Polylogarithmic Parallel Parsing of P(k) Languages [*] Technical Report 95-384

James P. Schmeiser [†] David T. Barnard [‡]
Department of Computing and Information Science
Queen's University, Kingston, Canada

June 19, 1995

Abstract

The Bird-Meertens theory of lists and two theorems by Bird are used to develop associative operators for parsing. The theorems guarantee the existence of an associative operator for computing a function if right and left directed reductions meeting certain criteria are known that compute the same function. A new class of languages, the P(k) languages, and their associated grammars are defined. An algorithm for parsing P(k) languages that runs in $O(log^2\ N)$ time using $O(N)$ processors (where N is the size of the input) is described.

Keywords: parsing, parallel, Bird-Meertens formalism, associative operators, log N parsing

1 Introduction

A grammar is a series of rewrite rules that generate a language. Recognition is the problem of determining whether a string is in the language generated by some grammar. Parsing is the problem of determining the series of grammar rules that generates the given string. It is a necessary part of the compilation of programming languages and is well understood in the sequential domain. However, success in adapting the usual sequential algorithms for parallel computers has been limited. Parallel algorithms for parsing languages with restrictive characteristics that can be exploited have succeeded in achieving polylogarithmic running time, but such languages tend to be too restrictive for general use. Attempts have been made at developing parallel parsing algorithms for more general classes of languages but these efforts have resulted in classes of languages with restrictions that make it difficult to develop appropriate grammars. Often, there is no relation to known grammar classes and no clear indication of valid grammar constructs. The algorithms are typically developed in an *ad hoc* manner.

Bird's theory of lists is used as a basis for developing associative operators that specify the parsing problem for a new class of grammars. These associative operators are used as the basis for several parsing algorithms that require feasible numbers of processors. For the P(k) class of languages defined in this paper, a polylogarithmic running time is achieved. The P(k) class of languages is defined and a parallel algorithm for this class is presented that requires a feasible number of processors (linear in the size of the input) and runs in polylogarithmic time. In addition, the algorithm has a strong theoretical basis.

The next section introduces concepts that are used throughout the paper, including a description of relevant portions of Bird's theory of lists. The third section describes the generation of a parsing machine that can parse in both left-to-right and right-to-left manners and a parallel parsing algorithm using this machine is outlined. Section 4 discusses the computation of the associative operator used in the parallel parsing algorithm. Finally, a class of grammars is described for which the parallel parsing algorithm computes the parse in polylogarithmic time using a number of processors that is linear in the size of the input.

[*]This work was supported by the Natural Sciences and Engineering Research Council of Canada, the Information Technology Research Centre of Ontario and Queen's University at Kingston

[†]James Schmeiser is a developer in the IBM Canada Laboratory C/C++ compiler group.

[‡]The preparation of this paper was completed while the second author was on sabbatical at INRIA (Institute National de Recherche en Informatique et en Automatique), Rocquencourt, France.

PARSING ALGORITHMS		
Language Class	Processors Required	Time Required
Regular Languages[17, 9]	$O(N)$	$O(log\ N)$
Subset of Dyck Languages[15]	$O(N/log\ N)$	$O(log\ N)$
Arithmetic Expressions[6, 20]	$O(N)$	$O(log^2\ N)$
Arithmetic Expressions [1]	$O(N/log\ N)$	$O(log\ N)$
NQLL [3]	$O(N)$	$O(log\ N)$
Regular Right Part Languages[13]	$O(N/log\ N)$	$O(log\ N)$
LE(p,q) Languages [14]	$O(N/log\ N)$	$O(log\ N)$
General CFLs [16, 7]	$O(N^8)$	$O(log\ N)$

RECOGNITION ALGORITHMS		
Language Class	Processors Required	Time Required
Input Driven Languages [8]	$O(N/log\ N)$	$O(log\ N)$
Bracket Languages [8]	$O(N/log\ N)$	$O(log\ N)$
Hierarchical Language Specifications [21]	$O(N)$	$O(log^2\ N)$
Deterministic CFLs[10]	$O(N^3)$	$O(log\ N)$
Deterministic CFLs[10]	$O(N^2 log\ N)$	$O(log^2\ N)$
Deterministic CFLs[5]	$O(N^2)$	$O(log\ N)$
General CFLs [16, 7]	$O(N^6)$	$O(log\ N)$

Table 1: Summary of Algorithm Requirements

2 Background

Table 2 summarises known results for parallel parsing and recognition algorithms. Polylogarithmic time recognition and parsing algorithms using a feasible number of processors (*i.e.*, $O(N)$) exist for only small subsets of deterministic context free languages (CFLs). Characterising the class of languages accepted by these parallel parsing algorithms can be difficult. Membership in several classes is not described in terms of some theoretical property but instead on whether the parsing technique will work on the language. Reading input in parallel is not considered; the input is assumed to be present in memory in the form of tokens (i.e., lexical analysis has been performed). This is reasonable since an algorithm for parsing regular languages in polylogarithmic time using a feasible number of processors is a known result [9].

2.1 Terminology

Standard parsing terminology is used throughout and a background in sequential parsing techniques is assumed. A grammar G is a 4-tuple $G = (V_T, V_N, S, \Phi)$ where V_T and V_N are the sets of terminal symbols and nonterminal symbols, respectively. $S \in V_N$ is the starting symbol of the grammar and Φ is the set of productions (grammar rules). A grammar is said to be *reduced* if all symbols in the grammar generate some (possibly null) part of a terminal string in the language, there are no rules of the form $A \rightarrow A$ in Φ and there exists some derivation $S \stackrel{+}{\Longrightarrow} A$ for all A in V_N (see [22] for more information on reduced grammars). It is assumed that all grammars are reduced.

In general, terminal symbols in the grammar are specified using lower-case letters. A symbol which can be either a terminal or a nonterminal symbol is specified using an upper-case letter. In situations where a symbol must be a nonterminal and the context in which the symbol is used does not indicate this, it will be specified. Greek letters (except ε and ω) represent (possibly empty) strings of symbols which may consist of terminal, nonterminal or a mixture of both types of symbols. ε always represents the empty string and ω is reserved to represent the input string. Subscripts on ω indicate the individual tokens of the input string. In general, a subscripted letter indicates a different symbol than the same letter with a different subscript. Roman letters may be used to indicate specific symbols within a string of symbols. For example $A\alpha$ indicates a string that must be at least one symbol in length, the first of which may be a terminal or a nonterminal. Productions are shown in the form $A \rightarrow \alpha$ where A and α are known as the left-hand side (*lhs*) and right-hand side (*rhs*) of the rule, respectively. A padding production, It is assumed that all logarithms are base 2.

2.2 Deques

Since parsing in both left-to-right or right-to-left manners will be considered, deques are used by the algorithms rather than stacks. A deque (*d*ouble *e*nded *que*ue) is a list in which only the head and the tail are directly accessible; essentially, it acts as a stack at both ends (see [12] for more general information on deques). Throughout the paper, deques are described horizontally, in terms of the left and right end. A deque containing α is shown as $[\alpha]$. There are six operations that can be performed on a deque. We define $PUSH_R$, $PUSH_L$, TOP_R, TOP_L, POP_R, POP_L (read push right, push left, top right, top left, pop right, and pop left, respectively) and concatenation in the following manner.

$$\begin{aligned}
TOP_R([\alpha X_1 \ldots X_n]) &= X_1 \ldots X_n \\
TOP_L([X_1 \ldots X_n \alpha]) &= X_1 \ldots X_n \\
PUSH_R(X_1 \ldots X_n, [\alpha]) &= [\alpha X_1 \ldots X_n] \\
PUSH_L(X_1 \ldots X_n, [\alpha]) &= [X_1 \ldots X_n \alpha] \\
POP_R(X_1 \ldots X_n, [\alpha X_1 \ldots X_n]) &= [\alpha] \\
POP_L(X_1 \ldots X_n, [X_1 \ldots X_n \alpha]) &= [\alpha] \\
[\alpha] +\!\!+ [\beta] &= [\alpha\beta]
\end{aligned}$$

The functions are generally used in a pattern matching form and, in general, TOP_R and TOP_L may refer to more than one element.

2.3 Bottom-up Parsing

LR(1) parsing is a left to right bottom-up parsing method using a push-down automaton [11]. A machine called the *characteristic finite state machine (CFSM)* is generated from the grammar and is encoded in tables used by a standard parsing algorithm to simulate the operation of the machine. Symbols are scanned from the left to the right and placed on a stack. When all of the symbols in the right hand side of a rule are on the stack, they are removed and replaced with the nonterminal on the left hand side. This process continues until all of the input has been scanned and only the starting nonterminal remains on the stack, or an error occurs.

An item is of the form $[N \rightarrow \alpha_1 \ldots \alpha_i \bullet \alpha_{i+1} \ldots \alpha_j ; x]$ and represents the partial recognition of $N \rightarrow \alpha_1 \ldots \alpha_j$. The symbols to the left of the dot (i.e., $\alpha_1 \ldots \alpha_i$) have been recognised or *scanned* while those to the right have not. α_{i+1} is called the *desired symbol*. The symbols to the right of the semi-colon are called *lookahead* and represent those terminal symbols that can legally occur immediately after the rule. Each state has a *basis* consisting of *basis items* which represent partially matched items and a *closure* consisting of *closure items* which represnet rules that can be matched starting from this state. Basis items are indicated with a star ($*$) to the left of the item. An item which has no desired symbols is a *reduce* item and represents the complete recognition of the rule. All other items are *shift* items. There are *transitions* between the states representing the matching of a symbol. A transition is said to be a *basis* transition if the symbol on which the transition is made is only the desired symbol of basis items; otherwise, it is said to be a *closure* transition.

RL(1) parsing (*R*ight-to-left with *L*eft-most reductions using one symbol of lookahead) is the right to left analog of LR(1) parsing. A CFSM is generated from the grammar and the algorithm operates in the same way except that the symbols are scanned from the right to the left. Note that the rules are the same as in the LR(1) parse; however, in general the order is different.

2.4 Top-down Parsing

The LR(1) and RL(1) parsing algorithms can be modified to produce the rules in the same order as the LL(1) and RR(1) parsing algorithms [19]. The modifications to the standard algorithms are minimal. A rule list describing the parse subtree below each symbol is built up whenever a reduction is performed. Note that the parse order produced by an LR(1) parse is the same as that produced by an RR(1) parse. Throughout the rest of the paper, it is assumed algorithms presented are directional and analagous algorithms in the opposite direction are assumed. It is further assumed that the appropriate changes for producing the altered parse order are incorporated into such algorithms.

2.5 Parallel Architectures

The algorithms presented are designed for an *SIMD* (*S*ingle *I*nstruction, *M*ultiple *D*ata stream) architecture. Each processor in an SIMD computer has its own data stream but the processors all execute the same instruction on each cycle. Processors can be prevented from performing any instruction through the use of masks. The instructions can be quite complex, and may even include examining a table to determine the next instruction. This allows the individual processors to effectively make different decisions at the same time. SIMD computers are typically massively parallel. It is assumed that the architecture is a *CREW-PRAM* architecture (*C*oncurrent *R*ead-*E*xclusive *W*rite, *P*arallel *R*andom *A*ccess *M*achine) with shared memory. Several processors are allowed to read the same memory location simultaneously, but only one processor may write to a specific memory location at a time.

