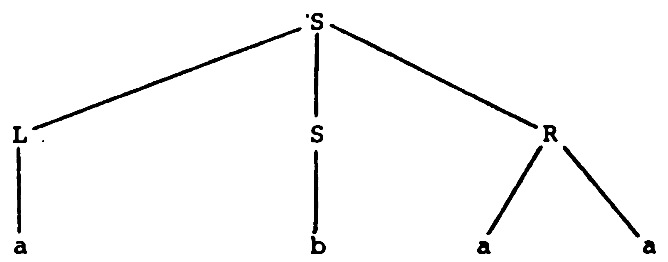
We say that γ_i is a *phrase* of a sentential form $w\gamma_i y$ if there is a derivation $S \to^* wA_i y \to^+ w\gamma_i y$; γ_i is a *simple phrase* if $wA_i y \to w\gamma_i y$. $w\gamma_i y$ may be *reduced* to $wA_i y$ by replacing γ_i with A_i. This operation is called a *reduction*. A leftmost simple phrase of a sentential form is called a *handle*. If a CFG is unambiguous, every sentential form has a unique handle.

To every sentential form there corresponds a *derivation tree*. This tree is rooted by S (the start symbol). If $A \in V_N \equiv (V-V_T)$ is rewritten using $A \to X_1 \ldots X_M$ then A has $X_1, \ldots, X_M$ as direct descendents. The phrases and simple phrases of a sentential form are immediately obvious in its derivation tree.

For example, if we have the CFG G:

$S \to LSR \quad S \to b \quad L \to a \quad R \to aa$,

then $L(G) = \{a^n b a^{2n} \mid n \geq 0\}$. The derivation tree of abaa is



A CFG is said to be <u>reduced</u> if $\forall$ $A \epsilon V_N$

(1) $S \to^* \ldots A \ldots$

(2) $A \to^+ x$ where $x \epsilon V_T^*$.

That is, a reduced CFG contains no "useless" non-terminals. In this dissertation, we shall assume that <u>all</u> CFG's are reduced.

2.2 Recognizers and Parsers

A recognizer is an algorithm which, given G, a CFG, and $x \in V_T^*$, determines whether or not $x \in L(G)$.

A canonical parser is an algorithm which, given G, a CFG, and $x \in V_T^*$,

(1) Signals an error if $x \notin L(G)$

(2) Otherwise, reduces x to S by repeatedly finding and reducing a handle. As output, it produces $(p_n, \ldots, p_1, p_0)$, the productions used to reduce x to S.

Note that $(p_n, \ldots, p_1, p_0)$ encodes the derivation tree of x for x can be canonically derived from S by applying, in order, the productions $p_0, p_1, \ldots, p_n$. Further, if G is unambiguous, then this sequence of productions is unique.

A non-canonical parser is an algorithm which, given G, a CFG, and $x \in V_T^*$,

(1) Signals an error if $x \notin L(G)$

(2) Otherwise, reduces x to S by repeatedly finding and reducing simple phrases. An encoding of the derivation tree of x is generated as output.

Note that a given derivation tree may have many different encodings, but each encoding uniquely specifies a single derivation tree. Thus, if necessary, an encoding may be transformed into an alternate, unique encoding (such as the canonical encoding used above).

Clearly, the canonical parser is a special case of the more general non-canonical parser.

In the balance of this dissertation we shall consider non-canonical parsing algorithms which can effectively utilize parallel computing environments.

2.3 Miscellaneous Notation

In this section we present a number of notational devices which will be of use to us in various parts of this dissertation.

Unless explicitly overridden, the following conventions hold for naming strings:

a,b,c,d,...	are terminal characters
A,B,C,D,...	are non-terminal characters
U,V,...,Z	are either terminal or non-terminal characters
$\alpha,\beta,\gamma,\ldots$	are elements of V^*
u,v,...	are elements of V_T^*

The notation $|\alpha|$ denotes the number of characters in the string α.

The set of characters that may follow a character U in some sentential form is given by

$$\mathrm{Follow}(U) = \{Y \mid S \rightarrow^* \ldots UY \ldots\}.$$

The first k characters of a string are given by

$$\mathrm{First}_k(\gamma)=\delta \text{ where } \gamma=\delta\delta' \text{ and } |\delta|=k.$$

A notation very similar to that of APL [Iv 62] is rather extensively used. A summary of this notation may be found in the appendix.

3 SIMPLE PRECEDENCE TECHNIQUES

We first review briefly the well known Simple Precedence techniques of Wirth and Weber [WW 66]. It is then seen that a strengthening of the Simple Precedence conditions yields the Strict Precedence Grammars. These grammars are well suited to parallel parsing, yielding compact and efficient parsers. Alternatively, the Simple Precedence conditions may be weakened to yield the Total Precedence Grammars. This class of grammars contains (properly) both the Simple and Strict Precedence Grammars and yet can still be parsed efficiently in parallel environments.

3.1 Simple Precedence

The basic problem of all canonical bottom-up parsing techniques is that of finding the handle in a given right sentential form. The Simple Precedence technique represents a most straightforward solution to the problem. Three disjoint binary relations are defined on the vocabulary. These relations, denoted <., =., >. are used to delimit the left end, interior, and right end of the handle. Thus the handle is found merely by finding the leftmost sequence of characters having the sequence of relations $<. \; (=.)^{*} >.$

Formally, let $G=(V,V_T,P,S)$ be a CFG. We define the following relations over V.

(1) $A \lambda Y$ if $A \to Y \ldots \varepsilon P$ (λ denotes left derives)

(2) $X \alpha Y$ if $A \to \ldots XY \ldots \varepsilon P$ (α denotes adjacent)

(3) $X \rho A$ if $A \to \ldots X \varepsilon P$ (ρ denotes right derives)

Using these, we define the Wirth-Weber precedence relations:

(1) $X <\cdot Y$ iff $X \alpha \lambda^{+} Y$

(2) $X \doteq Y$ iff $X \alpha Y$

(3) $X \cdot> Y$ iff $X \rho^{+} \alpha \lambda^{*} Y$

We may now define the class of Simple Precedence Grammars. Let G be some CFG which has no ε-productions. Then G is a Simple Precedence grammar if:

(1) At most one Wirth-Weber relation holds between any pair of symbols in V.

(2) The grammar is uniquely invertible, that is $A \to x \varepsilon P$ and $B \to x \varepsilon P$ implies that $A = B$.

We will assume that all Simple Precedence grammars are augmented with endmarkers \$, where \$ is added to V_T and $S' \to \$S\$$ is added to P. Further, S' is made the start symbol and added to V_N.

We recall the following well known result ([WW 66]).

Theorem 3.1 Let G be an augmented Simple Precedence grammar. Then

(1) G is unambiguous

(2) Let $\gamma = X_1 \ldots X_n \varepsilon SF(G)$. Then the handle of γ is the leftmost substring of γ, $X_i \ldots X_j$ st $X_{i-1} <\cdot X_i$, $X_i \doteq X_{i+1}, \ldots, X_{j-1} \doteq X_j$, $X_j \cdot> X_{j+1}$ $\square$

This is the basis of the Simple Precedence parser. We use cond (2) of Thm 3.1 to locate the handle. The unique invertibility property ensures that the non-terminal to be substituted for the handle is uniquely defined. We detail this in the following algorithm.

<u>Algorithm 3.2</u> (A Simple Precedence parser)

Input: A string x of the form \$x'\$ stored in INPUT

Output: A sequence of production indices representing the sequence of reductions performed during the parse

[1] (Initialization)

LIMIT← |INPUT|; POINTER←2

Stack INPUT[1]=\$ onto an empty push-down stack

[2] (Done with parse?)

If POINTER>LIMIT

then if the stack contains \$S\$

then stop and signal a correct parse

else stop and signal an error

[3] (Scan the next character)

If the top stack character <. or =. INPUT[POINTER]

then push INPUT[POINTER] onto the stack

POINTER←POINTER+1; Go to [2]

else if the top stack character >. INPUT[POINTER]

then pop characters $Y_j, Y_{j-1}, \ldots$ until the last character popped, Y_i, is such that the top stack character <.Y_i

If there exists some production $A \rightarrow Y_i \ldots Y_j$

then push A onto the stack

output k, the index of the production applied; Go to [3]

[4] (If we get here, an error has been detected in [3])

Stop and signal an error

$\square$

We recall the following well known facts about Simple Precedence parsing.

Theorem 3.3 Algorithm 3.2

(1) Produces a correct, canonical parse

(2) Operates in time and space bounded by $O(|\text{Input}|)$.

$\square$

One point which we glossed over in Alg 3.2 is how a string of characters can be "looked-up" to determine whether or not it is the right hand side of some production, and if so what its corresponding left hand side is.

We shall consider three methods (A,B, and C), each of which encodes a string of characters one by one, and then operates on the encoding to determine its validity as a right hand side.

In each case, we shall assume that symbols are encoded as integers in the range 1 to $|V|$, and that strings are encoded from right to left (the order in which they are popped from the stack).

(A) Numeric encoding

The string is treated as a number in base $1+|V|$ and the value of this number is stored as an integer.

(1) Encoding: Given the encoding of $X_1 \ldots X_i$, we obtain the encoding of $X_1 \ldots X_{i+1}$ by multiplying by $1+|V|$ and then adding X_{i+1}.

(2) Look-up: A binary search of a table of encoded right hand sides is performed.

(3) Limitations: Since there is a maximum integer value that can be stored in a computer word, there is a maximum length string that can be encoded. Typically, given $1+|V| = 100$ and a 60 bit word, we can store a string of length 8, which is usually sufficient.

(B) Bit vector

This method is similar to Domolki's Algorithm (Domolki [Do 67], Wise [Wi 71]). We create bit vectors $V(i,c)$ of length $|P|$, where $1 \leq i \leq$ Max_Rhs_Length and $c \in V$. $V_j(i,c)=1$ iff production j is $A \to X_n \ldots X_i \ldots X_1$ and $c = X_i$.

(1) Encoding: We let a vector of all 1's represent the empty word. Given the encoding of $X_{i-1} \ldots X_1$, we obtain the encoding of $X_i \ldots X_1$ by anding on the vector $V(i,X_i)$. Note that by assigning the indices of the productions carefully, we can place all productions ending with the same character together. After encoding the first (rightmost) character, only a moderate number of bits, all adjacent, will equal 1. This

means the encoding (and look-up) can be done conviniently by operations on a single word (usually 30 to 60 bits).

(2) Look-up: Given the encoding of $X_i \ldots X_l$, we and on a vector M where $|M|=|P|$ and $M_j=1$ iff production j is $A \rightarrow \gamma$ where $|\gamma|=i$. Note that at most one bit equals 1 in this encoding (since the k-th bit is on iff production k is $A \rightarrow X_i \ldots X_l$ for some A). We can easily test if no bits are on (meaning that $X_i \ldots X_l$ is not the right hand side of any production). We can find out what bit is equal to 1 in two ways. First, a binary search could be used as in method (A). A faster, but much less obvious approach is (believe it or not!) to use a quirk of floating point arithmetic. We pack the encoding obtained above into a floating point number as its coefficient with a zero exponent. Then a zero is added. During the add operation, the result is <u>normalized</u> by shifting the coefficient left until the most significant bit is a 1. The exponent is decreased by the number of left shifts needed to do this. Thus by looking at the exponent we easily find out where the leftmost (and only) 1 bit was.

(3) Limitations: We must store the V(i,c)'s. Typically, given $|P|=100$, Max_Rhs_length=8 and $|V|=100$ we require 80,000 bits = 1.33K (60 bit) words.

(C) Finite automaton

This is essentially the method used by many lexical scanners (see p 251 of [AU 72]). States of the automaton are of the form (i,S) where $1 \leq i \leq$ Max_Rhs_Length, $S \epsilon 2^{\{1,\ldots,|P|\}}$. The meaning of a state (i,S) is that having scanned i characters, (say $X_i \ldots X_1$) the elements of S are the productions having $X_i \ldots X_1$ as a suffix of their right hand sides. Thus the start state S_0 is $(0,\{1,\ldots,|P|\})$ and $(i,S) \overset{a}{\rightarrow} (i+1,S')$ where $i+1 \leq$ Max_Rhs_Length and $S' = \{j \mid j \epsilon S$ and production j is $A \rightarrow \ldots X_{i+1} \ldots X_1$ and $X_{i+1} = a\}$.

(1) Encoding: Simply look-up the successor of the current state under X_i.

(2) Look-up: Note that if we are in state (j,S) where $k \epsilon S$ and production k is $A \rightarrow \gamma$ and $|\gamma| = j$, then we have recognized production k. Since Simple Precedence grammars are uniquely invertible, there is at most one such k in each state. Further, for each k there is but one state satisfying the above condition (namely the state reached from S_0 via $X_1 \ldots X_j$ where production k is $A \rightarrow X_j \ldots X_1$). We simply number the states which recognize productions 1 to $|P|$ as 1 to $|P|$ and look-up is trivial.

(3) Limitations: The table size required to store the finite automaton's transition function depends greatly on the format used. If we use the simplest (a matrix

of states by characters) and consider a worst case in which each state represents a distinct position in a distinct production then typically (100 productions, average right hand side length=3, $|V|=100$) 30K words are required. This is a fairly large region. Naturally, this could be significantly reduced by a careful packing of data and/or an alternate table format.

3.2 Parallel Strict Precedence Parsing

As we have seen, the Simple Precedence relations are designed to make it very easy to locate the handle in any right sentential form. In converting these techniques to utilize parallelism, we shall be more ambitious. Rather than seeking only the leftmost simple phrase (the handle), we shall seek to identify (and reduce) all simple phrases in a sentential form. As we shall see, the Strict Precedence relations, defined below, do this.

Let λ, α, and ρ be as before. The Strict Precedence relations are:

(1) $X <_S Y$ iff $X\ \alpha\lambda^+ Y$

(2) $X =_S Y$ iff $X\ \alpha\ Y$

(3) $X >_S Y$ iff $X\ \rho^+\alpha\ Y$

(4) $X \gtrless_S Y$ iff $X\ \rho^+\alpha\lambda^+\ Y$

As with Simple Precedence, $<_S$, $=_S$, and $>_S$ delimit the beginning, interior, and end of a simple phrase. $\gtrless_S$ delimits the end of one simple phrase and the immediate beginning of another. We now define the Strict Precedence grammars.

Let G be some CFG which has no ε-productions. Then G is a Strict Precedence grammar if

(1) At most one Strict Precedence relation holds between any pair of symbols in V.

(2) The grammar is uniquely invertible.

We can relate Simple and Strict Precedence by the following.

Theorem 3.4 **Let G be a CFG. Then G is Strict Precedence iff G is Simple Precedence and G^{REV} is Simple Precedence where G^{REV} is G with all right hand sides reversed.**

Proof: If: Observe that $<_S \equiv <\!\cdot$, $=_S \equiv \doteq$, and $\cdot\!> \equiv >_S \cup *_S$. Thus if G is Simple Precedence, we need only show $>_S \cap *_S = \emptyset$ to establish Strict Precedence. But G^{REV} being Simple Precedence implies that $<\!\cdot \cap \cdot\!> = \emptyset$ (for G^{REV}). Further, $<\!\cdot$ (for G^{REV}) $=$ $>_S$ (for G) and $\cdot\!>$ (for G^{REV}) $= *_S \cup <_S$. Thus $>_S \cap (*_S \cup <_S) = \emptyset$ which implies that $>_S \cap *_S = \emptyset$.

Only if: That G is Strict Precedence immediately gives us that G is Simple Precedence. Further, since $<\!\cdot$, $\doteq$, and $\cdot\!>$ (for G^{REV}) are $>_S$, $=_S$, and $*_S \cup <_S$ we get that G^{REV} also is Simple Precedence.

$\square$

Further, since it is well known that Simple Precedence grammars are <u>not</u> closed under reversal, we get that Strict Precedence grammars are properly contained in Simple Precedence grammars.

It should be pointed out that some programming languages, when reversed are <u>not</u> deterministic (and thus <u>can't</u> have Simple Precedence grammars). For example, the program fragment (A,B,...,X,Y) in PL/I could either be part of a procedure declaration or part of a procedure call. Since the differentiation is made to the left of the list, this fragment can't be parsed deterministicly if it must be parsed differently in the two contexts (as normally would be the case). This then means that either some contemporary languages will not be parsable via Strict Precedence techniques, or that they will have to be modified to be parsable (for example syntax might be altered or additional syntax checking might be done in the semantics stage).

We also establish some properties of Strict Precedence grammars augmented with endmarkers.

<u>Theorem 3.5</u> Let G be an augmented Strict Precedence grammar. Then

(1) G is unambiguous

(2) Let $\delta = X_1 \ldots X_n \varepsilon SF(G)$. Then some substring of δ, $X_i \ldots X_j$ is a simple phrase iff $(X_{i-1} <_S X_i$ or $X_{i-1} *_S X_i)$, and $X_i =_S X_{i+1}, \ldots, X_{j-1} =_S X_j$, and $(X_j >_S X_{j+1}$ or $X_j *_S X_{j+1})$.

<u>Proof</u>: (1) Follows from Thm 3.1 since G must be Simple Precedence.

(2) Only if: Let $S \to^* X_1 \ldots X_{i-1} A X_{j+1} \ldots X_n$. Then $X_i =_S X_{i+1}, \ldots, X_{j-1} =_S X_j$ by the definition of $=_S$. Further, since X_{i-1} is adjacent to A in a sentential form, we have $X_{i-1} \rho^* \alpha \lambda^* A \lambda X_i$, that is $X_{i-1} \rho^* \alpha \lambda^+ X_i$. But $<_S \cup *_S \equiv \rho^* \alpha \lambda^+$. Also, $X_j \rho A \rho^* \alpha \lambda^* X_{j+1}$, that is $X_j \rho^+ \alpha \lambda^* X_{j+1}$. But $>_S \cup *_S \equiv \rho^+ \alpha \lambda^*$.
If: Let $\delta = X_1 \ldots X_{i-1} X_i \ldots X_j X_{j+1} \ldots X_n \varepsilon SF(G)$ and assume that $X_i \ldots X_j$ is <u>not</u> a simple phrase. Do all possible reductions in δ not involving $X_i \ldots X_j$ to obtain $\ldots Y X_i \ldots X_j Z \ldots$ All simple phrases in this must involve some part of $X_i \ldots X_j$. Take any one simple phrase. Either its left end $\neq X_i$ or its right end $\neq X_j$. Say its left end $\neq X_i$. If it is to the left of X_i then $Y \alpha X_i$ which implies $X_{i-1} \rho^* \alpha X_i$ which implies

$X_{i-1} =_S X_i$ or $X_{i-1} >_S X_i$. But by assumption $X_{i-1} \doteq_S X_i$ or $X_{i-1} <_S X_i$ and the relations are disjoint. If its left end is to the right of X_i, let it be X_k $(i<k\leq j)$. Then $X_{k-1}\rho^*\alpha\lambda^+X_k$ which implies $X_{k-1} <_S X_k$ or $X_{k-1} \doteq_S X_k$, but by assumption $X_{k-1} =_S X_k$. A similar argument holds for the right end. We thus conclude the simple phrase must be $X_i \ldots X_j$ contradicting our assumption.

□

Thus the Strict Precedence relations delimit all the simple phrases in a sentential form. Since our goal is to recognize and reduce in parallel all the simple phrases in a sentential form, it follows that the number of iterations such an approach would require is simply the depth of the derivation tree of the string being parsed. (This is because reducing all simple phrases reduces the depth of the derivation tree by one).

We would then be interested in ways to reduce the depth of derivation trees, to reduce the number of iterations needed to do the parse.

In general, of course, the structure of a grammar can not be tampered with. We shall consider, however, one common construct which can rather easily be replaced.

It is often necessary to generate a list of arbitrary length (for example a list of statements in a program block). Since the set of productions is fixed and finite, such lists are usually generated by the following:

LIST → LIST ELEMENT | ELEMENT or

LIST → ELEMENT LIST | ELEMENT.

This does indeed generate the desired result but it is necessary to generate a tree of depth n to create a list of n elements. As an alternative, we introduce the iterated production, denoted $A \rightarrow B^+$. This iterated production is simply the set of productions $\{A \rightarrow B, A \rightarrow BB, A \rightarrow BBB, \ldots\}$. Clearly, via this device a list of arbitrary length can be created with a corresponding tree of depth one.

We now establish conditions under which the list generator described above can be replaced with an iterated production.

Theorem 3.6 Let G be some Simple (Strict) Precedence grammar. Assume that G either contains two productions of the form $L \rightarrow L\ E|E$ or of the form $L \rightarrow E\ L|E$, and that no other production has L as a left hand side. Let G' be G with $L \rightarrow L\ E$ (or $L \rightarrow E\ L$) removed. Assume that in G':

(i) Neither $E \rightarrow^+ \ldots E$ nor $E \rightarrow^+ E \ldots$

(ii) If $E \rightarrow^* \ldots a$ and $E \rightarrow^* b \ldots$ then no precedence relation exists between a and b.

Let G" be G' with $L \rightarrow E$ replaced by $L \rightarrow E^+$.

Then

(1) $L(G") = L(G)$.

(2) G" is Simple (Strict) Precedence.

Proof: (1) Trivial given that no other production has L as a left hand side.

(2) Clearly G' is Simple (Strict) Precedence, since it is G with a production removed. Changing $L \to E$ to $L \to E^+$ adds only relations induced by two adjacent E's. But (i) insures that for each a and b st $E \to^* \ldots a$ and $E \to^* b \ldots$ only one new relation is induced, and (ii) insures that there are no conflicts with old relations.

□

Naturally, iterated productions can be used in grammars which do not satisfy (i) and (ii) of Thm 3.6. In this case however, it will be necessary to rewrite parts of the grammar to remove any precedence conflicts generated.

The most notable feature regarding the use of iterated productions is that it is possible to generate strings of arbitrary length with derivation trees of bounded depth. In Algol, for example, using iterated productions to generate statement lists, etc., the depth of the derivation tree of an entire program is essentially determined not by the number of statements, but by the maximum depth of the statements. Thus the derivation tree of a program of several hundred statements might have a depth of only 25 or so.

Let $\gamma \to_p \delta$ denote the derivation of δ from γ by replacing all non-terminals in γ in parallel. Let $L^n(G) = \{x \mid S \to_p \gamma_1 \to_p \ldots \to_p x$ in n or less steps and $x \in V_T^+\}$. Rather clearly, $L^n(G)$ is the set of all the strings of $L(G)$ which have a derivation tree of depth n or less.

As we shall see, we can define a simple and efficient

Strict Precedence parser for $L^n(G)$, allowing the use of iterated productions. We shall establish some conventions for the Strict Precedence parser we are about to introduce. Iterated productions will be used for simple phrases of length greater than that of the maximum length right hand side. New, ordinary productions will be used for right hand sides which are generated by an iterated production, but are of length less than or equal to the maximum length right hand side.

The output of the parser will be n sequences, where n is the depth of the derivation tree being parsed. Each sequence will be of the form $(p_1,i_1),(p_2,i_2),\ldots,(p_j,i_j),(p_1',i_1',L_1'),\ldots,(p_k',i_k',L_k')$ where p_r represents the position of some non-terminal replaced by the production i_r and p_s' represents the position of some non-terminal replaced by the iterated production i_s' of length L_s'.

The operation of the algorithm is straightforward. Given a sentential form, the precedence relations between adjacent characters are calculated. Matching pairs of ($<_S$ or $\doteq_S$) and ($>_S$ or $\doteq_S$) delimit simple phrases. For simple phrases of length less than the Max_Rhs_Length, the phrases are encoded via some method (such as Methods A,B, and C of section 3.1). Iterated right hand sides are verified by comparing each character with its neighbor. Given encoded phrases, or the first character of the iterated productions, the appropriate production is looked-up. The corresponding left hand side is found and substituted, a sequence of non-terminal positions and production indices is outputted and the sentential form is compressed (if necessary). The entire process is repeated until the form \$S\$ is reached or no further reductions can be done.

<u>Algorithm 3.7</u> (A parallel Strict Precedence parser)

Input: Some string in V_T^+ delimited by \$'s.

Output: A series of sequences representing the positions of non-terminals and the productions used to re-write them.

[1] (Calculate precedence relations and find simple phrases)

REL←Look_up_rels(INPUT[1,...,|INPUT|-1],INPUT[2,...,
|INPUT|])

REL_POS←(REL$\varepsilon\{<_S,>_S,\ast_S\}$)/(1,...,|REL|)

SP_RT_POS←(BALANCED_REL←REL[REL_POS[1,...,|REL_POS|-1]]
$\varepsilon\{<_S,\ast_S\}$ and REL[REL_POS[2,...,|REL_POS|]]$\varepsilon\{>_S,\ast_S\}$)/
REL_POS[2,...,|REL_POS]

If |SP_RT_POS|=0 then
if INPUT= \$S\$ then stop, signal a correct parse
else stop and signal an error

SP_LEN←SP_RT_POS - (BALANCED_REL)/REL_POS[1,...,|REL_POS|-1]

[2] (Encode simple phrases for ordinary productions)

POS←ORDINARY_RT_POS←(SP_LEN ≤ MAX_RHS_LENG)/SP_RT_POS

ORDINARY_LEN←(SP_LEN ≤ MAX_RHS_LENG)/SP_LEN

ENCODED_SP←|POS| <u>REPEAT</u> NULL_ENCODING

COUNT←1; ACTIVE←1,...,|POS|

```
[3] ENCODED_SP[ACTIVE]←Encode(ENCODED_SP[ACTIVE],
     INPUT[POS[ACTIVE]],COUNT)
    INPUT[POS[ACTIVE]]←' '
    ACTIVE←(ORDINARY_LEN[ACTIVE] ≥ (COUNT←COUNT+1))/ACTIVE
    POS←POS-1
    If |ACTIVE| > 0 then Go to [3]
[4] (Verify iterated right hand sides are valid, that is,
      do not contain two different symbols)
    ITER_RT_POS←(SP_LEN > MAX_RHS_LENG)/SP_RT_POS
    ITER_LEN←(SP_LEN > MAX_RHS_LENG)/SP_LEN
    COMPARE_FIELD←ITER_RT_POS[1],ITER_RT_POS[1]-1,...,
       ITER_RT_POS[1]+2-ITER_LEN[1],ITER_RT_POS[2],...,
       ITER_RT_POS[|ITER_LEN|],ITER_RT_POS[|ITER_LEN|]-1,
       ...,ITER_RT_POS[|ITER_LEN|]+2-ITER_LEN[|ITER_LEN|]
    If or/(INPUT[COMPARE_FIELD]≠INPUT[COMPARE_FIELD-1])
        then signal an error (invalid simple phrase found)
    INPUT[COMPARE_FIELD-1]←' '
[5] (Look-up productions)
    ORDINARY_PROD←Look_up_prod(ENCODED_SP,ORDINARY_LEN)
    ITER_PROD←ITER_PROD_TABLE[INPUT[ITER_RT_POS]]
    If 0∈ORDINARY_PROD or 0∈ITER_PROD
       then signal an error (invalid simple phrase found)
[6] (Substitute left hand sides)
    INPUT[ORDINARY_RT_POS CONCAT ITER_RT_POS] ←
      LHS_TABLE[ORDINARY_PROD CONCAT ITER_PROD]
```

[7] (Output a sequence of non-terminal positions and productions used; then squeeze out blanks)

```
#BLANKS←+\(INPUT=' ')
Output((ORDINARY_RT_POS - #BLANKS[ORDINARY_RT_POS]),
    ORDINARY_PROD)
Output((ITER_RT_POS - #BLANKS[ITER_RT_POS]),ITER_PROD,
    ITER_LEN)
INPUT←(INPUT≠' ')/INPUT
Go to [1]
```

□

To illustrate the algorithm's operation, we consider an example. The following grammar generates nested begin-end blocks with statement prototypes.

BLK → BEGIN STL END STL → ST^{+} ST → S ; | BLK ;

The maximum right hand side length is 3 and, as explained earlier, we add the new productions

STL → ST | ST ST | ST ST ST

Of course, endmarkers are included. We index the BLK production as 1, the 3 STL productions as 2, 3, 4, the two ST productions as 5 and 6 and the iterated STL production as 7. The following Strict Precedence table can be easily verified.

	BLK	BEGIN	END	STL	ST	S	;	$
BLK							$=_S$	$=_S$
BEGIN	$<_S$	$<_S$		$=_S$	$<_S$	$<_S$		
END							$>_S$	$>_S$
STL			$=_S$					
ST	$<_S$	$<_S$	$>_S$		$=_S$	$<_S$		
S							$=_S$	
;	$*_S$	$*_S$	$>_S$		$>_S$	$*_S$		
$	$=_S$	$<_S$						

We trace the main outline of the parse in the following.

Iteration 1: $ BEGIN S ; S ; BEGIN S ; S ; END ; S ; END $
$<_S \ <_S =_S *_S =_S *_S \ <_S =_S *_S =_S >_S \ >_S *_S =_S >_S \ >_S$
Output: ((3,5),(4,5),(6,5),(7,5),(10,5))

Iteration 2: $ BEGIN ST ST BEGIN ST ST END ; ST END $
$<_S \ <_S =_S <_S \ <_S =_S >_S \ >_S >_S >_S >_S$
Output: ((6,3))

Iteration 3: $ BEGIN ST ST BEGIN STL END ; ST END $
$<_S \ <_S =_S <_S \ =_S =_S >_S >_S >_S >_S$
Output: ((5,1))

Iteration 4: $ BEGIN ST ST BLK ; ST END $
$<_S \ <_S =_S <_S =_S >_S >_S >_S$
Output: ((5,6))

Iteration 5: $ BEGIN ST ST ST ST END $
$<_S \ <_S =_S =_S =_S >_S >_S$
Output: ((3,7,4)) (Note the use of an iterated production)

Iteration 6: $ BEGIN STL END $
$\lessdot_S \quad \doteq_S \quad \doteq_S \quad \gtrdot_S$

Output: ((2,1))

Iteration 7: $ BLK $ and this is the goal

Note that the original string can be obtained by starting with $ BLK $ and performing (in reverse order) the encoded reductions. Further, if the number of S's in the two blocks had been larger the parse would still have taken the same number of iterations.

We now establish the correctness and complexity of the algorithm.

Theorem 3.8 Let G be a Strict Precedence grammar. Let Alg 3.7 be run on $x \in V_T^+$ using encoding and look-up functions bounded by O(# of items encoded or look-ed up) in time and space.

Then

(1) If $x \in L^n(G)$ then x will be parsed correctly in space bounded by $O(|x|)$ and in time bounded by $O(n|x|)$.

(2) If $x \notin L(G)$ then an error will be signalled.

Proof: (1) Correctness follows from Thm 3.5. Space is bounded by $O(|x|)$ because a bounded number of vectors, each so bounded are used. Further, each main iteration (steps [1] to [6]) takes $O(|x|)$ given the linearity of the encoding and look-up functions. Each iteration of the algorithm reduces the depth of the derivation tree of the current sentential form by one. Therefore, by the definition of

$L^n(G)$ at most n iterations are needed.

(2) If $x \notin L(G)$ then at some point an incorrect sentential form will be detected via the fact that an indicated simple phrase is not the right hand side of any production, or by the fact that no more simple phrases are indicated and \$ S \$ has not been reached.

□

We note in passing that all three of the encoding/look-up methods of section 3.1 can easily be run, in parallel, in time and space = O(# elements), and thus may be used in Alg 3.7.

3.3 Parallel Total Precedence Parsing

Both Simple and Strict Precedence techniques were designed to realize very specific goals, namely that of always recognizing either the handle or all simple phrases of a sentential form. We can, however, lessen our requirements by demanding that we merely be able to recognize at least one simple phrase in any sentential form. In doing this we expect, of course, to be able to parse a larger class of grammars than the Simple (and thus Strict) Precedence grammars. As we shall see, such expectations will be realized.

The approach we shall take is that of Colmerauer [Co 70] in developing the Total Precedence grammars. Consider the Strict Precedence relations ($<_S$, $>_S$, $=_S$, $*_S$) of the last section. Recall that these were closely related to the Simple Precedence relations ($<\cdot \equiv <_S$, $\doteq \equiv =_S$, $\cdot> \equiv >_S \cup *_S$). In each case we require the relations to be pairwise disjoint. We can, however, weaken this requirement as follows. Assume that $<_S$ and $*_S$ are not disjoint, that is, $X <_S Y$ and $X *_S Y$. This means that when X and Y occur together the beginning of a simple phrase is <u>always</u> indicated and the end of another is <u>sometimes</u> indicated. We can safely deal with this by assuming only the condition that is always true, the beginning of a simple phrase. This amounts to assigning the conflicts to the $<_S$ relation and removing them from the $*_S$ relation. Clearly the same thing can be done with conflicts between

$>_S$ and $\#_S$.

Thus we define the following.

The Parallel Total Precedence relations are:

(1) $\doteq_P \;\equiv\; \doteq_S$

(2) $<_P \;\equiv\; <_S$

(3) $>_P \;\equiv\; >_S$

(4) $\#_P \;\equiv\; \#_S \cap \overline{>}_S \cap \overline{<}_S$

As noted above, conflicts are removed from $\#_S$ to yield $\#_P$.

The Total Precedence relations are:*

(1) $\doteq_T \;\equiv\; \doteq_P$

(2) $<_T \;\equiv\; <_P$

(3) $>_T \;\equiv\; >_P \cup \#_P$

Let G be some CFG which has no ε productions. Then G is a Parallel Total Precedence (Total Precedence) grammar if

(1) At most one Parallel Total Precedence (Total Precedence) relation holds between any pair of symbols in V.

(2) The grammar is uniquely invertible.

We establish the following.

Theorem 3.9 (1) The class of Parallel Total Precedence (PTP) grammars $\equiv$ the class of Total Precedence (TP) grammars.

(2) The class of Simple Precedence grammars is properly contained in both the PTP and TP grammars.

*As suggested in [Co 70], the parameter μ has been set to $\emptyset$.

Proof: (1) PTP => TP is obvious. To get TP => PTP note that the disjointness of all pairs but $>_P$ and $\doteq_P$ are obvious. But $>_P \cap \doteq_P = >_S \cap (\doteq_S \cap \bar{>}_S \cap \bar{<}_S) = \emptyset$ since $>_S \cap \bar{>}_S = \emptyset$.

(2) Recall the Simple Precedence relations can be written as $(<_S, =_S, >_S \cup \doteq_S)$ and the TP relations can be written as $(<_S, =_S, >_S \cup (\doteq_S \cap \bar{>}_S \cap \bar{<}_S))$. The final relation can be simplified to $(>_S \cup \doteq_S) \cap (>_S \cup \bar{<}_S) = (>_S \cup \doteq_S)$ since $(>_S \cup \doteq_S) \cap <_S = \emptyset$ for Simple Precedence. Proper containment is obtained by observing that the following grammar is TP but not Simple Precedence:

$S \to A\bar{B} \mid aaB \quad A \to a \quad B \to b \quad \bar{B} \to bb.$ □

Theorem 3.10 Let G be an augmented PTP grammar.

Then

(1) G is unambiguous.

(2) Let $\gamma = X_1 \ldots X_n \in SF(G)$. Let $X_i \ldots X_j$ be some substring of γ, st $(X_{i-1} <_P X_i$ or $X_{i-1} \doteq_P X_i)$, $X_i =_P X_{i+1}, \ldots, X_{j-1} =_P X_j$, $(X_j >_P X_{j+1}$ or $X_j \doteq_P X_{j+1})$.

Then

(i) $X_i \ldots X_j$ is a simple phrase.

(ii) $X_i \ldots X_j$ always exists.

Proof: (1) By Thm 3.9 G is a TP grammar. But by Thm 4.11(7) G must be TMSP and bt Thm 4.8(3) all augmented TMSP grammars are unambiguous.

(2i) Just as in Thm 3.5 we argue that if $X_i \ldots X_j$ isn't a simple phrase then a precedence conflict must exist.

