October 1988

An Efficient All-Parses Systolic Algorithm for General Context-Free Parsing

Oscar H. Ibarra
University of Pennsylvania

Michael A. Palis
University of Pennsylvania

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-88-86.

An Efficient All-Parses Systolic Algorithm for General Context-Free Parsing

Abstract

The problem of outputting all parse trees of a string accepted by a context-free grammar is considered. A systolic algorithm is presented that operates in $O(m\,n)$ time, where m is the number of distinct parse traces and n is the length of the input. The systolic array uses n^2 processors, each of which requires at most $O(\log n)$ bits of storage. This is much more space-efficient than a previously reported systolic algorithm for the same problem, which required $O(n \log n)$ space per processor. The algorithm also extends previous algorithms that only output a single parse tree of the input.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-88-86.

An Efficient All-Parses Systolic Algorithm for General Context-free Parsing

Oscar H. Ibarra[1]
Department of Computer Science
University of Minnesota
Minneapolis, MN 55455

Michael A. Palis[2]
Department of Computer and Information Science
University of Pennsylvania
Phildelphia, PA 19104

Abstract: The problem of outputting all parse trees of a string accepted by a context-free grammar is considered. A systolic algorithm is presented that operates in $O(m \cdot n)$ time, where m is the number of distinct parse trees and n is the length of the input. The systolic array uses n^2 processors, each of which requires at most $O(log\ n)$ bits of storage. This is much more space-efficient than a previously reported systolic algorithm for the same problem, which required $O(n \log n)$ space per processor. The algorithm also extends previous algorithms that only output a single parse tree of the input.

1 Research supported in part by NSF Grants DCR-8420935 and DCR-8604603.

2 Research supported in part by ARO Grant DAA29-84-9-0027, NSF Grants MCS-8219116-CER, MCS-82-07294, DCR-84-10413, MCS-83-05221, and DARPA Grant N00014-85-K-0018.

AN EFFICIENT ALL-PARSES SYSTOLIC ALGORITHM FOR GENERAL CONTEXT-FREE PARSING

Oscar H. Ibarra
Michael A. Palis

MS-CIS-88-86
LINC LAB 137

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104

October 1988

Acknowledgements: This research was supported in part by DARPA grant NOOO14-85-K-0018, NSF grants MCS-82-07294, MCS-8219196-CER, DCR-84-10413, MCS-83-05221, IRI84-10413-AO2 and U.S. Army grants DAA29-84-K-0061, DAA29-84-9-0027.

1. Introduction

General context-free language (CFL) recognition is an important problem with a wide range of applications: formal language theory, pattern recognition, natural language processing, compiler design, to name a few. To date, the Cocke-Kasami-Younger (CKY) algorithm [YOUN67] and Earley's algorithm [EARL70] remain the best known practical methods for solving this problem, both having a worst-case time complexity of $O(n^3)$ for inputs of length n. (In [VALI75], Valiant presented an asymptotically faster algorithm; however, the constant of proportionality is too large for practical applications.)

Kosaraju [KOSA75] first considered the problem of parallel CFL recognition and presented a parallelization of the CKY algorithm on a two-dimensional iterative array of n^2 processors. The array operates in linear time and only requires finite-state processors (i.e., the processor stores information whose size is independent of the length of the input). Another algorithm, using a systolic array, is also implied by the work of Guibas, Kung and Thompson [GUIB79], who gave a parallel implementation of the dynamic programming algorithm (similar to the CKY algorithm) for computing the cost of an optimum binary search tree. Both algorithms are optimal; the speed-up is linear in the number of processors used. A parallel algorithm which has a faster running time (in fact, $O(log^2 n)$) has been presented by Rytter [RYTT85]; however, the algorithm is implemented on parallel random-access machine (PRAM), a hypothetical model that ignores communication costs, and uses more processors (n^6).