2.6 Bird's Theory of Lists[1]

This section presents portions of Bird's theory of lists which are subsequently used to direct the search for a parallel parsing algorithm. A pair of theorems guarantees that an associative operator must exist for any function on lists for which left and right reductions are known. These theorems have applications in the area of parsing since recognition and parsing can be seen as operations on lists. The theorems ensure the existence of an associative operator given certain conditions and suggest characteristics of the operator due to type restrictions; however, they do not indicate how to compute the operator.

Applicable portions of Bird's theory of lists are presented. This includes descriptions of left and right reducing operators and the theorems that guarantee the existence of an associative operator given certain conditions. The proofs were originally developed by Richard Bird and published in [2]. The application of these theorems to parsing is then discussed.

2.6.1 Theory of Lists

A list is a finite sequence of values of some specified type, written as a comma-separated sequence enclosed in square brackets. List concatenation is associative and is represented by the $+\!\!+$ operator.

The Bird-Meertens formalism defines a rich set of second-order functionals. One of the most useful is *reduce*, written /. Informally, the effect of applying reduce by an operator to a list is

$$\oplus/\,[a_1, a_2, ..., a_n] = a_1 \oplus a_2 \oplus ... \oplus a_n$$

For operators which are not associative, it is essential to specify the direction of reduction – that is the order in which the operators are applied. Given a (non-associative) binary operator $\oplus$, the operator *left reduce* and the operator *right reduce* can be defined informally as follows:

$$\begin{array}{rcl} \oplus \not\to_e [a_1, a_2, ..., a_n] & = & ((e \oplus a_1) \oplus a_2) \oplus ... \oplus a_n \\ \oplus \not\leftarrow_e [a_1, a_2, ..., a_n] & = & a_1 \oplus (a_2 \oplus ... \oplus (a_n \oplus e)) \end{array}$$

where e is the appropriate identity element. Another useful second-order functional is *map* which applies a function element-wise to a list; in symbols

$$f * [x_0, x_1, \ldots, x_n] = [fx_0, fx_1, \ldots, fx_n]$$

For each function $f : \alpha \to \beta$, a homomorphism h from a list monoid $(\alpha*, +\!\!+, [\,])$ to a monoid $(\beta, \odot, e)$ can be defined by

$$\begin{array}{rcl} h([\,]) & = & e \\ h([a]) & = & fa \\ h(x +\!\!+ y) & = & h(x) \odot h(y) \end{array}$$

[1]Portions of this section were previously published (in a more complete form) in [2]. David Skillicorn was a co-author of [2] with the authors of this paper.

One way to find associative operators that compute interesting functions is given by a theorem of Bird, which guarantees that any function that can be computed both as a left reduction and as a right reduction can also be computed as an associative reduction. This suggests that interesting parallel algorithms can be found when left-to-right and right-to-left sequential algorithms are known.

Theorem 1 *If $\oplus$ and $\otimes$ are operators such that $h = \oplus \not\rightarrow_e$ and $h = \otimes \not\leftarrow_e$ then h is a homomorphism.*

The following theorem (First Homomorphism Theorem [4]) shows that all homomorphisms can be expressed in a canonical, and useful, way.

Theorem 2 *Any homomorphism h from a list monoid to any other monoid can be expressed in the form*

$$h = \odot/ \cdot (h\,[\cdot])*$$

This theorem means that any homomorphism can be expressed as the composition of a reduction with a function applied to all the singletons of the source list. Note that the right hand function really only does type coercion; the *map* applies the parenthesised function to the individual element of the operand list. This theorem shows that any homomorphism can be structured as a parallel algorithm with a logarithmic number of phases, although the total time complexity depends on the complexity of the operation $\odot$.

2.6.3 Applications to Parsing

The theorems guarantee that given appropriate left and right reductions for some function, there exists a way to compute the function using an associative operator. The first theorem states that if some function h is computed using equivalent left and right reductions, then h is a homomorphism. The second theorem states that any homomorphism can be expressed as a reduction using an associative operator. Some initialization function is applied to the singleton values of the list and then a composition involving a reduction is carried out over these values. Initially, the function is applied in parallel to each value in the list. Since the operator is associative, the reduction is accomplished in $O(log\ N)$ phases using $O(N)$ processors. Every second processor combines with its neighbour in parallel, effectively eliminating half of the remaining processors. This is then repeated on the resultant list, and so on. The actual time required for the reduction depends on the time required for computing the associative operator.

Recognition and parsing can be considered reduction problems over a list of tokens. Assume that an algorithm for parsing some language L in a left to right manner is known. This algorithm defines a left reduction. If an algorithm for parsing L in a right to left manner that produces the same result is known (i.e., a right reduction that satisfies the type constraints) then an associative operator for parsing L is guaranteed to exist. An algorithm for parsing L using this associative operator runs in $O(log\ N)$ phases. If the associative operator is computable in polylogarithmic time, the parallel parsing algorithm runs in polylogarithmic time using $O(N)$ processors.

The type restrictions of the first theorem require that the intermediate and final results in both directed reductions be of the same type. Therefore, to find an associative operator for parsing some class of languages, one may start with a known left to right parsing algorithm and then develop a right to left parsing algorithm which satisfies the type restrictions. In some cases, the type of the intermediate and final results in the left reduction may not be suitable for use in a right reduction. In such situations, it may be necessary to alter one or both of the algorithms to meet the type restrictions of the first theorem. When these type constraints are satisfied by the algorithms, the second theorem guarantees the existence of an associative operator for parsing the language. Unfortunately, Bird's theorem does not determine the associative operator. However, its type is specified and this knowledge can often provide significant information about how to compute the operator.

The type of the intermediate result must take into account the order of the rules in the list of rules produced. Although in both reductions a list of rules or rule numbers is output, the functions are not the same if the order of the rule applications is different. For the directed reductions to match, the order of the rule applications in both must be the same.

This methodology of developing parallel parsing algorithms has proven useful. It has previously been successfully applied to several known parsing algorithms, namely those for parsing regular languages, Dyck languages and for recognising simple precedence languages[2].

LR(1) parsing methods are used for a left to right reduction and a modified RL(1) parsing algorithm is used to get a directed reduction from right to left so the class of languages considered is limited to LR(1)$\cap$RL(1). Both reductions are described using a finite state machine generated from the grammar. The two reductions satisfy the conditions of Bird's theorems and an associative operator is guaranteed to exist. The associative operator for LR(1)$\cap$RL(1) is described.

3.1 PCFSM

Bottom-up parsing algorithms typically simulate the operation of a characteristic finite state machine that is generated from the grammar. A *PCFSM (Parallel Characteristic Finite State Machine)* is generated from the grammar and Bird's theorems are shown to be applicable to the computations carried out by the PCFSM. The PCFSM is defined such that it can parse the input string in a left-to-right or right-to-left manner and can represent all the partial parses of any input substring.

The PCFSM consists of a set of states, each state containing a *basis* and *left* and *right closures* that indicate the rules that can be recognised to the left and to the right from the current parsing position, respectively. *Left* and *right transitions* indicate transitions to the left and to the right, respectively.

3.1.1 Double Dotted Items

Since recognition may proceed in either direction in the PCFSM, *double dotted items* are used to specify the parse state information. A double dotted item contains two dots and has contextual information in both directions associated with it. The double dotted item $[A \rightarrow X; \alpha \bullet \beta \bullet \gamma; Y]$ represents a partial recognition of the rule $A \rightarrow \alpha\beta\gamma$ in which β has been recognised. α, the *left frontier* or *left desired symbols*, and γ, the *right frontier* or *right desired symbols*, are those symbols to the left and right, respectively, that have not been recognised. $X, Y \in V_T$ are the sets of terminal symbols that can immediately precede or follow the rule, respectively. The last symbol of α and the first symbol of γ are called the *left desired symbol* and the *right desired symbol*, respectively. The left and right designators are omitted when it will not cause confusion.

An item in which $\alpha = \varepsilon$ and $\gamma = \varepsilon$ is called a *reduce item*, otherwise it is a *shift item*. An item is said to be in a state if it is in either closure or in the basis. A non-basis item in the left or right closure is called a *left* or *right closure item*, respectively. Define $CLOSURE_L(S)$, $CLOSURE_R(S)$ and $BASIS(S)$ to be the left closure of some state S, the right closure of S and the basis of S, respectively.

3.1.2 Construction of a State

Two closures are generated from the basis items. Consider a state which contains $[D \rightarrow Y; \alpha A \bullet \beta \bullet R\rho; X]$. Assuming that the grammar contains the rules $R \rightarrow \phi$ and $A \rightarrow \gamma$, $\left[R \rightarrow \{m | Y\alpha A\beta \overset{*}{\Longrightarrow} \ldots m\}; \bullet \bullet \phi; \{n | \rho X \overset{*}{\Longrightarrow} n \ldots\}\right]$ and $\left[\{m | Y\alpha \overset{*}{\Longrightarrow} \ldots m\}; \gamma \bullet \bullet; \{n | \beta R \rho X \overset{*}{\Longrightarrow} n \ldots\} \leftarrow A\right]$ are added to the right and left closures, respectively. These items may, in turn, cause items to be added to the closure sets, the process continuing until no more can be added.

3.1.3 Starting States

The construction of the PCFSM begins with a starting state 0 containing two items, $[S' \rightarrow; \bullet\bullet \hookrightarrow S \hookleftarrow;]$ and $[; \hookrightarrow S \hookleftarrow \bullet\bullet; \leftarrow S']$, allowing the machine to parse in either directions. The closure operation is performed and directed transitions are made, creating new states if they do not already exist.

The PCFSM cannot yet describe a parse that starts in the middle of the input string since all basis items have at least one empty frontier. There may be unmatched symbols on both frontiers when a processor starts parsing in the middle of the input. The machine is expanded so that a processor can start parsing in either direction from an arbitrary point in the input by adding *parallel starting states* that represent states that could occur at either end of the deque. They are represented as having a negative state number.

The basis set of a parallel starting state is constructed from each shift state by advancing the first dot in each basis item forward until it is adjacent to the second dot. The closure operation is performed in both directions, and transitions are made, possibly creating more states. These new states may in turn also spawn more starting states finite number of rule and context combinations.

It is possible for inadequate states to exist in the PCFSM even though the grammar used to generate a PCFSM is in LR(1)∩RL(1). The bi-directional nature of the starting states results in inadequate states in which there are reduce/reduce conflicts or shift/reduce conflicts with the items having non-empty frontiers in the other directions. Reduction on such a state fails. Upon attempting a reduction it is known that the parsing situation is artificial and could not occur in a sequential parse so the configuration is ignored. The state transition functions $\xrightarrow{a}$ and $\xleftarrow{b}$ for the PCFSM are defined as a transition to the right on a and a transition to the left on b, respectively. Transitions in shift-reduce situations are undefined.