(2ii) Using TP relations, Colmerauer shows every sentential form has some simple phrase delimited by $<_T$, $(\doteq_T)^*$, $>_T$ and since $>_T \equiv (>_P \cup \ast_P)$ we immediately get (2ii). □

We now consider the problem of parsing PTP grammars. Our goal will be to produce a linear parser in an environment where not all simple phrases are discernable. This means that, unlike Alg 3.7, we cannot scan the entire input on every iteration. If we did, to parse an input where only a constant number of phrases were reducible, we would require $O(|x|)$ iterations and an $O(|x|^2)$ worst case would result. If we reconsider Alg 3.7 we will note that the only two reasons we needed to scan the entire input were to calculate precedence relations (to find simple phrases) and to "shrink" the input after right hand sides had been replaced. The latter can be removed by maintaining the input in a linked list structure. The former can be dealt with by observing that, if at a given iteration, all recognizable simple phrases have been reduced, then any simple phrases recognizable during the next iteration must have been made so by some new precedence relation involving a left hand side just substituted. In fact, let $X_1 \ldots X_i A X_j \ldots X_n \rightarrow X_1 \ldots X_i X_{i+1} \ldots X_{j-1} X_j \ldots X_n$. After reducing $X_{i+1} \ldots X_{j-1}$ to A, at most 3 new simple phrases can be made recognizable:

(C1) We had $X_i <_P X_{i+1}$ and $X_i >_P A$ or $X_i \ast_P A$.

(C2) $A >_P X_j$ or $A \ast_P X_j$.

(C3) $X_{j-1} >_P X_j$ and $A =_P X_j$ or $A <_P X_j$ or $A \ast_P X_j$.

This simply means, at most, a simple phrase to the left of A, to the right of A, and A itself may become recognizable.

The basic idea of the parser becomes very simple. We start at all $>_p$'s and $*_p$'s. We scan left, character by character, encoding as we scan, until we hit a $<_p$ or a $*_p$ or a $>_p$. If we hit a $<_p$ or a $*_p$ then we have found a simple phrase. If a $>_p$ is hit, we stop the scan and save the information we have (encoding, starting(rightmost) position, etc.). Later, an application of (C3) will restart the scan. Simple phrases are looked-up, corresponding left hand sides are substituted, and new scans are started ((C1) or (C2)) and old scans are continued (C3). This approach requires that we be able to encode arbitrarily long simple phrases since we have made no distinction between ordinary and iterated productions. Each of the three encoding/look-up methods of section 3.1 can be extended to handle iterated productions as follows:

(A) Numeric encoding: If COUNT >Max_Rhs_Length then we look-up the encoding of $C^{\text{Max\_Rhs\_Length}}$ (where C is the next character to be encoded). If the value is the same as the current encoding, we return this same encoding, otherwise an error is signalled. Thus we verify that we have seen all C's.

(B) Bit vector: If COUNT>Max_Rhs_Length, we use V(Max_Rhs_Length,C) in the encoding process. When we look-up the encoding, we use a mask corresponding to productions of length Max_Rhs_Length.

(C) Finite automaton: For each state S=(Max_Rhs_Length,{i}) where i indexes the production $A \to C \ldots C = \delta$ for some A and $|\delta|$=Max_Rhs_Length, we add the transition $S \xrightarrow{C} S$. Thus we stay in S as long as we keep scanning C's.

In each of the above, if we do a look-up and obtain i where production i is $A \to \delta = C \ldots C$ and $|\delta|$=Max_Rhs_Length, then we have recognized the production $A \to C^+$, with a length of at least the maximum right hand side length.

Finally, we consider the data structures used. The input is stored in CHAR, which is a doubly linked list. Left and right links are NEXT_CHAR and PREV_CHAR. Precedence relations are stored in REL which consists of the sub-fields GT and LT. That is REL contains two single bit fields. Thus $\doteq_p$= 1 1, $>_p$= 1 0, etc. As characters are scanned (as potential simple phrases) a DESCRIPTOR is maintained. This consists of the starting(rightmost) position (START_POS), the number of characters scanned (COUNT), and an encoding of the characters scanned (ENCODING). A DESCRIPTOR may be stored, between iterations, at RHS_DESCRIPTOR.

<u>Algorithm 3.11</u> (A Parallel Total Precedence recognizer)

Input: A string x of the form \$x'\$ where $x' \in V_T^+$, stored in CHAR.

Output: An indication of whether or not $x' \in L(G)$.

```
[1] (Initialization)
   NEXT_CHAR[1,...,|x|]←0,...,|x|-1
   PREV_CHAR[1,...,|x|]←2,...,|x|,0
   REL[2,...,|x|]←Look_up_rels(CHAR[1,...,|x|-1],
                                 CHAR[2,...,|x|])
   RHS.COUNT[1,...,|x|]←0
   CURRENT_POS←POS←GT[2,...,|x|]/(2,...,|x|)
   ACTIVE←1,...,|POS|
   DESCRIPTOR←|POS| REPEAT NULL_DESCRIPTOR
   START_POS←NEXT_CHAR[POS]
[2] (First test for <p or ≐p)
   ACTIVE←(~LT[POS] or COUNT[ACTIVE]=0)/ACTIVE
[3] (Scan and encode the next character)
   If |ACTIVE| = 0
     then Go to [6]
   NEXT←NEXT_CHAR[POS←CURRENT_POS[ACTIVE]]
   ENCODING[ACTIVE]←Encode(ENCODING[ACTIVE], CHAR[NEXT],
                          COUNT[ACTIVE])
   COUNT[ACTIVE]←COUNT[ACTIVE] + 1
   POS←CURRENT_POS[ACTIVE]←NEXT
[4] (Check for <p or ≐p; if found, we have a simple phrase)
   ACTIVE←(~LT[POS])/ACTIVE
[5] (Check for >p; if found, save DESCRIPTOR for a later
     restart of the scan)
   SAVE←(GT[POS←CURRENT_POS[ACTIVE]])/ACTIVE
   RHS_DESCRIPTOR[POS[SAVE]]←DESCRIPTOR[SAVE]
```

```
   ACTIVE←(~GT[POS])/ACTIVE
   Go to [3]
[6] (Look-up the left hand sides of simple phrases)
   If |SIMPLE_PHRASES←(LT[CURRENT_POS])/(1,...,|CURRENT_POS|)|
      = 0 then signal an error and stop
   LEFTMOST_POS←CURRENT_POS[SIMPLE_PHRASES]
   RIGHTMOST_POS←START_POS[SIMPLE_PHRASES]
   If 0∈LHS←Look_up_lhs(ENCODING[SIMPLE_PHRASES], COUNT[
     SIMPLE_PHRASES]) then signal an error and stop
[7] (Plug in the left hand sides and reset markers)
   CHAR[RIGHTMOST_POS]←LHS
   NEXT_CHAR[RIGHTMOST_POS]←NEXT_CHAR[LEFTMOST_POS]
   PREV_CHAR[NEXT_CHAR[LEFTMOST_POS]]←RIGHTMOST_POS
   RHS.COUNT[RIGHTMOST_POS]←0
[8] (Calculate new precedence relations)
   LEFT_RELS←Look_up_rels(CHAR[NEXT_CHAR[RIGHTMOST_POS]],LHS)
   RIGHT_RELS←Look_up_rels(LHS,CHAR[PREV_CHAR[RIGHTMOST_POS]])
[9] (Find new start positions for the next scan via condi-
     tions C3, C1, and C2 respectively)
   NEWSCAN1←(RHS.COUNT[PREV_CHAR[RIGHTMOST_POS]]≠ 0 and
     RIGHT_RELS ≠ >p)/(PREV_CHAR[RIGHTMOST_POS])
   NEWSCAN2←RIGHT_GT/PREV_CHAR[RIGHTMOST_POS]
   NEWSCAN3←(~GT[LEFTMOST_POS] and LEFT_GT)/RIGHTMOST_POS
[10] (Plug in the new precedence relations)
   REL[RIGHTMOST_POS]←LEFT_RELS
   REL[PREV_CHAR[RIGHTMOST_POS]]←RIGHT_RELS
```

[11] (Set up the next scan)

POS←CURRENT_POS←NEWSCAN1 CONCAT NEWSCAN2 CONCAT NEWSCAN3

If |POS| = 0

then if CHAR = \$S\$ then signal a correct parse; stop

else signal an error; stop

DESCRIPTOR←(RHS_DESCRIPTOR[NEWSCAN1] CONCAT

((|NEWSCAN2|+|NEWSCAN3|) REPEAT NULL_DESCRIPTOR))

ACTIVE←1,...,|POS|

START_POS[1+|NEWSCAN1|,...,|POS|]←

NEXT_CHAR[NEWSCAN2 CONCAT NEWSCAN3]

Go to [2]

□

A few comments are in order. First, the algorithm is more compact (and efficient) than it may, at first glance, appear. This is because on such parallel computers as the CDC Star-100, each line can be realized in a few, or even a single instruction. Second, we presented a recognizer to simplify the presentation of the algorithm. It could be extended to become a parser as follows. First, observe that the subtree derived from any non-terminal can be represented as $(p,L_1,\ldots,L_n)$ where p is the production used to rewrite the non-terminal and L_1 to L_n are the representations of the subtrees rooted by the non-terminals of production p. Thus given:

S → a S b | c S d S | e, the derivation tree of acaebdcedeb can be represented as (1,(2,(1,(3)),(2,(3),(3)))).

Now assume that with every non-terminal is stored a pointer to its subtree's representation. As a simple phrase is scanned (and encoded) the pointers of the non-terminals encountered can be linked together. Finally, when a reduction is made the index of the production used is prefixed to the list and a pointer to it is stored along with the left hand side.

As an example consider Alg 3.11 and the following PTP (but <u>not</u> Simple Precedence) grammar which generates $\{[\,][^i]^i \mid i \geq 1\}^+$. $L \rightarrow B^+$ $B \rightarrow [\,]C$ $C \rightarrow [C] \mid [\,]$

	\$	L	B	C	[	]
\$		$=_P$	$<_P$		$<_P$	
L	$=_P$					
B	$>_P$		$=_P$		$<_P$	
C	$>_P$		$>_P$		$*_P$	$=_P$
[				$=_P$	$<_P$	$=_P$
]	$>_P$		$>_P$	$=_P$	$<_P$	$>_P$

In the following an underscore will denote a character already scanned but not yet reduced.

Iteration 1: $\$ \;[\;]\;[\;[\;[\;]\;]\;]\;[\;]\;[\;]\;[\;]\;[\;[\;]\;]\;\$$
$<_P\ =_P\ <_P\ <_P\ <_P\ =_P\ >_P\ >_P\ <_P\ =_P\ <_P\ =_P\ <_P\ =_P\ <_P\ <_P\ =_P\ >_P\ >_P$

Iteration 2: $\$ \;[\;]\;[\;[\;C\;\underline{]}\;]\;[\;]\;[\;]\;[\;]\;[\;C\;\underline{]}\;\$$
$<_P\ =_P\ <_P\ <_P\ =_P\ =_P\ >_P\ <_P\ =_P\ <_P\ =_P\ <_P\ =_P\ <_P\ =_P\ =_P\ >_P$

Iteration 3: $\$ \;[\;]\;[\;C\;]\;[\;]\;[\;]\;[\;]\;C\;\$$
$<_P\ =_P\ <_P\ =_P\ =_P\ <_P\ =_P\ <_P\ =_P\ <_P\ =_P\ =_P\ >_P$

Iteration 4: $\$ <_P [=_P] <_P [=_P C =_P] <_P [=_P] <_P [=_P] >_P B >_P \$$

Iteration 5: $\$ <_P [=_P] <_P [=_P C =_P] <_P [=_P] =_P C >_P \underline{B} >_P \$$

Iteration 6: $\$ <_P [=_P] <_P [=_P C =_P] >_P B =_P \underline{B} >_P \$$

Iteration 7: $\$ <_P [=_P] =_P C >_P \underline{B} =_P \underline{B} >_P \$$

Iteration 8: $\$ <_P B =_P \underline{B} =_P \underline{B} >_P \$$

Iteration 9: $\$ =_P L =_P \$$

Finally, we prove correctness and linearity.

Theorem 3.12 Let G be a PTP grammar. Let Alg 3.11 be run on $x = \$x'\$$ where $x' \in V_T^+$ using encoding and look-up functions which are bounded by O(# of items encoded or looked-up) in time and space.

Then

(1) If $x' \in L(G)$ then x' will be recognized correctly in time and space bounded by $O(|x|)$.

(2) If $x' \notin L(G)$ then an error will be signalled.

Proof: (1) We first prove correctness. By Thm 3.10, every sequence of characters delimited by ($<_P$ or $\leq_P$) $(=_P)^*$ ($>_P$ or $\geq_P$) is a simple phrase and every sentential form has at least one simple phrase so delimited. But Alg 3.11 reduces at each iteration every phrase so delimited. To see this, note that it certainly does so during the first iteration since we scan from every $>_P$ or $\geq_P$. Phrases which become recognizable during later iterations must have become so because of a reduction. But there are only 3 ways a simple phrase can be made recognizable as a result of a reduction and these are conditions C1, C2, and C3 (noted above). Each of these conditions is provided for in the algorithm. Thus during each iteration the algorithm reduces (correctly) at least one simple phrase and finally the goal (S) must be reached and a correct parse signalled.

Linearity in space is easily shown for a bounded number of data structures, all bounded in length by $O(|x|)$ are used.

Finally, we note the well known fact that both the total number of reductions performed and the total number of characters scanned (input and non-terminals) during a parse are $O(|x|)$. We now charge execution time as follows. Step [1] and the first execution of [2] are bounded by $O(|x|)$ and occur once. Each iteration of the scan loop (Steps [3] to [5]) is bounded by O(# of characters scanned). All but the first line of the reduce sequence (Steps [6] to [11]) is bounded by O(# of simple phrases detected). The first line of [6] is bounded by $O(|x|)$ the first time it is executed and by O(# of simple phrases detected in the previous iteration) in subsequent executions. We now use Lemma 3.13 (see below) to conclude each character and reduction is charged exactly once, and since (as noted above) the total number of characters and reductions is $O(|x|)$, we get an $O(|x|)$ time bound for the entire algorithm.

(2) If $x' \notin L(G)$ then at some point an incorrect sentential form will be detected, either because an indicated simple phrase is not the right hand side of any production or because no more simple phrases are indicated, but \$S\$ has not been reached.

□

We also note that the modifications mentioned above to convert Alg 3.11 to a parser can easily be implemented without affecting the linearity established in Thm 3.12.

<u>Lemma 3.13</u> Let Alg 3.11 be run on $x=\$x'\$$ where $x' \varepsilon V_T^+$. Then every character occuring during the reduction of x is scanned but once. Further, during any iteration, a given reduction is detected and performed only once.

<u>Proof</u>: May be found in the appendix.

□

4 SIMPLE MIXED STRATEGY PRECEDENCE TECHNIQUES

As we have seen, Simple Precedence techniques make the detection of simple phrases particularly straightforward. The price paid for this simplicity is the fact that it is often a difficult task to produce a Simple Precedence grammar for a given Context Free language.

It is natural then to search for generalizations of Simple Precedence which have "nice" handle detection properties. One such generalization is the Weak Precedence technique of Ichbiah and Morse [IM 70]. In this method the >. of Simple Precedence is again used to detect the right end of the handle. The handle is then the longest sequence of characters that is the right hand side of some production. Unfortunately, this method does not generalize to allow the detection of many simple phrases in parallel. To see why consider the following grammar:

$S \rightarrow DA \mid DBa \qquad D \rightarrow d \qquad B \rightarrow b \qquad A \rightarrow a$

Now consider the sentential form $\$ \; d \underset{>.}{} \; b \underset{>.}{} \; a \underset{>.}{} \; \$$.

Using the above method to detect all simple phrases would reduce the a to A, an error. The difficulty, of course, is that the correct simple phrase DBa was not detected because the d and b had not yet been reduced.

An extension of Weak Precedence is the Simple Mixed Strategy Precedence technique of Aho, Denning and Ullman [AD 72]. Handles are determined as in Weak Precedence but right hand sides need not be uniquely invertible. A single character (called a look-behind character) to the immediate left of the handle is used to determine the appropriate reduction. This method can be extended to allow the concurrent reduction of many simple phrases by using the look-behind character in all reductions. As was the case for Simple Precedence, we may define Strict Mixed Strategy Precedence and Total Mixed Strategy Precedence by either strengthening or weakening the Simple Mixed Strategy Precedence requirements.

4.1 Simple Mixed Strategy Precedence

As noted above, we can extend the class of Simple Precedence grammars by retaining the >. relation to delimit the right end of the handle in sentential forms, but discarding the use of <. and =. to delimit its left end and interior. Further, the concept of a look-behind character allows us to drop the requirement that grammars be uniquely invertible.

Following Aho et al, define the Simple Mixed Strategy Precedence (SMSP) relations:

(1) $>_M \equiv >. \equiv \rho^+\alpha\lambda^*$

(2) $\leq_M \equiv <. \cup =. \equiv \alpha\lambda^*$

The set of look-behind characters for a given nonterminal is then defined:

$L(A) = \{X \mid X \leq_M A\}$.

Let G be some CFG which has no ε-productions. Then G is a SMSP grammar if:

(1) The SMSP relations $>_M$ and $\leq_M$ are disjoint, that is $>_M \cap \leq_M = \emptyset$.

(2) If $A \to \beta' X\beta$ and $B \to \beta$ are productions then $X \notin L(B)$.

(3) If $A \to \beta$ and $B \to \beta$ are productions then $A \neq B$ implies that $L(A) \cap L(B) = \emptyset$.

It is well known (see for example Aho and Ullman [AU 72]) that if G is an augmented SMSP grammar then:

(1) If $\$S\$ \xrightarrow{*}_{rt} \gamma = \gamma_1 \gamma_2 = Y_1 \ldots Y_n Z_1 \ldots Z_m$ where $Y_1 \lessdot_M Y_2 \lessdot_M Y_3 \lessdot_M \ldots \lessdot_M Y_n$ and $Y_n \gtrdot_M Z_1$, then the unique handle of γ is $Y_i \ldots Y_n$ where $Y_i \ldots Y_n$ is the longest suffix of γ_1 such that there exists some production $A \rightarrow Y_i \ldots Y_n$ for some non-terminal A.

(2) The leftmost reduction of any right sentential form is unique.

(3) G is unambiguous.

An alternate method of finding handle of a right sentential form is given by the following.

<u>Theorem 4.1</u> Assume that the conditions of (1) above hold. Then the unique handle of γ is $Y_j \ldots Y_n$ where there exists some production $B \rightarrow Y_j \ldots Y_n$ and $Y_{j-1} \in L(B)$.

<u>Proof</u>: From (1) we know Y_n is the rightmost character of the (unique) handle. Assume $\$S\$ \xrightarrow{*}_{rt} Y_1 \ldots Y_{k-1} C \gamma_2 \rightarrow Y_1 \ldots Y_{k-1} Y_k \ldots Y_n \gamma_2$.

If $k>j$ then $C \rightarrow Y_k \ldots Y_{j-1} Y_j \ldots Y_n$ and $B \rightarrow Y_j \ldots Y_n$ and $Y_{j-1} \in L(B)$ a violation of cond (2) of the SMSP definition. Likewise, if $k<j$ then $Y_{k-1} \alpha C$ which implies that $Y_{k-1} \in L(C)$, again violating cond (2) of the SMSP definition. If $k=j$ then $Y_{k-1} = Y_{j-1} \in L(B)$ as expected

□

We note that (1) is usually used to find the handle in SMSP parsers. We shall, however, use the equivalent method of Thm 4.1 in the parser presented below.

<u>Algorithm 4.2</u> (An SMSP parser)

Input: A string of the form \$x'\$ stored in the vector INPUT.

Output: A sequence of production indices representing the sequence of reductions performed during the parse.

[1] (Initialization)

LIMIT←|INPUT| ; POINTER←2

Stack INPUT[1]=\$ onto the empty push down stack.

[2] (Done with parse?)

If POINTER>LIMIT
  then if the stack contains \$S\$
    then signal a correct parse and stop
    else signal an error and stop.

[3] (Scan the next character)

If the top stack character $\leq_M$ INPUT[POINTER]
  then push INPUT[POINTER] onto the stack
    POINTER←POINTER+1 ; Go to [2]
  else if the top stack character $>_M$ INPUT[POINTER]
    then HANDLE←ε
    else signal an error and stop.

[4] (Find the correct reduction)

If the stack is empty
  then signal an error and stop

else Handle←the top stack character CONCAT HANDLE

Pop off the top stack character

If there exists some production k=A→HANDLE and

the top stack character ε L(A)

then push A onto the stack; output k;

Go to [2]

else Go to [4]

□

A few comments about Alg 4.2 are in order. First, this algorithm can parse CFG's extended by the use of the iterated productions of Chapter 3. To see this, we will consider typical means whereby a handle can be recognized in step [4] of the algorithm. In particular, the three encoding and look-up methods of Chapter 3 (A, B, C) will be re-examined. It will be seen that each, modified to deal with both iterated productions and look-behind characters, will be usable.

(A) Numeric encoding

As before, strings of length ≤ Max_Rhs_length are encoded as integers base $1+|V|$. Strings of greater length are the right hand side of some iterated production or invalid. Thus to encode them we merely verify that they are contained in X^+ where $A \to X^+$ is an iterated production in the grammar. This is done just as suggested in Chapter 3. During look-up, we first encode the look-behind character, then do a binary search in a table of encodings of the strings $Y\gamma$ where γ is the right hand side of some production $A \to \gamma$ and $Y \varepsilon L(A)$. As noted in (2) above, this suffices to recognize a handle and determine the correct reduction. We note that:

(a) The table size needed has been increased from size $|P|$ to $|L(\text{Left\_hand\_side}(1))| + \ldots + |L(\text{Left\_hand\_side}(|P|))|$.

(b) Since a look-behind character must be encoded, the maximum length right hand side which can be encoded is decreased by 1.

(B) <u>Bit vector</u>

As before, the encoding of $Y\gamma$ (where $|\gamma| = i$), given the encoding of γ is obtained by anding together this encoding and $V(1+i, Y)$ if $1+i \leq$ Max_Rhs_Length and by using V(Max_Rhs_Length,Y) otherwise. To decode, we and on, as before, $M(i)$ (where i = length of the encoded string). Additionally, we and on a bit string $LB(Y')$ where Y' is the look-behind character. $|LB(Y')| = |P|$, and $LB_i(Y') = 1$ iff production i is $A \to \gamma$ for some A and $Y' \varepsilon L(A)$. Thus M isolates productions of the right length and LB verifies that the look-behind character is correct. By conditions (2) and (3) of the SMSP definition, at most 1 bit is on after applying M and LB. Note that additional space is required to store LB. The space required is $|P| * |V|$ bits. (Typically 10000 bits ≈ 166 60 bit words).

(C) Finite automaton

The definition of states and transitions is the same as in Chapter 3 except as follows:

(1) The transitions to be added to recognize iterated productions are:

Let S=(Max_Rhs_Length,T). If $i \in T$ and the i-th production is an iterated production, then $S \xrightarrow{X} S'$ where S'= (Max_Rhs_Length,T') and $T'=\{i \mid i \in T \text{ and } P_i = A_i \rightarrow X^+\}$. Further, if $S' \neq S$ then $S' \xrightarrow{X} S'$.

(2) Transitions are added to new final states as follows: Let S=(j,T) and assume there exists $k \in T$ such that $P_k = A \rightarrow \gamma$ for some γ and $|\gamma|=j$. Then $S \xrightarrow{Y} (1+\text{Max\_Rhs\_Length},\{k\}) = S'$ and this is a final state, where $Y \in L(A)$.

Thus S recognizes the right hand side γ and then S' verifies the look-behind character. Note that conditions (2) and (3) of the SMSP definition insure that each S has at most one such k and that the automaton is deterministic. We can number the S' states from 1 to $|P|$ (that is, $S'=(1+\text{Max\_Rhs\_Length},\{i\})$ is the i-th state) to make recognition easy. Note further that $|P|$ new states have been added to deal with the look-behind characters. However since they have no transitions from them, they need not increase the region needed to store the transition function.

Finally, we can easily verify that each of the above three methods can encode a character and then test if the encoding is a handle in constant time. Thus a handle of size s can be recognized in O(s) time.

We can now state the following.

Theorem 4.3 Let G be a SMSP grammar possibly augmented with iterated productions. Let Alg 4.2 be run with encoding and look-up functions which add a character to an encoding and test whether the new encoding is a handle in constant time. Assume further that Alg 4.2 is run on $x = \$x'\$$ where $x' \epsilon V_T^+$. Then

(1) If $x' \epsilon L(G)$ then x' will be parsed correctly in time and space bounded by $O(|x|)$.

(2) If $x' \not\epsilon L(G)$ then an error will be signalled.

Proof: (1) Correctness follows from Thm 4.1.

Clearly, space is $O(|x|)$. Note that each input character is scanned once, and that each character occuring in the parse is stacked once, and there are $O(|x|)$ of these. During each iteration of steps [2] and [3] a character is scanned or popped from the stack. During each iteration of step [4] a character is popped. Since each step can be done in constant time, the $O(|x|)$ bound is immediate.

(2) If $x' \not\epsilon L(G)$ then at some point an incorrect handle will be indicated or no reductions at all will be indicated, but \$S\$ will not have been reached. Either case is covered.

□

4.2 Parallel Strict Mixed Startegy Precedence

The Simple Mixed Strategy Precedence technique of the previous section made it easy to recognize and reduce the handle of a given right sentential form. We now strengthen the SMSP requirements to obtain a class of grammars in which all simple phrases can be detected and reduced in the SMSP manner. We shall call this class the Strict Mixed Strategy Precedence (STMSP) grammars. As we shall see, these grammars are well suited to parsing in parallel environments. Further, they are properly contained in the SMSP class and they properly contain the Strict Precedence grammars.

Let $\overline{L}(A) = \{x \mid x \; \beta \; A\}$. Recall $\beta \equiv \rho^* \alpha \lambda^*$, denotes the "by" relation, that is, X β Y implies that X can occur by Y in some sentential form.

Let G be some CFG which has no ε-productions. Then G is a <u>Strict</u> <u>Mixed</u> <u>Strategy</u> <u>Precedence</u> grammar if:

(1) The SMSP relations $>_M$ and $\leq_M$ are disjoint, that is $>_M \cap \leq_M = \emptyset$.

(2) If $A \to \gamma' X \gamma$ and $B \to \gamma$ are productions then $Y \notin \overline{L}(B)$ where $X \to^* \ldots Y$.

(3) If $A \to \gamma$ and $B \to \gamma$ are productions then $A \neq B$ implies that $\overline{L}(A) \cap \overline{L}(B) = \emptyset$.

<u>Theorem 4.4</u> Let G be an augmented STMSP grammar and assume $\$S\$ \to^+ \gamma = X_1 \ldots X_{i-1} X_i \ldots X_j X_{j+1} \ldots X_n$.

Then

(1) $X_i \ldots X_j$ is a simple phrase iff

(a) $X_j >_M X_{j+1}$

(b) There exists some production $A \to X_i \ldots X_j$ and $X_{i-1} \varepsilon \overline{L}(A)$.

(2) $X_i \ldots X_j$ may be correctly reduced to A.

Proof: (1) Only if: Assume $\$S\$ \to^+ X_1 \ldots X_{i-1} A X_{j+1} \ldots X_n \to \gamma$. Then $A \beta X_{j+1}$ and $X_j \rho A \rho^* \alpha \lambda^* X_{j+1} \Rightarrow X_j \rho^+ \alpha \lambda^* X_{j+1} \Rightarrow X_j >_M X_{j+1}$. Further, $X_{i-1} \beta A \Rightarrow X_{i-1} \varepsilon \overline{L}(A)$.

If: Assume that $X_i \ldots X_j$ isn't a simple phrase. Perform all possible reductions on γ not involving $X_i \ldots X_j$, obtaining $\ldots Z X_i \ldots X_j Y \ldots$ Take any simple phrase in this. Part of $X_i \ldots X_j$ must be included. Consider the right end of the simple phrase. If it is to the right of X_j then $X_j \alpha Y \Rightarrow X_j \alpha \lambda^* X_{j+1} \Rightarrow X_j \underline{<}_M X_{j+1}$. But $X_j >_M X_{j+1}$ by assumption. If it is to the left of X_j, it must be at X_k where $i \underline{<} k < j$. But then $X_k \rho^+ \alpha X_{k+1} \Rightarrow X_k >_M X_{k+1}$. However, by assumption, $A \to \ldots X_k X_{k+1} \ldots \Rightarrow X_k \underline{<}_M X_{k+1}$. Thus the right end of the simple phrase must be at X_j. Now consider the left end of the simple phrase. If it is to the right of X_i, say at X_k, $i < k \underline{<} j$, then if $X_k \ldots X_j$ is reduced to (say) C, we have $X_{k-1} \beta C \Rightarrow X_{k-1} \varepsilon \overline{L}(C)$, a violation of cond (2) of the STMSP definition. If the left end is to the left of X_i, then we have $C \to \ldots Z X_i \ldots X_j$ where $Z \to^* X_{i-1}$, which again is a violation of cond (2). Thus the left end must be at X_i, implying $X_i \ldots X_j$ is a simple phrase, a contradiction of our earlier assumption.

(2) We know from (1) that $X_i \ldots X_j$ is a simple phrase. Assume that the correct reduction is to reduce $X_i \ldots X_j$ to

$B \neq A$. Then $X_{i-1}\beta B \Rightarrow X_{i-1}\varepsilon\overline{L}(B)$. But this contradicts cond (3) of the STMSP definition.

$\square$

Theorem 4.5 Let G be a STMSP grammar. Then G is unambiguous.

Proof: Follows immediately from the fact that G must be SMSP (see Thm 4.11(4)).

$\square$

We are now ready to consider a parallel parser for STMSP grammars. As we shall see, it will be quite similar to the Parallel Strict Precedence parser of Chapter 3 (Alg 3.7). The output of the parser will be n sequences, where n is the depth of the derivation tree of the string being parsed. Each sequence will be of the form $(p_1,i_1,L_1),\ldots,(p_m,i_m,L_m)$ where p_j is the position of some non-terminal replaced by production i_j, where the length of i_j's right hand side is L_j (L_j is needed in the case of iterated productions).

The algorithm starts a scan at each $>_M$. The scans proceed left, character by character. After each new character is scanned and encoded, the encoding, along with a look-behind character, is tested. If a simple phrase has been found, the scan stops. If the next relation is a $>_M$, the scan stops (since a $>_M$ cannot occur within a right hand side). Otherwise, another character is scanned, the encoding is tested, etc. After all scans have terminated, the characters in the simple phrases are blanked out, the correct left hand sides are substituted, and the string is compressed. Then, if \$S\$ has not been reached, another set of scans is initiated.

<u>Algorithm 4.6</u> (A parallel STMSP parser)

Input: Some string in V_T^+, delimited by \$'s.

Output: A series of sequences representing the positions of non-terminals and the productions used to rewrite them.

```
[1] (Calculate precedence relations and find >M's)
    REL←Look_up_rels(INPUT[1,...,|INPUT|-1],
           INPUT[2,...,|INPUT|])
    POS←GT_POS←(REL = >M)/(1,...,|REL|)
    If |GT_POS| = 0
       then if INPUT = $S$ then signal a correct parse, stop
                            else signal an error, stop
[2] (Set up scans)
    ENCODED_SP←|POS| REPEAT NULL_ENCODING
    RHS_COUNT←PRODUCTION#←|POS| REPEAT 0
    POS_SCANNED←SCAN_INDEX←NULL_VECTOR
    ACTIVE←1,...,|POS|; COUNT←1
[3] (Scan the characters and test for simple phrases)
    ENCODED_SP[ACTIVE]←Encode(ENCODED_SP[ACTIVE],
          INPUT[POS[ACTIVE]],COUNT)
    RHS_COUNT[ACTIVE]←COUNT
    PRODUCTION#[ACTIVE]←TEMP←Look_up_rhs(ENCODED_SP[ACTIVE],
           COUNT,INPUT[POS[ACTIVE]-1])
```

```
  POS_SCANNED←POS_SCANNED CONCAT POS[ACTIVE]
  SCAN_INDEX←SCAN_INDEX CONCAT ACTIVE
  ACTIVE←(TEMP = 0)/ACTIVE
  POS[ACTIVE]←POS[ACTIVE] - 1
  If or/(POS[ACTIVE] ≤ 0) then signal an error and stop
  ACTIVE←(REL[POS[ACTIVE]] ≠ >_H)/ACTIVE
  COUNT←COUNT + 1
  If |ACTIVE| = 0 then Go to [3]
[4] (Blank out characters in simple phrases)
  INPUT[(PRODUCTION#[SCAN_INDEX] ≠ 0)/POS_SCANNED]←' '
[5] (Plug in left hand sides, output reduction information,
      and squeeze out blanks)
  SIMPLE_PHRASE_POS←(PRODUCTION# ≠ 0)/GT_POS
  INPUT[SIMPLE_PHRASE_POS]←LHS_TABLE[
        (PRODUCTION# ≠ 0)/PRODUCTION#]
  #BLANKS←+\(INPUT = ' ')
  Output( (SIMPLE_PHRASE_POS - #BLANKS[SIMPLE_PHRASE_POS]),
          ( (PRODUCTION# ≠ 0)/PRODUCTION#),
          ( (PRODUCTION# ≠ 0)/RHS_COUNT)   )
  INPUT←(INPUT ≠ ' ')/INPUT
  Go to [1]
```

□

To illustrate the algorithm's operation, we consider the following well-known arithmetic expression grammar:

$E \rightarrow E + T \mid T \quad T \rightarrow T * P \mid P \quad P \rightarrow (E) \mid ID$

The following tables can easily be verified.

	E	T	P
$\overline{L}$:	{(,$}	{+,(,$}	{+,*,(,$}

	E	T	P	+	*	ID	(	)	$
E				$\leq_M$				$\leq_M$	$\leq_M$
T				$>_M$	$\leq_M$			$>_M$	$>_M$
P				$>_M$	$>_M$			$>_M$	$>_M$
+		$\leq_M$	$\leq_M$			$\leq_M$	$\leq_M$		
*			$\leq_M$			$\leq_M$	$\leq_M$		
ID				$>_M$	$>_M$			$>_M$	$>_M$
(	$\leq_M$	$\leq_M$	$\leq_M$			$\leq_M$	$\leq_M$		
)				$>_M$	$>_M$			$>_M$	$>_M$
$	$\leq_M$	$\leq_M$	$\leq_M$			$\leq_M$	$\leq_M$		

We now consider a parse of $ ID * (ID + ID) + ID * ID $

Iteration 1: $\$ \leq_M ID >_M * \leq_M (\leq_M ID >_M + \leq_M ID >_M) >_M + \leq_M ID >_M * \leq_M ID >_M \$$

Output: (2,6,1),(5,6,1),(7,6,1),(10,6,1),(12,6,1)

Iteration 2: $\$ \leq_M P >_M * \leq_M (\leq_M P >_M + \leq_M P >_M) >_M + \leq_M P >_M * \leq_M P >_M \$$

Output: (2,4,1),(5,4,1),(7,4,1),(10,4,1)

Iteration 3: $\$ \leq_M T \leq_M * \leq_M (\leq_M T >_M + \leq_M T >_M) >_M + \leq_M T \leq_M * \leq_M P >_M \$$

Output: (5,2,1),(10,3,3)

Iteration 4: $\$ \leq_M T \leq_M * \leq_M (\leq_M E \leq_M + \leq_M T >_M) >_M + \leq_M T >_M \$$

Output: (5,3,1)

Iteration 5: $\$ \leq_M T \leq_M * \leq_M (\leq_M E \leq_M) >_M + \leq_M T >_M \$$

Output: (3,5,3)

Iteration 6: $\$ \leq_M T \leq_M * \leq_M P >_M + \leq_M T >_M \$$

Output: (2,3,3)

Iteration 7: $\$ \leq_M T >_M + \leq_M T >_M \$$

Output: (2,2,1)

Iteration 8: $\$ \leq_M E \leq_M + \leq_M T >_M \$$

Output: (2,1,3)

Iteration 9: $\$ \quad E \quad \$$

We now establish the correctness and the time and space complexity of the algorithm.