In [CHIA84], Chiang and Fu considered the more general problem of CFL parsing, which unlike recognition, also requires a parse tree as output. They gave a parallel implementation of Earley's algorithm on a systolic array of n^2 processors. Besides recognizing the input, the array also outputs a parse tree in linear time. However, the processors are no longer finite-state since each is required to store $O(log\ n)$ bits of information. A fully finite-state systolic array for recognition and parsing was later given in [CHAN87]; the array uses n^2 processors and runs in linear time.

An interesting extension to the CFL parsing problem is that of outputting all parse trees of the input string. In some applications such as natural language parsing, the underlying grammar is usually ambiguous. Typically, one would be interested in generating all parse trees of the given string, which later can be disambiguated by applying some semantic rules. In [LANG86], Langlois considered the all-parses problem and gave a systolic algorithm based on the systolic architecture of [GUIB79]. The systolic array uses $O(n^2)$ processors. However, each processor is required to store $O(n \log n)$ bits of information, resulting in a total space complexity of $O(n^3 \log n)$. If the underlying grammar is unambiguous, the space complexity reduces to $O(n^2 \log n)$. Langlois posed indirectly the question of whether $O(n^2 \log n)$ space is sufficient to output all parses for an arbitrary CFL. In this paper, we settle this question in the affirmative. In particular, we give a systolic CFL parsing algorithm that outputs all parses in time $O(m \cdot n)$ using n^2 processors, each of which requires only $O(log\ n)$ bits of storage. Thus, the total space complexity is $O(n^2 \log n)$. The systolic algorithm is an extension of the one described in [CHAN87]. It should be pointed out that the algorithm in [CHAN87] does not give an explicit systolic array implementation, but rather gives an algorithm that runs on a sequential machine characterization of a systolic array. This paper gives the explicit "systolic version" of the algorithm in [CHAN87], and extends it to generate all parse trees of the input string with only a factor $\log n$ increase in the space complexity.

The paper is organized as follows. In Section 2, we first describe a sequential parsing algorithm on which the systolic algorithm is based. In Section 3, we introduce the systolic array model that implements the algorithm. Sections 4 and 5 describe the two phases of the systolic algorithm: the *recognition* and

parse generation phase, respectively. Finally, Section 6 gives an analysis of the time and space complexity of the algorithm.

2. A Sequential Context-Free Parsing Algorithm

We first describe the sequential parsing algorithm on which the systolic parsing algorithm is based. We assume familiarity with context-free grammars (CFG's); see, e.g., [AHO72]. Let $G = \langle V_N, V_T, P, S \rangle$ be a CFG where V_N and V_T are finite sets of nonterminal and terminal symbols, respectively, $S \in V_N$ is the start symbol, and P is a finite set of productions in Chomsky normal form. That is, every production in P is either of the form $A \rightarrow BC$ or $A \rightarrow a$, where $A, B, C \in V_N$ and $a \in V_T$. The language generated by G is $L(G) = \{w \in V_T^+ \mid S \Rightarrow w\}$.

Given an input string $w = a_1 a_2 \cdots a_n$, $a_i \in V_T$, the sequential algorithm starts by constructing sets $R(i,j)$, $1 \le i \le j \le n$, such that

$$R(i,j) = \{[A \rightarrow \alpha] \in P \mid A \Rightarrow a_i \cdots a_j\}.$$

The sets $R(i,j)$ are computed according to the following variant of the CKY dynamic programming algorithm [YOUN67]:

$$R(i,i) = \{[A \rightarrow a_i] \in P\} \qquad 1 \le i \le n,$$

$$R(i,j) = \bigcup_{i \le k < j} R(i,k) * R(k+1,j) \qquad 1 \le i < j \le n,$$

where $R_1 * R_2 = \{[A \rightarrow BC] \in P \mid$ there are productions $\pi_1 \in R_1$ and $\pi_2 \in R_2$ such that $\mathrm{LHS}(\pi_1) = B$ and $\mathrm{LHS}(\pi_2) = C\}$. ('LHS' stands for 'left-hand side'.) Thus, $w \in L(G)$ iff $R(1,n)$ contains a production whose LHS is the start symbol S.