3.1.4 Redundant States

The PCFSM is actually a superset of the set of states necessary for parallel parsing. It will compute representations for the partial parse in both directions; however, only one of these is necessary since the two compute the same result. The left-to-right direction is chosen to remain as the base of the parse as it is more familiar. Therefore, the basis set of state 0 consists of only $* \ [S' \rightarrow; \bullet\bullet \hookrightarrow \ S \ \hookleftarrow;]$. In addition, the direction in which a state is reached is taken into account when computing the closure functions and transitions from the state. If a state is only reached in one direction, only the corresponding closure set is computed and transitions are only made in this direction.

3.2 Associative Operator $\odot$

Parsing in either direction using the complete PCFSM can be performed using adaptations of standard bottom-up parsing algorithms. However, the obvious adaptations do not, in general, produce matching partial results since different parse orders are produced. The LR(1) parse order corresponds to a top-down parse order in a right-to-left parse so the right-to-left reduction is defined such that a top-down parse order is produced (see [18, 19] for more information on producing a top-down parse order from bottom-up parsing algorithms). Bird's theorems are applicable and an associative operator for parsing these languages is guaranteed to exist[2].

Corollary 3 *An associative operator $\odot$ exists for parsing languages in $LR(1) \cap RL(1)$.*

3.3 Parsing Algorithm

The associative operator operates on a pair of sets of deques. Each deque, called a *configuration*, represents a possible partial parse tree and *spans* the portion of the input that it describes and *comprises* those symbols on which transitions have been made. A set of configurations represents all of the possible partial parse trees for the spanned input. An attempt is made to combine each configuration from one set with each configuration from the other. Only those describing similar parsing situations will combine to form a new set of configurations representing all possible partial parse trees for the increased input substring.

3.4 GF(k) Grammars

The number of configurations within a set can grow to a size that is proportional to the size of the input. Consider the grammar given in Figure 1. The problem centres on the local ambiguities inherent in the symbol x. It is used as both an opening and a closing bracketting symbol and without enough context to determine the function of each x, all possibilities must be represented. Consider the input string "a a x x x x b b" in which the first two x's are closing bracketting symbols while the last two are opening bracket symbols. However, if the configuration only spans the four x's, there are five different possible configurations. Each x added to the symbols spanned by the set will result in another configuration being added to the set. Thus, the number of configurations in a set is $O(N)$.

The inherent restrictions in the LR(1)∩RL(1) class of languages limit the situations in which such growth in the number of configurations can occur. A symbol M is said to be *middle recursive* when $M \overset{*}{\Longrightarrow} A\sigma M\phi B$. σ and ϕ are called *opening* and *closing bracket symbols*, respectively. Note that the middle recursive symbol can be in the form of several symbols which produce the same effect. It can be shown that growth only occurs when the opening and closing bracket symbols of middle recursive symbols are similar.

[2]Directed reductions that satisfy the conditions of Bird's theorems are given in[18].

$$\begin{array}{lll}
0. & S' & \rightarrow A \hookleftarrow \\
1. & A & \rightarrow Y\ Z \\
2. & Y & \rightarrow a\ Y\ x \\
3. & & \mid\ \varepsilon \\
4. & Z & \rightarrow x\ Z\ b \\
5. & & \mid\ \varepsilon
\end{array}$$

Figure 1: Example Problem Grammar

Define the following relation on the symbols of the grammar.

$$\begin{aligned} FRONT_k(\alpha) &= \left\{\beta \mid \beta \in V_T^*, \alpha \overset{*}{\Longrightarrow} \beta\gamma, |\beta| = k\right\} \\ &\cup \left\{\beta \mid \beta \in V_T^*, \alpha \overset{*}{\Longrightarrow} \beta, |\beta| < k\right\} \end{aligned}$$

$FRONT_k(\alpha)$ is a typical definition of first sets on strings and is the set of strings of tokens of length k that are left-most produced by α. $BACK_k(\alpha)$ is defined to be the analogous right to left function and is the set of strings of tokens of length k that are right-most produced by α.

To detect possible growth, each pair (A, B) of middle recursive symbols is examined. There is a potential growth situation when $X \Rightarrow \alpha_1 \Rightarrow \cdots \Rightarrow \alpha_n \Rightarrow \mu A \phi B \psi$, $A \notin \alpha_i, 1 \leq i \leq n$. This avoids misinterpreting closing bracket symbols as the non-closing bracket symbols ϕ. There is no potential for growth with A and B when the left-most k closing symbols are different from the left-most k symbols of ϕ, therefore the $CLOSE_k$ function captures the left-most k symbols of the closing symbols of A is defined.

$$CLOSE_k(A) = \left\{ \gamma \;\middle|\; \begin{array}{rcl} \gamma &=& FRONT_k(\beta_n \ldots \beta_1), \\ |\gamma| &=& k, \\ A &\Rightarrow& \alpha_1 X_1 \beta_1 \\ &\Rightarrow^*& \alpha_{(n-1)} X_{(n-1)} \beta_{(n-1)} \\ &\Rightarrow& \alpha_n A \beta_n, \end{array} \right\}$$

These strings are compared with $FRONT_k(\phi B)$. If there is no symbol X in the grammar that results in a non-empty intersection between $CLOSE_k(A)$ and $FRONT_k(\phi B)$, there can be no growth because of A and B. Define $OPEN$ to be the analogous test in the other direction.

A grammar in LR(1)$\cap$RL(1) is *GF(k) (Growth Free within k symbols)* and cannot experience growth in the number of configurations when the following holds for each pair of middle recursive symbols (A, B) in the grammar.

$$\begin{array}{l} \forall X \mid X \Rightarrow \alpha_1 \Rightarrow \cdots \Rightarrow \alpha_n \Rightarrow \mu A \phi B \psi, \\ A \notin \alpha_i, B \notin \alpha_i, 1 \leq i \leq n, \\ ((CLOSE_k(A) \cap FRONT_k(\phi B) = \varphi) \vee \\ \qquad (OPEN_k(B) \cap BACK_k(A\phi) = \varphi)) \end{array}$$

It is easily seen that a grammar that is GF(k) is also GF($k+1$); however, a grammar that is GF($k+1$) is not necessarily GF(k). A GF(k) language is a language generated by a GF(k) grammar.

3.5 Parsing Algorithm

The parallel parsing algorithm, shown in Figure 2, works in the following manner. End markers are added to the input string and a processor is assigned to each token ω_i. In parallel, the processors compute the initial set

```
for $i \leftarrow 0$ to $N$ do in parallel
    if $i = 0$
    then $P[i] \leftarrow h \hookrightarrow$
    else if $i = N$
        then $P[i] \leftarrow h \hookleftarrow$
        else $P[i] \leftarrow h\omega_i$
parallel_reduce$([P_0 \ldots P_N], \odot)$
return $([([\,], 0, L)] \in P[0])$
```

Figure 2: Parsing Algorithm

of configurations for the token. The algorithm then carries out the well known logarithmic combining process, represented by the call to *parallel_reduce*. $\circledcirc$, called *join*,is defined to handle the joining of a pair of configurations. The associative operator, $\odot$, operates on two sets of configurations and applies $\circledcirc$ on all the pairs of configurations consisting of a configuration from each set.

The time required by the algorithm is $O(X * log\ N)$ where X is the time required to compute $\odot$.

3.6 Initialisation Function

The initialisation function h operates on singleton tokens and creates the initial sets of configurations. It is based on two operators, $\oplus$ and $\otimes$, which are the operators for the left-to-right and right-to-left directed reductions, respectively. They represent all of the actions that an LR(1) or RL(1) parser[3] (respectively) would perform on receiving the next token (i.e., all actions up to, and including, shifting the token).

$$\begin{aligned} h\omega_i &= \{[([\,], S, [\,])]\,|S\ is\ a\ starting\ state\} \oplus \omega_i \\ &\cup \quad \omega_i \otimes \{[([\,], S, [\,])]\,|S\ is\ a\ starting\ state\} \end{aligned}$$

4 Computing $\odot$

The computation of $\odot$ is significant in the timing of the parsing algorithm. Initialisation takes $O(log\ N)$ time and the parallel combining process will result in $O(log\ N)$ parallel sets of computations of $\odot$. The number of configurations in a set cannot grow because of the GF(k) restriction so the algorithm requires $O(X\ log\ N)$ time, where X is the time required to compute $\odot$. There are two steps in computing $\odot$. First, delayed reductions must be performed. Reductions are delayed when a configuration spans an entire rule but the lookahead symbol required for the reduction is spanned by the neighbouring configuration. After computing the delayed reductions, the configurations are joined. Therefore, the time required to compute $\odot$ is that needed to compute the delayed reductions plus that needed to join the configurations.

The way that two configurations combine depends on the shapes of the parse trees described by the configurations. A configuration in which the starting state is the left-most state in the deque is called a *right tree* while a configuration with the starting state at the right end is called a *left tree*. A configuration with the starting state in the middle of the deque is called a *tent*. A tent has a *left* and *right* side where a *side* is that portion of the deque that is between the starting state and the respective edge. The starting state in a tent is considered to be in both sides effectively making a tent a left and right tree that share a starting state.

When applying the associative operator, the sets of configurations being combined span adjacent substrings of the input. Those configurations from the set to the left are referred to as *left* configurations while those from the right are called *right* configurations.

There are several significant entries in the deque. Define $SS(D)$ to be a function which returns the starting state of a configuration. The *left edge* and *right edge* of a configuration are the states at the the respective end of the deque describing the tree. The right edge of a left configuration and the left edge of a right configuration are called *inner* edges. The *left edge symbol* and the *right edge symbol* are the left-most and right-most symbols, respectively, of the

[3]Recall that the RL(1) parsing algorithm is modified to produce a top-down parse order

configurations. Let $D = \alpha SS(D)\beta$ be a deque that spans $\omega_i \ldots \omega_j$ and define $LE(D) = \alpha SS(D)$, $RE(D) = SS(D)\beta$, $LES(D) = \omega_i$, and $RES(D) = \omega_j$, read *left-edge of D, right-edge of D, left-edge-symbol of D* and *right-edge-symbol of D*, respectively.

4.1 Delayed Reductions

A configuration may span all the symbols of a rule but reductions are delayed since the lookahead needed is in a neighbouring configuration. The first step in joining two configurations is resolving these delayed reductions, including any delayed null reductions in the left set. The lookahead symbol for these reductions is the left-edge symbol of the configurations in the right set. There may also be delayed reductions in the right configuration. However, reducing delayed null reductions in the right configuration will cause a duplication of rule applications. Consider the situation where a null reduction is expected on the inner edge of a configuration. Reductions are performed on the left configuration and the null reduction is performed among these reductions. The null production is also delayed on the left edge of the right configuration. If all delayed reductions were performed, this reduction would be performed twice. To avoid this, as many reductions in the right configuration are performed until a null production is indicated. A configuration is removed from the set if it does note have sufficient depth to perform a chain of reductions.