<u>Theorem 4.7</u> Let G be an augmented STMSP grammar, possibly extended with iterated productions. Let Alg 4.6 be run on $x=\$x'\$$, where $x' \in V_T^+$, with an encoding function that adds a new character to n encodings in O(n) time, and with a look-up function that determines whether any of n encodings are simple phrases in O(n) time.

Then

(1) If $x' \in L^n(G)$ then x' will be parsed correctly in $O(|x|)$ space and $O(n|x|)$ time.

(2) If $x' \notin L(G)$ then an error will be signalled.

<u>Proof</u>: (1) Correctness follows from Thm 4.4.

Observe that each scan starts from a $>_M$ and terminates when a simple phrase has been found or a $>_M$ is hit. Thus during each iteration, a given character is scanned at most once. The length of POS_SCANNED and of SCAN_INDEX is equal to the total number of characters scanned and thus is $O(|x|)$. The rest of the vectors are clearly bounded by the length of INPUT and thus the $O(|x|)$ bound is immediate. Now consider one main iteration of the algorithm (Steps [1],[2],$[3]^+$,[4],[5]). Steps [1] and [2] can be executed in $O(|x|)$ time. Consider one execution of step [3]. A number of characters equal to |ACTIVE| are scanned. Further, the entire step can be executed in O(|ACTIVE|) time. But since each character is scanned at most once, all executions of [3] are done in $O(|x|)$ time. Finally, each instruction in steps [4] and [5] can be done in time proportional to $|x|$ or (# of scans done) or (# of simple phrases found). Thus these steps can be done in $O(|x|)$ time, and therefore one main iteration takes $O(|x|)$ time. But by Thm 4.4, during each iteration all simple phrases are found and reduced. Therefore by the definition of $L^n(G)$ at most n main iterations are needed, and thus an $O(n|x|)$ time bound is obtained.

(2) If $x' \notin L(G)$ then at some point either no $>_M$'s will be found and \$S\$ will not have been reached, or the leftmost $>_M$ will not delimit a simple phrase, in which case the scan will stop when the left end of the input is reached. But both of these cases are checked for and an error is

signalled.

$\square$

We note that all three of the encoding/look-up functions described in section 4.1 satisfy the assumptions of Thm 4.7.

4.3 Parallel Total Mixed Strategy Precedence

The STMSP grammars of the previous section were obtained by strengthening the requirements of SMSP grammars to enable all simple phrases to be recognized and reduced. We may, however, weaken the SMSP requirements to obtain a class of grammars in which at least one simple phrase is always recognizable (and reducable). We shall call this class the Total Mixed Strategy Precedence (TMSP) grammars.

First, we shall reconsider the SMSP relations $>_M$ and $\leq_M$. We had

$$>_M \equiv \rho^+\alpha\lambda^* \equiv >_S \cup \doteq_S$$

$$\leq_M \equiv \alpha\lambda^* \equiv \doteq_S \cup <_S .$$

As was the case in Total Precedence (section 3.3), we shall allow certain conflicts between $>_M$ and $\leq_M$ to occur. In particular, if $>_M \cap \leq_M \neq \emptyset$ because $\doteq_S \cap <_S \neq \emptyset$, we shall remove the conflicts by assigning them to $\leq_M$. That is, if $X \doteq_S Y$ and $X <_S Y$, we shall assume that Y always begins some simple phrase, but ignore the fact that sometimes X also ends another simple phrase. Thus we have made it easier for a grammar to fit the "no conflicts" requirement by allowing the possibility that some simple phrases will not be reduced as soon as they might be.

We now define the TMSP relations:

(1) $\leq_{TM} \equiv \alpha\lambda^* \equiv \leq_M$

(2) $>_{TM} \equiv \rho^+\alpha\lambda^* - \alpha\lambda^* \equiv >_M - \leq_M$

As we shall see below, we shall require that $(\lessdot_M \cap >_S)=\emptyset=(\doteq_S \cap >_M)$. It should be noted that for both SMSP and STMSP it was the case that if $A \to X\ldots$ then $Y \lessdot_M A \Rightarrow Y \lessdot_M X$ and $Y >_M A \Rightarrow Y >_M X$. This, however, is not the case for TMSP grammars. For example, we may have $A \to a\ldots$, $A \to c\ldots$, $b >_{TM} A$ and $b >_{TM} a$, but $b \lessdot_{TM} c$. This means that look-behind sets may no longer be calculated as functions of non-terminals; rather, they must be functions of individual productions. The reason for this is that sometimes, as will be discussed below, it will be necessary to remove (say) b from the look-behind set associated with $A \to a\ldots$ if $b >_{TM} a$. On the other hand, we will require that b be in the look-behind set associated with $A \to c\ldots$, if $b \lessdot_{TM} c$. Clearly this could not be done if the old method were used. In what follows we shall assume that the i-th production is $A_i \to Z_i^1 \ldots Z_i^{n_i}$. Let $LT(i)=\{x \mid x \beta A_i \text{ and } x \lessdot_{TM} Z_i^1\}$, $GT(i)=\{x \mid x \beta A_i \text{ and } x >_{TM} Z_i^1\}$.

Clearly, $LT(i) \cap GT(i) = \emptyset$ by the definition of $\lessdot_{TM}$ and $>_{TM}$. Recall that the look-behind set is used to determine if a sequence of characters which is the right hand side of some production is a simple phrase, and, if so, what reduction should be done. Consider, for example, $\ldots CZ_i^1 \ldots Z_i^{n_i}\ldots$ We may distinguish two cases:

(1) $C \lessdot_{TM} Z_i^1$. In this case C must be sufficient to decide whether or not $Z_i^1 \ldots Z_i^{n_i}$ should be reduced, and, if so, to what. If C is not sufficient, the grammar must be rejected.

(2) $C >_{TM} Z_i^1$. If C is sufficient to make the decision, then all is well. If it is not, we not disqualify the grammar. Rather, since the $>_{TM}$ implies that there are reductions to be done to the left, we may postpone the decision until these reductions are done.

Thus the look-behind set for production i will consist of LT(i) plus some part of GT(i). The elements removed from GT(i) will be just those which are insufficient to make the decision of whether or not to reduce (and what to reduce to). We may consider two cases in which the look-behind character is insufficient:

(1) $C \to \ldots X\gamma$, $D \to \gamma$, $X \to^{+} \ldots Y$, $Y \beta D$. That is, having seen γ, we encounter a Y. Since $Y \beta D$ it may be correct to reduce γ to D. However, since $X \to^{+} \ldots Y$, it may also be correct to ignore the reduction, reduce $\ldots Y$ to X, and finally recognize $C \to \ldots X\gamma$. Thus the reduction must be postponed.

(2) $C \to a\gamma$, $D \to a\gamma$, $b \beta C$, $b \beta D$, $b >_{TM} a$. That is, having seen $a\gamma$ we encounter a b. Since b occurs by both C and D, we must postpone the reduction.

The following two sets contain those elements of GT(i) which satisfy the above two conditions.

$\nu(i) = \{Y \mid C \to \ldots X Z_i^1 \ldots Z_i^{n_i}$ and $X \to^{+} \ldots Y$ and $Y \beta A_i\}$

$\varepsilon(i) = \{Y \mid A_i \to a\gamma$ and there exists $B \neq A_i$ st $B \to a\gamma$ and $Y \beta A_i$ and $Y \beta B$ and $Y >_{TM} a\}$.

Note that if $x \in \nu(i)$ then $x >_{TM} Z_i^1$ since $x \rho^{+} \alpha Z_i^1$. We now define the set of look-behind characters.

$\hat{L}(i) = LT(i) \cup (GT(i) - (\nu(i) \cup \varepsilon(i)))$

The class of TMSP grammars can then be defined:

Let G be some CFG which has no ε-productions. Then G is a <u>Total</u> <u>Mixed</u> <u>Strategy</u> <u>Precedence</u> grammar if:

(1) $\rho^+\alpha \cap \alpha\lambda^* = \rho^+\alpha\lambda^+ \cap \alpha = \emptyset$

(2) If $B \to \delta' X\delta$ and $C \to \delta$, then $\sim X\beta C$

(3) If $A_i \to \gamma$ and $A_j \to \gamma$ then $A_i \neq A_j$ implies $LT(i) \cap LT(j) = \emptyset$

We note that (1) simply requires that:

(a) $>_S \cap \leq_{TM} = \emptyset$

(b) $=_S \cap \ast_S = \emptyset$

Further, if (3) is satisfied then $LT(i) \cap LT(j) = \emptyset$ implies that $\hat{L}(i) \cap \hat{L}(j) = \emptyset$ since any conflicts in GT appeared in ε and were thus removed from $\hat{L}$.

<u>Theorem 4.8</u> Let G be an augmented TMSP grammar, and assume that $\$S\$ \to^+ \gamma = X_1 \ldots X_{i-1} X_i \ldots X_j X_{j+1} \ldots X_n$.

Then

(1) If $X_j >_{TM} X_{j+1}$ and $A_k \to X_i \ldots X_j$ and $X_{i-1} \varepsilon \hat{L}(k)$ then

(a) $X_i \ldots X_j$ is a simple phrase.

(b) It is correct to reduce $X_i \ldots X_j$ to A_k.

(2) $X_i \ldots X_j$ st the conditions of (1) hold always exists.

(3) G is unambiguous.

<u>Proof</u>: (1a) Assume that $X_i \ldots X_j$ is not a simple phrase. Do all possible reductions in γ, not involving $X_i \ldots X_j$ to

obtain $\ldots ZX_i \ldots X_j Y \ldots$ It is the case that $X_j >_{TM} Y$ (if $X_j \leq_{TM} Y$ then $X_j \alpha\lambda^* Y \Rightarrow X_j \alpha\lambda^+ X_{j+1} \Rightarrow X_j \leq_{TM} X_{j+1}$, a contradiction). Take any simple phrase; some part of $X_i \ldots X_j$ must be involved. Consider the right end of this simple phrase. If it is to the right of X_j, then $X_j \alpha Y \Rightarrow X_j \leq_{TM} Y$, a contradiction. If it is to the left of X_j, it must be at X_m, $i \leq m < j$. But then $X_m \rho^+ \alpha X_{m+1}$, and also, by assumption, $X_m \alpha X_{m+1}$. However, by (1) of the TMSP definition, $\rho^+\alpha \cap (\alpha \cup \alpha\lambda^+) = \emptyset$. Thus the right end must be at X_j. Now consider the left end. If it is the the right of X_i at X_m, $i < m \leq j$, then if $X_m \ldots X_j$ is reduced to (say) C we have an immediate violation of cond (2) of the TMSP definition. If the left end is to the left of X_i then we have $C \rightarrow \ldots ZX_i \ldots X_j$. If $Z = X_{i-1}$ then since $X_{i-1} \varepsilon \hat{L}(k) \Rightarrow X_{i-1} \beta A_k$, we again have a violation of cond (2) of the TMSP definition. If $Z \rightarrow^+ \ldots X_{i-1}$ then $X_{i-1} \rho^+ \alpha X_i \Rightarrow X_{i-1} >_{TM} X_i$ and $X_{i-1} \varepsilon \nu(k)$ (by definition) $\Rightarrow X_{i-1} \not\varepsilon\ \hat{L}(k)$ a contradiction. Thus the left end must be at $X_i \Rightarrow X_i \ldots X_j$ is a simple phrase, again a contradiction.

(1b) Assume $X_i \ldots X_j$ should be reduced to $A_m \neq A_k$. If $X_{i-1} \leq_{TM} X_i$ then $X_{i-1} \varepsilon LT(m)$ and $X_{i-1} \varepsilon \hat{L}(k) \Rightarrow X_{i-1} \varepsilon LT(k)$, a violation of cond (3) of the TMSP definition. If $X_{i-1} >_{TM} X_i$ then $X_{i-1} \varepsilon\ \varepsilon(k)$ (by definition) $\Rightarrow X_{i-1} \not\varepsilon \hat{L}(k)$, a contradiction.

(2) There is always at least one $>_{TM}$ in γ (at the right \$). Consider the leftmost $>_{TM}$ in γ. Let it be at X_{q+1}. Since $X_q >_{TM} X_{q+1} \Rightarrow X_q \rho^+ \alpha\lambda^* X_{q+1}$, X_q must be the right end of some

phrase. Let this phrase be $X_p \ldots X_q$. But $X_p \ldots X_q$ must be a simple phrase. Otherwise, reduce it to $Y_{p'}, \ldots Y_q$, where $Y_q =$ $X_q, \ldots, Y_{s+1} = X_{s+1}$, but $Y_s \to^{+} \ldots X_s$ for $p \leq p' \leq s < q$. Then $X_s \rho^{+} \alpha X_{s+1} \Rightarrow X_s >_{TM} X_{s+1}$, contradicting our assumption that the leftmost $>_{TM}$ was at X_{q+1}. Assume that is correct to reduce $X_p \ldots X_q$ to A_t. Then $X_{q-1} \varepsilon LT(t) \Rightarrow X_{q-1} \varepsilon \hat{L}(t)$ and the requirements of (1) are met.

(3) Assume that G is ambiguous and that $y \varepsilon L(G)$ can be derived in (at least) two different ways. Perform reductions on \$y\$ as indicated by (1) above, until the reduction indicated would not be performed according to one (or both) of the two derivations. But then we use the arguments of the proof of (1a) and (1b) to show that the reduction indicated <u>must</u> eventually be performed.

□

We may now consider the problem of parsing TMSP grammars in parallel. Since Thm 4.8(2) only guarantees one simple phrase per iteration, we cannot scan the entire input during every iteration. If we did, an $O(|x|^2)$ worst case time bound would result. In Alg 4.6, the input string was scanned to calculate precedence relations and to "shrink" the input after reductions. The former can be eliminated by noting that new relations will appear only at those positions where reductions have been done. The latter can be dealt with by maintaining the input in a linked-list data structure.

Further, assume that, at a given iteration, all recognizable reductions (according to Thm 4.8(1)) have been made. Then any reductions that become recognizable during the next iteration must be the result of the substitutions just made. In fact, if $\ldots X_i X_{i+1} \ldots X_{j-1} X_j \ldots$ is reduced to $\ldots X_i A X_j \ldots$, then at most three new simple phrases can be recognized and reduced:

(D1) We had $X_i \lessdot_{TM} X_{i+1}$ but $X_i >_{TM} A$.

(D2) $A >_{TM} X_j$.

(D3) We had $X_{j-1} >_{TM} X_j \lessdot_{TM} X_{j+1} \ldots \lessdot_{TM} X_{k-1} >_{TM} X_k$.

Either $A \varepsilon \hat{L}(t)$ and $A_t \to X_j \ldots X_k$ or $A \lessdot_{TM} X_j$.

That is, we may recognize a simple phrase to the immediate left of A (cond(D1)), or recognize a simple phrase starting at A (cond(D2)), or recognize a simple phrase to the right of A because A is a valid look-behind character, or A may be part of a simple phrase starting to its right (cond (D3)).

The basic idea of the algorithm is then very simple. We start at all $>_{TM}$'s and scan left, character by character. At each step, we test the current sequence of characters already scanned, and a look-behind character, to see if a reduction is indicated. If it is, this scan is stopped. Otherwise, if a $>_{TM}$ has been hit, we save the information we have (encoded characters, starting (rightmost) position, etc.) and stop. Later, an application of condition (D3) will restart the scan. If a $>_{TM}$ has not been hit we continue the scan. When all scans have stopped, all indicated reductions are done, new precedence relations are calculated, and new scans are started (via D1 or D2) and old scans are continued (via D3). The above is iterated until \$S\$ is reached or an error is detected.

It should be noted that the algorithm that follows parallels closely the parallel Total Precedence recognizer (Alg 3.11) of section 3.3. In fact, the data structures used here are exactly those of Alg 3.11. (A short explanation of them may be found immediately before Alg 3.11).

Algorithm 4.9 (A parallel TMSP recognizer)

Input: A string x=\$x'\$, where $x' \varepsilon V_T^+$, stored in the vector CHAR.

Output: An indication of whether or not $x' \varepsilon L(G)$.

[1] (Initialization)

```
NEXT_CHAR[1,...,|x|]←0,...,|x|-1
PREV_CHAR[1,...,|x|]←2,...,|x|,0
REL[2,...,|x|]←Look_up_rels(CHAR[1,...,|x|-1],
                  CHAR[2,...,|x|])
RHS.COUNT[1,...,|x|]←0
CURRENT_POS←POS←(REL[2,...,|x|] = >TM)/(2,...,|x|)
ACTIVE←1,...,|POS|
DESCRIPTOR←|POS| REPEAT NULL_DESCRIPTOR
START_POS←NEXT_CHAR[POS]
PRODUCTION#←|POS| REPEAT 0
```

```
[2] (Test if restarted scans (via (D3)) can be reduced)
   PRODUCTION#←Look_up_rhs(ENCODING,COUNT,
                    CHAR[NEXT_CHAR[POS]])
   ACTIVE←(PRODUCTION# = 0 or COUNT = 0)/ACTIVE
[3] (Scan and encode the next character)
   If |ACTIVE| = 0 then Go to [6]
   NEXT←NEXT_CHAR[POS←CURRENT_POS[ACTIVE]]
   If or/(NEXT = 0) then signal an error and stop
   ENCODING[ACTIVE]←Encode(ENCODING[ACTIVE],CHAR[NEXT],
                         COUNT[ACTIVE])
   COUNT[ACTIVE]←COUNT[ACTIVE] + 1
   POS←CURRENT_POS[ACTIVE]←NEXT
[4] (Test if we have found any right hand sides)
   PRODUCTION#[ACTIVE]←Look_up_rhs(ENCODING[ACTIVE],
                           COUNT[ACTIVE],CHAR[NEXT_CHAR[POS]])
   ACTIVE←(PRODUCTION#[ACTIVE] = 0)/ACTIVE
[5] (Check for >TM, if found save DESCRIPTOR for a later
      restart (via D3) of the scan)
   SAVE←(>TM= REL[POS←CURRENT_POS[ACTIVE]])/ACTIVE
   PHS_DESCRIPTOR[POS[SAVE]]←DESCRIPTOR[SAVE]
   ACTIVE←(>TM≠ REL[POS])/ACTIVE
   Go to [3]
[6] (Look-up the left hand sides of the simple phrases)
   SIMPLE_PHRASES←(PRODUCTION# ≠ 0)/(1,...,|PRODUCTION#|)
   LEFTMOST_POS←CURRENT_POS[SIMPLE_PHRASES]
   RIGHTMOST_POS←START_POS[SIMPLE_PHRASES]
```

```
   LHS←LHS_TABLE[PRODUCTION#[SIMPLE_PHRASES]]
[7] (Plug in left hand sides and reset markers)
   CHAR[RIGHTMOST_POS]←LHS
   NEXT_CHAR[RIGHTMOST_POS]←NEXT_CHAR[LEFTMOST_POS]
   PREV_CHAR[NEXT_CHAR[LEFTMOST_POS]]←RIGHTMOST_POS
   RHS.COUNT[RIGHTMOST_POS]←0
[8] (Calculate new precedence relations)
   LEFT_RELS←Look_up_rels(CHAR[NEXT_CHAR[RIGHTMOST_POS]],LHS)
   RIGHT_RELS←Look_up_rels(LHS,CHAR[PREV_CHAR[RIGHTMOST_POS]])
[9] (Find the starting positions for the next scan via
      conditions D3, D1, and D2 respectively)
   NEWSCAN1←(RHS.COUNT[PREV_CHAR[RIGHTMOST_POS]] ≠ 0)/
              (PREV_CHAR[RIGHTMOST_POS])
   NEWSCAN2←(RIGHT_RELS = >TM)/PREV_CHAR[RIGHTMOST_POS]
   NEWSCAN3←(REL[LEFTMOST_POS] = ≤TM or
              LEFT_RELS = >TM)/RIGHTMOST_POS
[10] (Plug in the new precedence relations)
   REL[RIGHTMOST_POS]←LEFT_RELS
   REL[PREV_CHAR[RIGHTMOST_POS]]←RIGHT_RELS
[11] (Set up the next scan iteration)
   POS←CURRENT_POS←NEWSCAN1 CONCAT NEWSCAN2 CONCAT NEWSCAN3
   If |POS| = 0
       then if CHAR = $S$ then signal a correct input, stop
                           else signal an error and stop
   DESCRIPTOR←(RHS_DESCRIPTOR[NEWSCAN1] CONCAT
      ((|NEWSCAN2| + |NEWSCAN3|) REPEAT NULL_DESCRIPTOR)
```

```
ACTIVE←1,...,|POS|
START_POS[1+|NEWSCAN1|,...,|POS|]←
     NEXT_CHAR[NEWSCAN2 CONCAT NEWSCAN3]
PRODUCTIONS←|POS| REPEAT 0
Go to [2]
```

□

The following comments are in order:

(1) The encoding/look-up functions outlined in section 4.1 can all be employed in the above algorithm after very slight modification (recall look-behind sets are now functions of productions rather than non-terminals).

(2) The above algorithm may be converted from a recognizer to a parser in the same manner as Alg 3.11. That is, the subtree derived from a non-terminal is represented as $(p,L_1,\ldots,L_n)$ where p is the production used to rewrite the non-terminal, and L_1 to L_n are the representations of the subtrees rooted by the non-terminals of p. A pointer to this list may be stored with the non-terminal, and these pointers can be linked together as the non-terminals associated with them are scanned.

As an example, we consider a grammar very similar to that used to illustrate Alg 3.11. The language generated is $\{[]\ [^i]^i \mid i \geq 1\}^+$.

$L \rightarrow E^+ \qquad E \rightarrow BC \qquad C \rightarrow [C] \mid [] \qquad B \rightarrow []$

	\$	E	B	C	[	]
\$		$\leq_{TM}$	$\leq_{TM}$		$\leq_{TM}$	
E	$\leq_{TM}$	$\leq_{TM}$	$\leq_{TM}$		$\leq_{TM}$	
B				$\leq_{TM}$	$\leq_{TM}$	
C	$>_{TM}$	$>_{TM}$	$>_{TM}$		$>_{TM}$	$\leq_{TM}$
[				$\leq_{TM}$	$\leq_{TM}$	$\leq_{TM}$
]	$>_{TM}$	$>_{TM}$	$>_{TM}$	$>_{TM}$	$>_{TM}$	$>_{TM}$

Production #	1	2	3	4	5
LT	{\$}	{\$,E}	{B,[}	{B,[}	{\$,E}
GT	∅	∅	{C,]}	{]}	{C,]}
ν	∅	∅	∅	∅	∅
ε	∅	∅	∅	{]}	{]}
$\hat{L}$	{\$}	{\$,E}	{B,C,[,]}	{B,]}	{\$,E,C}

An underscore will denote a character already scanned, but not yet reduced.

Iteration 1:

\$	[	]	[	[	[	]	]	]	[	]
$\leq_{TM}$	$\leq_{TM}$	$>_{TM}$	$\leq_{TM}$	$\leq_{TM}$	$\leq_{TM}$	$>_{TM}$	$>_{TM}$	$>_{TM}$	$\leq_{TM}$	$>_{TM}$

[	]	[	]	[	[	]	]	\$
$\leq_{TM}$	$>_{TM}$	$\leq_{TM}$	$>_{TM}$	$\leq_{TM}$	$\leq_{TM}$	$>_{TM}$	$>_{TM}$	

Iteration 2:

\$	B	[	[	C	$\underline{]}$	$\underline{]}$	$\underline{[}$	$\underline{]}$	$\underline{[}$	$\underline{]}$
$\leq_{TM}$	$\leq_{TM}$	$\leq_{TM}$	$\leq_{TM}$	$\leq_{TM}$	$>_{TM}$	$>_{TM}$	$\leq_{TM}$	$>_{TM}$	$\leq_{TM}$	$>_{TM}$

$\underline{[}$	$\underline{]}$	[	C	$\underline{]}$	\$
$\leq_{TM}$	$>_{TM}$	$\leq_{TM}$	$\leq_{TM}$	$>_{TM}$	

Iteration 3: $\$ \le_{TM} B \le_{TM} [\le_{TM} C \le_{TM} \underline{]} >_{TM} \underline{[} \le_{TM} \underline{]} >_{TM} \underline{[} \le_{TM} \underline{]} >_{TM} \underline{[} \le_{TM} \underline{]} >_{TM} C >_{TM} \$$

Iteration 4: $\$ \le_{TM} B \le_{TM} C >_{TM} \underline{[} \le_{TM} \underline{]} >_{TM} \underline{[} \le_{TM} \underline{]} >_{TM} \underline{[} \le_{TM} \underline{]} >_{TM} C >_{TM} \$$

Iteration 5: $\$ \le_{TM} E \le_{TM} B \le_{TM} \underline{[} \le_{TM} \underline{]} >_{TM} \underline{[} \le_{TM} \underline{]} >_{TM} C >_{TM} \$$

Iteration 6: $\$ \le_{TM} E \le_{TM} B \le_{TM} C >_{TM} \underline{[} \le_{TM} \underline{]} >_{TM} C >_{TM} \$$

Iteration 7: $\$ \le_{TM} E \le_{TM} E \le_{TM} B \le_{TM} C >_{TM} \$$

Iteration 8: $\$ \le_{TM} E \le_{TM} E \le_{TM} E >_{TM} \$$

Iteration 9: $\$ \quad L \quad \$$

We now establish correctness and linearity.

Theorem 4.10 Let G be an augmented TMSP grammar, possibly extended by the use of iterated productions. Let Alg 4.9 be run on $x=\$x'\$$, where $x' \in V_T^+$, using an encoding function that adds a new character to each of n encodings in O(n) time, and a look-up function that determines whether any of n encoded strings can be reduced in O(n) time.

Then

(1) If $x' \in L(G)$ then x' will be recognized correctly in $O(|x|)$ time and space.

(2) If $x' \notin L(G)$ then an error will be signalled.

Proof: We first state the following lemma:

Let x be as above. Then every character occuring during the reduction of x by Alg 4.9 is scanned but once. Further, during any iteration of Alg 4.9 a given reduction is detected and performed only once.

Proof: Parallels exactly that of Lemma 3.13, using conditions D1, D2, and D3.

(1) We first prove correctness. Thm 4.8(1b) guarantees that in every sentential form at least one reduction will be found using the criteria of Thm 4.8(1a). At each iteration Alg 4.9 reduces all strings satisfying Thm 4.8(1a). To see this, observe that during the first iteration it certainly does, for scans are started from all $>_{TM}$'s and stopped only when a reduction is recognized or a $>_{TM}$ is hit (in which case Thm 4.8(1a) cannot be satisfied since a $>_{TM}$ cannot occur within a right hand side). Reductions which become recognizable during later iterations do so because of a reduction. But there are only three ways a reduction may become recognizable after a reduction is performed and these are conditions D1, D2, and D3. Each of these conditions is provided for in the algorithm and a scan is started if they apply. Thus during each iteration at least one reduction is performed and eventually \$S\$ is reached and success is signalled.

We note the well known result that the total number of characters in the derivation tree of a string x is $O(|x|)$ and that the number of reductions needed to fully reduce x is also $O(|x|)$.

Linearity in space follows from:

(i) A bounded number of data structures, each bounded in length by $O(|x|)$ are used.

(ii) Since each character is scanned only once, the total number of characters encoded at any instant is $O(|x|)$. Thus $O(|x|)$ space sufficies for all encodings.

We charge execution time as follows:

(i) Step [1] is executed once and bounded by $O(|x|)$.

(ii) The first execution of step [2] is bounded by $O(|x|)$ and charged to initialization. Subsequent executions are bounded by $O(n)$ where n equals the number of reductions performed during the last iteration. Execution time is charged to these n reductions.

(iii) Executions of Steps [3], [4], and [5] are bounded by $O(m)$ where m equals the number of characters scanned. Execution time is charged to these m characters.

(iv) The first line of Step [6] is bounded by $O(|x|)$ the first time it is executed and is charged to initialization. Subsequent executions are bounded by $O(n)$ (where n is the same as in (ii)). Execution time is charged to these n reductions.

(v) The rest of Step [6] and Steps [7] to [12] are bounded by $O(p)$ where p equals the number of reductions detected. Execution time is charged to these p reductions.

Since there are $O(|x|)$ characters scanned and $O(|x|)$ reductions, and since each character is scanned once, and each reduction is performed once, we get an $O(|x|)$ time bound.

(2) During each iteration of Alg 4.9 on x one of the following must occur:

(i) No $>_{TM}$'s occur. An error is signalled because \$S\$ cannot have been reached.

(ii) A scan starting at the leftmost $>_{TM}$ "runs off" the left end of the string. An error is signalled.

(iii) A scan starting at the leftmost $>_{TM}$ detects a reduction.

Condition (iii) may occur a number of times, but eventually either (i) or (ii) must occur since $x' \notin L(G)$.

□

It should be noted that the three encoding/look-up functions of section 4.1 satisfy the assumptions of the above theorem.

Further, the extention noted earlier to convert Alg 4.9 to a parser can easily be implemented without affecting the space or time complexity of the algorithm.

4.4 A Comparison of Simple Precedence and SMSP Grammar Classes

We compare the three Simple Precedence grammar classes (Simple Precedence, Strict Precedence, and Total Precedence) with the three SMSP grammar classes (SMSP, STMSP, and TMSP) in the following.

Theorem 4.11 The following relations hold:

$$
\begin{array}{ccccc}
\text{STMSP} & \subsetneq_{(4)} & \text{SMSP} & \subsetneq_{(5)} & \text{TMSP} \\
\cup\!\!\!\neq_{(6)} & & \cup\!\!\!\neq_{(3)} & & \cup\!\!\!\neq_{(7)} \\
\text{STP} & \subsetneq_{(1)} & \text{SP} & \subsetneq_{(2)} & \text{TP}
\end{array}
$$

Further:

(8) TP is incomparable with SMSP.

(9) TP is incomparable with STMSP.

(10) SP is incomparable with STMSP.

Proof: (1) Thm 3.4.

(2) Thm 3.9(2).

(3) Well known. (See, for example, [AU 72]).

(4) Let, for example, SMSP(2) denote the second condition of the SMSP definition.

Containment: (i) SMSP(1) is the same as STMSP(1).

(ii) Clearly, $\bar{L}(A) \supseteq L(A)$. STMSP(2) with $\% \rightarrow^{*} \% \Rightarrow$ SMSP(2).

(iii) STMSP(3) => SMSP(3) since $\overline{L}(A) \supseteq L(A)$.

Proper containment:

G_1= $S \rightarrow AC \mid BD$ $\quad A \rightarrow ab \quad C \rightarrow c \quad B \rightarrow bb \quad D \rightarrow c$

G_1 is SMSP but not STMSP.

(5) Containment:

(i) SMSP(1) is $(\rho^+\alpha\lambda^*) \cap (\alpha\lambda^*) = \emptyset$. But $(\rho^+\alpha\lambda^*) \supseteq (\rho^+\alpha)$ $\Rightarrow \rho^+\alpha \cap \alpha\lambda^* = \emptyset$. Also $(\rho^+\alpha\lambda^*) \supseteq (\rho^+\alpha\lambda^+)$ and $\alpha\lambda^* \supseteq \alpha \Rightarrow$ $\rho^+\alpha\lambda^+ \cap \alpha = \emptyset \Rightarrow$ TMSP(1).

(ii) SMSP(2) $\Rightarrow \sim X \underline{<}_M B \Rightarrow \sim X\alpha\lambda^* B$. If $X >_M B$ then $X\rho^+\alpha\lambda^* B \Rightarrow$ $X\rho^+\alpha\lambda^+ Y_1$ (recall $B \rightarrow \beta = Y_1 \ldots Y_n$) $\Rightarrow X >_M Y_1$. But $X\alpha Y_1 \Rightarrow$ $X \underline{<}_M Y_1 \Rightarrow \sim X\rho^+\alpha\lambda^* B$. Thus $(\sim X\alpha\lambda^* B)$ and $(\sim X\rho^+\alpha\lambda^* B) \Rightarrow$ $\sim X\rho^*\alpha\lambda^* B \Rightarrow \sim X\beta B \Rightarrow$ TMSP(2).

(iii) $>_M \cap \underline{<}_M = \emptyset \Rightarrow L(B) \supseteq LT(i)$ (where $A_i = B$). Thus SMSP(3) => TMSP(3).

Proper containment:

G_2= $S \rightarrow BC \mid bD \quad B \rightarrow b \quad C \rightarrow cd \quad D \rightarrow ce$

G_2 is TMSP but not SMSP.

(6) Containment:

(i) $>_M = >_S \cup \doteq_S$, $\underline{<}_M = <_S \cup \dot{=}_S$. Since $<_S$, $=_S$, $>_S$, and $\doteq_S$ are pairwise disjoint we get $>_M \cap \underline{<}_M = \emptyset \Rightarrow$ STMSP(1).

(ii) Consider STMSP(2). If $Y \epsilon \overline{L}(B)$ then $Y\rho^*\alpha\lambda^* B \Rightarrow$ $Y\rho^*\alpha\lambda^+ Z_1$ (where $B \rightarrow Z_1 \ldots Z_n$) $\Rightarrow (Y <_S Z_1$ or $Y \doteq_S Z_1)$. But $X\alpha Z_1 \Rightarrow Y\rho^*\alpha Z_1 \Rightarrow (Y >_S Z_1$ or $Y =_S Z_1)$. Since the STMSP relations are pairwise disjoint, we get $Y \notin \overline{L}(B) \Rightarrow$ STMSP(2).

(iii) STP(2) => STMSP(3).

Proper containment:

G_3= $S \to aB \mid b\bar{B}$ $B \to b$ $\bar{B} \to b$

G_3 is STMSP but not STP.

(7) Containment:

(i) By TP(1) $>_T \cap <_T \equiv >_T \cap =_T \equiv <_T \cap =_T \equiv \emptyset$. But $>_T \supseteq \rho^+\alpha \Rightarrow$ $\rho^+\alpha \cap (\alpha\lambda^*) = \emptyset$. Further, $(>_T \cup <_T) \cap =_T = \emptyset$. $(>_T \cup <_T) \supset \rho^+\alpha\lambda^+ \Rightarrow \rho^+\alpha\lambda^+ \cap \alpha = \emptyset \Rightarrow$ TMSP(1).

(ii) Consider TMSP(2). $X\beta C \Rightarrow X\rho^*\alpha\lambda^*C \Rightarrow X\rho^*\alpha\lambda^+Z_1$ (where $C \to Z_1 \ldots$) $\Rightarrow$ ($X <_T Z_1$ or $X >_T Z_1$). But $X =_T Z_1$ and $=_T \cap$ $(<_T \cup >_T) = \emptyset \Rightarrow \sim X\beta C \Rightarrow$ TMSP(2).

(iii) TP(3) => TMSP(3).

Proper containment:

G_3 is TMSP but not TP.

(8) G_3 is SMSP but not TP.

G_2 is TP but not SMSP.

(9) G_3 is STMSP but not TP.

G_2 is TP but not STMSP.

(10) G_3 is STMSP but not SP.

G_4= $S \to AC \mid \bar{A}c$ $A \to a$ $\bar{A} \to aa$ $C \to c$

G_4 is SP but not STMSP.

□

5 LR PARSING TECHNIQUES

We now consider the family of LR parsing techniques, including LR(k) [Kn 65], SLR(k), and LALR(k) [De 69], [De 71]. Such techniques are well suited to automatic parser generation; further, many yield compact parsers for a wide variety of context free grammars (including virtually all grammars used to generate programming languages).