An example of a CFG G and the corresponding matrix of $R(i,j)$'s for the string $w = abaa$ is illustrated in Figure 2.1. Henceforth, the matrix $\mathrm{R} = \{R(i,j) \mid 1 \le i \le j \le n\}$ shall be referred to as the *recognition matrix*. For the given example, we see that $abaa \in L(G)$ since $R(1,4)$ contains a production whose LHS is S.

If $w \in L(G)$ then w has one or more parse trees, where a parse tree is a binary tree of productions used in the derivation $S \Rightarrow w$. For the example in Figure 2.1, the string *abaa* has five distinct parse trees, as shown in Figure 2.2. For each production, the pair of numbers (i,j) denotes the matrix entry $R(i,j)$ to which the production belongs.

We now describe a procedure *PARSE* for generating all parse trees of the input string. *PARSE* is a recursive procedure that takes four arguments (A,i,j,tag), where $A \in V_N$, $1 \le i \le j \le n$ and $tag \in \{FIRST, CURRENT, NEXT\}$. Informally, $PARSE(A,i,j,tag)$ returns a parse tree for the derivation $A \Rightarrow a_i \cdots a_j$. The parse tree is represented as follows: if a production π in the parse tree belongs to $R(i,j)$, then the occurrence of π in $R(i,j)$ is "marked" by some special symbol, say *. (There is no ambiguity here since all productions in a parse tree belong to distinct $R(i,j)$'s.) For example, the first parse tree in Figure 2.2 would be represented as shown in Figure 2.3. Note that the actual tree can be retrieved since for every marked production, its left (right) child in the actual tree is simply the next marked production above it along the same column (diagonal).

$G = < \{S,A,B,C\}, \{a,b\}, P, S >$, where

$$P = \{ [S \rightarrow AA], \quad [A \rightarrow AC], \quad [B \rightarrow BC], \quad [C \rightarrow CC],$$
$$[S \rightarrow AB], \quad [A \rightarrow CB], \quad [B \rightarrow b], \quad [C \rightarrow a] \}$$
$$[A \rightarrow a],$$

Recognition Matrix for $w = abaa$

a	b	a	a
R(1,1)	R(2,2)	R(3,3)	R(4,4)
[A->a] [C->a]	[B->b]	[A->a] [C->a]	[A->a] [C->a]
R(1,2)	R(2,3)	R(3,4)	
[S->AB] [A->CB]	[B->BC]	[S->AA] [A->AC] [C->CC]	
R(1,3)	R(2,4)		
[S->AA] [S->AB] [A->AC] [A->CB]	[B->BC]		
R(1,4)			
[S->AA] [S->AB] [A->AC] [A->CB]			

Figure 2.1. A CFG G and the recognition matrix R for $w = abaa$.

The argument *tag* dictates which parse tree is returned. If $tag = FIRST$, then $PARSE(A,i,j,tag)$ returns an initial parse tree for $A \overset{*}{\Rightarrow} a_i \cdots a_j$. If $tag = NEXT$, then it returns the next (distinct) parse tree following the one last generated. Finally, if $tag = CURRENT$, then it returns the current parse tree.

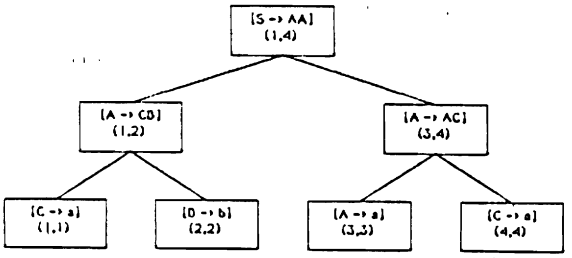
To keep track of the order of parse tree generation, the procedure makes use of a number of auxiliary variables. For each (i,j), $1 \le i \le j \le n$, there are boolean variables $done(i,j)$ and $last\_id(i,j)$, and an integer variable $id(i,j)$. The variables are utilized as follows: Let t be the tree that results after a call to $PARSE(A,i,j,tag)$. Then,