Define two operators $\Uparrow_R (C, t)$ and $\Uparrow_L (C, t)$ which operate on a configuration and a lookahead symbol and compute delayed reductions in the left and right configurations, respectively. $\Uparrow_L$ is defined such that the production reduced is not a null production.

4.2 Algorithm for Reductions

The first step in joining two configurations is to perform reductions that are delayed because of unavailable lookahead. Computing the delayed reductions in polylogarithmic time is accomplished in parallel by viewing the reductions as chains to be linked together.

All delayed reductions, including those associated with null productions, are recognised in the left configuration while only those reductions up to, but not including, the first null production are recognised in the right configuration. Similar methods are used in both situations but the discussion will center on the method for recognising all delayed reductions, including those associated with null productions.

The algorithm for computing the delayed reductions in $O(log\ N)$ time and the data required are shown in Figure 4 and Figure 3, respectively. The algorithm determines which entries are to be popped from the deque and the rules that are applied. It is assumed that concatenation of strings is performed in constant time across the entire string since $O(N)$ processors are available. As well, it is assumed for the sake of clarity that the table entries that the initial chains are read from do not have entries for states that have inadequacies. This will result in such reductions failing, which is the appropriate action.

The elements in the various data structures used by the algorithm are related to the states on the right side of the deque. A reduction cannot extend past the starting state of the deque so the starting state, for the purposes of the algorithm, is considered to be the left-most state in the deque. The values in the arrays at index i refer to the state at position i, with the states in the deque being numbered from right to left. Therefore, the top right state of the deque is numbered 0 while the starting state is numbered $N - 1$, there being N states being considered. This reverse ordering of the states was chosen to simplify the description of the algorithm.

The values in the array $CHAIN$ describe the chains of reductions and there are two different data structures that can exist within $CHAIN[i]$. The first, a triple (P, d, Q), represents an incomplete chain of reductions, starting at state i and extending to the left to state d. This chain is expecting some symbol Q and produces some symbol P. If $i = 0$ (*i.e.*, it is the top right state), Q is ε, otherwise Q is a nonterminal symbol. P is always a nonterminal symbol. The triple is interpretted as meaning that state i has a basis item that has $Q\sigma$ for right desired symbols. $\sigma \overset{*}{\Longrightarrow} \varepsilon$ with the lookahead symbol and the reductions to the right of this state must reduce to Q for this chain to be valid. Consider Figure 5 which represents the chain of reductions (P, d, Q). The dotted portion represents several repetitions. The chain is made of several links, the link at the top representing the left end of the chain. It indicates that if Q is produced by the reductions that occur lower in the configuration (to the right in the deque), ϕ_i will produce ε with the lookahead symbol. This will result in A_i, which is the expected symbol of the next link. The symbols in $\phi_{(i-1)}$ will produce null, and the process repeats, until the end of the chain is reached. At the end, state

$$TC[S,t] = \begin{cases} \{(P, |\alpha\beta|, \varepsilon)\} & \text{if } [P \rightarrow w; \alpha \bullet \beta \bullet \gamma; t] \in BASIS(S),\ \gamma \stackrel{*}{\Longrightarrow} \varepsilon \\ \{(0, \varepsilon)\} & \text{if } [P \rightarrow p; \alpha \bullet \beta \bullet \gamma; q] \in BASIS(S),\ \gamma \stackrel{*}{\Longrightarrow} t\sigma \end{cases}$$

$$TR[S,t] = \begin{cases} [\alpha\ R] & \text{if } [N \rightarrow w; \bullet\beta \bullet \gamma; t] \in BASIS(S), \\ & \quad \gamma \stackrel{*}{\Longrightarrow} \varepsilon \textit{ using rules } \alpha,\ N \rightarrow \beta\gamma \textit{ is rule } R \\ \varepsilon & \textit{otherwise} \end{cases}$$

$$MC[S,t] = \left\{ (P, |\alpha\beta|, Q) \mid \begin{array}{l} [P \rightarrow w; \alpha \bullet \beta \bullet \gamma; t] \in BASIS(S), \\ \gamma \stackrel{*}{\Longrightarrow} Q\phi \stackrel{*}{\Longrightarrow} Q,\ Q \in V_N \end{array} \right\}$$

$$MR[S,Q] = \begin{cases} [\alpha\beta\gamma\ R] & \text{if } [N \rightarrow p; \bullet\phi \bullet A\delta; q] \in BASIS(S), \\ & \quad A \stackrel{*}{\Longrightarrow} Q\sigma \textit{ using rules } \beta,\ \sigma \stackrel{*}{\Longrightarrow} \varepsilon \textit{ using rules } \alpha, \\ & \quad \delta \stackrel{*}{\Longrightarrow} \varepsilon \textit{ using rules } \gamma,\ N \rightarrow \phi A\delta \textit{ is rule } R \\ \varepsilon & \textit{otherwise} \end{cases}$$

$$EC[S,t] = \left\{ Q \mid [P \rightarrow w; \alpha \bullet \beta \bullet \gamma; v] \in S\ \wedge (\gamma \stackrel{*}{\Longrightarrow} Q\phi\ t\sigma \stackrel{*}{\Longrightarrow} Q\ t\sigma) \wedge (Q \in V_N) \right\}$$

$$ER[S,Q] = \begin{cases} [\alpha\beta\gamma] & \text{if } [N \rightarrow p; \mu \bullet \phi \bullet A\delta; q] \in BASIS(S), \\ & \quad A \stackrel{*}{\Longrightarrow} B\lambda t\rho,\ \lambda \stackrel{*}{\Longrightarrow} \varepsilon \textit{ using rules } \gamma, \\ & \quad B \stackrel{*}{\Longrightarrow} Q\sigma \textit{ using rules } \beta,\ \sigma \stackrel{*}{\Longrightarrow} \varepsilon \textit{ using rules } \alpha \\ \varepsilon & \textit{otherwise} \end{cases}$$

$$Produces[S,Q] = \begin{cases} A & \text{if } [N \rightarrow p; \alpha \bullet \beta \bullet B\gamma; q] \in BASIS(S), \\ & \quad B \stackrel{*}{\Longrightarrow} A\delta \stackrel{*}{\Longrightarrow} Q\delta \\ \varphi & \textit{otherwise} \end{cases}$$

Figure 3: Definition of Tables for *reduce* Algorithm

d, the symbol produced is P. This chain could link with another incomplete chain at either end or with a chain ending in a shift at the left end, which is the other data structure that can exist in the set in $CHAIN[i]$.

A chain ending in a shift is represented by a pair representing the left end of a chain of reductions. After performing the reductions in the chain, making a transition on the nonterminal symbol produced by the reductions and making a series of null reductions, the lookahead becomes the desired symbol. This situation is represented by a pair (d, Q) where d is the left end of the chain. This state has an item with $Q\gamma$ for right desired symbols such that $\gamma \stackrel{*}{\Longrightarrow} \mu t\rho \stackrel{*}{\Longrightarrow} t\rho$. The chain of reductions has reached its left end since after these reductions are performed, the lookahead symbol is the right desired symbol and would be shifted onto the deque. A chain represented by a pair is represented in Figure 6. Again, the dotted portion represents several repititions. The end of the chain doesn't produce a nonterminal; instead, some (possibly 0) number of reductions are performed and then the lookahead symbol is t, terminating the chain of reductions.

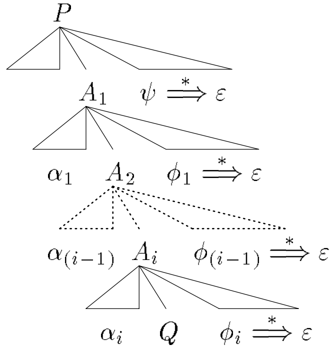
The value in $LINK[i]$ is non-ε only when state i is a reduce state in the chain of reductions that has the inner top state as its right end. $LINK[i]$ is the symbol that is the desired nonterminal in the item that forms a link in the chain.

There may be several chains originating from any state. These chains represent possible reductions but there can only only be one chain originating from the inner top state. If there were more than one chain, this would indicate a reduce-reduce conflict in the grammar. Since the grammar is LR(1), this is impossible. This lone chain is extended to the left and the left extreme is found. Only those i involved in this chain have non-ε values for $LINK[i]$. These values in $LINK$ indicate the nonterminal produced at each stage in the chain which is sufficient to determine the rules in the reduction chain. After the transition on the symbol, any remaining desired symbols in the item must be nullable so the nonterminal must determine the rule that is recognised for the grammar to be LR(1).

Null reductions are only performed on the inner half of the left deque. This avoids performing any delayed null reductions in both configurations. The algorithm for delayed reductions without nullable reductions functions in a similar manner except that when the chains are initially set up, chains are only created for those states that have a single nonterminal expected; no null reductions following the expected nonterminal are allowed. As well, the computation of the final rule list differs in that it is the rules associated with each of the symbols in a rule followed

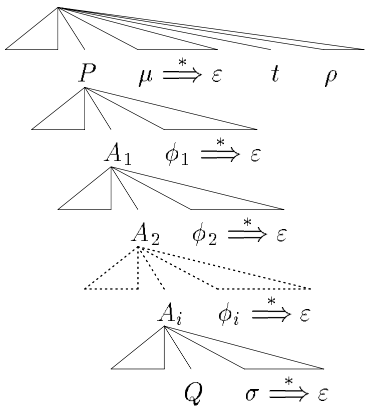
```
$reduce(D[N-1 \ldots 0], N, LookAhead, NewDeque, Size)$

  for $i \leftarrow 0$ to $N-1$ do in parallel
    if $i = 0$
    then $CHAIN[i] \leftarrow TC[D[i], LookAhead]$
    else $mid \leftarrow MC[D[i], LookAhead]$
      $C1 \leftarrow \{(P, d+i, Q) \mid (P, d, Q) \in mid\}$
      $C2 \leftarrow \{(i, Q) \mid Q \in EC[D[i], LookAhead]\}$
      $CHAIN[i] \leftarrow C1 \cup C2$
  for $i \leftarrow 0$ to $N-1$ do in parallel
    $LINK[i] \leftarrow \varepsilon$
  if $CHAIN[0] = \varphi$
  then for $i \leftarrow 0$ to $N-1$ do in parallel
      $NewDeque[i] \leftarrow D[i]$
    $Size \leftarrow N$
  else for $j \leftarrow 1$ to $\lceil log\ N \rceil$
      for $i \leftarrow 0$ to $N-1$ do in parallel
        $NC[i] \leftarrow \varphi$
        for $(P, d, Q) \in CHAIN[i]$ do
          if $d < N$
          then if $(A, x, P) \in CHAIN[d]$
            then $NC[i] \leftarrow NC[i] \cup \{(A, x, Q)\}$
              if $LINK[i] = Q$
              then $LINK[d] \leftarrow P$
            if $(B, P) \in CHAIN[d]$
            then $NC[i] \leftarrow NC[i] \cup \{(B, Q)\}$
              if $i = 0$
              then $LINK[d] \leftarrow P$
        $NC[i] \leftarrow \{(S, Q) | (S, Q) \in CHAIN[i]\}$
      for $i \leftarrow 0$ to $N-1$ do in parallel
        $CHAIN[i] \leftarrow NC[i]$
    if $CHAIN[0] = (P, i, Q)$
    then $Size \leftarrow -1$
    else $(S, P) \leftarrow CHAIN[0]$
      for $i \leftarrow 0$ to $N-1$ do in parallel
        if $i = 0$
        then $Rules[i] \leftarrow TR[D[i], LookAhead]$
        else if $i = S$
          then $Rules[i] \leftarrow ER[D[i], Q]$
          else if $LINK[i] \neq \varepsilon$
            then $Rules[i] \leftarrow MR[D[i], LINK[i]]$
            else $Rules[i] \leftarrow \varepsilon$
      $R \leftarrow D[S].right \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} D[1].right$
      $R \leftarrow R \mathbin{+\!\!+} Rules[0] \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} Rules[S]$
      $D[S].right \leftarrow R$
      $T \leftarrow (\varepsilon, D[S] \xrightarrow{Produces[S,P]}, \varepsilon)$
      $TEMP \leftarrow PUSH_R(T, D[N, \ldots, S-1])$
      $NewDeque \leftarrow POP_R(TEMP \oplus LookAhead)$
      $Size \leftarrow SIZE(NewDeque)$
```