First, serial parsing methods are reviewed. It is seen that LR methods can be readily abstracted into a common form. Then we generalize these methods to operate in parallel. It is proposed that a number of Synchronous Parsing Machines (SPM's) be run on an input string, each of which operates in a manner similar to that of an LR parser. Two types of SPM's are then considered. The first, the <u>Basic SPM</u>, is designed to operate as efficiently as possible, and yet is as powerful as any serial LR technique. The second, the <u>Full SPM</u>, is able to parse a broader class of grammars than serial techniques at the expense of additional overhead.

5.1 Serial LR Parsers

We first consider the problem of abstracting LR parsing techniques into a common form. This is done so that algorithms and properties may be developed independent of any particular LR technique.

If S is the start symbol of some CFG, we will augment the grammar by adding a new non-terminal S', and a new production $S' \rightarrow \$S\$$. S' will be the new start symbol of the grammar. Recognition of the new production $(S' \rightarrow \$S\$)$ will successfully terminate the LR parsing algorithms. Further, if a particular LR technique uses a k $(k \geq 0)$ character look-ahead string (see below), then \$ and $\$^{k+1}$ will be used as left and right endmarkers, where \$ is a new symbol added to V_T.

We will consider an LR parsing technique to be defined by (PA,R) where PA is a deterministic finite automaton called the Parsing Automaton, and R is the Reduce function.

PA is used, during the parse, to keep track of the set of reductions which may be applicable. Its states are usually identified with various (finite) sets of so called "LR(k) items" (see [Kn 65]).

The Reduce function, R is defined as

$$R: \hat{S} \times V_T^k \rightarrow \{0,1,\ldots,|P|\}.$$

$\hat{S}$ is the set of states of PA, k is the size of the look-ahead string used by the LR method, and P is the set of productions of the CFG being parsed. Given some state of PA, s, and a look-ahead string y, of length k, R(s,y)=i, $1 \le i \le |P|$ if the i-th production has been recognized, and R(s,y)=0 if no reduction is indicated.

We say an LR parsing technique (PA,R) is <u>valid</u> if

(1) For $\$S\$^{k+1} \rightarrow^{*}_{rt} \alpha A w \rightarrow_{rt} \alpha\beta w = \gamma y w$

(a) If $|y| > 0$ then $s_0 \xrightarrow{\gamma} s'$, where s_0 is PA's start state, and R(s',$First_k$(yw))=0.

(b) $s_0 \xrightarrow{\alpha\beta} s''$ and R(s'',$First_k$(w))=i, where $A \rightarrow \beta$ is the i-th production.

(2a) If R(s,y)=i and the i-th production is $A \rightarrow \beta$, then $s_0 \xrightarrow{\delta} s \Rightarrow \delta = \ldots\beta$.

(2b) If $s_0 \xrightarrow{\delta} s$ then $S'\$^k \rightarrow^{*}_{rt} \alpha A w \rightarrow_{rt} \alpha\beta w$ and $\alpha\beta = \delta\ldots$ Further, for any state s', there exists ν st $s_0 \xrightarrow{\nu} s'$.

Essentially, (1a) and (1b) require that we always be able to distinguish whether or not the right end of the handle has been reached. (2a) requires that if a reduction is signalled, then the corresponding right hand side <u>must</u> just have been scanned. (2b) requires that any sequence of characters which takes us to a PA state be a prefix of some right sentential form which does not include <u>any</u> characters to the right of the handle (that is, a viable prefix). It is also required that every PA state be reachable from the start state.

Not surprisingly, the LR(k), SLR(k), and LALR(k) techniques are all valid, as are most other LR techniques. A few LR techniques, such as Szymanski's non-canonical LR(k,∞) method [Sz 73], however, are not.

We now establish the following property of valid LR parsers.

<u>Theorem 5.1</u> Assume G, some CFG, has a valid LR parser. Then G is unambiguous.

<u>Proof</u>: Assume G is ambiguous. Then

$\$S\$^{k+1} \xrightarrow{*}_{rt} \alpha A w \rightarrow_{rt} \alpha\beta w$ and

$\$S\$^{k+1} \xrightarrow{*}_{rt} \gamma B x \rightarrow_{rt} \gamma\delta x = \alpha\beta w$, but $\alpha A w \neq \gamma B x$.

Assume without loss of generality that $|\alpha\beta| \leq |\gamma\delta|$.

(1) If $|\alpha\beta|=|\gamma\delta|$ then let $A\rightarrow\beta$ be production i and $B\rightarrow\delta$ be production j. $\alpha A w \neq \gamma B x \Rightarrow i \neq j$. Let $s_0 \xrightarrow{\alpha\beta} s'$. We require that $R(s',First_k(w))=i$ and $R(s',First_k(x))=j$. But x=w and R is single valued (by definition).

(2) If $|\alpha\beta|<|\gamma\delta|$ then $\gamma\delta=\alpha\beta z$, $|z|>0$. As in (1) we require $R(s',First_k(w))=i$, but cond (1a) of the valid LR definition requires $R(s',First_k(zx))=0$, and zx=w.

□

We shall assume that the index of $S'\rightarrow\$S\$$ (the production used to augment G) is always $|P|$, that is, it is the last production in P.

A general parsing algorithm for any valid LR parsing method is now considered.

<u>Algorithm 5.2</u> (A parsing algorithm for valid LR parsers)

Input: (1) (PA,R), a valid LR parser for some augmented CFG, G.

(2) $x=\$x'\$^{k+1}$ stored in the vector INPUT, where $x'\varepsilon V_T^*$ and R uses k character look-ahead strings.

Output: A sequence of production indices representing a right parse of x'.

[1] (Initialization)

POINTER←1

Push s_0, PA's start state, onto an empty push down stack

[2] (Scan a character)

Let Top state in stack $\xrightarrow{\text{INPUT[POINTER]}}$ s'

If s' is undefined

then signal an error, and stop

else push s' onto the push down stack

[3] (Reduce?)

Let L=INPUT[POINTER+1,...,POINTER+k]

Let TOP_S = the top state in the push down stack

If R(TOP_S,L) = i ≠ 0

then pop $|\gamma_i|$ states from the stack, where $A_i \rightarrow \gamma_i$ is the i-th production

INPUT[POINTER]←A_i

Output(i)

If i = |P| then signal a correct parse and stop

else POINTER←POINTER + 1

Go to [2] □

Correctness and linearity are established for Alg 5.2.

Theorem 5.3 Let (PA,R) be a valid LR parsing technique for G, some CFG. Let Alg 5.2 be run on $x=\$x'\$^{k+1}$ using (PA,R) and assume $x'\varepsilon V_T^*$.

Then

(1) If $\$x'\$\varepsilon L(G)$, then a right parse will be produced in $O(|x|)$ time and space.

(2) If $\$x'\$\not\varepsilon L(G)$, then an error will be signalled.

Proof: This proof, as well as the statement and proof of Lemma 5.4, may be found in the appendix.

□

Up to this point we have allowed ourselves to consider grammars with ε-productions. In succeeding sections it will be convinient to restrict ourselves to LR techniques for grammars without ε-productions. Thus we shall now briefly consider how to remove ε-productions and how much their removal will cost us.

Algorithm 5.5 (An algorithm to remove ε-productions from CFG's)

Input: $G=(V,V_T,P,S)$

Output: $G'=(V',V_T,P',S)$. $L(G)=L(G')$, but G' contains no ε-productions.

[1] Partition $(V-V_T-\{S\})$ into three disjoint sets A, B, C where

(a) $N \epsilon A$ if $N \to^+ \delta \Rightarrow \delta = \epsilon$.

(b) $N \epsilon B$ if $N \to^+ \nu \Rightarrow \nu \neq \epsilon$.

(c) $N \epsilon C$ if $N \to^+ \omega \neq \epsilon$ <u>and</u> $N \to^+ \epsilon$.

Set $V' = B \cup C \cup V_T \cup \{S\}$

[2] If $S \to^* \epsilon$ is in P, then $S \to \epsilon$ is in P'

[3] If $D \to X_1 \ldots X_k$ is in P, then each production of the form

$D \to Y_1 \ldots Y_k$ is in P' where

(a) If $X_i \epsilon A$ then $Y_i = \epsilon$.

(b) If $X_i \epsilon B \cup V_T$ then $Y_i = X_i$.

(c) If $X_i \epsilon C$ then Y_i may be X_i <u>or</u> ϵ.

(d) In no case may $Y_1 \ldots Y_k = \epsilon$.

$\square$

A proof of the equivalence of G and G' may be found in [HU 69] (pp 62-62). The following relates G and G'.

<u>Theorem 5.6</u> Let G be some CFG and let G' be obtained from G by Alg 5.5.

Then

(1) If $D \to \gamma \ \epsilon \ P'$ can be obtained from both $D \to \gamma' \ \epsilon \ P$ and $D \to \gamma'' \ \epsilon \ P$ for $\gamma' \neq \gamma''$ then G is ambiguous.

(2) If (1) is not true, then any derivation in G' is isomorphic to one in G.

(3) If G' is ambiguous, so is G.

<u>Proof</u>: (1) Follows immediately from the fact that G is reduced and that $\gamma \to^+ x \Rightarrow \gamma' \to^+ x$ and $\gamma'' \to^+ x$.

(2) Every production in P' corresponds uniquely to one in P,

with (perhaps) a number of non-terminals required to derive ε.

(3) If (1) holds, the result is immediate. Otherwise, (2) holds, and again the result follows.

□

A natural question to ask is whether, given that G is parsable by some valid LR technique, is G' (its ε-free equivalent) parsable by the same LR technique? The answer, as we shall see, is sometimes. However, it is also true that every grammar parsable by a valid LR technique has <u>some</u> ε-free equivalent which is SLR(1).

<u>Theorem 5.7</u> Let G be a CFG, and let G' be its ε-free equivalent produced by Alg 5.5.

Then

(1) If G is LR(k) then G' is LR(k).

(2) There exists a G which is both SLR(k) and LALR(k), for some k, but for which G' is neither SLR(k') nor LALR(k') for any k'.

(3) There exists G", an ε-free CFG, st L(G)=L(G") and G" is SLR(1).

<u>Proof</u>: (1) May be found in the appendix.

(2) Let G be: $S' \rightarrow \$S\$$ $\quad S \rightarrow a\bar{B}Caa \mid bBCbb \mid aB\bar{C}bb$

$S \rightarrow b\bar{B}\bar{C}aa \quad B \rightarrow \varepsilon \quad \bar{B} \rightarrow \varepsilon \quad C \rightarrow c \quad \bar{C} \rightarrow c$

G is both SLR(2) and LALR(2).

G': $S' \to \$S\$$ $\quad S \to aCaa \mid bCbb \mid a\bar{C}bb \mid b\bar{C}aa$

$C \to c \qquad \bar{C} \to c$

But G' is neither SLR(k) nor LALR(k) for any k.

(3) Since G is augmented with endmarkers, ε is not in L(G); further, L(G) must be deterministic since Alg 5.2 can clearly be implemented on a deterministic PDA. But every deterministic language not containing ε has an SMSP grammar and every SMSP grammar is SLR(1).

□

5.2 Basic Synchronous Parsing Machines

As we saw in the previous section, the pair (PA,R) which defines a valid LR parsing technique can be used to drive a serial parser (as per Alg 5.2). We can extend this concept to utilize parallelism by using the (PA,R) pair to drive a number of Synchronous Parsing Machines (SPM's). Thus we may envision many SPM's operating along the entire length of an input string, rather than a single parser operating at the far left of the input. In fact, the number of SPM's started will be limited only by the length of the input; their starting positions will be completely arbitrary (although for reasons of efficiency some positions may be better than others). In practice then, to parse an Algol program one might start SPM's at the beginning of each procedure, or each block, or even each statement.

We first consider the modifications needed to make the pair (PA,R) usable in SPM parsing. As noted in section 5.1, we will require that only grammars without ε-productions be considered. This is to simplify the SPM technique; Thm 5.7 assures us that no languages need be disallowed by the restriction. Since parsing need not now proceed in a strictly left to right manner, we must extend R, the reduce function, to include non-terminal look-ahead strings. This is done by extending two of the requirements for valid LR parsers.

(1^*) For $\$S\$^{k+1} \xrightarrow{*} \alpha A\delta \rightarrow \alpha\beta\delta = \gamma\rho\delta$ where α is fully reduced (that is there does not exist α' st $\$S\$^{k+1} \xrightarrow{*} \alpha' \ldots$ and $\alpha' \xrightarrow{+} \alpha$).

(a) If $|\rho|>0$ then $s_0 \xrightarrow{\gamma} s'$ and $R(s', First_k(\rho\delta))=0$.

(b) $s_0 \xrightarrow{\alpha\beta} s''$ and $R(s'', First_k(\delta))=i$ where $A \rightarrow \beta$ is the i-th production.

Note that, as in (1a) and (1b) β is the handle. Now, however, the string to the right of β may be partially (or fully) reduced. Also, if an ε-free grammar G satisfies (1a) and (1b) then it also satisfies (1^*a) and (1^*b). (For if R fails to satisfy (1^*a) and (1^*b) because $R(s,\xi)=0$ and $R(s,\xi)=i\neq 0$ in different contexts, then let $\xi \xrightarrow{+} x\ldots$ where $|\xi|=|x|=k$ and $x\varepsilon V_T^k$. But then $R(s,x)=0$ and $R(s,x)=i\neq 0$ in these same contexts). Conditions (2a) and (2b) are not modified.

Clearly, SLR(k), LALR(k), and LR(k) techniques can easily be modified to yield (PA,R) with R duly extended. Also, Alg 5.2 using an extended reduce function will be able to reduce any sentential form correctly. This is because (1^*a) and (1^*b) still guarantee that the handle will always be detected and reduced correctly.

We are now ready to consider the fundamental extention needed to make SPM parsing possible. Simply put, it is that while serial LR parsers need carry only a single PA state, SPM's, in general, will need to carry a set of PA states. That is, since the state a serial parser is in depends on all the characters it has already scanned, it may not be possible for an SPM, starting in the middle of a string to know what this correct state will be. Thus it will instead carry a set of states, guaranteed to include the correct one. This set may be quite small if, for example, only a few states are possible in a given context. On the other hand, this set may include all the PA states (and thus certainly include the correct one!).

We shall assume that each SPM carries a set of push down stacks, one stack for each state carried. The stacks will be named (uniquely) by their bottommost elements. $(s_1:s_2)$ will denote the push down stack s_1; further s_2 is the top element on the stack. Also, we let $S^A=\{s \mid (s:s')\epsilon S \text{ for some } s'\}$, $S^B=\{s' \mid (s:s')\epsilon S \text{ for some } s\}$. S^A is the set of bottom elements (and thus names) of the stacks in S; S^B is the set of top elements.

As mentioned earlier, we shall investigate two types of SPM's, the Basic SPM, and the Full SPM. In this section the Basic SPM will be considered. As suggested by its name, the Basic SPM method is intended to be the simplest possible parallel extention of serial LR parsing which includes all LR parsable grammars. Thus if G (without ϵ-productions) can be parsed serially by an LR technique (PA,R), we require that it be possible to parse G, in parallel, via a Basic SPM parser using (PA,R) (and R extended as outlined above).

We now re-examine each step of the serial LR parsing algorithm (Alg 5.2), outlining the modifications needed to accomodate the Basic SPM extentions.

1. (Initialization)

Let $N(s,c)=s'$ if $s \xrightarrow{c} s'$; $N(s,c)=\emptyset$ otherwise.

(a) Start one SPM at the far left of the input string. S will contain one stack, with s_0 in it. (This is the equivalent of the serial LR parser).

(b) Optionally, start a number of other SPM's. If an SPM is to be started to the immediate left of some character c, then S will contain $|S_0|$ stacks, each with an element of S_0 in it, where $S_0=\{s \mid N(s,c)\neq\emptyset$ and s is a state of PA$\}$. (That is, the SPM is started with just those states that can validly scan the first character). Note that the choice of c determines the number of states (stacks) the SPM must initially carry.

2. (Scan the next character).

Let d be the character to be scanned. For each stack (s:s') in S:

(a) If $N(s',d)=s''$ then push s" onto the stack named s.

(b) If $N(s',d)=\emptyset$ then delete the stack (s:s') from S. This means the sequence of states stored in the stack has been shown to be invalid.

In general, we can expect that the number of stacks in S will decrease as new characters are scanned, because additional information has been made available.

3. (Reduce?)

Let $S'=S^B=\{s_1,\ldots,s_n\}$ (that is the set of top states of the stacks in S). Assume x is the appropriate k character look-ahead string. Then $R(S',x) = \bigcup_{s\epsilon S'} R(s,x)$.

(a) If $R(S',x)=\{0\}$ then go to the scan step (each state in S' agrees that no reduction is indicated).

(b) If $R(S',x)=\{i\}\neq\{0\}$ then if the stacks in S are deep enough (have enough elements) to pop off $|\gamma_i|$ states and not be empty, then do reduction i. (If the stacks are not deep enough, then even though we know what reduction should be done, it cannot be performed (at this time) because all of γ_i has not been scanned by this SPM).

(c) Otherwise, we have insufficient stack depth or $|R(S',x)|>1$ (implying not all states agree on the same reduction). We then:

(1) Stop this SPM (for the time being).

(2) Start a new SPM to the immediate right (if possible).

Step 3(c) raises two immediate questions:

(1) How do we restart an SPM which has been stopped? The answer is, via the merge of two adjacent SPM's. If an SPM is about to scan a character already scanned by another SPM (to its immediate right) then a merge of the two SPM's is attempted.

(a) If S of the SPM to the left contains a single stack, (s:s'), then we know s' is the correct state to be in at this point. The state s' is used to select the stack named s' in the SPM to the right. That is, when the SPM to the right was started, the correct state may not have been known, so a set of states (each with its own stack) was selected. We now know the correct state, so the correct stack (the one started with s') is now known. Any others may be discarded. We merge the two SPM's by merging the stack (s:s') in the left SPM with the selected stack (say (s':s")). This done by popping s' from the left stack (so it doesn't occur twice) and then concatenating the two. Thus $(s_1,s_2,\ldots,s_i)$ and $(s_i,\ldots,s_j)$ would be merged to $(s_1,s_2,\ldots,s_{i-1},s_i,$ $s_{i+1},\ldots,s_j)$. The newly merged SPM is then made active and begins to do reductions and scans until it encounters a character scanned by another SPM (in which case another merge is attempted) or until it is stopped (because it has insufficient stack depth to do a reduction, or because the parse has finished).

(b) If, on the other hand, the left SPM contains more than one stack, then it is stopped. Later it will be restarted by an SPM to its left. Note that the leftmost SPM <u>always</u> contains a single stack and thus will never itself be stopped. (Clearly, it will also always have sufficient stack depth to do any reduction indicated).

(2) How is a new SPM started?

In particular, what set of states is it to be started in? The basic requirement, of course, is that the correct state be included. The approach we shall take is to use a completer function $C: (s:s') \times \{0,1,\ldots,|P|\} \rightarrow 2^{\text{States of PA}}$. That is, using a stack of PA states and the index of a reduction as arguments, C will return a set of PA states. The interpretation of C((s:s'),i) will be that if an LR parser with (s:s') as the uppermost states of its push down stack does reduction i (and perhaps other reductions), the state it will be in when the character to its right is scanned will be included in C((s:s'),i). Then D, the set of states the new SPM is to be started in is:

$$D = \bigcup_{(s:s')\in S} C((s:s'),R(s',x)),$$ where x is the current look-ahead string. In other words, for each stack (s:s') in S, we take the reduction it suggests, R(s',x), and reason that if this stack is the correct one, and the reduction indicated were done, then the correct state to be in when the next character is scanned is included in C((s:s'), R(s',x)). Since we don't know which is the correct stack, we include them all. Further, we note that of necessity:

(1) $C((s:s'),\emptyset)=\emptyset$. (2) $C((s:s'),0)=s'$.

(recall that 0 indicates no reduction is to be done).

We are now ready to incorporate the modifications noted above into a Basic SPM recognizer algorithm. We will associate with each Basic SPM the triple (S,d,A):

S: As noted above, S is a finite set of non-empty stacks all of the same depth, containing states of PA as elements.

d: The depth of the stack(s) in S less 1. (That is, the number of characters scanned by the SPM).

A: The status of the SPM: a = <u>a</u>ctive, i = <u>i</u>nactive.

<u>Algorithm 5.8</u> (A Parallel Basic SPM Recognizer)

Input: (1) (PA,R) a valid LR parser (with R extended) for G some augmented, ε-free CFG.

(2) $x=\$x'\$^{k+1}$ where $x'\varepsilon V_T^+$ and k = the look-ahead length used by R.

Output: An indication of whether or not $x'\varepsilon L(G)$.

Assume that $b\varepsilon V$ is the next character to be scanned, $\delta\varepsilon V^k$ is the appropriate look-ahead string, and that Y is the SPM being considered.

Basic_SPM_cycle:

[1] (Initialization)

Start one SPM to the left of X_1 (where $x=X_1\ldots X_n$) with $S=\{(s_0)\}$, d=0, A=a. (Recall s_0 is PA's start state).

Optionally start i additional SPM's ($1\leq i\leq|x'|+1$) at positions $X_{j_1},X_{j_2},\ldots,X_{j_i}$ where $1<j_1<j_2<\ldots<j_i\leq|x'|+2$ with (for $1\leq k\leq i$), $S_k=\{(s) \mid N(s,X_{j_k})\neq\emptyset\}$, $d_k=0$, $A_k=a$.

```
[2] (Execute the Basic SPM cycle)
   For all SPM's with A=a, execute the following in parallel.
[3] (Scan the next character)
   If b is marked as scanned by another SPM (Z)
     then setup Merge(Y,Z)
     else mark b as scanned (by Y)
          d←d + 1
          For each (s:s')∈S
             If N(s',b)≠∅
                then push N(s',b) onto the stack (s:s')
                else delete the stack (s:s') from S
   If |S| = 0 then stop, signal an error
[4] Do all Merges which have been setup
[5] (Any reductions?)
   If R(S^B,δ) = {p} ≠ {0} and d ≥ |γ_p|
     then if p = |P|
             then stop, signal success
             else pop |γ_p| states from each stack in S
                  d←d - |γ_p|
                  insert A_p as the next character to be
                       scanned
     else if R(S^B,δ) ≠ {0}
             then Start_new_SPM(Y,δ)
[6] Go to [2]
End of Basic_SPM_cycle.
```

Merge(Y,Z): (Y = some SPM, Z = some SPM to the immediate right of Y)

[1] If $|S_Y| = 1$

then (Do the merge)

$d_Z \leftarrow d_Z + d_Y$

Let $S_Y = \{(s,\ldots,s',s")\}$

If $(s":s'") \varepsilon S_Z$ (for some s'")

then $S_Z \leftarrow \{(s,\ldots,s',s",\ldots,s'")\}$

else $S_Z \leftarrow \emptyset$

$A_Z \leftarrow a$

Eliminate SPM Y

else (No merge)

$A_Y \leftarrow i$

End of Merge.

Start_new_SPM(Y,δ) (Y = some SPM,δ = some look-ahead string)

[1] $A_Y \leftarrow i$

[2] If the next character to the right of SPM Y's current position has <u>not</u> been marked as scanned

then

$$D = \bigcup_{(s:s')\varepsilon S_Y} C((s:s'),R(s',\delta))$$

Start SPM Y' to the immediate right of Y where:

$A_{Y'} = a$

$d_{Y'} = 0$

$S_{Y'} = \{(s) \mid s \varepsilon D\}$

End of Start_new_SPM

□

We assume the last k $'s are always marked as scanned (so no SPM tries to read past them). Note that this algorithm may be modified to become a parser in the same manner as Alg 3.11.

Now we consider an example.

Let G be $E \to E + T \mid T \quad T \to T * P \mid P$

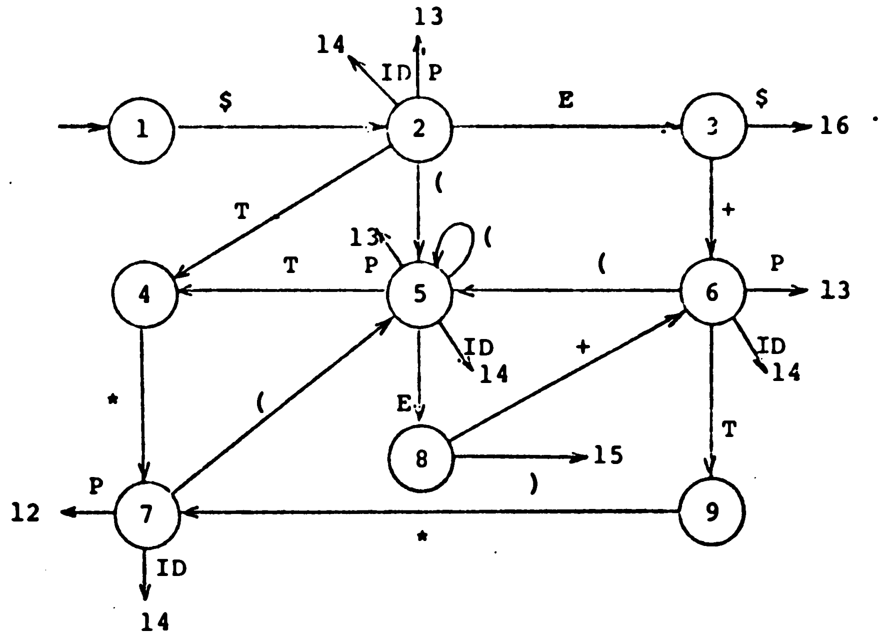
$P \to ID \mid (E) \quad S' \to \$ E \$$

An SLR(1) type parser is considered.

R:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
E		0			0											
T		0			0	0										
P		0			0	0	0									
ID		0			0	0	0									
(		0			0	0	0									
)				2				0	1	1	2	3	4	5	6	
+			0	2				0	1	1	2	3	4	5	6	
*				0					0			3	4	5	6	
$	0		0	2					1	1	2	3	4	5	6	7

PA:



For the completer function, C, we use a method (to be described later) which uses the top two states on the stack. This method is denoted as $C'(s,s',i)$. Since s' is a function of s and i ($s' = N(s,Z_i^{n_i})$ where $A_i \rightarrow Z_i^1 \ldots Z_i^{n_i}$), it may be elided to obtain $C'(s,i)$.

C':	$(2,5) = \{3,4\}$	$(2,4) = \{3,4\}$	$(2,2) = \{3\}$
	$(5,5) = \{4,8\}$	$(5,4) = \{4,8\}$	$(5,2) = \{8\}$
	$(6,5) = \{3,8,9\}$	$(6,4) = \{3,8,9\}$	$(6,1) = \{3,8\}$
	$(7,5) = \{3,4,8,9\}$	$(7,3) = \{3,4,8,9\}$	
	$(8,6) = \{3,4,8,9\}$		

Two SPM's are started.

```
   $      ID    +   ID    *    (   ID   )    $ $     Comment
| 1   2      14            | 2    5    14             Each SPM will
|                          | 5    5    14             do the reduc-
|                          | 6    5    14             tion P→ID.
|_______________           |_7____5____14_

   $      P     +   ID    *    (   P    )    $ $
| 1   2      13            | 2    5    13             Again, both
|                          | 5    5    13             do the same
|                          | 6    5    13             reduction
|_______________           |_7____5____13_            T→P.

   $      T     +   ID    *    (   T    )    $ $
| 1   2      4             | 2    5    4              Again, both
|                          | 5    5    4              do F→T.
|                          | 6    5    4
|_______________           |_7____5____4_

   $      E     +   ID    *    (   E    )    $ $
| 1   2      3     6       | 2    5    8   15         The second
|                          | 5    5    8   15         SPM does
|                          | 6    5    8   15         P→(E).
|_____________________     |_7____5____8___15_

   $      E     +   ID    *      P      $   $
| 1   2      3     6    14  | 2   13 | | 3            The second SPM is
|                           | 5   13 | | 4            stopped because
|                           | 6   13 | | 8            R(13,$)=4 and
|___________________________|_7___12_| |_9            R(12,$)=3. A new
                                                      SPM is started
                                                      using C'.

   $   E   +   P   *    P        $   $
| 1   2   3   6   14 | 2 13 | | 3  16 |               The third SPM is
|                    | 5 13 | |       |               stopped because its
|                    | 6 13 | |       |               stack is not deep
|____________________|_7_12_| |_______|               enough to reduce
                                                      S'→$E$.

   $   E   +   T   *    P        $   $
| 1   2   3   6   9   7 | | 2 13 | | 3  16 |          The first and second
|                       | | 5 13 | |       |          SPM's are about to
|                       | | 6 13 | |       |          merge.
|_______________________| |_7_12_| |_______|
```

```
     $   E  +  T  *   P       $    $             Comment
|1    2   3  6  9  7   12  |3   16|
                                           Now T→T*P may be
                                            reduced.

     $   E  +  T    $     $
|1    2   3  6  9 |3  16|                  E→E+T is done.

     $    E        $      $
|1    2       3  |3   16|              Another merge is
                                        imminent.

     $    E          $     $
|1    2       3        16          S'→$E$ is recognized;success!
```

We are now ready to consider various realizations of the completer function C.

(1) We could simply return the set of all PA states as the value of C((s:s'),i) for $i \neq 0$. This is what is done when SPM's other than the very leftmost are initialized at the beginning of the parse (clearly, states with no successors under the first character scanned are immediately eliminated). This approach certainly ensures that the correct state is included; further, the "calculation" is very fast. Unfortunately, <u>all</u> the information in the stack, as well as the suggested reduction, i, are ignored.

(2) On the other hand, we could try to calculate the correct state exactly by "simulating" the SPM, that is, popping off $|\gamma_i|$ states, then scanning A_i to obtain s". If R(s",x)=0 then s" is the answer, otherwise, $R(s",x)=j \neq 0$ and the above process must be iterated. This approach has two drawbacks:

(a) Since it could be necessary for the SPM to do arbitrarily many reductions before the next character is scanned, the calculation could become arbitrarily expensive.

(b) The calculation may not even be able to be completed because, at some point, the stack is not deep enough to pop off the required number of states (recall this is exactly one reason why SPM's are stopped!).

(3) We can use a bounded amount of information from the stack. Using this information, a set of possible states can be calculated. This is the approach used below.

We are now ready to consider in detail the function C'(s,s',i), a particular realization of the completer function. Let s and s' be the top two states on the stack; we are always guaranteed they exist. Further, as noted earlier, s' is a function of s and i ($s'=N(s,Z_i^{n_i})$ where $A_i \rightarrow Z_i^1 \dots Z_i^{n_i}$) and thus C' has only two degrees of freedom. Thus we can expect to be able to table needed values in a reasonable amount of space.

The calculation of C'(s,s',i) is quite straightforward:

(1) We calculate T, the set of states obtainable as top elements after popping $|\gamma_i|$ states from a stack in which s and s' are uppermost.

(2) For each $\bar{s} \varepsilon T$ we calculate $\bar{s}'=N(\bar{s},A_i)$.

(a) If $\bar{s}'$ has <u>any</u> transitions from it, it is included in C'(s,s',i).

(b) If $\bar{s}'$ can signal any reduction $j \neq 0$ then $C'(\bar{s},\bar{s}',j)$ is calculated and included in $C'(s,s',i)$.

As an example, consider the calculation of $C'(7,12,3)$ in the example presented earlier. Production 3 is $T \to T * P$. $T = \{2,5,6\}$ since just these three states go to state 7 after scanning $T *$ (and $s_7 \xrightarrow{P} s_{12}$ of course). Now:

(a) $s_2 \xrightarrow{T} s_4$. s_4 has a transition from it so $4 \in C'(7,12,3)$. Further, since $R(4,+)=2$, we evaluate $C'(2,4,2)$ obtaining $\{3\}$ which is also included in $C'(7,12,3)$.

(b) $s_5 \xrightarrow{T} s_4$. Once more s_4 is included. Further, $C'(5,4,2) = \{8\}$ is also included.

(c) $s_6 \xrightarrow{T} s_9$. s_9 has a transition out and is thus included. Further, $C'(6,9,1)$ is evaluated (because $R(9,+) = 1$), yielding $\{3,8\}$.

These sets are unioned together to obtain $C'(7,12,3) = \{3,4,8,9\}$ which is just what is tabled (as $C'(7,3)$) in the example.

With a little new notation, C' can be given a very compact definition. Let $A(s,w) = \{s' \mid s' \xrightarrow{w} s\}$. A is the "ancestor" function. Let $\bar{R}(s) = \{p \mid$ there exists w st $p \in R(s,w)$, $w \in V^k\}$. $\bar{R}(s)$ includes p if p can ever be signalled in s, and includes 0 if s has a transition out.

Now:

(a) $C'(s,s',0) = \{s'\}$.

(b) For $p \neq 0$ let $A_p \to Z_p^1 \ldots Z_p^{n_p}$.

Then $C'(s,s',p) =$

$$\bigcup_{s'' \in A(s, Z_p^1 \ldots Z_p^{n_p - 1})} \quad \bigcup_{q \in \overline{R}(N(s'', A_p))} C'(s'', N(s'', A_p), q).$$

<u>Theorem 5.9</u> Let $\$S\$^{k+1} \to^* \alpha XY\beta$. Assume $\alpha XY\beta$ is parsed by Alg 5.2 using some valid LR technique (PA,R) with R exetnded. Further assume that after X is scanned s and s' are the top two states on the stack. Then when Y is scanned, s" will be the top stack state where $s'' \in C'(s,s', R(s', First_k(Y\beta))$.

<u>Proof:</u> (A) If no reductions are done after X is scanned then: (1) $s''=s'$. (2) The right end of the handle is at Y or to its right => (by (1^*a)) $R(s', First_k(Y\beta))=0$. Thus $C'(s,s',0) = \{s'\}$.

(B) Otherwise, assume reductions $i_1, \ldots, i_m$ ($m \geq 1$) are done before Y is scanned, and that $(s_1,s_1'), \ldots, (s_m,s_m')$ are the top two states when each reduction is done.

Clearly, $(s_1,s_1') \equiv (s,s')$. Now $s'' \in N(A(s_m, Z_m^1 \ldots Z_m^{n_m - 1}), A_m)$ (that is, $|\gamma_m|$ states are popped, then A_m is scanned). $R(s'', First_k(Y\beta))=0$ (by (1^*a)) => $s'' \in C'(s_m, s_m', i_m)$. In like manner, $s_m \in A(s_{m-1}, Z^1_{i_{m-1}} \ldots Z^{(n_{i_{m-1}})-1}_{i_{m-1}})$ and $i_m \in R(s_m', First_k(Y\beta))$

and $s_m' = N(s_m, A_{m-1})$ => $C'(s_m, s_m', i_m) \subseteq C'(s_{m-1}, s_{m-1}', i_{m-1})$. Continuing, we finally get $C'(s_2,s_2',i_2) \subseteq C'(s_1,s_1',i_1) = C'(s,s',R(s',First_k(Y\beta))$. Thus $\{s''\} \subseteq C'(s_m,s_m',i_m) \subseteq \ldots$

$\subseteq C'(s,s',R(s',First_k(Y\beta)))$.

$\square$

We now make precise the concept of the "correct" PA state of a character in a sentential form. Let $\gamma X\delta$ be a sentential form of G. Then the correct PA state of X (in $\gamma X\delta$) is s' where $s_0 \xrightarrow{\bar{\gamma}} s'$ and $\bar{\gamma}$ is γ fully reduced. Further, the correct PA successor state of X (in $\gamma X\delta$) is s" where $s_0 \xrightarrow{\bar{\gamma}} s' \xrightarrow{X} s"$.

Observe that the correct (correct successor) PA state of X is independent of reductions performed in γ and δ. Note that we will often speak of the correct (correct successor) state of X, when the sentential form in which it appears is clear from the context.

It should be obvious that the correct state and correct successor state of X (in $\gamma X\delta$) are simply the states a serial LR parser would be in just before, and just after, X is scanned.