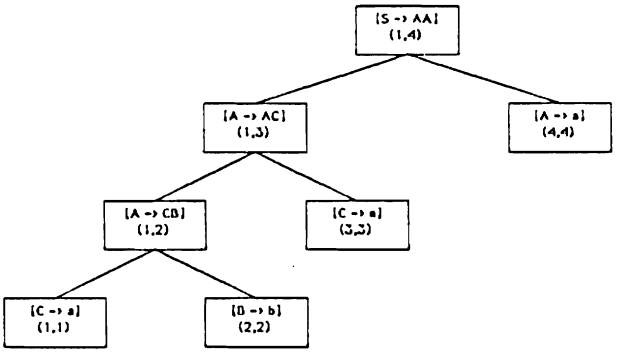
(1) $done(i,j) = true$ iff t is the last parse tree for $A \overset{*}{\Rightarrow} a_i \cdots a_j$.

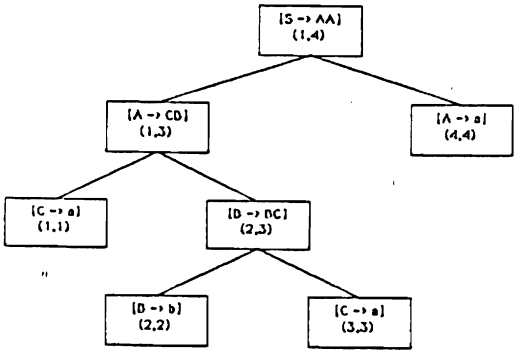
(2) $id(i,j) = k$, $i \le k < j$, iff the root of t has a left subtree whose root is a production in $R(i,k)$ and a right subtree whose root is a production in $R(k+1,j)$. (*id* stands for "index of decomposition".)

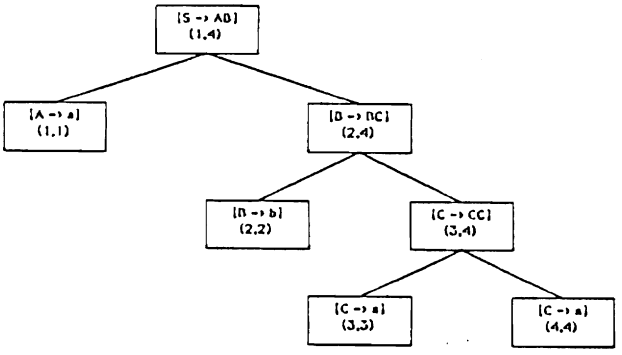
(3) $last\_id(i,j) = true$ iff $id(i,j)$ is the largest integer k satisfying (2).

Procedure *PARSE* is given below. In the procedure, each $R(i,j)$ is treated as an *ordered* subset of productions, so that we can refer to the first, second, etc., production in the set.









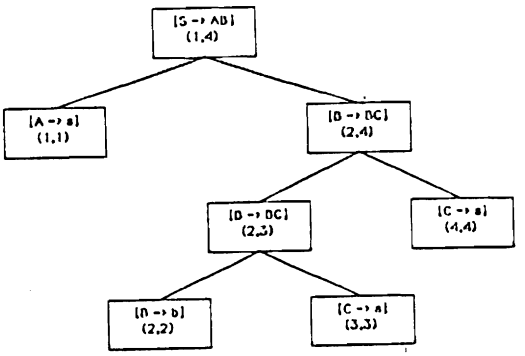


Figure 2.2. Parse trees for $S \stackrel{*}{\Rightarrow} abaa$.