Figure 4: Algorithm for Delayed Reductions



P
A_1 $\psi \overset{*}{\Longrightarrow} \varepsilon$
α_1 A_2 $\phi_1 \overset{*}{\Longrightarrow} \varepsilon$
$\alpha_{(i-1)}$ A_i $\phi_{(i-1)} \overset{*}{\Longrightarrow} \varepsilon$
α_i Q $\phi_i \overset{*}{\Longrightarrow} \varepsilon$

Figure 5: Chain Represented by Triple



P $\mu \overset{*}{\Longrightarrow} \varepsilon$ t ρ
A_1 $\phi_1 \overset{*}{\Longrightarrow} \varepsilon$
A_2 $\phi_2 \overset{*}{\Longrightarrow} \varepsilon$
A_i $\phi_i \overset{*}{\Longrightarrow} \varepsilon$
Q $\sigma \overset{*}{\Longrightarrow} \varepsilon$

Figure 6: Chain Represented by Pair

by the reductions on the completion of that rule. This is repeated for the next link and so on, resulting in the proper parse order.

Consider the grammar and relevent PCFSM states in Figure 7. The deque starting in state 0 and making transitions on $\hookrightarrow\ f\ *\ f$ is the following.

$$[([\,],0,[\,])([\,],1,[7])([\,],4,[\,])([\,],10,[\,])([\,],15,[\,])]$$

A trace of the effects of having $\hookleftarrow$ as the lookahead symbol is shown in Figure 8. Recall that the deque is numbered from right to left. Figure 8 shows the computation of $CHAIN$ and $LINK$. After these values are computed, the production lists for the recognised rules are computed. The values read from the tables for $Rules$ are as follows.

4	3	2	1	0
[ ]	[3, 1]	[4]	[6, 5]	[7]

These are used to build up the rule list that is associated with E, the nonterminal produced by the reduction. Recall that the values in $Rules$ are concatenated in numerical order which is from the right to the left.

$$[7] +\!\!+ [\,] +\!\!+ [\,] +\!\!+ [\,] +\!\!+ [7] +\!\!+ [6,5] +\!\!+ [4] +\!\!+ [3,1] +\!\!+ [\,]$$

$\hookleftarrow$ is then scanned and popped from the deque resulting in no change as there are no delayed empty reductions after the chain and reductions are complete resulting in the following deque.

$$[([\,],0,[\,])([\,],1,[7,7,6,5,4,3,1])([\,],2,[\,])]$$

0. $S' \rightarrow$ E $\hookleftarrow$

1. E $\rightarrow$ T Ts

2. Ts $\rightarrow$ + T Ts
3. $\quad | \ \varepsilon$

4. T $\rightarrow$ F Fs

5. FS $\rightarrow$ * F Fs
6. $\quad | \ \varepsilon$

7. F $\rightarrow$ f

$STATE$	$\Leftarrow$	$ITEM$	$\Rightarrow$
0		$*[S' \rightarrow; \bullet\bullet \ \hookrightarrow \ E \ \hookleftarrow;]$	$\hookrightarrow \Rightarrow 1$
1		$*[S' \rightarrow; \bullet \ \hookrightarrow \bullet \ E \ \hookleftarrow; \leftarrow S']$ $[E \rightarrow \hookrightarrow; \bullet\bullet \ T \ Ts; \ \hookleftarrow]$ $[T \rightarrow \hookrightarrow; \bullet\bullet \ F \ Fs; \ \hookleftarrow \ +]$ $[F \rightarrow \hookrightarrow; \bullet\bullet \ f; \ \hookleftarrow \ + *]$	$E \Rightarrow 2$ $T \Rightarrow 3$ $F \Rightarrow 4$ $f \Rightarrow 5$
4		$*[T \rightarrow \hookrightarrow; \bullet F \bullet \ Fs; \ \hookleftarrow \ + \leftarrow T]$ $[Fs \rightarrow \ f; \bullet\bullet \ * \ F \ Fs; \ \hookleftarrow \ +]$ $[Fs \rightarrow \ f; \bullet\bullet; \ \hookleftarrow \ +]$	$Fs \Rightarrow 9$ $* \Rightarrow 10$
10		$*[Fs \rightarrow \ f; \bullet \ * \bullet \ F \ Fs; \ \hookleftarrow \ + \leftarrow Fs]$ $[F \rightarrow \ *; \bullet\bullet \ f; \ \hookleftarrow \ + *]$	$F \Rightarrow 14$ $f \Rightarrow 15$
15		$*[F \rightarrow \ *; \bullet \ f \bullet; \ \hookleftarrow \ + * \leftarrow F]$	

Figure 7: Example Grammar and Relevant States From PCFSM

	$chain$					$LINK$				
j	4	3	2	1	0	4	3	2	1	0
$initialisation$	φ	$(3, E)$ $(3, T)$ $(3, F)$	$(T, 3, Fs)$	$(Fs, 2, F)$	$(F, 1, \varepsilon)$	ε	ε	ε	ε	ε
1	φ	$(3, E)$ $(3, T)$ $(3, F)$	$(3, Fs)$	$(T, 3, F)$	$(Fs, 2, \varepsilon)$	ε	ε	ε	F	ε
2	φ	$(3, E)$ $(3, T)$ $(3, F)$	$(3, Fs)$	$(3, F)$	$(3, Fs)$	ε	T	Fs	F	ε
3	φ	$(3, E)$ $(3, T)$ $(3, F)$	$(3, Fs)$	$(3, F)$	$(3, Fs)$	ε	T	Fs	F	ε

Figure 8: Computation of $CHAIN$ and $LINK$

4.3 Joining Configurations

The joining of two configurations is accomplished using $\circledcirc$. Delayed reductions are resolved and $\circledcirc$ is applied to all possible combinations of configurations from the left and right sets.

$$\begin{aligned} L \odot R = \{A \circledcirc B\}\ where A &\in \{\Uparrow_R (M, LES(N))\}, \\ B &\in \{\Uparrow_L (N, RES(M))\}, \\ M &\in L, N \in R \end{aligned}$$

The definition of $\circledcirc$ is in Figure 9. Choice (1) is a special case that checks for the final combining of configurations. Choices (2) and (3) result from recursive calls and are discussed with choice (7). Choices (4) and (5) also express base cases for the recursion. There is assumed to be no overlap with these choices and choices (2) through (6). As well, it is assumed that choices (2) and (3) take precedence over choices (4) and (5), respectively. Choice (6) is a special case where both configurations are outside trees.

Choice (4) is used when the left configuration is a left tree[4]. The starting state of the left configuration represents starting to parse in the same position as the inner edge of the right configuration. The transitions to the left from the

[4]For the sake of brevity, many definitions of functions, such as $left\_tree$, are omitted. They are presented in detail in [18].

$$
A \circledcirc B = \begin{cases}
([\,], 0, \alpha +\!\!+ [0]) & \text{if } (LE(A) = SS(A) = RE(A) \wedge B\ spans\ \hookrightarrow S \hookleftarrow) \quad (1) \\
 & \quad \vee (A\ spans\ \hookrightarrow S \hookleftarrow \wedge LE(B) = SS(B) = RE(B) \\
 & \quad \vee (A\ spans\ \hookrightarrow S \wedge B\ spans\ \hookleftarrow) \\
 & \quad where\ S\ is\ the\ starting\ nonterminal\ of\ the \\
 & \quad grammar\ and\ \alpha\ is\ associated\ with\ S \\
 & \\
B & \text{if } LE(A) = SS(A) = RE(A), \quad (2) \\
 & \quad [N \rightarrow p; \alpha \bullet \bullet \beta\gamma; q] \in BASIS(RE(A)) \\
 & \qquad \equiv [N \rightarrow p; \alpha \bullet \beta \bullet \gamma; q] \in BASIS(LE(B)) \\
 & \\
A & \text{if } LE(B) = SS(B) = RE(B), \quad (3) \\
 & \quad [N \rightarrow p; \alpha \bullet \beta \bullet \gamma; q] \in BASIS(RE(A)) \\
 & \qquad \equiv [N \rightarrow p; \alpha\beta \bullet \bullet \gamma; q] \in BASIS(LE(B)) \\
 & \\
left\_tree(A, B) & \text{if } [N \rightarrow p; \alpha \bullet \bullet \beta\gamma; q] \in BASIS(RE(A)) \quad (4) \\
 & \qquad \equiv [N \rightarrow p; \alpha \bullet \beta \bullet \gamma; q] \in BASIS(LE(B)) \\
 & \\
right\_tree(A, B) & \text{if } [N \rightarrow p; \alpha \bullet \beta \bullet \gamma; q] \in BASIS(RE(A)) \quad (5) \\
 & \qquad \equiv [N \rightarrow p; \alpha\beta \bullet \bullet \gamma; q] \in BASIS(LE(B)) \\
 & \\
A' +\!\!+ B' & \text{if } [N \rightarrow p; \alpha \bullet \bullet \beta; q] \in BASIS(RE(A)) \quad (6) \\
 & \qquad \equiv [N \rightarrow p; \alpha \bullet \bullet \beta; q] \in BASIS(LE(B)), \\
 & \quad B' = POP_L(([\,], SS, \sigma), B), \\
 & \quad A' = PUSH_R((\mu, SS, \sigma), POP_R((\mu, SS, [\,]), A)) \\
 & \\
join(A, B, N, |\alpha|, |\beta|, R) & \text{if } \exists [N \rightarrow p; \bullet\alpha \bullet \beta; q] \in BASIS(RE(A)) \quad (7) \\
 & \quad where\ [p; \alpha \bullet \beta \bullet; q \leftarrow N] \in BASIS(LE(B)), \\
 & \quad N \rightarrow \alpha\beta\ is\ production\ R
\end{cases}
$$

Figure 9: Joining Configurations

starting state in A are transferred to the right side of B by making appropriate transitions on the symbols spanned by the subtree. The number of symbols spanned is bounded by some constant based on the grammar so this is accomplished in constant time. Case (5) (the right configuration is a right tree) is the analogous adding of B to A.