Theorem 5.10 Let Alg 5.8 be run on $\$x'\$^{k+1}$ where $\$x'\$ \in L(G)$, using C' and some extended, valid LR parsing technique for G. Assume x' has been reduced to ξ. Let ξ be written as $\gamma X_1 \ldots X_n X_{n+1} \delta$ where $X_1 \ldots X_n$ have been scanned by some SPM Y. Then

(1) S_Y contains the stack $(s_1, \ldots, s_n, s_{n+1})$ where s_1 to s_n are the correct states of X_1 to X_n. Further, when X_{n+1} is scanned s_{n+1} is its correct state.

(2) Let U be the last character scanned by SPM Y. If U is the rightmost character of some phrase in ξ then SPM Y either does a reduction or stops.

(3) All reductions done by an SPM are correct.

<u>Proof</u>: Induction on m, the number of SPM cycles executed during the parse.

For m=0, (2) and (3) are vaccuously true. Also the leftmost SPM is started in s_0 => (1), and all other SPM's are started with <u>all</u> states that can scan the first character to their right => (1).

Assume (1), (2) and (3) are true after m cycles; we show they are still true after m+1 cycles.

If SPM Y is inactive then (1), (2), and (3) will still hold at the end of the cycle, unless Y is reactivated by a merge (a case considered below).

If SPM Y is active, it will try to scan X_{n+1}.

(a) If X_{n+1} is not marked as already scanned, then X_{n+1} is scanned and the stack is of the form $(s_1,\ldots,s_{n+1},s')$ (denoted as (**)), where s_1 to s_{n+1} are the correct states of X_1 to X_{n+1} and s' is the correct successor of X_{n+1}.

(b) If X_{n+1} is marked as already scanned, then if $|S_Y^A| > 1$ then SPM Y is stopped and (1), (2), and (3) will still hold at the end of the cycle, unless Y is immediately reactivated by a merge (a case considered below).

(c) $|S_Y^A| = 1$ and a merge is done. Assume SPM Y merges with SPM Z. Then by (1), Z has a stack of the form $(s_{n+1},\ldots,$

s_p,s'). After the merge, Z's stack will be of the form (**), where n+1 is now the number of characters scanned by both Y and Z. If more than two SPM's are merged, then the resulting SPM will still be of the form (**) since the merges will be done pairwise, and (**) will hold after each intermediate merge.

Now SPM Y will consider possible reductions.

By (2), no phrase can end with X_1 to X_n, thus $\gamma X_1 \ldots X_n$ can be fully reduced to $\bar{\gamma} X_1 \ldots X_n$. By definition, $s_0 \; \bar{\gamma} X_1 \ldots X_{n+1} \to s'$. If X_{n+1} is not the right end of a phrase, then (by (1^*a)) $R(s', \mathrm{First}_k(\delta))=0$. So too, if X_{n+1} is the right end of a phrase (in $\gamma X_1 \ldots X_{n+1}\delta$) then it is the right end of a simple phrase (in $\bar{\gamma} X_1 \ldots X_{n+1}\delta$) and (by ($1^*$b)) $R(s', \mathrm{First}_k(\delta))=i\neq 0$, where i is the reduction to be applied. Now:

(a) If $R(S_Y^B, \mathrm{First}_k(\delta))=\{0\}$, then no phrase ends at X_{n+1}. Further, (by definition) s' is the correct state for the first character in δ. Thus (1), (2), and (3).

(b) If $R(S_Y^B, \mathrm{First}_k(\delta))=\{i\} \neq \{0\}$ and $d_Y=n+1\geq|\gamma_i|$, then the reduction is done, and since all states (including the correct one) agree, the reduction must be correct => (2) and (3). $|\gamma_i|$ states are popped off leaving $(s_1,\ldots,s_{j+1})$ in S_Y where $\gamma X_1 \ldots X_{n+1}\delta$ has been reduced to $\gamma X_1 \ldots X_j A_i \delta$. Clearly, s_1 to s_j are still the correct states for X_1 to X_j. Also, since $\gamma X_1 \ldots X_j$ can be fully reduced to $\bar{\gamma} X_1 \ldots X_j$, s_{j+1}

is the correct state fo A_i. Thus we have (1).

(c) Otherwise, SPM Y is stopped => (2) and (3). Also, since it will not try to scan next cycle, we get (1). Further, by definition of correct and correct successor states, a serial LR parser would have s_{n+1}, s' on the top of its stack after X_{n+1} was scanned. Thus by Thm 5.9, it would be in $s'' \varepsilon C'(s_{n+1}, s', First_k(\delta))$ when it scanned the first character in δ. That is, s" is the correct state for the first character in δ. Since s" will be one of the states the new SPM is started in (if one is started), we get (1) for the new SPM.

$\square$

The above forms the basis of the correctness proof which follows.

Theorem 5.11 Let Alg 5.2 be run on $x = \$x'\$^{k+1}$ where $x' \varepsilon V_T^+$, using C' and some extended, valid LR parsing technique for G, some ε-free CFG. Assume n SPM's are started ($1 \leq n \leq 2+|x'|$) at the leftmost character of x and n-1 other, unique positions. Then

(1) If $\$x'\$ \varepsilon L(G)$ then \$x'\$ will be recognized correctly in $O(|x|)$ time and space.

(2) If $\$x'\$ \not\varepsilon L(G)$ then an error will be signalled.

Proof: (1) Correctness: The algorithm cannot stop and signal an error for then $S = \emptyset$ for some SPM, contradicting Thm 5.10(1). Further, all reductions done are correct (Thm 5.10(3)), thus we always have a valid sentential form.

But now consider the handle. Clearly all characters are eventually scanned (if an SPM stops, another is started if possible). By Thm 5.10(2), when the rightmost character of the handle is scanned, it is reduced (correctly) or the SPM stops. The leftmost SPM never stops (it has a single state stack, and the stack is always deep enough to do any reduction signalled (by cond (2b) of the valid LR definition)). After merges with other SPM's the leftmost SPM still has a single (correct) state stack (by Thm 5.10(1)) and thus the handle is always reduced. Eventually, \$x'\$ is reduced to \$S\$, S'→\$S\$ is recognized and success is signalled.

Linearity: (Space) Each SPM clearly uses space O(number of characters scanned). Since each character is scanned once, and there are O(|x|) characters in any sentential form, the result is immediate.

(Time) Initialization can be done in O(|x|) time. Each SPM cycle (Steps [2] to [6], including, if necessary, one execution of Merge and/or Start_new_SPM) for one SPM can be done in constant time. Each cycle is charged as follows: if a character is scanned, then it is charged, else a merge will be attempted and the last character scanned will be charged. A character is charged for (at most) one unsuccessful merge (in which case the SPM is stopped and when it is reactivated it contains a single state stack which implies that future merges must be successful) and one successful merge (the SPM merged with will have already scanned at least one character). Thus each character is charged a bounded amount of time and there are $O(|x|)$ characters in all.

(2) Whenever a reduction is signalled and performed, (say $A\to\beta$) then β must just have been scanned (conditions (2a) and (2b) of the valid LR definitions). Thus \$x'\$ cannot be reduced to \$S\$ and success cannot be signalled. Either failure is signalled or the algorithm does not halt. As noted above the leftmost SPM is never stopped, so if the algorithm doesn't halt then neither does the leftmost SPM. But if this SPM continues to scan characters and merge with other SPM's then eventually it will have scanned all of \$$\xi$\$ (where x' has been reduced to ξ). But $S'\not\to\$\xi\$$ (by assumption) and thus by cond (2b) of the valid LR definition there does not exist s' st $s_0 \xrightarrow{\$\xi\$} s'$. So if the algorithm hasn't already been stopped, it will be at this point. Thus the leftmost SPM must stop scanning new characters at some point. That is, we have (say) $\nu X\delta$, and the SPM has scanned all of ν but not X. It must then repeatedly do a reduction, scan the left hand side, etc. generating ν_1 (= ν), ν_2, etc. where $\nu_{i+1}\to\nu_i$. Since we have no ε-rules, $|\nu_{i+1}|\le|\nu_i|$ and thus eventually we get $\nu_i=\nu_j$ for j>i. But by cond (2b) of the valid LR definition, $S'\$^k\overset{*}{\to}\nu_i\rho$ for all ν_i and thus $\nu_j\rho\overset{+}{\to}\nu_i\rho=\nu_j\rho \Rightarrow$ G is ambiguous, contradicting Thm 5.1. Thus the algorithm must eventually signal an error and halt.

□

It should be noted that the above still holds if Alg 5.8 is extended (in the manner suggested) to become a parser.

We now return briefly to the claim, advanced earlier, that Algol-like programs might be parsed efficiently, and with a high degree of concurrency, by running a number of Basic SPM's along their entire length. Implicit in this claim are two requirements:

(1) That it is possible to operate concurrently, during the entire parse, a number of Basic SPM's each of which will carry no more than a few state stacks.

(2) That the concurrent operation of such SPM's can be efficiently realized on suitable parallel computers.

The second requirement of course is highly dependent upon the particular computer involved. We can note however that the Basic SPM parser is quite straightforward; in fact, as long as the number of state stacks maintained is small, it is not a great deal more complex than the conventional serial LR parser. Therefore in lieu of actual testing on parallel computers, we will consider the requirement plausable but not yet demonstrated.

The former requirement however may be tested by simulating a Basic SPM parser. In fact, we will now consider the results of a series of such simulations.

An SLR(1) parser was constructed for a CFG (written by David Gries) which generated a slight variant of Algol 60.

In order to start a large number of Basic SPM's it was decided to start a machine at each BEGIN, PROCEDURE, THEN, ELSE, and ; occuring in the program to be parsed. Rather than starting the SPM's immediately before these symbols (as we have usually done), we chose to start them immediately after the symbols. This was done because the symbols in question has a particular property - they never occured as the rightmost symbol in any production. Thus an SPM started immediately after such a symbol (say d) initially needed to carry only those states of the Parsing Automaton which had a transition into them under d. In fact, the SPM's started after BEGIN's and PROCEDURE's needed to carry only one state. Those SPM's started after THEN's and ELSE's needed to carry 2 states (since THEN and ELSE occur in both conditional statements and conditional expressions), and those started after ;'s needed to carry 4 states (since ;'s separate statements and declarations in both blocks and procedures).

The Algol programs to be parsed were chosen at random from the Algorithms section of the Communications of the ACM. The results of the simulations are summarized in Table 5.1. The first column, the number of SPM's started for a given program, may be taken as a measure of the program's size (in terms of the number of declarations and statements it contains). The second column, the ratio of serial parser cycles to SPM cycles, compares the number of iterations of Algorithm 5.2 a serial LR parser would need to parse the program in

Number of SPM's Started	Ratio of Serial Cycles to Basic SPM Cycles	Average Number of PA States per SPM	Algorithm Number
32	3.67	2.67	A303
84	12.78	2.50	A355
59	6.47	3.33	A357
54	9.17	2.65	A374
89	14.71	2.48	A394
30	7.49	2.68	A482

Table 5.1

question with the number of times one or more Basic SPM's would need to concurrently execute the Basic SPM cycle (one iteration of Algorithm 5.8) to parse the same program. Thus this measures the average amount of concurrency obtained by using the Basic SPM's to do the parse. The third column lists the average number of state stacks per SPM during the parse. It thus measures the average overhead required by the Basic SPM's to do the parse.

The most significant result of this series of simulations is the fact that even for the rather small programs considered a fairly large amount of concurrency is obtainable. Further, the overhead required in carrying extra state stacks is small enough to be acceptable. Thus we feel the first requirement outlined above can be considered to be realizable for Algol and Algol-like languages. Coupled with the plausability of the second requirement, the Basic SPM technique must be considered a most promising approach to the problem of parsing Algol-like programs efficiently in parallel computational environments.

5.3 Full Synchronous Parsing Machines

As noted earlier, Basic SPM's can parse any deterministic language. Thus they are powerful enough to parse virtually any grammar used to generate a programming language. Nevertheless, we can consider the result of removing some of the constraints placed on Basic SPM's. We obtain a still more powerful parsing method which shall be called the Full SPM technique.

Let us first consider some of the limitations placed on Basic SPM's.

(1) They are not really "non-canonical" in that the leftmost SPM is not allowed to postpone the decision of what, if any, reduction is to be made.

(2) Information only passes in a left to right direction; that is, once an SPM is stopped, only an SPM to its left may restart it.

(3) The definition of SPM merge is limited by the requirement that the SPM to the left have but a single state stack.

We now investigate ways in which these limitations may be lifted.

(1) The reason why the leftmost (Basic) SPM cannot postpone a reduction is simple - we require that the reduce function be single valued. The solution then is obvious - redefine R so that $R : \hat{S} \times V^k \rightarrow 2^{\{0,1,\ldots,|P|\}}$, while retaining the valid LR requirements, (1^*a), (1^*b), (2a), and (2b).

This is, however, a rather bold extention in that now virtually any (ϵ-free) CFG will now have a valid LR parsing technique. Not all such grammars will be parsable for it will now be possible that all SPM's will stop before the parse is completed. In such a situation we will say the parse has failed. We will later concern ourselves with the problem of recognizing those grammars for which a Full SPM parse cannot fail.

(2) In light of the above, it is imperitive that a means be found to pass information from right to left, for without such means, once the leftmost SPM was stopped, it could never be restarted!

As an example of how information might profitably be passed from right to left, consider the following situation. At some point $S = \{(s,\ldots,s',s''),(t,\ldots,t',t'')\}$ for some SPM. Further, $R(s'',x)=p_1$ and $R(t'',x)=p_2 \neq p_1$. That is, the SPM does not know whether p_1 or p_2 is the correct reduction, so a new SPM is started. Assume $C'(s',s'',p_1)=\{s_1\}$ and $C'(t',t'',p_2)=\{t_2\}$, so that $S = \{(s_1),(t_2)\}$ for the new SPM. Now if, after scanning a few (or even one) character(s), the state stack with t_2 as the bottom element is removed (because its top state cannot scan the next character) then we have enough information to decide that p_1 was the correct reduction. The question is how is this information to be returned to the stopped SPM? We cannot simply "match" s" with s_1 (in general they are different). Therefore we introduce an

<u>interface</u> <u>element</u> of the form $(s,p:s')$ to do the matching through. The interpretation of $(s'',p:s')$ is simply that if, with a state stack $(\ldots,s'',s)$, reduction p (and maybe others) are done, then s' could be the correct state when the next character to the right is scanned. Thus the fact that s' is related to s" is stored. Now, whenever an SPM is stopped, an <u>interface</u> I (a set of interface elements) is calculated where $(s'',p:t)\varepsilon I$ iff $(\ldots,s'',s)\varepsilon S$ and $p\varepsilon R(s,x)$ and $t\varepsilon C'(s'',s,p)$. Information between two SPM's may then be passed through the interface by:

(a) Removing $(s'',p:t)$ from I if some stack $(\ldots,s'',s)$ is not in the left SPM or if some stack $(t:t')$ is not in the right SPM.

(b) Removing the state stack $(\ldots,u'',u)$ from the left SPM if there is not some interface element $(u'',p':t'')$ in I.

(c) Removing the state stack $(v:v')$ from the right SPM if there is not some interface element $(w,p'':v)$ in I.

In the above example, before the information exchange we would have $S_{Left}=\{(s,\ldots,s',s''),(t,\ldots,t',t'')\}$, $I=\{(s',p_1:s_1),(t',p_2:t_2)\}$, $S_{Right}=\{(s_1,\ldots)\}$ and after the exchange we would have $S_{Left}=\{(s,\ldots,s',s'')\}$, $I=\{(s',p_1:s_1)\}$, $S_{Right}=\{(s_1\ldots)\}$. Thus the correct reduction, and the correct state stack have been determined from information in the right SPM.

(3) We allow merges in which the left SPM has more than one stack. Now, for each stack (s:s') in the left SPM, we merge with the (unique) stack (s':s") in the right SPM, if it exists. Note that since more than one stack with the same top state may exist, the same stack in the right SPM may be merged with many different stacks (in the left SPM). This can be done without copying the entire stack, but requires something more elaborate than a simple pointer. This is why it was avoided in the Basic SPM.

We are now ready to extend Alg 5.8 into a Full SPM recognizer. The main difference will be that the Merge procedure has been replaced by an IEM (Information Exchange and (possible) Merge). This procedure will first try to exchange information through the interface; then it will try to do a merge. Merges will be done if the interface is empty (meaning that no reductions were possible) or if $I^P=\{0\}$ (where $I^P=\{p \mid (s,p:s')\varepsilon I\}$) which again means that no reductions are indicated.

Each Full SPM will have the quadruple (S,I,δ,Λ) associated with it. S and d are as before. I is the interface, and Λ, the SPM status, has a new status, m (for <u>m</u>odified) added. When SPM's are restarted after an information exchange, their status is m, indicating that they are to attemp a reduction <u>before</u> doing a scan.

Algorithm 5.12 (A Parallel Full SPM Recognizer)

Input: (1) (PA,R) a valid LR parser, with R extended and possibly not single valued, for G some augmented ε-free CFG.

(2) $x=\$x'\$^{k+1}$ where $x'\varepsilon V_T^+$ and k = the look-ahead length used by R.

Output: An indication of whether or not $\$x'\$\varepsilon L(G)$.

Assume that $b\varepsilon V$ is the next character to be scanned, $\delta\varepsilon V^k$ is the appropriate look-ahead string and that Y is the SPM being considered.

Full_SPM_cycle:

[1] (Initialization)

Start one SPM to the left of X_1 (where $x=X_1\ldots X_n$), with $S=\{(s_0)\}$, d=0, A=a, $I=\emptyset$.

Optionally, start i additional SPM's ($1\le i\le|x'|+1$) at positions $X_{j_1},X_{j_2},\ldots,X_{j_i}$ for $1<j_1<j_2<\ldots<j_i\le|x'|+2$

with (for $1\le k\le i$)

$S_k=\{(s)\mid N(s,X_{j_k})\neq\emptyset\}$, $d_k=0$, $A_k=a$, $I_k=\emptyset$.

[2] (Execute the Full SPM cycle)

For all SPM's with $A\neq i$, execute the following in parallel.

```
[3] (Scan the next character)
  If A_Y=a
    then if b is marked as scanned by another SPM (Z)
           then setup IEM(Y,r); Go to [4]
           else mark b as scanned
                d←d + 1
                For each (s:s')εS
                   If N(s',b) ≠ Ø
                     then push N(s',b) onto the stack (s:s')
                     else delete the stack (s:s') from S
                If |S|=0 then signal an error and stop
                Q←R(S^B,δ); I_Y←Ø
    else Q←I^P_Y; A_Y←a
[4] (Any reductions?)
  If Q = {p} ≠ {0} and d≥|γ_p|
    then if p = |P|
           then signal success and stop
           else pop |γ_p| states from each stack in S
                d←d - |γ_p|
                I_Y←Ø
                insert A_p as the next character to be scanned
    else if Q ≠ {0}
           then Start_new_SPM(Y,δ)
[5] Do all IEM's which have been setup
    Go to [2]
End of Full_SPM_cycle.
```

Start_new_SPM(Y,δ): (Y=some SPM,δ = some k character look-ahead string)

[1] $A_Y \leftarrow i$

[2] If $I_Y = \emptyset$

then (Calculate the interface)

$\forall$ $(s:s') \in S_Y$

$\forall$ $p \in R(s',\delta)$

$\forall$ $t \in C((s:s'),p)$

$(s,p:t) \in I_Y$

[3] Setup IEM(Y,ℓ)

[4] If the next character to the right is marked as scanned

then setup IEM(Y,r)

else start SPM Y' to the immediate right of Y with

$A_{Y'}=a$, $d_{Y'}=0$, $I_{Y'}=\emptyset$

$S_{Y'}=\{(s) \mid (t,p:s) \in I_Y\}$

End of Start_new_SPM.

IEM(Y,DIR): (Information Exchange and (possible) Merge)

(Y = some SPM, DIR = ℓ (left) or r (right))

[1] If DIR = r

then if there is some SPM (Z) to the immediate right of Y

then $a \leftarrow Y$; $b \leftarrow Z$

else return

If DIR = ℓ and the first character to the left of SPM Y
has been scanned by SPM Z'

then $a \leftarrow Z'$; $b \leftarrow Y$

else return

[2] If $I_a \neq \emptyset$

then (Do an Information exchange)

$I'_a \leftarrow \{(s'',p:s') \mid (\ldots s'',s) \varepsilon S_a$ and $(s':t') \varepsilon S_b$ and $(s'',p:s') \varepsilon I_a\}$

$S'_a \leftarrow \{(\ldots s',s) \mid (\ldots s',s) \varepsilon S_a$ and $(s',p:t) \varepsilon I'_a\}$

$S'_b \leftarrow \{(s:s') \mid (s:s') \varepsilon S_b$ and $(t,p:s) \varepsilon I'_a\}$

$I'_b \leftarrow \{(s,p:s') \mid (s,p:s') \varepsilon I_b$ and $(\ldots s,s'') \varepsilon S'_b\}$

If $I'^{P}_a = \{0\}$

then $S_a \leftarrow S'_a$; $S_b \leftarrow S'_b$; $S'_b \leftarrow \emptyset$

[3] If $I^P_a = \emptyset$ or $I'^{P}_a = \{0\}$

then (Do a merge)

$d_b \leftarrow d_b + d_a$

For each $(s,\ldots,s',s'') \varepsilon S_a$

If $(s'':t) \varepsilon S_b$

then $(s,\ldots,s',s'',\ldots,t) \varepsilon S'_b$

If $|S'_b| = \emptyset$

then signal an error and stop

else $S_b \leftarrow S'_b$

$I_b \leftarrow \{(s,p:s') \mid (s,p:s') \varepsilon I_b$ and $(\ldots s,t) \varepsilon S_b\}$

If $A_b = i$ then $A_b \leftarrow m$

Eliminate SPM a

else for IND = a, b

If $S_{IND} = \emptyset$

then signal an error and stop

If A_{IND}=i and $(|I'_{IND}|<|I_{IND}|$ or $|S'_{IND}|<|S_{IND}|)$

then $A_{IND} \leftarrow m$

$S_{IND} \leftarrow S'_{IND}$; $I_{IND} \leftarrow I'_{IND}$

End of IEM.

□

As a matter of convention, we shall assume:

(1) The last k \$'s are marked as scanned.

(2) $C((s:s'),|P|) = \{0\}$. (Recall production $|P|$ is $S' \to \$S\$$; this is done so that the recognition of reduction $|P|$ can be delayed, if necessary. That is, an interface element of the form $(s,|P|:0)$ may, if necessary, be created).

Further, note that the above algorithm can be extended into a parser in the same manner as Alg 5.8.

We now consider an example based on the following non-deterministic grammar, G_5:

$S \to Ac \mid Bd \quad A \to ab \mid aAb \quad B \to abb \mid aBbb \quad S' \to \$S\$$

Note that G_5 <u>does not</u> (as all other parsing methods do) pass back the identity of the rightmost character by reducing the b's to one of two different non-terminals (this is called "rippling back context"). Rather, the information is passed back through the state stacks of the (Full) SPM's.

An SLR(1) type parser is considered.

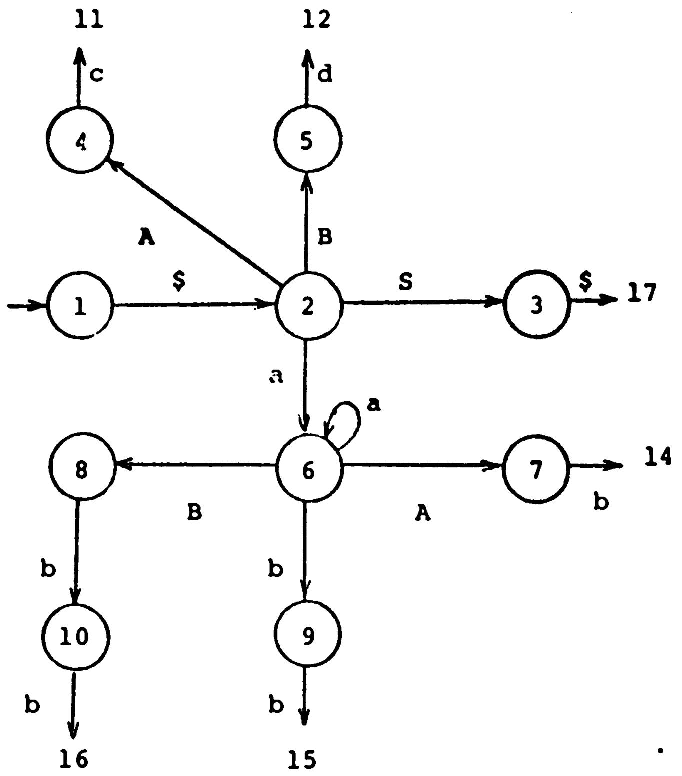
R:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
S		0															
A		0				0											
B		0				0											
a		0				0											
b						0	0	0	{0,3}	0			3	4	5	6	
c				0					3				3	4			
d					0										5	6	
$	0		0								1	2					7

C': (3,7) = {0} (4,1) = {3} (5,2) = {3} (6,3) ={4,7}

(7,4) = {4,7} (9,5) = {5,8} (10,6) = {5,8}

PA:



To show the effect of IEM's, only one SPM is started.

```
    $    a    a    b                                b    c    $    $
 1    2    6    6    9  (6,0:9)   | 9
                        (6,3:4)   | [illegible]
                        (6,3:7)   | [illegible]
```

(After scanning the first b, the SPM is stopped because R(9,b) = {0,3}).

```
    $    a    a    b                      b                     c    $    $
 1    2    6    6    9  (6,0:9)   |  9    15  (9,5:5) | 5
                        (6,3:7)   |  7    14  (9,5:8) | 8
                                                (7,4:4) | 4
                                                (7,4:7) | 7
```

(Another SPM is started; note that an IEM has reduced the size of the first interface).

```
 $  a  a  b         b            c          $    $
|1 2  6  6  9 (6,0:9)| |7 14(7,4:4)| |4 11(4,1:3)| |3
              (6,3:7)|
```

. (c has been scanned, another IEM has occured).

```
 $  a  a  b          b              c              $            $
|1 2  6  6  9 (6,3:7)| |7 14 (7,4:4)| |4 11(4,1:3)| |3 17(3,7:0)|
```

(The \$ has been scanned. Another IEM reduced the first interface to a single element; now it can do reduction 3 (A→ab)).

```
 $  a  A      b              c               $              $
|1 2  6  7| |7 14 (7,4:4)| |4 11 (4,1:3)| |3 17 (3,7:0)|
```

(A merge is about to occur, then reduction 4 (A→aAb)).

```
 $     A    c                 $             $
|1  2  4| |4 11 (4,1:3)|   |3 17 (3,7:0)|
```

(Another merge, then reduction 1 (S→Ac) is recognized).

```
 $     S      $            $
|1  2  3|  |3 17 (3,7:0)|
```

(Another merge, then reduction 7 (S'→\$S\$) is recognized; success!)

Theorem 5.13 Let Alg 5.12 be run on $\$x\$^{k+1}$ where $\$x\$\in L(G)$, using G' and some extended, valid LR parsing technique for G. Assume $\$x\$$ has been reduced to ξ. Let ξ be written as $\gamma X_1 \ldots X_n X_{n+1}\delta$ where X_1 to X_n have been scanned by some SPM Y. Then

(1i) S_Y contains the stack $(s_1,\ldots,s_n,s_{n+1})$ where s_1 to s_n are the correct states of X_1 to X_n. Further, when X_{n+1} is scanned, s_{n+1} is its correct state.

(1ii) If S_Y scans X_{n+1} and the SPM stops, then I_Y contains (s',P:p) where s' is the correct state of X_{n+1}, p is the correct state of the first character in δ (if it exists), and P is the correct production to be applied. Further, (s',P:p) is never removed from I_Y by an IEM.

(2) Let U be the last character scanned by SPM Y. If U is the rightmost character of some phrase in ξ then Y either does a reduction or stops.

(3) All reductions done by an SPM are correct.

Proof: Essentially the same as that of Thm 5.10. Simply note that since an SPM must contain a stack of correct states, (s',P:p) must be included in I_Y when it is calculated. Further, during an IEM the correct state stacks in the two SPM's will match through (s',P:p).

□

As noted earlier, by extending R to allow multiple values, we have created a class of grammars having valid LR parsers which is too broad to be parsed using Alg 5.12. We therefore define a subclass of interest.

An ϵ-free CFG, G, having an extended, valid LR parser (PA,R) is <u>Full SPM parsable</u> if $\forall\ x \epsilon L(G)$, Alg 5.12, using C' and (PA,R) will never stop (with all SPM's inactive) before completing the parse.

The following should be rather obvious.

<u>Theorem 5.14</u> Assume G, some ϵ-free CFG, is ambiguous. Then G is never Full SPM parsable.

<u>Proof</u>: There exists some $x \epsilon L(G)$ which has two (or more) distinct derivations. At some point during the parse of x, Alg 5.12 would recognize a reduction correct in one derivation, but not in another. Thus two different reductions, p_1 and p_2, would be signalled, one indicating the reduction correct in one derivation, the other another possible reduction or 0 (if the right end of the alternate simple phrase hasn't been reached). The SPM will be stopped with both $(s,p_1:s')$ and $(t,p_2:t')$ in I. Since both are correct, Thm 5.13(lii) assures us that neither will be removed. Thus eventually all characters will be scanned, and all unambiguous reductions done; but then all SPM's will be stopped, and the input will not have been fully reduced.

□

Theorem 5.15 Let Alg 5.12 be run on $x=\$x'\$^{k+1}$ where $x'\epsilon V_T^+$, using C' and some extended, valid LR parsing technique for some ϵ-free CFG, G, which is Full SPM parsable. Assume n SPM's are started $(1\leq n\leq 2+|x'|)$ at the leftmost character of x and at n-1 other, unique positions.

Then

(1) If $\$x'\$\epsilon L(G)$ then $\$x'\$$ will be recognized correctly in $O(|x|)$ time and space.

(2) If $\$x'\$\not\epsilon L(G)$ then an error will be signalled or all SPM's will stop without signalling success. (This may be taken as a sign of failure).

Proof: (1) Correctness: The algorithm cannot stop and signal an error for then $S = \emptyset$ for some SPM, contradicting Thm 5.13 (1i and 1ii). Further, by Thm 5.13(3), all reductions done are correct. So too, by assumption, the SPM's will not all stop before the parse is finished. Then either the algorithm runs forever without finishing the parse, or it finishes the parse and signals success. If the algorithm is to run forever, then at some point no more reductions must be performed (else since all reductions are correct, the parse would finish). If the SPM's are not all to stop, then every cycle one of the following must occur:

(i) A character is scanned (ii) A new SPM is started
(iii) A merge occurs (iv) $|I|$ or $|S|$ is decreased by an IEM

But an SPM is started only if it can scan an unmarked character. Therefore (i) and (ii) occur only a bounded number of times. Clearly (iii) can also occur only a bounded number of times, and (since $|I|$ and $|S|$ are bounded above), so too with (iv). Thus eventually either a reduction is done, or all the SPM's stop. Since the latter is ruled out (G is Full SPM parsable), eventually the input is fully reduced and success is signalled.

Linearity in space: Clearly the size of any sentential form, as well as the number of SPM's extant is $O(|x|)$. The space required for I is bounded above, and the space required for S is O(# of characters scanned). Since each character is scanned but once, the result is immediate.

Linearity in time: Initialization is clearly $O(|x|)$; further, each Full SPM cycle (Steps [2] to [4], with perhaps an execution of an IEM and/or a Start_new_SPM) can be done in bounded time. Recognize four types of SPM cycles:

(a) A=m and a merge occured last cycle.

(b) A=m and an IEM occured last cycle and $|S|$ or $|I|$ decreased.

(c) A=a and a character is scanned.

(d) A=a and the next character is marked as already scanned.

Type (d) cannot occur twice in a row (an A=m cycle will intervene), so it can be ignored by charging the others double. Type (a) will charge the last character scanned by the left SPM; type(b) will charge the last character scanned by the SPM; type(c) will charge the character scanned.

Now each character will be charged once for a type(c), once for a type(a) (after the merge it will no longer be the last character scanned by the merged SPM), and a bounded number of times for type(b)'s (after each type(b) cycle, $|I|$ or $|S^B|$ will have decreased). Since there are $O(|x|)$ characters in all, and each is charged a bounded amount, the $O(|x|)$ time bound is immediate.

(2). Whenever a reduction is signalled and performed (say $A \to \beta$) then β must just have been scanned (by conds (2a) and (2b) of the valid LR definition). Thus $\$x'\$$ cannot be reduced to $\$S\$$, and thus success cannot be signalled. Therefore, either failure is signalled, or all SPM's halt (an indication of failure), or the algorithm does not halt. Assume the algorithm does not halt. Observe that all nonterminals are scanned immediately after they appear. Eventually, all the terminals that will ever be scanned are => no new SPM's are started (since they need at least one unscanned character). Further, all merges that will ever occur eventually do. If the algorithm is to run forever, reductions must occur arbitrarily often => at least one SPM (Y) recognizes arbitrarily many reductions. Write what $\$x'\$$ has been reduced to as $\nu_1\delta_1\rho_1$, where δ_1 is what SPM Y has scanned. Thus $\nu_{i+1}\delta_{i+1}\rho_{i+1} \to \nu_i\delta_i\rho_i$ as reductions occur. Since $|\delta_i| \leq |\$x'\$|$, eventually, for $j>i$, we have $\delta_j = \delta_i$. But then $\delta_j \to^+ \delta_i = \delta_j$. Further, by cond (2b) of the valid LR definition, $S'\$^k \to^+ \xi\delta_i\omega$ for some ξ and ω. But then $\xi\delta_i\omega \to^+ \xi\delta_j\omega =$

$\xi\delta_1\omega$, that is, G is ambiguous. But Thm 5.14 contradicts this. Thus the algorithm must halt and signal failure, or all SPM's will stop (again indicating failure).

□

The reader is invited to see the appendix for G_6, a CFG which is Full SPM parsable, but for which all SPM's can halt on an $x \notin L(G)$. Further, it should be noted that the above still holds if Alg 5.12 is extended (in the manner suggested) to become a parser.

We are now ready to consider the problem of deciding whether or not a grammar is Full SPM parsable. The following characterizes the situation when a parse fails because all SPM's have become inactive.

<u>Theorem 5.16</u> Let $\delta \varepsilon SF(G)$ be parsed by Alg 5.12 using C' and some extended, valid LR parsing technique for G, an ε-free CFG. Assume δ is reduced to $\delta' \neq S'$ and that all SPM's are inactive (the parse has failed).

Then

(1) Every character in δ' has been scanned.

(2) Every SPM has $|I^P|>1$ or $I^P=\{q\}\neq\{0\}$ and its stack depth is $< |\gamma_q|$ where $A_q \rightarrow \gamma_q$.

(3) If an IDM is done between any two adjacent SPM's nothing will happen, thus (if the two SPM's are denoted as Y and Z):

(a) $\forall$ $(s",p:s')\varepsilon I_Y$ there exists $(\ldots s",s)\varepsilon S_Y$ and there exists $(s':t')\varepsilon S_Z$.

(b) $\forall$ $(\ldots s'', s) \varepsilon S_Y$ there exists $(s'', p{:}s') \varepsilon I_Y$ and there exists $(s'{:}t') \varepsilon S_Z$.

(c) $\forall$ $(s'{:}t') \varepsilon S_Z$ there exists $(s'', p{:}s') \varepsilon I_Y$ and there exists $(t, \ldots, s'', s) \varepsilon S_Y$ and $s' \varepsilon C'(s'', s, p)$.

(d) $I_Y^P \neq \{0\}$.

Proof: (1) All non-terminals are scanned immediately after being inserted. Further, whenever an SPM is stopped, another is started if the next character has not already been scanned.

(2) These are the only conditions which stop an SPM. Also, if $|I|$ is decreased, or the stack depth is increased, then the SPM is restarted.

(3) We note first that (a) to (d) are the exact requirements that neither a merge nor an information exchange occur. Now consider the SPM which stopped last. It did an IEM with its neighbors before it stopped and if anything had happened (an information exchange or a merge) then it would not have been stopped. Further, if at any later time $|S|$ or $|I|$ were decreased in either SPM then that SPM would have been restarted.