Case (6) handles the case when the left configuration is a left tree and the right configuration is a right tree, both starting states are the same and the two combine to form a tent. The two deques are joined by removing the starting state from one of the deques and joining the two together.

When neither inside edge is a starting state, the inside halves of the configurations are joined and then the outside halves of the two configurations are added. For the configurations to join, the starting states must represent different portions of a similar parsing situation.

Consider a series of transitions on the inner half of a configuration starting from the starting state. A closure transition indicates that the configuration does not comprise all of the symbols that constitute the right hand side of the basis desired symbol. The rest of this half of the configuration is expanding the desired symbol. The inside half of the other configuration must also be expanding the same desired symbol since the two configurations are neighbours. In effect, there is a series of partially recognised rules in both configurations. The desired symbols of the states in the left configuration are the recognised symbols of the right configuration and *vice versa.*

The joining of the two configurations can be pictured as a zipper. The inner-most states join in the recognition of a rule, producing a nonterminal symbol. A transition is made on this symbol by one of the configurations, producing a new set of inner-most states which are then joined, and the process repeats. Like a zipper closing, the rules are brought together one by one until the two configurations are joined. Choice (7) is recursive and produces this effect.

The inner-most states of the two configurations will not, in general, contain the same items. However, if the two configurations can combine, there is an intersection of at least one basis item between the two states. There is only the configurations do not combine. The process of joining the inner halves fails when the deques do not have enough depth to perform a reduction.

There are three cases that must be handled in the definition of *join*, distinguished by the completeness of the inner-most transitions of the configurations. Therefore, the following functions are defined. $A\_complete(A, X, L, A')$ returns $TRUE$ if X states can be popped from the right side of the deque before encountering the starting state and returns $FALSE$ otherwise. When true, L is set to the ordered list of pairs, each pair consisting of the symbol on which a transition was made and the rules associated with each state that is removed from the deque. When false, L is set to the longest list of pairs before the starting state is encountered. In both cases, A' is set to the remaining deque. *B_complete* is analogous to *A_complete* and removes states from the left side of the deque.

join, shown in Figure 10, has several options. In the first choice of *join*, the inner states span the symbols desired by the inner state of the other deque. A combined production list is computed and *join*1 is called to push the nonterminal onto one of the two deques. The nonterminal may cause a chain of reductions which are performed by *join*1 (also shown in Figure 10). Note that *join*1 recursively calls itself and that the call to *A_complete* is used to remove the recognised rule from the list. After no more reductions can be performed, $\circledcirc$ is recursively called to complete the joining of the deques.

In the second choice of *join*, the left configuration is incomplete and the right configuration is complete. The desired symbols are transferred from the right configuration to the left configuration and appropriate adjustments to the production lists of the right deque are made. Finally, the resulting configurations are joined using $\circledcirc$. The final choice is similar except that the states are removed from the left deque and transitions are made in the right deque. Again, a recursive call is made to complete the joining of the two configurations. The call to *join* may have removed all of the elements from one deque, leaving a deque with just the starting state and spanning no symbols. When this happens, the recursive call of $\circledcirc$ results in case (2) or (3). In both of these cases, the starting state is compared with the inner edge of the other configuration to ensure that the two configurations join.

4.4 LL(1)∩RL(1)

The number of configurations cannot grow because of the GF(k) restriction and delayed reductions are computed in $O(log\ N)$. The last step is to develop a polylogarithmic time algorithm for joining configurations. The outer halves join in constant time but a polylogarithmic time algorithm is needed for joining the inside halves of the deques. This is done by limiting the languages considered to LL(1)∩RL(1). It can be shown that every LL(1) grammar is also an LR(1) grammar and properties of the PCFSM that are guaranteed by the LL(1) nature of the grammar are presented. Finally, an $O(log\ N)$ algorithm for joining the insides of the deque is presented.

LL(1) grammars are normally parsed from left to right in a top-down manner. This is not a necessary requirement as any LL(1) grammar is also an LR(1) grammar and can be parsed in a bottom-up fashion. A PCFSM must exist for any LL(1) language and the associative operator for LR(1)∩RL(1) grammars also parses those languages in LL(1)∩RL(1). The added structure imposed by the top-down nature of the LL(1) languages is exploited to achieve joining the inside halves of the configurations in $O(log\ N)$ time. Due to their top-down nature, the LL(1) languages give the PCFSM certain characteristics which are used to achieve an $O(log\ N)$ algorithm for joining configurations. These are presented now so that they can be used in the algorithm for joining the inner halves of configurations.

Recall that the generation of the PCFSM begins with the generation of the PCFSM with only one basis item in the initial state. Since there are no left recursive symbols, there can only be one basis item in each state in the PCFSM. States with multiple basis items are introduced by transitions to the left on right recursive symbols. All states in the PCFSM, even those states with multiple basis items retain certain characteristics due to the LL(1) nature of the grammar and the generation of the PCFSM.

Lemma 4 *The states of a PCFSM for a grammar that is LL(1) have the following characteristics.*

(1) A transition to the right is either a basis or a closure transition.

(2) For every string S of right desired symbols of basis items, every shorter string of right desired symbols of items in the same basis are prefixes of S.

(3) All basis items have the same right desired symbol or have no right desired symbols.

$$
\begin{array}{l}
join(A, B, N, \\
\quad LA, LB, R)
\end{array}
= \begin{cases}
D & \text{if } A\_complete(A, LA, [(S_1, \phi_1) \ldots (S_{LA}, \phi_{LA})], A') = TRUE, \\
& B\_complete(B, LB, [(T_1, \rho_1) \ldots (T_{LB}, \rho_{LB})], B') = TRUE, \\
& \gamma = \phi_1 \mathbin{++} \cdots \mathbin{++} \phi_{LA} \mathbin{++} \rho_1 \mathbin{++} \cdots \mathbin{++} \rho_{LB} \mathbin{++} [R], \\
& D = join1(A', B', N, \gamma) \qquad (1) \\[2ex]
D & \text{if } A\_complete(A, LA, [(S_M, \phi_M) \ldots (S_{LA}, \phi_{LA})], A') = FALSE, \qquad (2) \\
& B\_complete(B, LB, -, -) = TRUE, \\
& B' = POP_L(([\,], W, \mu), B), \\
& V_i = \xleftarrow{S_i \ldots S_{LA}} W, \\
& B'' = PUSH_L(([\,], V_M, [\,])(\phi_M, V_{(M+1)}, [\,]) \ldots \\
& \qquad (\phi_{(LA-1)}, V_{LA}, [\,])(\phi_{LA}, W, \mu), B'), \\
& D = A' \circledcirc B'' \\[2ex]
D & \text{if } B\_complete(B, LB, [(T_1, \rho_1) \ldots (T_{LB}, \rho_{LB})], B') = FALSE, \qquad (3) \\
& A' = POP_R((\delta, S, [\,]), A), \\
& V_i = S \xrightarrow{T \ldots T_i}, \\
& A'' = PUSH_R((\delta, S, \rho_1)([\,], V_1, \rho_2) \ldots \\
& \qquad ([\,], V_{(M-1)}, \rho_M)([\,], V_M, [\,]), A'), \\
& D = A'' \circledcirc B' \\[2ex]
\varphi & otherwise \qquad (4)
\end{cases}
$$

$$
join1(A, B, N, \gamma) = \begin{cases}
join1(A''', B, M, L) & \text{if } A' = POP_R((\alpha, S, [\,]), A) \\
& A'' = PUSH_R((\alpha, S, \gamma)([\,], S \xrightarrow{N}, [\,]), A'), \\
& reduce(A'', B, M, \alpha) = TRUE, \\
& M \rightarrow \alpha \ is \ rule \ R, \\
& A\_complete(A'', B, |\alpha|, [(T_1, \delta_1) \ldots (T_N, \delta_N)], A'''), \\
& L = \delta_1 \mathbin{++} \cdots \mathbin{++} \delta_N \mathbin{++} [R] \\[2ex]
A'' \circledcirc B & \text{if } A' = POP_R((\alpha, S, [\,]), A) \\
& A'' = PUSH_R((\alpha, S, \gamma)([\,], S \xrightarrow{N}, [\,]), A'), \\
& reduce(A'', B, -, -) = FALSE \\[2ex]
A \circledcirc B'' & \text{if } TOP_R(A) \not\xrightarrow{N}, \\
& B' = POP_L(([\,], S, \alpha), B), \\
& B'' = PUSH_L(([\,], \xleftarrow{N} S, [\,])(\gamma, S, \alpha), B')
\end{cases}
$$

$$
reduce(A, B, M, \alpha) = \begin{cases}
TRUE & \text{if } [M \rightarrow p; \bullet\alpha\bullet; q] \in BASIS(TOP_R(A)), \\
& \quad [M \rightarrow p; \bullet\alpha\bullet; q] \in BASIS(TOP_L(B)) \\[2ex]
FALSE & otherwise
\end{cases}
$$

Figure 10: Definition of *join* and *join*1

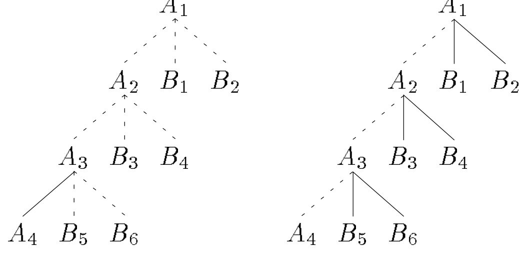


Figure 11: Partial Parse Tree Rooted at A_1 and Its Match

(4) For each right closure item, there can only be one series of productions from the basis right desired symbol to the left hand side symbol of the closure item.

(5) The number of rules in the derivation from the right desired symbol of a basis item to the left hand side symbol of the closure item is finite.

(6) The only states that can be reached by transitions to the right that have more than one basis item are reached by a series of basis transitions from some starting state.