$\square$

We will be unable to solve the general problem of whether or not a CFG G using an arbitrary extended, valid LR parsing technique (PA,R) is Full SPM parsable. The problem will (as we shall see) be solved for the case in which R uses

0 or 1 character look-ahead. Of course, in practice this would be a minimal restriction for look-aheads of more than one character are never used (the tables are too big).

A key result in the solution of the problem is the following.

<u>Theorem 5.17</u> Let $\phi' \varepsilon SF(G)$ be parsed by Alg 5.12 using C' and some extended, valid LR technique for G an ε-free CFG. Assume R, the reduce function, uses 0 or 1 character look-ahead, and that the algorithm reduces ϕ' to $\nu \neq S'$ and stops.
Then if Alg 5.12, using the same LR technique, is run on ϕ where $\nu \rightarrow^* \phi$ then only reductions already done in ν will be performed and the parse will thus fail.

<u>Proof</u>: Unfortunately, the proof is both long and tedious. It may be found in the appendix. The idea, however, is simply that, for a given SPM in the second parse, both S and I will contain the S and I of the corresponding SPM in the first parse. Thus only reductions already done in the first parse will be possible.

□

The utility of the above result is that we may restrict our attention to those sentential forms of G for which an attempted parse will fail without making <u>any</u> reductions. We know that if a parse of $x \varepsilon L(G)$ can fail, then such a sentential form will exist, and further, if it exists, then a parse of $x \varepsilon L(G)$ will fail for some x.

Unfortunately, Thm 5.17 is not true for LR techniques with more than one character look-ahead.

Consider G_7:

$S \rightarrow xdaBc \mid xEB'd \mid yEBc \quad B \rightarrow b \quad B' \rightarrow b \quad E \rightarrow DA$

$D \rightarrow d \quad A \rightarrow a \quad S' \rightarrow \$S\$$

If an LALR(2) parser is constructed for G_7 (recall that its R function need not be single valued), then if a single Full SPM tries to parse \$xdabc\$, the input is reduced to \$xdaBc\$ and the parse stops (and thus fails). But a single Full SPM parsing \$xdaBc\$ will succeed!

To detect those strings for which Alg 5.12 will fail without making any reductions, we construct a non-deterministic finite automaton called the <u>Associated Non-deterministic Parsing Automaton</u> (ANPA). This automaton will simulate an SPM, verifying at each step that no reduction is possible.

We define the ANPA as (D,V,δ,D_0,D_f) where

$D = \{D_i\}=\{(U,d)\}$ = the states of the ANPA

$U = \{(s',s) \mid s',s\varepsilon\hat{S}$ (the states of PA)$\}$

$d\varepsilon \{0,1,\ldots,$Max_Rhs_Length in G $(\geq 3)\}$

V = The same vocabulary as is used by the PA

δ (defined below)

$D_0 = (\{(0,s_0)\},0)$ where s_0 is the PA's start state

$D_f = \{D_i \mid U_i^B=\{s_f\}$ where $s_0 \xrightarrow{\$S\$} s_f$ and $d_i=1$ or $d_i=2\}$

Let $U^A \equiv \{s' \mid (s',s)\varepsilon U\}$, $U^B \equiv \{s \mid (s',s)\varepsilon U\}$,

$a \oplus 1 = \mathrm{Min}(a+1,$ Max_Rhs_Length in G$)$

$L(U,T,Q',a) = \underline{\text{true}}$ iff

(1) $\forall$ $(s',s)\varepsilon U$ there exists $p\varepsilon Q'$ st $p\varepsilon R(s,a)$ and
$T^B \cap C'(s',s,p) \neq \emptyset$

(2) $\forall$ $p\varepsilon Q'$ there exists $(s',s)\varepsilon U$ st $p\varepsilon R(s,a)$ and
$T^B \cap C'(s',s,p) \neq \emptyset$

$N'(U,a)$ = If there exists $(s',s)\varepsilon U$ st $N(s,a) = \emptyset$ then $\emptyset$
else $\{(s,N(s,a)) \mid (s',s)\varepsilon U\}$.

We now define δ, the transition function.

To generate transitions from $D_i \varepsilon D$ $(D_i \equiv (U_i, d_i))$ do the following:

$\forall$ $a\varepsilon V$ let $Q = R(U_i^B, a)$

$\forall$ Q' = non-empty subsets of Q

$\forall$ T where $T \subseteq T'$ and $T' = \{(0,u) \mid u\varepsilon C'(s',s,p)$ and $p\varepsilon R(s,a)$ and $(s',s)\varepsilon U_i$ and $p\varepsilon Q'\}$

If $L(U_i,T,Q',a)$ is true and $N'(T,a) \neq \emptyset$

then if $Q' = \{0\}$

then add the transition $D_i \xrightarrow{a} (N'(T,a), d \oplus 1)$

else

if $Q' = \{p\} \neq \{0\}$ and $d_i \geq |\gamma_p|$

then generate $\underline{\text{no}}$ transition

else add the transition

$D_i \xrightarrow{a} (N'(T,a), 1)$

We now investigate how the ANPA simulates an SPM. U represents the top 2 states of each state stack in S, d represents the stack depth. Thus D_0 represents the start state s_0 and 0 characters scanned. The look-ahead, denoted a, is guessed; Q is calculated and Q' a non-empty subset of Q is guessed. Q' represents those elements of Q which are <u>not</u> later removed because of IEM's. T, a subset of T' is guessed. T represents those states in T' which are <u>not</u> later removed because of IEM's or because they lack successors. It is verified (via L) that:

(1) No state in U will later be removed because of an IEM.

(2) No reductions in Q' will later be removed because of an IEM.

It is then verified that no reduction is possible; either the next character is scanned or a "new" SPM is started. If, when the entire input has been scanned, we are in $D_n \epsilon D_f$ then we have verified that (1) the input might have been reduced to S' (since we get to s_f from s_0 by \$S\$), (2) the final reduction could not be made (because the stack depth <3).

<u>Theorem 5.18</u> Assume $\delta = X_1 \ldots X_n \neq S'$ and that $\delta \epsilon SF(G)$. Further, assume δ is accepted by some ANPA based on (PA,R) an extended, valid LR parsing technique, using 0 or 1 character look-ahead, for G some ϵ-free CFG. If Alg 5.12, using G' and (PA,R), is run on δ with a single Full SPM then no reductions will be performed and thus the parse will fail.

<u>Proof</u>: Let the sequence of states in the ANPA be $D_0 \xrightarrow{X_1} D_1 \xrightarrow{X_2} \ldots \xrightarrow{X_n} D_n$. Let $Q'(D_i,D_{i+1})$ be the value of Q' used in the generation of the transition from D_i to D_{i+1} via X_i (if more than one Q' exists, then choose any). Assume some SPM has scanned $X_i \ldots X_j$ and thus contains stacks of depth $2+j-i$. Then let $V_k=\{s \mid (s':s")\varepsilon S$ and s is the $(1+k-i)$th element in $(s':s")\}$; let $V_k'=\{s \mid (s':s")\varepsilon S$ and s is the $(2+k-i)$th element in $(s':s")\}$. V_i then corresponds to the states in S just before X_i is scanned, V_i' to those just after X_i is scanned.

We now show by induction on the number of SPM cycles executed that the following conditions hold, and further that no reductions are performed. Thus the parse will fail. (Recall $D_i \equiv (U_i,d_i)$)

(1) $U_i^A \subseteq V_i$ and $U_i^B \subseteq V_i'$

(2) If $i \geq 1$ and $d_{i+1} = 1$ then

(a) An SPM will stop after scanning X_i and will never do a reduction or merge.

(2b) Whenever an SPM stops, its interface, I_i, will satisfy: (assume X_i was just scanned)

(i) $\forall\, p\varepsilon Q'(D_i,D_{i+1})$ there exists $(s,p:s')\varepsilon I_i$ st $s\varepsilon U_i^A$ and $s'\varepsilon U_{i+1}^A$

(ii) $\forall\, s\varepsilon U_i^A$ there exists $(s,p:s')\varepsilon I_i$ st $s'\varepsilon U_{i+1}^A$

(iii) $\forall\, s'\varepsilon U_{i+1}^A$ there exists $(s,p:s')\varepsilon I_i$ st $s\varepsilon U_i^A$

Base step: $U_1^A=\{s_0\} \subseteq V_1=\{s_0\}$ implies (1); (2) holds vacuously.

Induction step: Consider each step of the SPM cycle.

[3] (Scan a character)

$V_i'=N(V_i,X_i)$ and $U_i^B=N(U_i^A,X_i)$. Thus $U_i^A \subseteq V_i$ implies $U_i^B \subseteq V_i'$. Further, since each state in U_i^A has a successor under X_i, $U_i^A \subseteq V_i$ still holds. Thus (1) is verified.

Calculation of Q: We claim $Q \supseteq Q'(D_i,D_{i+1})$.

(a) If A=a, $Q=R(V_i',X_{i+1}) \supseteq R(U_i^B,X_{i+1}) \supseteq Q'(D_i,D_{i+1})$
(Recall $Q' \subseteq Q = R(U_i^B,X_{i+1})$).

(b) If A=m then $Q \supseteq Q'(D_i,D_{i+1})$ by cond (2b(i)).

[4] (Do a reduction if possible)

If $Q = \{0\}$ then $Q'(D_i,D_{i+1}) = \{0\}$ and also $U_i^B= U_{i+1}^A$. No new SPM is started, thus $V_i'=V_{i+1}$ which implies cond (1) and cond (2) since no SPM stops.

Otherwise, if $Q = \{p\}\neq\{0\}$ then $Q'(D_i,D_{i+1}) = \{p\}$ and by construction of the transition function $d_i<|\gamma_p|$. But every time d_i is set to 1 (by cond (2a)) an SPM is stopped at the same point. Therefore, the SPM's stack depth $<|\gamma_p|$ and in any event <u>no</u> <u>reduction</u> <u>is</u> <u>done</u>, and the SPM is stopped which implies cond (2a). Further, recall that $U_i^A \subseteq V_i$ and $U_i^B \subseteq V_i'$, $Q'(D_i,D_{i+1}) \subseteq Q$ and $U_{i+1}^A \equiv T^B$ (used in the transition definition). Now, since $L(U_i,T,Q',X_{i+1})$ must be true, cond L(2) implies cond (2b(i)) and cond L(1) implies cond (2b(ii)), and from the definition of $U_{i+1}^A \equiv T^B$ we get cond (2b(iii)). ($\forall$ $s\varepsilon T^B$ there exists $(u,u')\varepsilon U$,

$p \epsilon Q'$, $p \epsilon R(u', X_{i+1})$, $s \epsilon C'(u,u',p)$). And for the new SPM which will be started, (by cond (2b(iii))) $U^A_{i+1} \subseteq I^B_i =$ V_{i+1} which implies cond (1).

[5] (Do all IEM's)

Information exchanges:

Conds (1) and (2) imply cond (2) after the exchange.

Conds (2b(ii)) and (2b(iii)) imply cond (1) holds for V'_i and V_{i+1} after the exchange. Also note that only those stacks containing states not in U^A_{i+1} and U^A_i are removed. Therefore cond (1) still holds for any j st X_j was scanned by one of the two SPM's doing the IEM.

Merge:

To merge, we require $I^P_i = \{0\}$ which implies (by cond (2b(i))) that $Q'(D_i, D_{i+1}) = \{0\}$ and thus $d_{i+1} \neq 1$ which implies cond (2a). Further, $U^B_i = U^A_{i+1}$ (by construction, $T^B \subseteq \{s \mid s\epsilon$ $C'(s',s'',0)$ where $(s',s'')\epsilon U_i\} = U^B_i$, and cond L(1) requires $T^B \supseteq U^B_i$, and $U^A_{i+1} \equiv T^B$). Since $V'_i \supseteq U^B_i$ and $V_{i+1} \supseteq U^A_{i+1}$, after the merge $V'_i = V_{i+1}$ and thus cond (1) holds. Also, cond (1) holds for any j st X_j was scanned by the merged SPM (by the same argument as above).

$\square$

Theorem 5.19 Assume $\delta = X_1 \ldots X_n \neq S'$ and that $\delta \epsilon SF(G)$. Further, assume Alg 5.12 is run on δ using some extended, valid LR parsing technique, (PA,R) with 0 or 1 character look-ahead, for G an ϵ-free CFG. Also, assume C' is used and that a single Full SPM is started.

Then if no reductions are made (and the parse thus fails) then δ will be accepted by the ANPA based on (PA,R).

Proof: Let V_i and V_i' be as in Thm 5.18, and let $V_0' \equiv V_1$. From Thm 5.16 we know every character in δ has been scanned by some SPM. Thus there are a number of adjacent, inactive SPM's covering the input. Now consider the ANPA starting at X_1. We start with $D_0 \equiv (\{(0,s_0)\},0)$. Also, V_0' $(\equiv V_1)$ = $\{s_0\}$. Thus the following is true for i=0, and the first SPM.

(a) $V_i' = U_i^B$ (b) $V_i = U_i^A$ for i>0

(c) The number of characters scanned by the SPM that scanned X_i, up to and including X_i, is $\geq d_i$.

A: (Scan step)

Now assume X_{i+1} was scanned by the SPM under consideration and that (a), (b), and (c) hold for i. Since X_{i+1} was scanned, $0 \in R(V_i', X_{i+1})$ which implies $0 \in R(U_i^B, X_{i+1})$. Therefore, choose $Q=\{0\}$, and choose $T^B = U_i^B = V_i'$. Now both conditions of $L(U_i, T, Q', X_{i+1})$ are true (clearly) and $N'(T, X_{i+1}) \neq \emptyset$ (because each element of $V_i' = T^B$ has a transition under X_{i+1}, otherwise it would have been removed). Thus the transition $D_i \xrightarrow{X_i} D_{i+1} = (N'(T, X_{i+1}), d_i \oplus 1)$ exists. Now $U_{i+1}^A = T^B = V_i' = V_{i+1}$ (since no SPM stopped at X_i) which implies cond (b) for i+1; further, by definition, $V_{i+1}' = N(V_{i+1}, X_{i+1}) = N(T^B, X_{i+1}) = U_{i+1}^B$ which implies cond (a) for i+1. Finally, $d_i \oplus 1 \leq d_i + 1$ and thus cond (c) holds for i+1.

(End of A)

We repeat the above, (A), for each character scanned by the SPM under consideration until we scan X_j, the last character scanned by the SPM. Conds (a), (b), and (c) hold for j and the ANPA has gone from D_0 to D_j by scanning $X_1 \ldots X_j$.

B: (Start a "new" SPM)

At this point the ANPA "guesses" that an SPM is stopped at X_j. It then verifies its guess and proceeds to simulate the next SPM. That is:

We note that since an SPM is stopped at this point, either $|I_j^P|>1$ or $I_j^P=\{q\}\neq\{0\}$ and the SPM's stack depth $<|\gamma_q|$. Now $R(U_j^B,X_{j+1}) = R(V_j',X_{j+1}) \supseteq I_j^P$ (when I_j was calculated, $I_j^P \equiv R(V_j',X_{j+1})$, later (perhaps) elements were removed by IEM's). Thus we choose $Q'=I_j^P$. By Thm 5.16(3), $V_{j+1}= I_j^B$ and $\forall\ s\varepsilon I_j^B$ there exists $s_1 \varepsilon V_j$, there exists $s_2 \varepsilon V_j'$ and there exists $(s_1,p{:}s)\varepsilon I_j$ st $s\varepsilon C'(s_1,s_2,p)$ and $p\varepsilon R(s_2,X_{j+1})$ (by construction of the interface elements). And recall, $V_j'= U_j^B$, $V_j= U_j^A$. Thus we may choose $T^B= I_j^B= V_{j+1}$. Now, $L(U_j,T^B,Q',X_{j+1})$ is true (Thm 5.16(3b) implies L(1), 5.16 (3a) implies L(2)) and $N'(T,X_{j+1}) \neq \emptyset$ (since $T^B= V_{j+1}$). Since $Q'= I_j^P$ and $d_j \leq$ the SPM's stack depth (cond (c)), we choose the transition $D_j \xrightarrow{X_{j+1}} D_{j+1}=(N'(T,X_{j+1}),1)$. $U_{j+1}^A = T^B= V_{j+1}$ thus cond (b); also, $U_{j+1}^B=N(T^B,X_{j+1})=N(V_{j+1},X_{j+1})= V'_{j+1}$ thus cond (a). Finally, $d_{j+1}= 1$ as does the stack of the new SPM thus cond (c).

(End of B)

Once again, (a), (b), and (c) hold and the ANPA has gone from D_0 to D_{j+1} by scanning $X_1 \ldots X_{j+1}$. We now repeat step A until the last character of the current SPM is reached, then do step B, etc. Finally, we scan X_n=\$. Since the SPM must contain the correct state (Thm 5.13(1)), $s_f \varepsilon V'_n$. But if any other state, s, is in V'_n, we require $s_0 \xrightarrow{\omega\$} s$ where $|\omega\$|>1$ and ω\$ satisfies cond (2b) of the valid LR parser definition. This implies that $\omega\$=\$S\$$, thus $s=s_f$ (since $s_0 \xrightarrow{\$S\$} s_f$) and $V'_n = \{s_f\} = U^B_n$ by cond (a). Further, the stack depth of the SPM must be <3 (else \$S\$ would be reduced to S'). This implies that $d_n<3$ and d_i always >0 except for d_0. But this means $D_n \varepsilon D_f$ and δ is therefore accepted.

□

We now use the above results to establish the following.

<u>Theorem 5.20</u> Let G be an ε-free CFG, and let (PA,R) be an extended, valid LR parsing technique for G, using 0 or 1 character look-ahead.

Then G (using (PA,R)) is Full SPM parsable iff

$SF(G) \cap T(ANPA) = \emptyset$

where ANPA is based on (PA,R).

<u>Proof</u>: If: Assume $SF(G) \cap T(ANPA) = \emptyset$ but that for $\delta \varepsilon L(G)$ Alg 5.12, using C' and (PA,R) fails, having reduced δ to $\delta' \neq S'$. By Thm 5.17, a single Full SPM started on δ' will fail; further, <u>no</u> reductions will be made. But then by Thm 5.19 $\delta' \varepsilon T(ANPA)$ and $\delta' \varepsilon SF(G)$ (since all reductions done

are correct), contradicting our assumption.

Only if: Assume G (using (PA,R)) is Full SPM parsable, but $SF(G) \cap T(AMPA) \neq \emptyset$. Let δ be in the intersection. By Thm 5.18, Alg 5.12 will make no reductions while trying to parse δ and thus will fail. But by Thm 5.17, Alg 5.12 will fail on $x \in L(G)$ where $\delta \to^* x$, contradicting our assumption that G is Full SPM parsable.

□

<u>Corollary 5.21</u> Given G an ε-free CFG and (PA,R) an extended, valid LR parsing technique for G, using 0 or 1 character look-ahead, it is decidable whether G, using (PA,R) is Full SPM parsable.

<u>Proof</u>: $SF(G) \cap T(AMPA)$ is context free and thus emptiness is decidable.

□

As noted earlier, the above method cannot be extended to 2 or more character look-aheads because Thm 5.17 can't. Thus the question is currently open.

For all practical purposes the Basic SPM method is powerful enough to do the job; the Full SPM method is of interest theoretically. In the next chapter we shall compare it with a number of other non-canonical parsing methods.

6 A COMPARISON OF FULL SPM TECHNIQUES WITH OTHER NON-CANONICAL METHODS

6.1 A Comparison of Grammar Classes

In order to determine how powerful Full SPM Parsing Techniques are, we shall compare Full SPM LR(k) Techniques (the most powerful commonly known method) with other non-canonical methods which are of linear time and space complexity. In particular, Total Mixed Strategy Precedence (Chapter 4), Bounded Context Parsable (Williams [Wi 69]), LR(k,∞) (Szymanski [Sz 73]), and LR Regular (Cohen and Culik [CC 71]) shall be considered. While a few inclusions will be demonstrated, for the most part the classes will be seen to be mutually incomparable.

We first consider the Total Mixed Strategy Precedence Technique of Chapter 4.

Theorem 6.1 Let G be an augmented TMSP Grammar. Then G is Full SPM parsable using an LR(1) technique.

Proof: We will establish that it cannot be the case that G is not Full SPM parsable (using an LR(1) technique) and yet is TMSP. Assume this could be the case and that in fact Alg 5.12 parsing $x' \in L(G)$ reduces x' to $x \neq S'$ and stops (and thus fails). By Thm 4.8(2), we may write x as $\alpha YX_1 \ldots X_N Z\delta$, where the leftmost $>_{TM}$ is between X_N and Z; further the leftmost recognizable reduction is $A_q \to X_1 \ldots X_N$, where $Y \varepsilon \hat{L}(q)$. Now an SPM (V) must be stopped at X_N with $q \varepsilon I_V^P$ (since no reduction is missed). No SPM can be stopped at X_i $1 \le i < N$. If one was, then we would have $X_i \rho^+ \alpha \lambda^* X_{i+1}$ and this cannot be by cond (1) of the TMSP definition because $X_i \alpha X_{i+1}$ (by assumption). Further $0 \not\varepsilon I_V^P$ for if it were then $X_N \alpha \lambda^* Z \Rightarrow X_N \le_{TM} Z$, but $X_N >_{TM} Z$ by assumption. Thus SPM V cannot be stopped because its stack is not deep enough to reduce $X_1 \ldots X_N$ (recall no SPM is stopped at X_i $1 \le i < N$). Therefore, there exists j st $j \varepsilon I_V^P$ and $0 \ne j \ne q$. Let production j be $A_j \to \delta_j$. Now consider the cases:

(i) $|\delta_j| < N$. Then $A_q \to X_1 \ldots X_N = X_1 \ldots X_k \delta_j$. Further $\$S\$ \to^* \ldots X_k A_j \ldots$ and thus $X_k \beta A_j$, violating cond (2) of the TMSP definition.

(ii) $|\delta_j| = N$. That is, $X_1 \ldots X_N = \delta_j$. Now $\$S\$ \to^* \ldots WA_j \ldots$ where $W \to^* \ldots Y$. Thus $Y \beta A_j \Rightarrow Y \varepsilon LT(j) \Rightarrow Y \varepsilon \hat{L}(j)$ but recall $Y \varepsilon \hat{L}(q) \Rightarrow$ cond (3) of the TMSP definition is violated.

(iii) $|\delta_j| > N$. Then $\delta_j = \ldots WX_1 \ldots X_N$ where $W \to^* \ldots Y$. If $W=Y$ then since $Y \beta A_q$ (by assumption $X_1 \ldots X_N$ should be reduced to A_q) we have a violation of cond (2) of the TMSP definition; otherwise $W \to^+ \ldots Y$. But then $Y \rho^+ \alpha X_1 \Rightarrow Y >_{TM} X_1$, contradicting the assumption that the leftmost $>_{TM}$ was at X_N.

$\square$

<u>Corollary</u> Let G, some augmented CFG be

(a) Simple Precedence, or

(b) Strict Precedence, or

(c) Total Precedence, or

(d) Simple Mixed Strategy Precedence, or

(e) Strict Mixed Strategy Precedence.

Then G is Full SPM parsable using an LR(1) parsing technique.

<u>Proof</u>: By Thm 4.11, each of the above classes is contained in TMSP.

□

We now consider two special cases of Bounded Context Parsable (BCP) Grammars.

<u>Theorem 6.2</u> Let G be a CFG augmented with endmarkers. Then

(a) If G is BCP(1,0) then G is Full SPM parsable using an LR(1) technique.

(b) If G is BCP(0,1) then G is Full SPM parsable using an LR(1) technique.

<u>Proof</u>:

(a) If G is BCP(1,0) then the handle must always be recognizable, independent of context to the right. Thus it is certainly LR(1) and therefore Full SPM parsable using an LR(1) technique.

(b) Assume G is BCP(0,1) but not Full SPM parsable using an LR(1) technique. In particular, assume Alg 5.12 parsing

$x' \epsilon L(G)$ reduces x' to $\theta \neq S'$ and stops (and thus fails). Let $\theta = X_1 \ldots X_i \ldots X_j X_{j+1} \ldots X_N$ where $X_i \ldots X_j$ is the rightmost simple phrase. A rule to reduce $X_i \ldots X_j$ to (say) A_q in context (ϵ, X_{j+1}) must exist; otherwise $X_i \ldots X_j$ could never be reduced. But this implies that if $\rho X_k \ldots X_N \epsilon SF(G)$, then

(1) If $k>i$ then part of the rightmost simple phrase must be in ρ.

(2) If $k \leq i$ then $X_i \ldots X_j$ must be the rightmost simple phrase and $A_q \rightarrow X_i \ldots X_j$ is the correct reduction.

Note that if (1) or (2) were not true then either (1) an incorrect reduction might occur in x, or (2) $X_i \ldots X_j$ might be incorrectly reduced. There must be an SPM (call it V) stopped at X_j (since no reductions are missed) and $q \epsilon I_V^P$. Since no reduction was done either:

(i) $|I_V^P| > 1$ or (ii) V's stack depth $< |X_i \ldots X_j|$.

Now if (i) holds, remove all elements of the form $(s, q{:}s')$ from I_V and if (ii) holds then an SPM W, to the immediate left of V, is stopped at X_k, $i \leq k < j$ (otherwise, V's stack depth would be $\geq |X_i \ldots X_j|$). Further, $0 \epsilon I_W^P$ (since the correct reduction is no reduction). Remove all elements of the form $(s, 0{:}s')$ from I_W.

Now, proceeding from left to right, choose a sequence of state stacks and interface elements (one stack and one interface element from each SPM) such that each stack "matches" the interface element on each side. Thus we have for example:

$(s_1 \ldots s_1', s_1'')$ $(s_1', p_1 : s_2)$ $(s_2 \ldots s_2', s_2'')$ $(s_2', p_2 : s_3) \ldots$
$\ldots$ $(s_{n-1}', p_{n-1} : s_n)$ $(s_n \ldots s_n'\ s_n'')$ $(s_n', p_n : 0)$ where $(s_1 \ldots s_1'')$ was the first SPM's stack, etc.

Thm 5.16(3) guarantees such a sequence can always be chosen.

If $p_i = 0$, then merge $(s_i, \ldots, s_i'')$ and $(s_{i+1}, s_{i+1}', \ldots, s_{i+1}'')$ into the single stack $(s_i, \ldots, s_i'', s_{i+1}', \ldots, s_{i+1}'')$ (recall $s_i'' = s_{i+1}$ if $p_i = 0$). We now have $m \leq n$ stacks and interface elements.

Find $r \leq m$, the rightmost interface element such that (*): the stack to its left, $(s_r, \ldots s_r'')$ has a depth $\geq |\delta_{p_r}|$ where $A_{p_r} \to \delta_{p_r}$. Observe that p_1 will always satisfy this requirement; further, p_m never will since reduction p_m must be $S' \to \$S\$$. We can write x as $\gamma_1 \ldots \gamma_m$ where for $1 \leq i \leq m$, $s_i \xrightarrow{\gamma_i} s_i''$. Observe that for $m \geq j > r$, $|\gamma_j| < |\delta_{p_j}|$ (by construction). Since $s_{r+1} \varepsilon C'(s_r', s_r'', p_r)$, there exists $B_r \varepsilon V_N$ st $s_0 \xrightarrow{\nu B_r} s_{r+1} \xrightarrow{\gamma_{r+1}} s_{r+1}''$ and $S' \to^+ \nu B_r \gamma_{r+1} \ldots$ and $B_r \to^+ \ldots \delta_{p_r}$ where $\gamma_r = \ldots \delta_{p_r}$. And since $|\gamma_{r+1}| < |\delta_{p_{r+1}}|$, $A_{p_{r+1}} \to \ldots B_r \gamma_{r+1}$. Repeating the above, we get $s_0 \xrightarrow{\nu' B_{r+1}} s_{r+2} \xrightarrow{\gamma_{r+2}} s_{r+2}''$ and $S' \to^+ \nu' B_{r+1} \gamma_{r+2} \ldots$ and $B_{r+1} \to^+ \ldots A_{p_{r+1}} \to^+ \ldots B_r \gamma_{r+1}$. Finally, we get $S' = A_{p_m} \to^+ \ldots B_r \gamma_{r+1} \ldots \gamma_m \to^+ \xi \delta_{p_r} \gamma_{r+1} \ldots \gamma_m$. Thus,

$\zeta\delta_{p_r}\gamma_{r+1}\cdots\gamma_m = \zeta X_k\ldots X_N$ and either (1) or (2) should hold. By construction the rightmost simple phrase is δ_{p_r}. Thus (1) certainly doesn't hold. If $k \neq i$ or $\delta_{p_r} \neq X_i\ldots X_j$ then (2) doesn't hold. Otherwise, by (i) or (ii) $p_r \neq q$ for either q was removed from SPM V or 0 was removed from the SPM to the left, insuring that no merges of stacks could occur, and thus q could not satisfy (*). Thus if a Full SPM parse can fail, G is not BCP(0,1) a contradiction.

$\square$

Finally, we establish the following.

<u>Theorem 6.3</u> Let FSPM(k) be the class of grammars which are Full SPM parsable using an LR(k') technique for $k \geq k' \geq 1$. Then FSPM(k) is incomparable with each of the following classes.

(a) BCP(m,n) for $m \geq 1$, $n \geq 1$.

(b) LR(k,∞) for $k \geq 1$.

(c) LR-Regular.

<u>Proof</u>:

(a) 1. G_5 (used to illustrate Full SPM parsing) is Full SPM parsable using an LR(1) technique; however, it is not BCP(m,n) for any $m \geq 1$, $n \geq 1$.

2. G_8^L (see below) is BCP(1,1) but not Full SPM parsable with an LR(L') technique, $L' \leq L$. (It fails on $\$eccd^Lg\$$).

(b) 1. G_5 is not LR(k,∞) for any k.

2. G_9 (see below) is LR($1,\infty$) (proved in Szymanski [Sz 73]) but fails on $a^{k+1}cb^{k+1}$ for an SPM using an LR(k) parser.

(c) 1. G_{10} (see below) is Full SPM parsable using an LR(1) parser; however, it is not LR-Regular.

2. G_8^L is LR-Regular, but, as noted above, is not Full SPM parsable with an LR(L') technique for $L' \leq L$.

G_8^L: $S' \to \$S\$$ $\quad S \to EAg \mid E'A'h \mid jA'g \quad E \to e \quad E' \to e$

$\quad A \to ccd^L \quad C \to cc \quad A' \to CD^L \quad D \to d$

G_9: $S' \to \$S\$ \quad S \to A \mid B \quad A \to aAA' \mid acA'$

$\quad B \to aBB'B' \mid acB'B' \quad A' \to b \quad B' \to b$

G_{10}: $S' \to \$S\$ \quad S \to AA' \mid BB' \quad A \to d \quad B \to d$

$\quad A' \to aA' \mid aB' \quad B' \to aB'a \mid b$

□

As Theorem 6.3 suggests, a Full SPM parser can, at times, be very clever, and, at other times, rather stupid. In fact, the Full SPM technique is possessed of a great strength and a great weakness. Its strength, as exemplified in G_5, is its ability to bring back right context internally (in its state stacks) rather than having to do it externally by doing reductions to "ripple context back". Its weakness is that it does not always retain exact knowledge of what it has scanned. Rather, when a new SPM is started, erroneous states, not compatable with what already has been scanned can be introduced. Thus in G_8, for example, the parse fails because when a new SPM is started (just after scanning \$ecc) it contains a state corresponding to \$EC (this state is correct) and another corresponding to \$jC (erroneous). The method has "forgotten" that e and not j was scanned after the \$.

7 PARSING OPERATOR INFIX EXPRESSIONS

All of the parsing methods considered so far are rather general in scope; they are able to parse all deterministic languages, or, at the very least, large and interesting subsets of them. We now turn our attention briefly to the special problem of parsing operator infix expressions. Such expressions are of interest because in some programming languages (such as APL and, to a lesser extent, Fortran) they represent the only non-trival structure present.

We first introduce Operator Infix Grammars (OIG's) as a means of describing the class of operator infix expressions. It is then seen that an encoding of the input string can be transformed, in parallel, to an encoding of the derivation tree of the string. Thus the overhead of finding and reducing simple phrases is, for the most part, eliminated. Further, this approach can be extended to by-pass parsing completely and generate object code (in the form of quadruples) directly.

7.1 Operator Infix Grammars

The structure of an operator infix expression is determined by the properties of its operators. Usually the set of operators used is partitioned into a number of classes. Each operator class is distinguished by its priority (sometimes called precedence) and its associativity (either right or left).

Thus, a given Operator Infix language will be characterized by n pairwise disjoint operator classes $OP_1,\ldots,OP_n$. All unary operators will be placed in OP_n. Operators in OP_{i+1} will have a higher priority than those in OP_i, OP_{i-1}, etc. Thus, the highest priority operators (the unary ones) are applied first, then the next highest, etc. Further, each operator class, OP_i, has an associativity ASSOC[i]. -1 denotes left associativity, 1 denotes right associativity. By definition, ASSOC[n] = 1 (right). Finally, we shall assume a class $OP_0 = \{\#\}$ is added. # is a null operator used to separate adjacent expressions. We assume ASSOC[0] = -1.

For example, an operator infix language having the operators +,-,*,/,**,SQRT and the usual semantics would be characterized as follows:

$OP_0=\{\#\}$ $OP_1=\{+,-\}$ $OP_2=\{*,/\}$ $OP_3=\{**\}$ $OP_4=\{SQRT\}$

ASSOC[0]=-1 ASSOC[1]=-1 ASSOC[2]=-1 ASSOC[3]=1 ASSOC[4]=1

We may now define an <u>Operator Infix Grammar</u> which generates the Operator Infix language characterized by $OP_0,\ldots$

OP_n and ASSOC[0],...,ASSOC[n] as $G=(V,V_T,P,S)$ where:

$V_T=\{\$,ID,(,)\} \cup OP_0 \cup \ldots \cup OP_n$

$V= V_T \cup \{S,N_0,\ldots,N_{n+1}\}$

P is the union of the following sets of productions:

(1) $\{S \rightarrow \$N_i\$ \mid 0\leq i\leq n+2\}$

(2) $\{N_n \rightarrow \theta N_i \mid \theta\varepsilon OP_n \text{ and } n\leq i\leq n+2\}$

(3) $\{N_{n+1} \rightarrow (N_i) \mid 0\leq i\leq n+2\}$

(4) For $0\leq j\leq n-1$:

If ASSOC[j]=-1

then $\{N_j \rightarrow N_i\theta N_k \mid \theta\varepsilon OP_j \text{ and } j\leq i\leq n+2 \text{ and } j<k\leq n+2\}$

else $\{N_j \rightarrow N_i\theta N_k \mid \theta\varepsilon OP_j \text{ and } j<i\leq n+2 \text{ and } j\leq k\leq n+2\}$

Note that we assume $N_{n+2}\equiv ID$.

<u>Theorem 7.1</u> If G is an OIG then G is unambiguous.

<u>Proof</u>: It can be established that G is LR(1)

□

7.2 A Parallel OIG Parser

In general, a parser serves two functions. It verifies that an input string is, in fact, in L(G) and, given that it is, produces a parse of it. We shall see that it is especially simple to test whether an input is in L(G) for OIG's. Further, given that it is, an efficient parse is possible.

Let Count(x,a)=+\(a=x). Count counts the cumulative number of occurences of a in x. For example, Count(aabaab, a)=1,2,2,3,4,4.

Let $x \epsilon V_T^+$. We will say x has a Well-Formed Parenthesis Structure iff:

(1) $x=\$x'\$$ and $\$ \notin x'$

(2) Count(x',() - Count(x',)) = $c_1, \ldots, c_{|x'|}$ where $c_{|x'|}=0$ and for $1 \leq i < |x'|$ $c_i \geq 0$.

That is, we require balanced parentheses and correct endmarkers. Let UN (the set of unary operators)= OP_n and let BIN (the set of binary operators)= $OP_0 \cup \ldots \cup OP_{n-1}$.
It is easily verified that the follow sets listed below are correct.

Set:	UN	BIN	{(}	{\$}	{)}	{ID}
Follow:	S	S	S	S	S'	S'

where $S \equiv UN \cup \{(,ID\}$ and $S' \equiv BIN \cup \{),\$\}$.

Let $x \epsilon V_T^+$. Then x is Well Formed iff

(1) x has a Well-Formed Parenthesis Structure

(2) x=...ab... => $b \epsilon$ Follow(a).

Theorem 7.2 Let G be an OIG. Then $x \in L(G)$ iff x is Well Formed.

Proof: Clearly, $x \in L(G)$ => x is Well Formed.

Assume x is Well Formed. Then x=\$x'\$. Write x' as ...(x")... where x" contains no "(" or ")"; otherwise, let x"=x'. Using Follow sets, we may determine that x" is of the form $(UN^* X)\ (BIN\ UN^*\ X)^*$ where X = ID or N_{n+1}. Occurences of unary operators can be reduced to obtain the form $Y\ (BIN\ Y)^*$ where Y = ID or N_{n+1} or N_n. Now if x" fits this pattern, then $N_i \xrightarrow{*} x"$ for $0 \le i \le n+2$ (recall $N_{n+2} \equiv ID$). This can be verified by a simple induction on the number of operators in x". Now if x"=x' we are done since $S \to N_i$ is a production. Otherwise, (x") is replaced by N_{n+1} (since $N_{n+1} \to (N_i)$) and the process is repeated.

□

The order of application of operators in an expression depends on a number of factors:

(1) The depth of the parenthesis nesting.

(2) The priority of the operators.

(3) The associativity of the operators.

(4) The position of the operators in the expression.

We shall encode these factors, uniquely, for each operator, in an integer. This integer, the precedence of the operator, is an extention of the concept of Operator Precedence [Fl 63]. Let x'=\$x\$ be a Well Formed string and let INDEX= the positions of the operators in x. Further, let

(1) NEST = Count(x,( ) - Count(x,))

(2) PRIO[i] = If X_i is an operator (where $x=X_1 \ldots X_{|x|}$)
then j where $X_i \epsilon OP_j$ else undefined.

(3) POS[i] = If X_i is the k-th operator then k*ASSOC[j]
where $X_i \epsilon OP_j$ else undefined.

Finally, PREC = POS[INDEX] + 2 * (m+1) * PRIO[INDEX]
+ 2 * (m+1) * (n+1) * NEST[INDEX]
where m=|INDEX| and n = (as usual) the number operator classes.

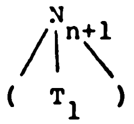
PREC thus encodes the type, position, and parenthesis level of each operator. It should be clear that (given m and n) the encoding is unique, for it can easily be inverted.

As an example, using the same operator classes as before, let x = ID * (ID + ID ** ID) - SQRT ID. Then
INDEX = 2,5,7,10,11; NEST = 0,0,1,1,1,1,1,1,0,0,0,0;
PRIO[INDEX] = 2,1,3,1,4; POS[INDEX] = -1,-2,3,-4,5
m = 5; n = 4; PREC = 23,70,99,8,53.

For each infix expression we may define a derivation tree (called a <u>reduced derivation</u> tree) which involves only ID's and operators. This is done (given the original derivation tree) by:

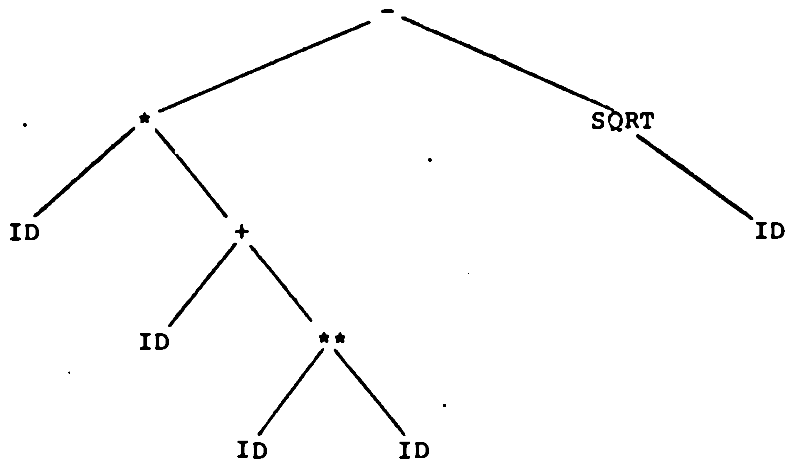
(1) Replacing each N_i $(0 \leq i \leq n)$ by the operator it derives.

(2) Replacing subtrees of the form



by T_1.

Thus, for example, the reduced derivation tree corresponding to the above example is:



Observe that the PREC values have the property that <u>all</u> operators occuring in a subtree rooted by θ have a <u>larger</u> PREC value than that of θ. This, of course, is by no means accidental.

<u>Theorem 7.3</u> Let some operator O_i occur in a subtree rooted by the operator O_j in the reduced derivation tree of x, some operator infix expression.

Then PREC[i] > PREC[j].

<u>Proof</u>: Consider the corresponding derivation tree in G, the OIG used to define the set of valid infix expressions.

If O_i is in a subtree rooted by O_j then

(1) The parenthesis level of O_i is $\geq$ that of O_j.

(2) If the parenthesis levels of O_i and O_j are equal then $O_i \varepsilon OP_k$ and $O_j \varepsilon OP_q$ and $k \geq q$.

(3) If $k=q$ then

(a) If ASSOC[k] = -1 then O_i is to the left of O_j, that is, i<j.

(b) If ASSOC[k] = 1 then O_i is to the right of O_j, that is, i>j.

But these conditions ensure that (by the definition of PREC) that PREC[i] > PREC[j].

□

The problem of finding the reduced derivation tree of an expression is now considered. As will be seen, the following functions are key.

Let Q be a vector of unique integers. We then define:

$SR(i,Q)=\{j \mid i<j\leq|Q|$ and there does not exist k st $(i<k\leq j$ and $Q[k] < Q[i])\}$.

$SL(i,Q)=\{j \mid 1\leq j<i$ and there does not exist k st $(j\leq k<i$ and $Q[k] < Q[i])\}$.

Simply, SR scans right from i, getting the largest set of consecutive integers having Q values greater than Q[i]. SL does the same, scanning left.

We also define

$L(i,Q)=0$ if $SL(i,Q) = \emptyset$, otherwise $j \varepsilon SL(i,Q)$ st $\forall k \varepsilon SL(i,Q)$ $Q[j]\leq Q[k]$.

$R(i,Q) = 0$ if $SR(i,Q) = \emptyset$, otherwise $j \in SR(i,Q)$ st $\forall k \in SR(i,Q)$ $Q[j] \leq Q[k]$.

L (R) simply finds the index in SL (SR) that has the smallest Q value.

The import of these functions is demonstrated in the following.

<u>Theorem 7.4</u> Let $x \in L(G)$ for some OIG and let PREC = the precedence values of the operators in x.

Then

(1) SR(i,PREC) (SL(i,PREC)) = the set of indices of all the operators in the right (left) subtree of the i-th operator.

(2) R(i,PREC) (L(i,PREC)) = the index of the operator that roots the right (left) subtree of the i-th operator, if it exists, otherwise 0.

<u>Proof</u>: (1) Consider SR (SL is analogous). All operators in O_i's right subtree are to the immediate right of O_i and by Thm 7.3 will be included in SR(i,PREC). Let O_j for $j>i$ be the first operator <u>not</u> in O_i's right subtree. If O_i is a descendent of O_j then $PREC[i]>PREC[j]$ and $SR(i,PREC)=\{i+1,\ldots,j-1\}$ as expected. Otherwise, let O_k root the smallest subtree containing both O_i and O_j. Then O_i must be in O_k's left subtree and O_j in O_k's right subtree. But then O_k occurs <u>between</u> O_i and O_j, contradicting the assumption that O_j was the first operator to O_i's right not in its right subtree.

(2) If $SR(i,PREC) = \emptyset$ then no operators are in O_i's right subtree and $R(i,PREC)=0$ as required. Otherwise, the root of O_i's right subtree (by Thm 7.3) is that operator in the subtree with the minimum PREC value. But this is just R(i, PREC). A similar argument holds for L(i,PREC).

□

Observe that L and R give us just what we need to determine the reduced derivation tree, for L(i,PREC) (for example) either points to the root of the left subtree or tells us (by returning 0) that the first ID to the left is the correct subtree.

We now present a parallel OIG parser.

Algorithm 7.5 (A Parallel OIG Parser)

Input: $x \varepsilon V_T^+$.

Output: ROOT, L_ROOT, R_ROOT (encodings of x's reduced derivation tree).

Tables used:

(1) OP_TABLE: A vector of valid operators.

(2) OP_CLASS: The operator classes to which the operators belong.

(3) OP_ASSOC: The asociativities of each operator class.

[1] (Check for Well-Formedness)

If x ≠ \$x'\$ or \$εx'

then signal an error and stop

NEST←(+\("("εx')) - (+\(")"εx'))

```
  If NEST[|x'|] ≠ 0 or (there exists j st 1≤j<|x'| and
      NEST[j]<0) or (there exists k st 1≤k<|x| and x[k+1]∉
      Follow(x[k]))
     then signal an error and stop
[2] (Input is Well-Formed; calculate PREC)
  NUM_OF_OP_CLASSES←|OP_ASSOC| - 1
  OPS_&_IDS←(x'∉{(,)})/x'
  IND_OF_OPS←(OPS_&_IDSεOP_TABLE)/(1,...,|OPS_&_IDS|)
  NUM_OF_OPS←|IND_OF_OPS|
  If NUM_OF_OPS = 0
    then ROOT←-1; stop
  NEST_LEVEL←(x'εOP_TABLE)/NEST
  PREC_LEVEL←OP_CLASS[(OPS_&_IDS[IND_OF_OPS]) LOOKUP OP_TABLE]
  ASSOC_LEVEL←(1,...,NUM_OF_OPS) * OP_ASSOC[PREC_LEVEL]
  PREC←(NEST_LEVEL * 2 * (1+NUM_OF_OP_CLASSES) * (1 +
        NUM_OF_OPS)) + (PREC_LEVEL * (2 * NUM_OF_OP_CLASSES))
        + ASSOC_LEVEL
[3] (Calculate L and R)
  L_VAL←L((1,...,NUM_OF_OPS),PREC)
  R_VAL←R((1,...,NUM_OF_OPS),PREC)
  ROOT←I st PREC[I] = Min(PREC)
[4] (Encode the reduced derivation tree)
    (First, if a value = 0, then insert -1 * the position of
     the ID to the immediate left (or right) of the corres-
     ponding operator)
  L_ROOT←((L_VAL = 0) * -(IND_OF_OPS - 1)) + L_VAL
```

R_ROOT←((R_VAL = 0) * -(IND_OF_OPS + 1)) + R_VAL

(Now zero out values in L_ROOT corresponding to unary operators)

L_ROOT←L_ROOT * (PREC_LEVEL ≠ NUM_OF_OP_CLASSES)

□

The encoding of the reduced derivation tree is quite straightforward. ROOT encodes the root of the tree. An encoding i>0 denotes the i-th operator. It is stored at OPS_&_IDS[IND_OF_OPS[i]]. An encoding i<0 denotes an ID. It is stored at OPS_&_IDS[|i|]. The encodings of the left and right subtrees of the j-th operator are to be found at L_ROOT[j] and R_ROOT[j].

Reconsider the expression used earlier:

x = ID * (ID + ID ** ID) - SQRT ID.

Its PREC value was 23,70,99,8,53.

Now: L_VAL = 0,0,0,1,0; R_VAL = 2,3,0,5,0;

L_ROOT = -1,-3,-5,1,0; R_ROOT = 2,3,-7,5,-10; ROOT = 4.

The reader may verify that the derivation tree is correctly encoded.

<u>Theorem 7.6</u> Let G be some OIG.

Then

(1) If $x \in L(G)$ then Alg 7.5 produces a correct encoding of x's reduced derivation tree in $O(|x|)$ time and space, given that L_VAL and R_VAL are calculated in $O(|x|)$ time and space.

(2) If $x \notin L(G)$ then Alg 7.5 stops and signals failure.

Proof: Clearly, the algorithm always halts.

(1) Correctness: Follows from Thm 7.2 and Thm 7.4.

Linearity: Each step is executed once and is bounded by $O(|x|)$.

(2) Follows from Thm 7.2

□

We now consider the problem of calculating L and R. Let $\overline{L}(Q) = L((1,\ldots,|Q|),Q)$ and similarly for $\overline{R}(Q)$. Clearly, $\overline{L}(Q)$ and $\overline{R}(Q)$ are related, in fact

$$\overline{L}(Q) = ((\overline{R}^{REV}(Q^{REV}) \neq 0) * (|Q| + 1)) - \overline{R}^{REV}(Q^{REV}),$$

where A^{REV} denotes A reversed.

Thus we need consider only methods of calculating $\overline{R}(Q)$. One method (originally suggested by John Hopcroft) is the following.

Algorithm 7.7

Input: A vector Q of unique integers.

Output: $\overline{R}(Q)$ stored in RESULT.

[1] NEG←Some very large negative integer

Stack (NEG,NEG) onto an empty push down stack where doublets are of the form (STK_POINTER,STK_Q)

POINTER←1; LIMIT←|Q| + 1; Q[LIMIT]←NEG

[2] If POINTER > LIMIT

then stop

```
[3] If STK_Q(Top stack element) ≤ Q[POINTER]
      then stack (POINTER,Q[POINTER])
           POINTER←POINTER + 1
          PREV←0
      else RESULT[STK_POINTER(Top stack element)]←PREV
           PREV←STK_POINTER(Top stack element)
           Pop off the top doublet on the stack
[4] Go to [2]
```

□

The algorithm proceeds from left to right. Those indices for which the last (rightmost) element of SR has not yet been reached are stacked. When the last element is reached, its R value has been calculated and can be stored.

It may appear that the above algorithm is too "serial" to utilize parallel environments efficiently. However this need not be the case. Recall that our intention is to parse <u>all</u> infix expressions occuring in a program (or perhaps even a group of programs). Thus the input we shall parse will, in general, be of the form $E_1\#E_2\#\ldots\#E_n$ where the E_i are individual expressions. Since each E_i is independent of the other expressions, the input may be broken into a number of segments, each containing one or more expressions. Algorithm 7.7 may then be run, in parallel, on this set of segments. In fact, this approach is similar to that taken by Zozel, et al [Zo 73] in performing a conventional Operator Precedence parse in parallel.

We now consider an algorithm which is "more parallel" in structure.

Algorithm 7.8

Input: A vector Q of unique integers and a scalar LIMIT ≤ |Q|.

Output: A vector RESULT and a bit vector HAVERESULT.

```
[1] (Encode indices and Q in QI)
  BIG←Some large, positive integer
  QI←((1,...,|Q|) + ((|Q| + 1) * Q)) CONCAT
       (LIMIT REPEAT ( -BIG * (|Q| + 1)))
  MINVAL←|Q| REPEAT (BIG * (|Q| + 1))
  HAVERESULT←|Q| REPEAT 0
  BIGVECTOR←MINVAL
  POS←1
[2] HAVERESULT←HAVERESULT or (QI[1+POS,...,POS+|Q|]
                   < QI[1,...,|Q|])
  COMP_VAL←QI[1+POS,...,POS+|Q|], BIGVECTOR SELECT HAVERESULT
  MINVAL←Min(MINVAL,COMP_VAL)
  POS←POS + 1
  If POS≤LIMIT
    then Go to [2]
[3] (Decode R̄)
  RESULT←MINVAL mod (|Q| + 1)
```

□

The algorithm is actually much simpler than it looks. Since we wish to output indices, but will compare values, we encode the index i and the value Q[i] in QI[i] in such a way that QI[i] < QI[j] iff Q[i] < Q[j]. MINVAL is initialized to an encoding of a very large value of Q and an index of 0. In [2], we compare Q[i+POS] and Q[i], and then compare Q[i+POS] with MINVAL (if HAVERESULT[i]=0). HAVERESULT[i] is set to 1 when we encounter Q[i+POS] < Q[i] (the stopping criterion for SR(i,Q)).

We now characterize the complexities of Algorithms 7.7 and 7.8.

Theorem 7.9

(1) Given input Q, Alg 7.7

(a) Computes $\overline{R}(Q)$ correctly,

(b) in time and space bounded by $O(|Q|)$.

(2) Given Q and LIMIT as inputs to Alg 7.8

(a) RESULT[i] contains $\overline{R}_i(Q)$ if HAVERESULT[i]=1.

(b) The algorithm is bounded in space by $O(|Q|)$ and in time by $O(\text{LIMIT} * |Q|)$.

Proof: May be found in the appendix.

$\square$

One immediate question is how do we ensure that HAVERESULT[i]=1 in Alg 7.8? One way (easily verified) is to set LIMIT = $|Q|$. Does this mean Alg 7.8 is really $O(|Q|^2)$? In practice not. Recall that, in general, the input string will be of the form $E_1\#E_2\#\ldots\#E_n$. If the number of operators in E_i is < k for all E_i then we can set LIMIT = k. Since in practice, individual expressions are fairly short, we can set LIMIT to some reasonably small value (say 25) and obtain a linear algorithm.

The question of how effective Alg 7.5 would be in an actual parallel environment remains unanswered. Clearly, careful implementations and tests are called for. However, preliminary tests are favourable. Using APL\360 as an admittedly crude approximation of a parallel environment, it was found that Alg 7.5 (using Alg 7.8 to calculate L and R) ran an average of four times faster than a serial SLR(1) parser on common arithmetic expressions. This, of course, is by no means conclusive, but it does suggest that more careful tests are indicated.

7.3 Generating Code For OIG's In Parallel

In compilation, parsing is not an end unto itself. Rather, it is closely connected with another phase of compilation - code generation. Since the actual code generated by a compiler depends on the particular computer for which it is targeted, we shall deal with a generalized machine instruction - the quadruple. A quadruple is of the form (OP,A,B,C) and has the semantics C←A OP B.

It is remarkably easy to generate quadruples from the output of Alg 7.5. For each operator O_j, we generate

(O_j, A, B, T_j).

T_j is a temporary associated uniquely with j. A and B are obtained from L_ROOT[j] and R_ROOT[j] as follows:

(1) If L_ROOT[j] (R_ROOT[j]) is an encoding of an ID then A (B) is this ID.

(2) If L_ROOT[j] (R_ROOT[j]) is an encoding of operator O_i then A (B) is T_i.

After the quadruples are generated, they are sorted by the PREC values of the operators, in decreasing order.

For example, reconsidering the expression ID_1 * (ID_2 + ID_3 ** ID_4) - SQRT ID_5 which had L_ROOT = -1,-3,-5,1,0 and R_ROOT = 2,3,-7,5,-10, we generate the following (sorted) quadruples.

(** $,ID_3,ID_4,T_3$)

(+ $,ID_2,T_3\ ,T_2$)

(SQRT, $,ID_5,T_5$)

(* $,ID_1,T_2\ ,T_1$)

(- $,T_1\ ,T_5\ ,T_4$)

These quadruples correctly compute the above expression.

To see that the quadruples generated are always correct, simply note that for any operator:

(1) If an operator's operands are ID's they are clearly obtained correctly. If the operands are expressions, then the result of the expression is expected in T_i where O_i is the root of the expression. But note that T_i is exactly where O_i stores its result.

(2) Once a result is stored in a temporary, it is never changed (each temporary is stored into exactly once).

(3) Sub-expressions are always calculated and stored before they are used (by Thm 7.3 and the sorting requirement).

One possible objection to the above method is its profligate use of temporaries. Clearly, each quadruple does not need its own unique temporary. We therefore seek a method that "reuses" temporaries. As it happens, such a method can be obtained by slightly modifying the above method.

Let $L'(i,Q) = 0$ if $SL(i,Q) = \emptyset$, otherwise $j \epsilon SL(i,Q)$ st

$$\forall k \epsilon SL(i,Q)\ Q[j] \geq Q[k]$$

and $R'(i,Q) = 0$ if $SR(i,Q) = \emptyset$, otherwise $j \in SR(i,Q)$ st

$$\forall k \in SR(i,Q)\ Q[j] \geq Q[k].$$

L' and R' take the index in SL (SR) having the maximum rather than minimum Q value. We calculate L_ROOT and R_ROOT exactly as in Alg 7.5 <u>except</u> that L' and R' are used. The quadruples are generated and sorted as before, except that D (the fourth quadruple argument) is calculated as follows:

Assume a quadruple for the j-th operator O_j is being generated. Then D is T_k where O_k is that operator in the subtree rooted by O_j having the <u>largest</u> PREC value. This means:

(1) If O_j has only ID's as operands then k=j.

(2) If O_j has only <u>one</u> operand that is an expression, then k=L_ROOT[j] (or R_ROOT[j]).

(3) Otherwise, if PREC[L_ROOT[j]]>PREC[R_ROOT[j]] then k=L_ROOT[j] else k=R_ROOT[j].

Reconsidering our favourite example,

ID_1 * (ID_2 + ID_3 ** ID_4) - SQRT ID_5, we now have

L_ROOT = -1,-3,-5,3,0 and R_ROOT = 3,3,-7,5,-10 which yields

(** ,ID_3,ID_4,T_3)

(+ ,ID_2,T_3 ,T_3)

(SQRT, ,ID_5,T_5)

(* ,ID_1,T_3 ,T_3)

(- ,T_3 ,T_5 ,T_3)

Two, rather than five, temporaries are used. In general, the number of different temporaries used will be equal to the number of operators having <u>only</u> ID's as operands. While this method does not always use the minimum number of temporaries necessary, it usually comes rather close.

To see that the above method generates correct quadruples, note that:

(1) As before, if O_j has an operand that is an expression, then, by construction, the value of the expression is stored in the same temporary that O_j expects the result to be in.

(2) Sub-expressions are calculated and stored before they are used as operands (Thm 7.3 and the sorting requirement).

(3) A new value is not stored into a temporary before the current value is used. Assume O_j is a direct descendent of O_i and that O_j stores its result in T_k. O_i will, of course, expect the result in T_k. If another operator O_q $(O_i \neq O_q \neq O_j)$ stores into T_k then O_k must occur in O_q's subtree. But then either O_q is a descendent of O_j (in which case O_q stores into T_k <u>before</u> O_j does) or O_i is a descendent of O_q (in which case O_i uses the result in T_k <u>before</u> O_q stores into it).

We now consider methods of calculating L' and R'. As before, L' and R' are related, so we need only consider algorithms for calculating $\overline{R}$ $(Q)=R'((1,\ldots,|Q|),Q)$.

An algorithm similar to Alg 7.7 can be constructed to generate $\overline{R}'(Q)$.

Algorithm 7.10

Input: Q, a vector of unique integers.

Output: $\overline{R}'(Q)$, stored in RESULT.

```
[1] POINTER←1;  LIMIT←|Q| + 1
    Q[LIMIT]←Some large negative integer
    CURRENT_MAX_PTR←LIMIT
    Stack (LIMIT,LIMIT) onto an empty push down stack where
       doublets are of the form (PTR,MAX_PTR)
[2] If POINTER > LIMIT
      then stop
[3] If Q[PTR(Top stack element)] ≤  Q[POINTER]
      then PT←If Q[CURRENT_MAX_PTR] > Q[POINTER]
                then CURRENT_MAX_PTR else POINTER
           Stack (POINTER,PT)
           POINTER←POINTER + 1
           CURRENT_MAX_PTR←LIMIT
      else RESULT[PTR(Top stack element)] ←
                (CURRENT_MAX_PTR ≠ LIMIT) * CURRENT_MAX_PTR
           CURRENT_MAX_PTR←If Q[CURRENT_MAX_PTR] >
                                Q[MAX_PTR(Top stack element)]
                            then CURRENT_MAX_PTR
                            else MAX_PTR(Top stack element)
           Pop off the top doublet on the stack
[4] Go to [2]
```

□

<u>Theorem 7.11</u> Given input Q, a vector of unique integers, Alg 7.10

(1) Computes $\overline{R}'(Q)$

(2) In time and space bounded by $O(|Q|)$.

<u>Proof</u>: See the appendix.

□

<u>Theorem 7.12</u> Let Alg 7.8 be modified as follows:

(a) In [1] initialize MINVAL by:

MINVAL←|Q| <u>REPEAT</u> (-BIG * (|Q| +1))

(b) In [2] compute MINVAL by:

MINVAL←Max(MINVAL,COMP_VAL)

Then

(1) RESULT[i] contains $\overline{R}'_i(Q)$ if HAVERESULT[i] = 1.

(2) The algorithm is bounded in space by $O(|Q|)$ and in time by $O(\text{LIMIT} * |Q|)$.

<u>Proof</u>: Essentially the same as that of Thm 7.9(2). Simply observe that:

(1) BIGVECTOR and MINVAL (initially) contain large negative integers and thus always compare low.

(2) The maximum over SR(j,Q) is now being accumulated in MINVAL (pardon the misnomer).

□

Finally, note that if the input is of the form $E_1 \# E_2 \# \ldots \# E_n$ where the number of operators in E_i is $\leq k$ for all E_i then either modified algorithm can be run with Alg 7.5

to produce a linear algorithm. Moreover, since the operators in each E_i can be sorted in constant time, quadruples can be generated in linear time and space.

8 CONCLUSIONS

8.1 Summary

We feel this dissertation will help to establish parallel parsing as a viable and fruitful area of investigation. Both theoretical and practical results have been established.

Theoretically, non-canonical parsing has been complemented and extended. In particular:

(1) Two new non-canonical parsing techniques (Total Mixed Strategy Precedence and Synchronous Parsing Machine) have been introduced and studied.

(2) Non-canonical parsing techniques (Total Precedence, Total Mixed Strategy Precedence, Basic and Full SPM) have been converted to perform concurrent reductions, that is to do parallel parsing. Further, efficient (linear) parsers have been presented.

(3) Grammar classes (Strict Precedence, Strict Mixed Strategy Precedence) in which it is possible, for all sentential forms, to recognize and reduce all simple phrases have been defined. The parsers for these classes represent the "ultimate" in parallel parsing in that, at each step, all possible reductions are performed. Thus the number of iterations needed to parse an input is exactly equal to the depth of its derivation tree. Iterated productions are presented as a means of reducing the depth of derivation trees.

(4) It is a well known property of LR parsers that reductions are determined by all of the left context and k characters of the right context. The Full SPM parsing technique has extended this in that arbitrarily distant right context can be returned (via Information Exchanges) to determine reductions. Thus this method exhibits what Szymanski [Sz 73] terms "sequencing"; that is, information is obtained not only from local context but also by the order in which reductions are determined and made.

Many of the results developed in this dissertation have, we believe, practical significance also. As parallel computers are installed, it seems inevitable that parallel compilers will be written (or at the very least attempted). The parallel parsing techniques developed herein should either be of direct use or should form the basis of further research and development. In particular:

(1) A wide variety of techniques have been presented, ranging from the special purpose OIG parser to the very general Basic SPM parser.

(2) With the possible exception of the OIG parser, each of the techniques presented is an extension of a well known serial parsing method. Thus a minimum of new concepts need be assimilated.

(3) Each of the parsers presented is described in considerable detail; further, each is straightforward enough to

suggest that efficient implementations are possible.

(4) Correctness is established for each parsing technique as well as a means of determining whether a given CFG is parsable using the technique.

8.2 Future Work

As suggested earlier, we believe parallel parsing to be a fruitful area of future research. In particular consider the following:

(1) It would be very informative to compare implementations on parallel computers of various parallel and serial parsing techniques. We could then determine the relative merits (in a practical sense) of the various algorithms and develop indicated improvements.

(2) We have considered the extention to parallel environments of but a few parsing techniques. Surely others can be extended and their properties studied.

(3) The problem of integrating a parallel parser into the whole of a parallel compiler must surely be dealt with. Given the output of the parser, how can object code be efficiently generated, how can semantic checking be applied, and how can optimization be performed? The viability of a parallel compiler rests upon the adequate solution of these problems.

(4) The techniques developed for SPM parsing may be of value in the related area of detecting and correcting syntatic errors in programs. Recall that SPM's have the ability to carry, in parallel, a number of state stacks. These state stacks could be associated with a number of proposed corrections to an error which has been detected. Then, as state

stacks are shown to be incorrect and removed, the corresponding suggested correction can be discarded as erroneous.

(5) Recall that, as noted in Chapter 5, the problem of deciding whether a CFG is Full SPM parsable using an LR parsing technique is open if a look-ahead of 2 or more characters is used. We conjecture that the problem is recursively decidable, but this too is open.

(6) The function C' used throughout Chapter 5 is a particular realization of C, the completer function. It was chosen because it was easy to define and prove correct; further, it can be tabled using a moderate amount of space. It would be most interesting to investigate the possibility of obtaining a better realization of the completer function, that is, one that is comparable in space requirements but which includes fewer incorrect states in the set used to bound the correct state. Such a function would be of interest because it is almost always the incorrect states introduced by C' which cause a Full SPM parse to fail. Thus a better realization of the completer function would increase the set of grammars which are Full SPM parsable.

APPENDIX

I A Summary of APL Notation

Let V and W denote vectors ($V=V_1,V_2,\ldots,V_n$; $W=W_1,\ldots,W_m$), let B denote a vector of 0's and 1's (a boolean vector), and let s be a scalar. OP is any binary operator.

(1) V OP $V' = V_1$ OP V_1', V_2 OP V_2' $,\ldots,V_n$ OP V_n'

V OP $s = V_1$ OP s, V_2 OP $s,\ldots,V_n$ OP s

(2) (Subscripting)

$V[W] = V[W_1], V[W_2],\ldots,V[W_m]$

(3) (Compression and expansion)

B/V = V' where |V'| = the number of 1's in B, and

V'[i] = V[j] where the i-th 1 in B occurs in position j.

B\V = V' where |V'| = |B| and

V'[i] = 0 if B[i] = 0;

V'[i] = V[j] if B[i] is the j-th 1 in B.

Thus (1 0 0 1 1 0)/(1 2 3 4 5 6) = 1 4 5 and

(1 0 0 1 1 0)\(1 2 3) = 1 0 0 2 3 0.

(4) (Operator reduction and operator scan)

OP/V = V_1 OP V_2 OP V_3 OP V_4 ... V_n

OP\V = V_1, V_1 OP V_2, V_1 OP V_2 OP $V_3,\ldots,V_1$ OP V_2OP ... OP V_n

Thus +/(1 2 3 4) = 10 and +\(1 2 3 4) = 1 3 6 10.

(5) s <u>REPEAT</u> s' = The vector s',s',s',... of length s.

(6) (Concatenation)

V <u>CONCAT</u> W = $V_1,\ldots,V_n$,$W_1,\ldots W_m$

(7) (Look-up)

V <u>LOOKUP</u> W = V' where $|V'| = |V|$ and V'[i] = j where the first occurence of V[i] in W occurs at position j (j = $|W|$+ 1 if there is no occurence).

Thus 1 3 9 8 <u>LOOKUP</u> 3 8 1 = 3 1 4 2.

(8) (Selection)

V,W <u>SELECT</u> B = V' where V'[i] = V[i] if B[i] = 0

V'[i] = W[i] if B[i] = 1.

Thus (1 2 3 4),(5 6 7 8) <u>SELECT</u> 0 1 1 0 = 1 6 7 4.

II Proof of Lemma 3.13

Assume some character is scanned more than once. Then

(1) The second scan occured on a different iteration than the first <u>or</u>

(2) The second scan occured on the same iteration.
In this case, either

- (2.1) The two scans that scanned the same character started at different positions <u>or</u>
- (2.2) They started at the same position.
  All scans are started for one of three reasons, conditions C1, C2, or C3.
  Then either
  - (2.2.1) The two scans were started at the same position but for different reasons <u>or</u>
  - (2.2.2) The two scans were started at the same position for the same reason.

We consider each of these in turn.

(1) After the first scan we must have had the following sequence of relations: $(>_p)(=_p)^*(>_p \text{ or } *_p)$, otherwise a reduction would have been done. The $>_p$ or $*_p$ at the right cannot be changed to a $<_p$ or a $=_p$ by reductions to its right ($X <_p B$ or $X =_p B$ and $B\lambda Y \Rightarrow X\alpha\lambda^* B\lambda Y \Rightarrow X\alpha\lambda^+ Y \Rightarrow X <_p Y$). Thus if characters are to be rescanned, the second scan must start at the $>_p$ or $*_p$. But new scans are only started because of C1, or C2, or C3 and none

apply.

(2.1) Consider the scan started farther to the left. If it was started because of C1 or C2, it was started at a $>_P$ or a $*_P$. The scan to the right cannot proceed beyond these relations. If the scan started farther to the left was started because of C3, then the character to the immediate right of the starting position has already been scanned (since its RHS.COUNT value $\neq 0$). The scan to the right cannot then scan this by (1).

(2.2.1) This cannot be. The conditions for starting are mutually exclusive.

(2.2.2) This implies the same reduction has been detected and done more than once, since different reductions, using the same reason to start a scan, would start at different positions. But a reduction cannot be detected and done more than once.

Assume it could and consider the first iteration in which it occurs. It cannot be the very first since each scan starts at a different $>_P$ or $*_P$. Consider the two scans which detect the same reduction. They cannot start at different positions by (2.1) and they cannot start at the same position for different reasons by (2.2.1). Finally, they cannot start at the same position for the same reason since this implies some reduction has already been done more than once, contradicting our assumption. □

III Proof of Theorem 5.3

(1) Correctness: Whenever step [2] is executed it is true that:

(*) If x' has been reduced to αXy where Xy has not yet been scanned, then

(1) αXy is a right sentential form.

(2) The top state stack is s' where $s_0 \xrightarrow{\alpha X} s'$.

(3) The right end of the handle is at X or to its right.

The first time [2] is executed, $\alpha=\varepsilon$, $Xy=\$x'\$$ and (*) clearly holds.

Assuming (*) true, we scan X, s' becomes s'' where $s' \xrightarrow{X} s''$. If X is not the right end of the handle, $R(s'', \text{First}_k(y))=0$, we go back to [2] and (*) still holds. If X is the right end, then $R(s'',\text{First}_k(y))=i$, where i is the correct reduction. γ_i is replaced by A_i (the i-th reduction is $A_i \rightarrow \gamma_i$). (*(1)) still holds. Further, we pop off $|\gamma_i|$ states and (*(2) and (3)) still hold. We then return to [2].

But (*) simply says that a right parse is being produced, and when $\$S\$$ is recognized, the parse terminates successfully.

Linearity: Since the total number of characters in the parse tree of x' is $O(|x|)$, the stack depth is bounded by $O(|x|)$ as is the total space requirement.

Further, each step of the algorithm can be done in constant time and each iteration of [2] and [3] can be charged to the character scanned. Each character is scanned exactly once, and an $O(|x|)$ time bound follows.

(2) By cond (2a) of the valid LR definition, if Alg 5.2 signals the reduction $A\to\beta$ then β has just been scanned. Thus if x' is reduced to γ then $\gamma\to^{+}x'$. Thus Alg 5.2 cannot signal success for then $\$S\$$ has been recognized which would imply $x'\in L(G)$. If the algorithm tries to scan past $\$\gamma\$$ (where x' has been reduced to γ) then it will be stopped for $\gamma \neq S$ and by cond (2b) of the valid LR definition there does not exist s st $s_0 \xrightarrow{\$\gamma\$} s$. Further, it cannot loop, for if it did, having scanned x_1 but not x_2 ($x=x_1x_2\$^k$) then it must loop by finding a reduction, scanning its left hand side, etc. forever. Thus we must have an infinite number of δ_i's where $\delta_{i+1}\to\delta_i\to^{*}x_1$. We cannot have $\delta_i=\delta_j$ for $i \neq j$ for then $\delta_i\to^{+}\delta_j$ (or $\delta_j\to^{+}\delta_i$) and since by (2b) of the valid LR definition, $S'\$^k\to^{*}_{rt}\delta_i\beta$, G would be ambiguous, which is forbidden by Thm 5.1. This means there does not exist an n st $|\delta_i|\leq n$ for all i. But Lemma 5.4 contradicts this. Thus the algorithm must, sooner or later, be stopped because it has reached a state that has no successor under the next character to be scanned.