4.5 Joining Inner Halves

These properties are now used to develop a polylogarithmic time algorithm for joining the inner halves of the configurations. Property 4 of the Lemma is key to achieving sublinear time for joining the inside halves of the two configurations. Consider a closure transition from a state on the inside half of the left configuration. There is only one item with the desired symbol on which the transition is made and only one derivation from the basis desired symbol to the symbol so a section of the parse tree is uniquely specified. If the closure transition is not the left-most closure transition in the inner half of the configuration, there is only one item in the basis so the basis desired symbols are also known. The desired symbols in the items of the derivation are the unmatched nodes in the partial parse tree. The symbols spanned by the inner half of the right configuration must match for the two configurations to combine. Therefore, the strings of symbols that are indicated by the closure transitions in the left deque are joined together to form a string that specifies the symbols that must be spanned by the right configuration. The predictions are built and the symbols are compared in parallel. If all match, the joining of the inner halves is complete and a recursive call to ⊚ ensures that the starting states match. The transitions between the states in the right configuration specify the symbols spanned by the configuration. These symbols are determined and then verified against the prediction made by the left configuration.

Consider the following state in some LL(1)∩RL(1) PCFSM. Assume that a transition to the right has been made on the terminal symbol A to the inner top state and the expansion of A_i is rule R_i.

$$\begin{array}{l} *[N \rightarrow x_1; \bullet\alpha \bullet A_1\beta; y_1] \\ [A_1 \rightarrow x_2; \bullet \bullet A_2B_1B_2; y_2] \\ [A_2 \rightarrow x_3; \bullet \bullet A_3B_3B_4; y_3] \\ [A_3 \rightarrow x_4; \bullet \bullet AB_5B_6; y_4] \end{array}$$

The partial parse tree rooted at A_1 that has A recognised and the lower portion of the right configuration are shown in Figure 11. However, it may not be possible to determine this form by only examining the right configuration.

The lengths of the strings of predicted symbols are constant since they are based on the grammar rather than the input. The positions in the right deque of the predicted strings are computed in parallel. Each prediction is validated by a single processor, appending together the rule lists associated with each symbol and adding the recognised rules in the appropriate positions. This takes constant time since the number of symbols validated by each processor is bounded by a constant based on the grammar. Assuming that all of the symbols are valid and that μ_i is the production list associated with B_i, the resulting production list for A_1 is $[\mu_5\mu_6R_3\mu_3\mu_4R_2\mu_1\mu_2R_1]$. A parallel suffix sums (analogous to prefix sums except from the right to left) is used to compute the index of the entries in the

expected	*actual symbols*	*rule list*
$[B_5 B_6 [R_3] B_3 B_4 [R_2] B_1 B_2 [R_1]]$	$B_5 B_6 B_3 B_4 B_1 B_2$	$[\,]$
$[B_6 [R_3] B_3 B_4 [R_2] B_1 B_2 [R_1]]$	$B_6 B_3 B_4 B_1 B_2$	μ_5
$[[R_3] B_3 B_4 [R_2] B_1 B_2 [R_1]]$	$B_3 B_4 B_1 B_2$	$\mu_5\mu_6$
$[B_3 B_4 [R_2] B_1 B_2 [R_1]]$	$B_3 B_4 B_1 B_2$	$\mu_5\mu_6 R_3$
$[B_4 [R_2] B_1 B_2 [R_1]]$	$B_4 B_1 B_2$	$\mu_5\mu_6 R_3\mu_3$
$[[R_2] B_1 B_2 [R_1]]$	$B_1 B_2$	$\mu_5\mu_6 R_3\mu_3\mu_4$
$[B_1 B_2 [R_1]]$	$B_1 B_2$	$\mu_5\mu_6 R_3\mu_3\mu_4 R_2$
$[B_2 [R_1]]$	B_2	$\mu_5\mu_6 R_3\mu_3\mu_4 R_2\mu_1$
$[[R_1]]$		$\mu_5\mu_6 R_3\mu_3\mu_4 R_2\mu_1\mu_2$
$[\,]$		$\mu_5\mu_6 R_3\mu_3\mu_4 R_2\mu_1\mu_2 R_1$

Figure 12: Validation of Prediction

right deque. Typically, the list of expected symbols is stored in a table indexed by states and symbols. This list has markers of the form $[R]$ indicating that rule R has been recognised. The processor validates the string and builds the rule list as shown in Figure 12. The validations are done in parallel and take constant time since the sizes of the strings validated are based on the grammar. The suffix sums algorithm requires $O(log\ N)$ time so the joining of the two halves of the configurations takes $O(log\ N)$ time.

4.5.1 Algorithm

The algorithm for joining two inner halves of a pair of adjacent configurations is divided into three parts. The tables used by the algorithm and part of the algorithm are shown in Figures 13 and 14. The algorithm for handling the case of the starting state of the left configuration being higher than the right's starting state is shown in Figures 15 and 16.

The algorithm for handling a higher right starting state is shown in Figure 17. The algorithm for matching all of the symbols in a group of levels is shown in Figure 18. The algorithm for creating the production lists of the matched group of levels in shown in Figure 19. *rules* initially contains the production lists that were formed by matching a group of levels while *prods* contains the production lists of the symbols on which basis transitions where made. At the lowest level in the parse tree, the desired list is the rules associated with the basis transitions followed by the rules from the matched symbols. The production list for the next higher level is the list associated with the basis transitions followed by the list from the lower level followed the list from the matched symbols and so on. The production list for n levels is $prods[n] +\!\!+ \cdots +\!\!+ prods[0] +\!\!+ rules[0] +\!\!+ \cdots +\!\!+ rules[n]$ The values in *rules* and *prods* are not adjacent, nor are they in the proper order. A suffix sums computation is used to compute the position of each portion of the rule and the values in *prods* are stored in *list* in reverse order. The values in *rules* are stored in *list* to the right of the *prods* values in the order specified by the suffix sums computation. Note that $A[0]$ is the number of non-ε elements of *rules*.

5 P(k) Languages

The algorithm for joining the inner halves of two configurations relies on the LL(1) characteristics of the grammar so the grammar must be in LL(1)∩RL(1). As well, the grammar must be GF(k) for some finite k to avoid growth in the number of configurations. An analogous algorithm works for those grammars that are in LR(1)∩RR(1), where RR(1) is the analogous right to left parsing method as LL(1), again assuming that the grammar is GF(k) for some k. All of these requirements are captured in the following language class.

The *P(k)* (*P*arallel with GF(k)) grammars are defined as the set of grammars that are GF(k) and are in LL(1)∩RL(1) or LR(1)∩RR(1). A *P(k) language* is any language generated by a P(k) grammar. For P(k) languages, the parsing algorithm using $\odot$ runs in $O(log^2\ N)$ time using $O(N)$ processors since there are $O(log\ N)$ phases of combining partial answers using $\odot$. Each phase of combining requires $O(log\ N)$ time since there are at most some constant number of configurations that are combined in a phase and each combining of a pair of configu-

$$
\begin{aligned}
BC[S] &= [\gamma\,[R] \mid [N \rightarrow p; \alpha \bullet \beta \bullet \gamma; q] \in BASIS(S), N \rightarrow \alpha\beta\gamma \textit{ is rule } R] \\
BCL[S] &= |BC[S]| \\
BD[S] &= A \mid [N \rightarrow p; \alpha \bullet \beta \bullet A\gamma; q] \in BASIS(S) \\
BLHS[S] &= N \mid [N \rightarrow p; \alpha \bullet \beta \bullet \gamma; q] \in BASIS(S) \\
BE[S] &= [\gamma\,[R] \mid [N \rightarrow p; \alpha \bullet \beta \bullet A\gamma; q] \in BASIS(S), N \rightarrow \alpha\beta\gamma \textit{ is rule } R] \\
BEL[S] &= |BE[S]| \\
CE[S, A_i] &= \ \ [\phi_{(i-1)}\,[R_{(i-1)}] \ldots \phi_1\,[R_1]] \mid \\
&\qquad\qquad [N \rightarrow x; \alpha \bullet \beta \bullet A_1\gamma; y] \in BASIS(S), \\
&\qquad\qquad [A_1 \rightarrow q; \bullet \bullet A_2\phi_1; p] \in CLOSURE(S), \\
&\qquad\qquad\qquad\qquad \vdots \\
&\qquad\qquad [A_{(i-1)} \rightarrow q; \bullet \bullet A_i\phi_{(i-1)}; p] \in CLOSURE(S), \\
&\qquad\qquad [A_i \rightarrow q; \bullet \bullet A_{(i+1)}\phi_i; p] \in CLOSURE(S), \\
&\qquad\qquad A_j \rightarrow A_{(j+1)}\phi_j \textit{ is rule } R_j \\
CEL[S, A] &= |CE[S, A]| \\
NRS[S] &= |\beta| \textit{ such that } [N \rightarrow p; \alpha \bullet \beta \bullet \gamma; q] \in BASIS(S) \\
RTS[S_1, S_2] &= T \mid S_1 \xrightarrow{T} S_2 \\
LTS[S_2, S_1] &= T \mid S_2 \xleftarrow{T} S_1 \\
CT[S_1, S_2] &= \begin{cases} TRUE & \text{if } \textit{closure\_transition}[S_1, S_2] \\ FALSE & \textit{otherwise} \end{cases} \\
GL[S, A] &= T \mid T \xleftarrow{A} S \\
GR[S, A] &= T \mid S \xrightarrow{A} T
\end{aligned}
$$

Figure 13: Table Entries for Algorithm

rations requires $O(log\ N)$ time using $O(N)$ processors. Therefore, the algorithm runs in $O(log^2\ N)$ time using $O(N)$ processors.

6 Conclusion

A test was presented for a grammar that ensures that the number of configurations in a partial result cannot grow without bound. Grammars for which the test succeeds are said to be GF(k), growth free within k symbols. A polylogarithmic time algorithm for resolving the delayed reductions in the configurations in $O(log\ N)$ time using $O(N)$ processors was presented. A polylogarithmic time algorithm for joining the inner halves of two configuration that runs in $O(log\ N)$ time using $O(N)$ processors was described. This algorithm requires that the grammar be LL(1)$\cap$RL(1). Finally, the P(k) class of languages was defined. The parallel parsing algorithm has a strong theoretical basis in list theory and for P(k) grammars, it runs in $O(log^2\ N)$ time using $O(N)$ processors. Unfortunately, membership in the class of grammars is not intuitive as is the case for the LL(1) and LR(1) classes of grammars.

References

[1] I. Bar-on and U. Vishkin. Optimal parallel generation of a computation tree form. *ACM Transactions on Programming Languages and Systems*, 7(2):348–357, April 1985.