□

<u>Lemma 5.4</u> Let (PA,R) be a valid LR parsing technique for some CFG, G. Then, given x, there exists n st if $\$S\$^{k+1}\to^{*}_{rt}\beta y\$^{k+1}\to^{+}_{rt}xy\$^{k+1}$ then $|\beta|\leq n$.

<u>Proof</u>: Assume otherwise. For any particular y $|\beta|$ is bounded, thus there must be infinitely many y's. Choose two, y_1 and y_2 st $y_1=zw_1$, $y_2=zw_2$ and $|z| = k$. Further, for some m $\$S\$^{k+1} \xrightarrow[rt]{*} \beta zw_1 \xrightarrow[rt]{+} xy_1 \Rightarrow |\beta| \leq m$, and $\$S\$^{k+1} \xrightarrow[rt]{*} \beta' zw_2 \xrightarrow[rt]{+} xy_2$ and $|\beta'| > m$.

But Alg 5.2 cannot parse both xzw_1 and xzw_2 correctly for all decisions on how to reduce x will be made without consulting w_1 or w_2 (since $|z| = k$). Thus either x is reduced to β' (eventually) or it is not, and in either case an incorrect parse will result. But Thm 5.3(1) guarantees a correct parse. Thus such an n must exist.

□

IV Proof of Theorem 5.7(1)

Assume G' is not LR(k) but that G is.

Then in G':

(a) $\$S\$^{k+1} \xrightarrow[rt]{*} \alpha A\ell w \xrightarrow[rt]{} \alpha\beta\ell w \xrightarrow[rt]{*} z\ell w$.

(b) $\$S\$^{k+1} \xrightarrow[rt]{*} \gamma Bx \xrightarrow[rt]{} \gamma\delta x = \alpha\beta\ell y \xrightarrow[rt]{*} z\gamma y$.

(c) $\mathrm{First}_k(\ell w) = \mathrm{First}_k(\ell y) = \ell$.

But $\alpha A\ell y \neq \gamma Bx$ (if G' were LR(k) equality would be required).

Consider the corresponding derivations in G:

(a) $\$S\$^{k+1} \xrightarrow[rt]{*} \alpha' A\ell w \xrightarrow[rt]{} \alpha'\beta'\ell w = \rho'\ell w \xrightarrow[rt]{*} z\ell w$.

(b) $\$S\$^{k+1} \xrightarrow[rt]{*} \gamma' Bx \xrightarrow[rt]{} \gamma'\delta' x = \rho''\ell y \xrightarrow[rt]{*} z\ell y$.

ρ' and ρ'' are $\alpha\beta$ with (perhaps) some non-terminals which derive ε added. Similarly, α' is α with such non-terminals perhaps added, etc.

(1) If $\rho' = \rho''$ then since G is LR(k), we have $\alpha' A\ell y = \gamma' Bx \Rightarrow$ $\ell y = x$, $A=B$, $\alpha' = \gamma' \Rightarrow \alpha \neq \gamma$ (from above) $\Rightarrow \alpha\beta = \gamma\delta$.

Assume $\alpha\beta\ell w \xrightarrow[rt]{*} \alpha z_2 \ell w \xrightarrow[rt]{*} z_1 z_2 \ell w = z\ell w$ and $\gamma\delta x = \gamma\delta\ell y \xrightarrow[rt]{*} \gamma z_4 \ell y$ $\xrightarrow[rt]{*} z_3 z_4 \ell y = z\ell y$. Then $z_1 \neq z_3$ and $z_2 \neq z_4$ (since $\gamma \neq \alpha$ and no ε-productions exist in G').

But $\gamma' Bx \xrightarrow[rt]{} \gamma'\delta' x = \gamma'\delta'\ell y \xrightarrow[rt]{*} \gamma' z_4 \ell y \xrightarrow[rt]{*} z_3 z_4 \ell y$.

Also, $\gamma'\delta' x = \alpha'\beta' x = \alpha'\beta'\ell y \xrightarrow[rt]{*} \alpha' z_2 \ell y \xrightarrow[rt]{*} z_1 z_2 \ell y$.

That is, $\underbrace{\gamma'}_{z_1}\underbrace{B}_{z_2}\ell y$ and $\underbrace{\gamma'}_{z_3}\underbrace{B}_{z_4}\ell y$

Thus G is ambiguous and could not be LR(k), a contradiction.

(2) Thus $\rho' \neq \rho''$.

During the parse of $z\ell y$, the sentential form (sf) $\rho''\ell y$ must occur, as must $\rho'\ell w$ during the parse of $z\ell w$. But since the LR(k) parser cannot see beyond the look-ahead string (ℓ), the sf $\rho'\ell y$ must also occur (that is, the parses are the same until ℓ is scanned). Thus $\rho''\ell y \xrightarrow[rt]{+} \rho'\ell y$ or $\rho'\ell y \xrightarrow[rt]{+} \rho''\ell y$. Without loss of generality, assume the former. The corresponding sf's in G' are both $\alpha\beta\ell y$ and if $\alpha\beta\ell y \xrightarrow{+} \alpha\beta\ell y$ then G' (and G) are ambiguous (by Thm 5.6(3)) and thus not LR(k). Thus the derivation of $\rho'\ell y$ from $\rho''\ell y$ must involve only the use of ε-productions, or the derivation of non-terminals which later derive ε (otherwise, there is a corresponding step in G'). Thus either (A) $\rho'=\rho''\nu$ or (B) $\rho''=\rho'\nu$ where $\nu \xrightarrow{+} \varepsilon$. Assume $\nu \xrightarrow[rt]{*} E \to \varepsilon$.

Then (A) (a) $\rho''\nu\ell w \xrightarrow[rt]{*} \rho''E\ell w \xrightarrow[rt]{} \rho''\ell w$.

(b) $\gamma'Bx \xrightarrow[rt]{} \gamma'\delta'x = \rho''\ell y$.

But if $\rho''E\ell y = \gamma'Bx$ then $\delta'=\varepsilon$ which is impossible since δ' is $\delta \neq \varepsilon$ with (perhaps) some non-terminals added.

So too, (B) (a) $\alpha'A\ell w \xrightarrow[rt]{} \alpha'\beta'\ell w = \rho'\ell y$.

(b) $\rho'\nu\ell y \xrightarrow[rt]{*} \rho'E\ell y \xrightarrow[rt]{} \rho'\ell y$.

If $\alpha'A\ell y = \rho'E\ell y$ then $\rho'=\alpha' \Rightarrow \beta' = \varepsilon$, again impossible for $\beta \neq \varepsilon$.

Thus G' must be LR(k) for if it is not, then G is not, contradicting our original assumption.

□

V Grammar G_6

Consider G_6: $S \rightarrow Ae \mid Bd \mid aAf \mid bBf \qquad A \rightarrow Cd \qquad B \rightarrow \bar{C}d$

$C \rightarrow c \qquad \bar{C} \rightarrow c \qquad S' \rightarrow \$S\$$

G_6 is not SLR(2) because there exists a state containing the LR(0) items $C \rightarrow c.$ and $\bar{C} \rightarrow c.$ which is inadequate. This is because $df \in Follow_2(C)$ and $df \in Follow_2(\bar{C})$. But this inadequate state is reached only in parsing two strings in L(G), cde and cdd, neither of which contains df as a look-ahead. Thus G_6 is Full SPM parsable. But (as the reader can verify) a single Full SPM, trying to parse cdf (not in L(G)) will fail, that is all SPM's in the parse will become inactive without signalling success or failure.

VI Proof of Theorem 5.17

Let $\nu = X_1 X_2 \ldots$ and $\phi = Y_1 Y_2 \ldots$ Let parse A be the one which reduces ϕ' to ν, and parse B be the other. Each character in ϕ corresponds to one in ν, where either it is the same or derived from it. Thus Y_j either $= X_i$ or $Y_j = Z_i^k$ where $X_i \to^+ \delta_i = Z_i^1 \ldots Z_i^k \ldots$ Assume some SPM in parse A has scanned characters $X_i \ldots X_j$. Then S contains stacks of depth 2+j-i. Let $S[k] = \{s \mid (s':s'') \varepsilon S$ and s is the k-th state in $(s':s'')\}$. Then define ${}_AT_m = S[m+1-i]$ for $i \leq m \leq j$ and ${}_AT'_m = S[m+2-i]$. Roughly, ${}_AT_m$ corresponds to the states in the SPM just before X_m was scanned, and ${}_AT'_m$ corresponds to the states in the SPM just after X_m was scanned. Use a similar notation for parse B. We now establish some notation and useful results.

Lemma A.1 If $s'' \varepsilon C'(s,s',p)$ for $p \neq 0$ and $N(s'',a) \neq \emptyset$ then $p \varepsilon R(s',a)$.

Proof: By the definition of C' there exists $B \varepsilon V_N$ st

(1) $\$S\$^{k+1} \to_{rt}^{*} \alpha B a \gamma$ where $s_0 \xrightarrow{\alpha B} s''$ and α is fully reduced.

(2) $B \to \rho_0 B_1 \to \rho_0 \rho_1 B_2 \to \ldots \to \rho_0 \ldots \rho_{n-1} B_n \to \rho_0 \ldots \rho_n = \rho$ where production p is $B_n \to \rho_n$.

(3) $s_0 \xrightarrow{\alpha\rho} s'$.

But since α is fully reduced, so is $\alpha\rho_0 \ldots \rho_{n-1}$. Now $\$S\$^{k+1} \to^{*} \alpha\rho_0 \ldots \rho_{n-1} B_n a\gamma \to \alpha\rho_0 \ldots \rho_{n-1}\rho_n a\gamma$. But then by cond ($1^*$b) of the valid LR definition, $p \varepsilon R(s',a)$

□

It should be rather clear that states occuring during the reduction of δ_i (where $X_i \to^+ \delta_i$) are related to those which are associated with X_i. This relation is established in the following.

We define the <u>fully reduced successor function</u>:

Let $A \to^+ \delta = Z_1 \ldots Z_n$ and assume $N(s,A) \neq \emptyset$. Then $F(s,A,\delta,i) =$

(1) If $i<n$ then let δ' be $Z_1 \ldots Z_i$ fully reduced.

$F(s,A,\delta,i)=s'$ where $s \xrightarrow{\delta'} s'$.

(2) If $i=n$ then $F(s,A,\delta,i) = \bigcup_{q \in \overline{R}(s'')} C'(s,s'',q)$

where $N(s,A)=s''$.

<u>Lemma A.2</u> Let $A \to^+ \delta = Z_1 \ldots Z_n$.

(1) Assume $N(s,A) \neq \emptyset$. Then $F(s,A,\delta,i) \neq \emptyset$ for $i<|\delta|=n$.

(2) Let $F(s,A,\delta,i)=s'$ for $i<n$ and let $N(s',a)=s''$ for $a = Z_{i+1}$. Let γ be $Z_1 \ldots Z_i$ fully reduced, but assume $\gamma a = \gamma' \alpha a$ is not fully reduced, and in fact production $j = B_j \to \alpha a$ should be applied.

Then $C'(s',s'',j) \supseteq F(s,A,\delta,i+1)$.

(3) Let $X_i \to^+ \delta$. Then $F({}_AT_i, X_i, \delta, |\delta|) \supseteq {}_AT_{i+1}$.

(4) Let $X_i \to^+ \delta$. Then $\forall s \in {}_AT_i$, there exists $s' \in {}_AT_{i+1}$ st

$s' \in F(s,X_i,\delta,|\delta|)$.

<u>Proof</u>:

(1) We know $s_0 \xrightarrow{\alpha} s \xrightarrow{A} s'$ where $\$S\$ \xrightarrow{k+1}{}^* \alpha A \rho$ and α is fully reduced. Let $\delta' = Z_1 \ldots Z_i$ when fully reduced. Then $\$S\$ \xrightarrow{k+1}{}^* \alpha\delta'\nu\rho$ and $\alpha\delta'$ is fully reduced. Thus there exists s'' st $s_0 \xrightarrow{\alpha\delta'} s''$.

(2) Assume $\gamma'\alpha a$ can be fully reduced to $\gamma_t B_t$ by applying productions $j,k,\ldots,s,t$, that is, $\gamma_t B_t \to \gamma_t \gamma_s B_s \to \ldots \to \gamma_t \ldots \gamma_j B_j = \gamma' B_j \to \gamma' \alpha a$. Now $C'(s',s'',j) \supseteq \bigcup_{u_1 \in A(s',\alpha)} C'(u_1, N(u_1,B_j),k) \supseteq$

$$\bigcup_{u_2 \in A(u_1,\gamma_j)} C'(u_2,N(u_2,B_k),\ell) \supseteq \ldots \supseteq \bigcup_{u_m \in A(u_{m-1},\gamma_r)} C'(u_m,N(u_m,B_s),t)$$

$$\supseteq \bigcup_{v \in A(u_m,\gamma_s)} \; \bigcup_{q \in \overline{R}(N(v,B_t))} C'(v,N(v,B_t),q).$$

Now $v \in A(s',\gamma_s \ldots \alpha)$ and let $s \xrightarrow{\gamma_t} w \xrightarrow{\gamma_s \ldots \alpha} s' \xrightarrow{a} s''$. Thus $w \in A(s',\gamma_s \ldots \alpha)$.

If $i+1<|\delta|$ then $0 \in \overline{R}(N(w,B_t))$ (since $\gamma_t B_t$ is fully reduced). Thus $C'(s',s'',j) \supseteq C'(w,N(w,B_t),0) = N(w,B_t) = w'$. But $s \xrightarrow{\gamma_t} w \xrightarrow{B_t} w' =$ (by def) $F(s,A,\delta,i+1)$.

If $i+1=|\delta|$ then $\gamma_t=\varepsilon, B_t=A$, $C'(s',s'',j) \supseteq$

$$\bigcup_{q \in \overline{R}(N(s,A))} C'(s,N(s,A),q) = \text{(by def) } F(s,A,\delta,|\delta|).$$

(3) and (4)

By definition, $F(s,X_i,\delta,|\delta|) = \bigcup_{q \in \overline{R}(s')} C'(s,s',q)$

where $N(s,X_i)=s'$. If no SPM is stopped at X_i in parse A, then $0 \in \overline{R}(N({}_AT_i,X_i)) = \overline{R}({}_AT'_i)$ and ${}_AT'_i = {}_AT_{i+1}$. Thus $F(s,X_i,\delta,|\delta|) \supseteq C'(s,s',0) = \{s'\}$ where $s' \in {}_AT_{i+1}$ (since $s \in {}_AT_i$) and (3) and (4) follow.

If an SPM is stopped at X_i then by Thm 5.16(3):

(a) $\forall r \in {}_AT_{i+1}$ there exists $(r'',p:r) \in {}_AI_i$ and there exists

$r'' \epsilon {}_A T_i$ st $N(r'',X_i) = r'$ and $C'(r'',r',p) \supseteq \{r\}$. Thus (3).

(b) $\forall r'' \epsilon {}_A T_i$ there exists $r \epsilon {}_A T_{i+1}$ and there exists $(r'',p:r) \epsilon$ ${}_A I_i$ st $N(r'',X_i) = r'$ and $C'(r'',r',p) \supseteq \{r\}$. Thus (4).

$\square$

We will show, by induction on the number of SPM cycles executed that the following conditions hold, and that the only reductions done are those already done in parse A. Thus parse B will fail.

(1a) If $Y_j = X_i$ then ${}_A T_i \subseteq {}_B T_j$ and ${}_A T'_i \subseteq {}_B T'_j$.

(1b) If $Y_j = Z_i^k$ then ${}_B T_j \supseteq F({}_A T_i, X_i, \delta_i, k-1)$

(2) Assume $Y_j = X_i$ has been scanned and an SPM is stopped at X_i in parse A.

(a) Then an SPM is stopped at Y_j, and ${}_A I_i \subseteq {}_B I_j$.

If an SPM is stopped at Y_j in parse B, but no SPM is stopped at X_i in parse A

(b) Then $\forall s' \epsilon {}_A T_i$ $(s',0:s) \epsilon {}_B I_j$ where $s = N(s',X_i)$.

Assume $Y_j = Z_i^k$ has been scanned and the SPM stops.

Let $U = F({}_A T_i, X_i, \delta_i, k-1)$

$V = F({}_A T_i, X_i, \delta_i, k)$ if $Y_{j+1} = Z_i^{k+1}$; ow ${}_A T_{i+1}$.

(c) Then $\forall s \epsilon U$ there exists $s' \epsilon V$ and there exists $\overline{p}$ st $(s,\overline{p}:s') \epsilon {}_B I_j$.

Also, $\forall s' \epsilon V$ there exists $s \epsilon U$ and there exists $\overline{p}$ st $(s,\overline{p}:s') \epsilon {}_B I_j$.

Base step: (1a and 1b) follow from the initialization step ([1]); (2a,b,c) hold vacuously.

Induction step:

([3], Scan step):

Assume Y_j is scanned.

If A = a (the SPM is active)

then if $Y_j=X_i$ then ${}_A T_i \subseteq {}_B T_j$ implies ${}_A T'_i \subseteq {}_B T'_j$.
Consider the calculation of Q.

1. Assume $Y_j=X_i$.

(i) If no SPM is stopped at X_i in parse A then $0\epsilon Q$ since ${}_A T'_i \subseteq {}_B T'_j$ and $N({}_A T'_i, X_{i+1}) = {}_A T_{i+1} \neq \emptyset$.

(ii) If some SPM is stopped at X_i, then ${}_A I^P_i \subseteq Q$. To see this let $p\epsilon {}_A I^P_i$. Then by Thm 5.16(3), there exists $s"\epsilon {}_A T_i$ and there exists $(s",p:s')\epsilon {}_A I_i$ st $s\epsilon {}_A T'_i$, $s'\epsilon C'(s",s,p)$, $s'\epsilon {}_A T_{i+1}$. Now $s"\epsilon {}_B T_j$ (since ${}_B T_j \supseteq {}_A T_i$) and $s\epsilon {}_B T'_j$ (since $s=N(s",X_i)$). Further, $N(s',Y_{j+1}) \neq \emptyset$ (since $N(s',X_{i+1}) \neq \emptyset$ and $X_{i+1} \to^* Y_{j+1} \ldots$). Thus Lemma A.1 implies $p\epsilon R(s,Y_{j+1}) \Rightarrow p\epsilon Q \Rightarrow {}_A I^P_i \subseteq Q$.

2. Assume $Y_j=Z^k_i$. By Thm 5.13(2), Q contains the correct reduction (and if $Y_{j+1} \neq Z^{k+1}_i$ then $Q \neq \{0\}$ since some reduction must occur (recall $X_i \to^+ \ldots Z^k_i$)).

If A = m, then reconsider the three above cases:

1. $(Y_j=X_i)$

(i) (No SPM is stopped at X_i in A) $0\epsilon Q$ by cond (2b).

(ii) (An SPM is stopped at X_i) ${}_A I^P_i \subseteq Q$ by cond (2a).

2. $(Y_j=Z^k_i)$ Q contains the correct reduction by Thm 5.13(2).

([4], Reduction step)

No reduction involving both X and Z type characters can occur (since it would be ambiguous and only unambiguous reductions are done).

If $Y_j = X_i$ was the last character scanned, then either case (1i) or (1ii) above applies. To do a reduction we require $Q = \{p\} \neq \{0\}$ and stack depth $\geq |\gamma_p|$. Thus case (1i) implies no reduction is done. So consider case (1ii). By Thm 5.16(2), the SPM stopped at X_i had either $|{}_A I_i^P| > 1$ or stack depth $< |\gamma_p|$. But $Q \supseteq {}_A I_i$ and by cond (2a) B's stack depth is $\leq$ A's. Thus no reduction occurs.

If a reduction is done, it involves only Z type characters, and thus already has been done in parse A. After the reduction, cond (1b) still holds (because $F(s,A,\delta,i)$ is independent of reductions done in $Z_{i+1}\ldots$).

Now we consider the case in which an SPM is started (because the current SPM must be stopped). Again consider the three above cases:

(1i) $0 \varepsilon Q$. ${}_A T_i \subseteq {}_B T_j$. $\forall s' \varepsilon {}_A T_i$, $N(s',X_i) = s \varepsilon {}_A T_i' \subseteq {}_B T_j'$. Also $R(s,X_{i+1}) \supseteq \{0\}$. Thus (2b). Further, ${}_B T_{j+1} \supseteq {}_A T_i' = {}_A T_{i+1}$. Thus cond (1a) or (1b).

(1ii) $Q \supseteq {}_A I_i^P$. Since ${}_A T_i \subseteq {}_B T_j$ and ${}_A T_i' \subseteq {}_B T_j'$ we get cond (2a). Further, ${}_A T_{i+1} = {}_A I_i^B$ (where $I^B = \{s' \mid (s,p{:}s') \varepsilon I\}$) and ${}_B T_{j+1} = {}_B I_j^B$. Thus cond (1a) or (1b).

(2) ${}_B T_j \supseteq F({}_A T_i, X_i, \delta_i, k-1)$ by cond (1b). We know Q contains p' the correct reduction.

If $p' = 0$ then $\forall s \epsilon {}_A T_i$ $N(F(s,X_i,\delta_i,k-1),Z_i^k) = F(s,X_i,\delta_i,k)$ and $Y_{j+1} = Z_i^{k+1}$ (else $p' \neq 0$). Thus cond (2c).

If $p' \neq 0$ then $C'({}_B T_j, {}_B T_j', p') \supseteq C'(F({}_A T_i, X_i, \delta_i, k-1),$ $N(F({}_A T_i,X_i,\delta_i,k-1),Z_i^k),p') \supseteq$ (by Lemma A.2(2)) $F({}_A T_i,X_i,\delta_i,k) \supseteq$ (if $k = |\delta_i|$ by Lemma A.2(3)) ${}_A T_{i+1}$. Thus we have the $\forall s'$ there exists s part of cond (2c). Further, $\forall u \epsilon {}_A T_i$, we use Lemma A.2(1) to conclude $F(u,X_i,\delta_i,k-1) = u' \neq \emptyset$ and thus (as above) $C'(u',N(u',Z_i^k),p') \supseteq F(u,X_i,\delta_i,k) \neq \emptyset$ (and if $k = |\delta_i|$ then by Lemma A.2(4) $F(u,X_i,\delta_i,k) \supseteq \{s'\}$ and $s' \epsilon {}_A T_{i+1}$). Thus the $\forall s$ there exists s' part of cond (2c).

And also, if $Y_{j+1} = Z_i^{k+1}$ then ${}_B I_j^B = {}_B T_{j+1} \supseteq$ $F({}_A T_i,X_i,\delta_i,k)$ and thus cond (1b). If $Y_{j+1} = X_{i+1}$ or Z_{i+1}^1 then ${}_B I_j^B = {}_B T_{j+1} \supseteq {}_A T_{i+1}$ and thus cond (1a) or (1b).

Finally, if $Q = \{0\}$ then if $Y_j = X_i$ then ${}_A T_i' \subseteq {}_B T_j'$ implies ${}_A T_{i+1} \subseteq {}_B T_{j+1}$ (since no reduction is done). Thus cond (1a) or (1b). If $Y_j = Z_i^k$ then $Y_{j+1} = Z_i^{k+1}$ (else $Q \neq \{0\}$). Thus ${}_B T_j' \supseteq N(F({}_A T_i,X_i,\delta_i,k-1),Z_i^k) = F({}_A T_i,X_i,\delta_i,k)$ and ${}_B T_j' =$ ${}_B T_{j+1} \supseteq F({}_A T_i,X_i,\delta_i,k)$ and thus cond (1b).

([5], Do IEM's)

Assume that adjacent SPM's C and D execute an IEM and that the last character scanned by C is Y_j.

Consider first Information Exchanges, using the same three cases as above.

(1i) ($Y_j = X_i$ and no SPM is stopped at X_i in parse A)

By cond (2b) $\forall s' \epsilon {}_A T_i$, $(s',0{:}s) \epsilon {}_B I_j$, where $s \epsilon {}_A T_i' = {}_A T_{i+1}$

(since no machine is stopped in parse A). Thus cond (1a) and (1b) imply (2b) (after the exchanges) and also (2b) implies (1a) and (1b) (after the exchange).

(1ii) ($Y_j = X_i$ and an SPM is stopped in parse A)
By Thm 5.16(3) ${}_A T_i = {}_A I_i^A$ and ${}_A T_{i+1} = {}_A I_i^B$. Then cond (1a) and (1b) imply cond (2a) and cond (2a) implies (1a) and (1b) (after the exchange).

(2) ($Y_j = Z_i^k$)
By cond (2c) ${}_B I_j^A \supseteq F({}_A T_i, X_i, \delta_i, k-1) \Rightarrow$ cond (1b). Also, ${}_B I_j^B \supseteq F({}_A T_i, X_i, \delta_i, k)$ (or $\supseteq {}_A T_{i+1}$) implies cond (1b) (or (1a)). Further, cond (1b) implies ${}_B T_j \supseteq F({}_A T_i, X_i, \delta_i, k-1)$ and cond (1b) (or (1a)) implies ${}_B T_{j+1} \supseteq F({}_A T_i, X_i, \delta_i, k)$ (or $\supseteq {}_A T_{i+1}$) and thus we get (2c).

Also note that since cond (1a) or (1b) holds for ${}_A T_{i+1}$ and ${}_B T_{j+1}$, conds (2a or 2b or 2c) will hold for D's interface after the exchange.

Finally, consider a merge (${}_B I_j^P = \{0\}$ or $\emptyset$).
If $Y_j = X_i$, then cond (2a) and Thm 5.16(3) imply no SPM is stopped at X_i in parse A. Thus ${}_A T_i' = {}_A T_{i+1}$. Using cond (1a) and the definition of merge, we may conclude that cond (1a) or (1b) holds after the merge.
If $Y_j = Z_i^k$ then $Y_{j+1} = Z_i^{k+1}$ (else ${}_B I_j^P \neq \{0\}$ or $\emptyset$). Then ${}_B T_j' \supseteq N(F({}_A T_i, X_i, \delta_i, k-1), Z_i^k) = F({}_A T_i, X_i, \delta_i, k)$ and cond (1b) is preserved.

Thus over the entire SPM cycle conds (1a), (1b), (2a), (2b), and (2c) have held and <u>no</u> reduction not already done in parse A has occurred. Parse B must then fail.

□

VII Proof of Theorem 7.9

(1) (a) Consider the i-th element of $\overline{R}(Q) = R(i,Q)$.

Case (i): If $SR(i,Q) = \emptyset$ then $R(i,Q) = 0$ and either $Q[i+1]<Q[i]$ or $i = |Q|$. In this case i is stacked and PREV←0; POINTER←POINTER + 1. Then [3] is executed, the else path is taken, and RESULT[i]←PREV=0.

Case (ii): $|SR(i,Q)| = j>0$. Then $Q[i]<Q[i+1],\ldots,Q[i]<Q[i+j]$ and $i+j=n$ or $Q[i]>Q[i+j+1]$. Let $i+1\le k\le i+j$ and $Q[k] = \mathrm{Min}(Q[i+1],\ldots,Q[i+j])$, that is, $k = R(i,Q)$. But then i is stacked in the algorithm, and maybe others until POINTER=k. Then everything above i is popped from the stack (Q[k] is the minimum) and k is placed just above i in the stack. Since k is the minimum, i is not popped until POINTER = i+j+1. Then it is popped, PREV←k and then i is popped (since $Q[i+j+1]<Q[i]$) and RESULT[i]←PREV=k=R(i,Q).

(b) Clearly the stack is bounded by $|Q|$. Further, each iteration either pushes or pops an element of Q. Since each element is pushed and popped exactly once, the execution time is bounded by $O(|Q|)$.

(2) (a) HAVERESULT[i] is set to 1 the first time that $QI[i+POS]<QI[i] \Rightarrow Q[i+POS]<Q[i]$. Once HAVERESULT[i]=1, MINVAL[i] is unchanged since it is always compared with BIG * $(|Q| + 1)$. Thus MINVAL[i] = Min(BIG * $(|Q| + 1)$, $QI[i+1],\ldots,QI[i+POS-1]$) where $QI[i]<QI[i+1],\ldots,QI[i]<QI[i+POS-1]$, and $QI[i]>QI[i+POS]$. If POS = 1 then MINVAL =

BIG * ($|Q|$ + 1) and then RESULT[i]=0=$\overline{R}_i(Q)$ since in this case SR(i,Q) = $\emptyset$. Otherwise, MINVAL[i] = QI[k] = (Q[k] * ($|Q|$ + 1)) + k where Q[k] is the minimum Q value over SR(i,Q). Thus RESULT[i] = k = $\overline{R}_i(Q)$.

(b) Space is clearly bounded by O($|Q|$) since all vectors are O($|Q|$) in size. Since the loop in [2] is executed LIMIT times, and each iteration takes O($|Q|$), we get a time bound of O(LIMIT * $|Q|$).

$\square$

VIII Proof of Theorem 7.11

First we establish that the following two conditions are true whenever step [3] in Alg 7.10 is executed:

(i) Let i = PTR(Top stack element) and j = POINTER. Then CURRENT_MAX_PTR = q st $Q[q] = \text{Max}(\{Q[k] \mid k = \text{LIMIT or } i<k<j\})$.

(ii) Let p = PTR(Second from top stack element) and n = PTR(Top stack element). Then MAX_PTR(Top stack element) = q st $Q[q] = \text{Max}(\{Q[m] \mid p<m\leq n \text{ or } m = \text{LIMIT}\})$.

(A) The first time step [3] is executed:

i = CURRENT_MAX_PTR = LIMIT and $j = 1$ => (i).

$p = n$ = LIMIT = MAX_PTR(Top stack element) => (ii).

(B) Assume the above is true before the execution of step [3], show it is still true after execution.

(B1) The then path is taken:

(i) qualifies CURRENT_MAX_PTR. We stack MAX_PTR = q st $Q[q]$ = Max(Q[CURRENT_MAX_PTR],Q[POINTER]) => (ii) after stacking. Further, after stacking, PTR(Top stack element) + 1 = POINTER and CURRENT_MAX_PTR = LIMIT => (i).

(B2) The else path is taken:

(i) qualifies CURRENT_MAX_PTR, (ii) qualifies MAX_PTR(Top stack element). CURRENT_MAX_PTR←q st $Q[q] = \text{Max}(\{Q[m] \mid p<m\leq n \text{ or } i<m<j \text{ or } m = \text{LIMIT}\}) = \text{Max}(\{Q[m] \mid p<m<j \text{ or } m = \text{LIMIT}\})$ since $n = i$ = PTR(Top stack element) and after the pop this is cond (i). Further, (ii) still holds

because the elements in the stack haven't been changed.

Now we can establish (1). Clearly, POINTER is stacked only if $Q[\mathrm{POINTER}] \geq Q[\mathrm{PTR}(\text{Top stack element})]$. Consider some index i, $1 \leq i \leq |Q|$. At some point POINTER = i and is stacked. When it is popped POINTER = $j > i$ and $Q[j] < Q[i]$. Further, $Q[i] < Q[i+1], \ldots, Q[i] < Q[j-1]$, otherwise i would have been popped earlier (as well as all elements above it in the stack). But this just means $SR(i,Q) = \{i+1, \ldots, j-1\}$. If $SR(i,Q) = \emptyset$ then $R'(i,Q) = 0$ and since cond (i) is true (and $j=i+1$), CURRENT_MAX_PTR = LIMIT => RESULT[i] = 0. Otherwise, since Q[LIMIT] < any other Q value, cond (i) gives us that

$$\mathrm{RESULT}[i] = k \text{ st } Q[k] = \max_{t \in SR(i,Q)} (Q[t]) = R'(i,Q).$$

(2) Same as the proof of Thm 7.9(1b).

□

REFERENCES

[AD 72] Aho, A., P. Denning and J. Ullman. "Weak and Mixed Strategy Precedence Parsing". JACM 19:2, pp. 225-243. 1972.

[AU 72] Aho, A., J. Ullman. The Theory of Parsing, Translation and Compiling, Vol. I. Prentice Hall. 1972.

[BB 68] Barnes, G., R. Brown, M. Kato, D. Kuck, D. Slotnick and R. Stokes. "The Illiac IV Computer". IEEE Trans, Vol. C-17, No. 8, pp. 746-757. 1968.

[CC 71] Cohen, R. and K. Culik II. "LR-Regular Grammars - an Extension of LR(k) Grammars". IEEE Conference Record of 12th Symposium on Switching and Automata Theory, pp. 153-165. 1971.

[Co 70] Colmerauer, A. "Total Precedence Relations". JACM, 17:1, pp. 14-30. 1970.

[Co 73] Control Data STAR-100 Computer System, Publication number 6025600, Control Data Corporation, Arden Hills, Minnesota. 1973.

[De 69] DeRemer, F. "Practical Translators for LR(k) Languages". Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, Mass. 1969.

[De 71] DeRemer, F. "Simple LR(k) Grammars". CACM, 14:7, pp. 453-460. 1971.

[Do 67] Domolki, B. "Algorithms for the Recognition of Properties of Sequences of Symbols". USSR Computational Mathematics and Mathematical Physics, Vol 5, pp. 101-130. Pergamon 1967.

[Fl 63] Floyd, R. "Syntactic Analysis and Operator Precedence". JACM, 10:3, pp. 316-333. 1963.

[IM 70] Ichbiah, J. and S. Morse. "A Technique for Generating Almost Optimal Floyd-Evans Productions for Precedence Grammars". CACM, 13:8, pp. 501-508. 1970.

[Iv 62] Iverson, K. A Programming Language, John Wiley and Sons. 1962.

[HU 69] Hopcroft, J. and J. Ullman. Formal Languages and Their Relation to Automata, Addison-Wesley. 1969.

[Kn 65] Knuth, D. "On the Translation of Languages from Left to Right". Inf. and Control, 8:6, pp. 607-639. 1965.

[La 74] Lamport, L. "The Parallel Execution of DO Loops". CACM, 17:2, pp. 83-93. 1974.

[Li 70] Lincoln, N. "Parallel Programming Techniques for Compilers". SIGPLAN Notices, Vol 5, No. 10. 1970.

[Mi 73] Millstein, R. "Control Structures in Illiac IV Fortran". CACM, 16:10, pp. 621-627. 1973.

[Sz 73] Szymanski, T. "Generalized Bottom-Up Parsing". Ph.D. Thesis, Cornell University, Ithaca, NY. 1973.

[WB 72] Wulf, W. and C. Bell. "C.mmp - A Multi-mini Processor". Proc. AFIPS Fall Joint Computer Conference, Vol 41 II, pp. 765-777. 1972.

[Wi 69] Williams, J. "Bounded Context Parsable Grammars". Technical Report #58, Computer Sciences Department, University of Wisconsin, Madison, Wisc. 1969.

[Wi 71] Wise, D. "Domolki's Algorithm Applied to Generalized Overlap Resolvable Grammars". Proc. Third Annual ACM Symposium on Theory of Computing, pp. 171-184. 1971.

[WW 66] Wirth, N. and H. Weber. "Euler - A Generalization of Algol and its Formal Definition, Parts 1 and 2". CACM, 9:1, pp. 13-23. CACM 9:2, pp. 89-99. 1966.

[Zo 73] Zosel, M. "A Parallel Approach to Compilation". ACM Symposium on the Principles of Programming Languages. pp. 59-70. 1973.

[Zw 72] Zwakenberg, R. "APL* Language Specification and Compiler Progress". UCID-30052, Lawrence Livermore Laboratory, University of California, Livermore, Ca. 1972.