[2] D.T. Barnard, J.P. Schmeiser, and D.B. Skillicorn. Deriving associative operators for language recognition. *Bulletin of the EATCS*, 43:131–139, February 1991.

```
$join(left[0 \ldots LN-1], LN,$
        $right[0 \ldots RN-1], RN,$
        $new[0 \ldots NN-1], NN)$
    $find\_closure_transition()$
    $closure\_expected()$
    $suffix\_sums(A, LL, LN-1)$
    $bounds()$
    $transition\_symbols()$
    if $upper[LL] > RN-1$
    then $left\_higher()$
    else $right\_higher()$

$find\_closure_transition()$
    $i \leftarrow 0$
    while $(i < LN-1) \wedge \neg CT[left[i], left[i+1]]$
        $i \leftarrow i+1$
    $LL \leftarrow i$

$closure\_expected()$
    for $i \leftarrow LL$ to $LN-1$ do in parallel
        if $i = LN-1$
        then $E[i] \leftarrow BC[left[i]]$
            $A[i] \leftarrow BCL[left[i]]$
        else if $i = LL$
            then $T \leftarrow RTS[left[i], left[i+1]]$
                $E[i] \leftarrow CE[BD[left[i]], T]$
                $A[i] \leftarrow CEL[BD[left[i]], T]$
            else if $CT[left[i], left[i+1]]$
                then $Z \leftarrow CE[BD[left[i]], T]$
                    $ZL \leftarrow CEL[BD[left[i]], T]$
                    $E[i] \leftarrow BE[left[i]] +\!\!+ Z$
                    $A[i] \leftarrow BEL[left[i]] + ZL$
                else $A[i] \leftarrow 0$

$bounds()$
    for $i \leftarrow LL$ to $LN-1$ do in parallel
        if $i = LN-1$
        then $lower[i] \leftarrow 0$
        else $lower[i] \leftarrow A[i+1]$
        $upper[i] \leftarrow A[i] - 1$

$transition\_symbols()$
    for $i \leftarrow 0$ to $RN-1$ do in parallel
        if $i = RN-1$
        then $symbols[i] \leftarrow \varepsilon$
        else $symbols[i] \leftarrow LTS[right[i+1], right[i]]$
```

Figure 14: Algorithm to Join Inner Halves

```
$left\_higher()$
    for $i \leftarrow LL$ to $LN - 1$ do in parallel
        $rules[i] \leftarrow [\,]$
        $prods[i] \leftarrow [\,]$
        $matches[i] \leftarrow TRUE$
        $complete\_match(i)$
    $replicate()$
    if $complete$
    then $do\_complete()$
    else $do\_incomplete()$

$complete\_match(i)$
    if $((i = LN - 1) \vee CT[left[i], left[i+1]])$
        $\wedge(lower[i] \leq RN - 1)$
    then if $upper[i] \leq RN - 1$
        then $rules[i] \leftarrow match\_all(i)$
            if $rules[i] = [\,]$
            then $matches[i] \leftarrow FALSE$
            else for $j[i] \leftarrow i - NRS[left[i]]$ to $i - 1$
                    $prods[i] \leftarrow prods[i] +\!\!+ left[j[i]].right$
            if $upper[i] = RN - 1$
            then $highest \leftarrow i$
                $complete \leftarrow TRUE$
                $R \leftarrow LL - 1$
        if $upper[i] > RN - 1$
        then $highest \leftarrow i$
            $complete \leftarrow FALSE$
            $R \leftarrow highest$

$replicate()$
    for $i \leftarrow 0$ to $R$ do in parallel
        if $i = R$
        then $new[i].right \leftarrow [\,]$
        else $new[i] \leftarrow left[i]$

$do\_complete()$
    if $and(matches, LL, LN - 1)$
    then $new[R].right \leftarrow prod\_list()$
        $S \leftarrow GR[left[R], BLHS[left[highest]]]$
        $new[R+1] \leftarrow ([\,], S, [\,])$
        $NN \leftarrow R + 2$
    else fail
```

Figure 15: Algorithm for Left Higher (i)

```
$do\_incomplete()$
    if $and(matches, LL, LN-1)$
    then $new[R].right \leftarrow left[R].right$
        $top \leftarrow R$
        $j \leftarrow 0$
        $i \leftarrow lower[highest]$
        while $(i \leq RN-2) \wedge (top \geq 0)$
            $reduce\_recognised()$
            if $E[highest][j] = symbols[i]$
            then $top \leftarrow top + 1$
                $new[top-1].right \leftarrow right[i+1].left$
                $S \leftarrow GR[new[top-1], symbols[i]])$
                $new[top] \leftarrow ([\,], S, [\,])$
                $i \leftarrow i + 1$
            else $top \leftarrow -2$
            $j \leftarrow j + 1$
        $NN \leftarrow top + 1$
    else fail

$reduce\_recognised()$
    while $E[highest][j] = [r]$
        $prods \leftarrow [\,]$
        for $k \leftarrow top - rule\_size[r]$ to $top - 1$
            $prods \leftarrow prods +\!\!+ new[k].right$
        $lhs \leftarrow BLHS[new[top]]$
        $top \leftarrow top - rule\_size[r]$
        $new[top].right \leftarrow prods +\!\!+ [r]$
        $top \leftarrow top + 1$
        $S \leftarrow GR[new[top-1], lhs]$
        $new[top] \leftarrow ([\,], S, [\,])$
        $j \leftarrow j + 1$
```

Figure 16: Algorithm for Left Higher (ii)

[3] D.T. Barnard and D.B. Skillicorn. Parallel parsing on the Connection Machine. *Information Processing Letters*, 31, No.3:111–117, 8th May 1989.

[4] R.S. Bird. Lectures on constructive functional programming. Oxford University Programming Research Group Monograph PRG-69, 1988.

[5] M.P. Chytil and B. Monien. Caterpillars and context-free languages. In C. Choffrut and T. Lengauer, editors, *STACS90 7th Annual Symposium on Theoretical Aspects of Computer Science*, Springer Lecture Notes in Computer Science 415, pages 70–81, February 1990.

[6] E. Dekel and S. Sahni. Parallel generation of postfix and tree forms. *ACM Transactions on Programming Languages and Systems*, 5(3):300–317, July 1983.

[7] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, 1988.

[8] A. Gibbons and W. Rytter. Optimal parallel algorithms for dynamic expression evaluation and context-free recognition. *Information and Computation*, 81(1):32–45, April 1989.

[9] W. Daniel Hillis and G.L. Steele. Data parallel algorithms. *Communications of the ACM*, 29, No.12:1170–1183, December 1986.

```
$right\_higher()$
    $match\_symbols()$
    if $and(matches, LL, LN-1)$
    then $j \leftarrow upper[LL]+1$
        for $i \leftarrow j$ to $RN-1$ do in parallel
            if $i = j$
            then $new[LL+1] \leftarrow right[i]$
            $new[LL+1].left \leftarrow prod\_list()$
            else $new[LL+1+(j-i)] \leftarrow right[i]$
        $new[LL].right \leftarrow [\,]$
        $S \leftarrow GL[new[LL+1], BD[left[LL]]]$
        $new[LL].state \leftarrow S$
        $NN \leftarrow (LL+1)+(j-(RN-1)+1)$
        $i \leftarrow LL$
        $basis\_transitions()$
        $new[0].left \leftarrow [\,]$
    else fail

$match\_symbols()$
    for $i \leftarrow LL$ to $LN-1$ do in parallel
        $rules[i] \leftarrow [\,]$
        $prods[i] \leftarrow [\,]$
        $matches[i] \leftarrow TRUE$
        if $(i = LN-1) \vee CT[left[i], left[i+1]]$
        then $rules[i] \leftarrow match\_all(i)$
            if $rules[i] = [\,]$
            then $matches[i] \leftarrow FALSE$
            else for $j[i] \leftarrow i - NRS[left[i]]$ to $i-1$
                    $prods[i] \leftarrow prods[i] \mathbin{+\!\!+} left[j[i]].right$

$basis\_transitions()$
    while $i \geq 1$
        $S \leftarrow GL[new[i], RTS[left[i-1], left[i]]]$
        if $S \neq ERROR$
        then $new[i] \leftarrow (left[i-1].right, S, [\,])$
            $i \leftarrow i-1$
        else fail
```

Figure 17: Algorithm for Right Higher

```
$match\_all(entry)$
/* match all expected symbols */
    $prods \leftarrow [\,]$
    $j \leftarrow 0$
    if $lower[entry] > upper[entry]$
    then $prods \leftarrow [\,]$
        while $j < |E[entry]|$
            $prods \leftarrow prods +\!\!+ E[entry][j]$
    else for $i \leftarrow lower[entry]$ to $upper[entry]$
            while $E[entry][j] = [r]$
                $prods \leftarrow prods +\!\!+ r$
                $j \leftarrow j + 1$
            if $E[entry][j] = symbols[i]$
            then $prods \leftarrow prods +\!\!+ deque[i+1].left$
            else return $[\,]$
            $j \leftarrow j + 1$
        $prods \leftarrow prods +\!\!+ E[entry][j]$
    return $prods$
```

Figure 18: Algorithm for Matching Symbols

[10] P. N. Klein and J. H. Reif. Parallel time O(log n) acceptance of deterministic CFLs on an exclusive-write P-RAM. *Siam Journal of Computing*, 17(3):463–485, June 1988.

[11] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, November 1967.

[12] D. E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms.* Addison Wesley, 1972.

[13] P.H.W.M. Oude Luttighuis. Parallel parsing of regular right-part grammars. Technical Report INF89-63, Faculteit der Informatica, Universiteit Twente, 1989.

[14] P.H.W.M. Oude Luttighuis. Optimal parallel parsing of almost all ll(k) grammars. Technical report, Memoranda Informatica INF 91-78, Department of Computer Science, University of Twente, Oct. 1991.

[15] R. Mattheyses and C.M. Fiduccia. Parsing Dyck languages on parallel machines. In *Proceedings of the 20th Allerton Conference on Communication, Control and Computing*, pages 272–280, 1982.

[16] W. Rytter. On the complexity of parallel parsing of general context-free languages. *Theoretical Computer Science*, 47:315–321, 1986.

[17] R. M. Schell, Jr. *Methods for Constructing Parallel Compilers for use in a Multiprocessor.* PhD thesis, University of Illinois at Urbana-Champaign, 1979.

[18] James. P. Schmeiser. *Polylogarithmic Parallel Parsing.* PhD thesis, Queen's University at Kingston, 1992.

[19] James. P. Schmeiser and David T. Barnard. Producing a top-down parse order with bottom-up parsing. *Information Processing Letters*, (accepted March, 1995).

[20] Y. N. Srikant and P. Shankar. A new parallel algorithm for parsing arithmetic infix expressions. *Parallel Computing*, 4:291–304, 1987.

[21] Y.N. Srikant and P. Shankar. Parallel parsing of programming languages. *Information Sciences*, 43:55–83, 1987.

[22] J.P. Tremblay and P.G. Sorenson. *The Theory and Practice of Compiler Writing.* McGraw-Hill, 1985.

```
$prod\_list()$
/* build up the rule list for the joined levels */
    for $i \leftarrow 0$ to $LN - 1$ do in parallel
        if $rules[i] \neq [\,]$
        then $A[i] \leftarrow 1$
        else $A[i] \leftarrow 0$
    $suffix\_sums(A, 0, LN - 1)$
    for $i \leftarrow 0$ to $LN - 1$ do in parallel
        if $rules[i] \neq [\,]$
        then $list[A[0] - A[i]] \leftarrow prods[i]$
            $list[A[0] + A[i] - 1] \leftarrow rules[i]$
    return $list[0] \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} list[2 * A[0] - 1]$
```

Figure 19: Algorithm to Construct Production Lists