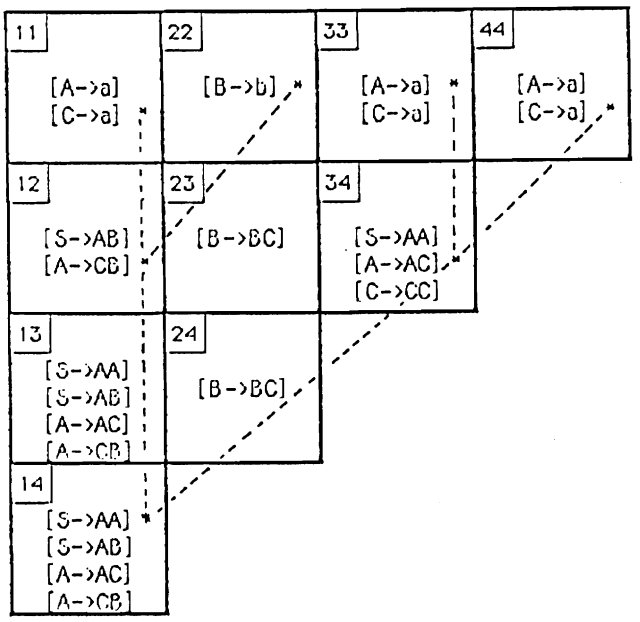


Figure 2.3. Representing a parse tree in the recognition matrix.

```
procedure PARSE (A ,i ,j ,tag );
begin
    if (i = j) then
        if R (i ,i) has a marked production then UNMARK (i ,i) endif;
        mark the production [A → a_i] in R (i ,i);
        id (i ,i) ← 0; done (i ,i) ← last_id (i ,i) ← true
    else
        case tag of
        CURRENT:
            /* there is a marked production in R (i ,j) */
            let [A → BC] be the marked production in R (i ,j);
            k ← id (i ,j);
            PARSE (B ,i ,k ,CURRENT);
            PARSE (C ,k+1,j ,CURRENT);
        FIRST:
            if R (i ,j) has a marked production then UNMARK (i ,j) endif;
            mark the first production π = [A → BC] in R (i ,j) whose LHS = A;
            (id (i ,j), last_id (i ,j)) ← MATCH (B ,C ,i ,j ,i);
            PARSE (B ,i ,id (i ,j),FIRST);
            PARSE (C ,id (i ,j)+1,j ,FIRST);
```

```
        NEXT:
            /* there is a marked production in R(i,j) */
            let [A → BC] be the marked production in R(i,j);
            k ← id(i,j);
            if not done(k+1,j) then
                PARSE(B,i,k,CURRENT);
                PARSE(C,k+1,j,NEXT)
            elseif done(k+1,j) and not done(i,k) then
                PARSE(B,i,k,NEXT);
                PARSE(C,k+1,j,FIRST)
            else /* done(i,k) and done(k+1,j) */
                UNMARK(i,k); UNMARK(k+1,j);
                if not last_id(i,j) then
                    (id(i,j), last_id(i,j)) ← MATCH(B,C,i,j,k+1);
                    PARSE(B,i,id(i,j),FIRST);
                    PARSE(C,id(i,j)+1,k,FIRST)
                else
                    unmark the currently marked production in R(i,j);
                    mark the next production π' = [A → DE] whose LHS = A;
                    (id(i,j), last_id(i,j)) ← MATCH(D,E,i,j,i);
                    PARSE(D,i,id(i,j),FIRST);
                    PARSE(E,id(i,j)+1,j,FIRST)
                endif;
            endif;
        endcase;
        temp ← done(i,id(i,j)) and done(id(i,j)+1,j) and last_id(i,j);
        if (temp) and (π is the last production in R(i,j) whose LHS = A) then
            done(i,j) ← true
        else
            done(i,j) ← false
        endif;
    endif;
end PARSE.
```

In the procedure, subroutine *UNMARK*(i,j) deletes all marks on productions in the subset of entries $\{R(a,b) \mid i \le a \le b \le j\}$. This has the effect of deleting the subtree whose root is a production in $R(i,j)$ (this subtree no longer belongs in the parse tree being generated).

Subroutine *MATCH*(B,C,i,j,k) returns a pair of values $(l,last)$, where l is an integer satisfying $k \le l < j$ and $last \in \{true, false\}$. Specifically, *MATCH* does the following: It looks at the pairs $[R(i,l), R(l+1,j)]$, $k \le l < j$, in increasing value of l then returns the least l such that

(*) there is some production in $R(i,l)$ whose LHS = B and
there is some production in $R(l+1,j)$ whose LHS = C.

In addition, if there is no other integer $> l$ satisfying (*), it returns $last = true$; otherwise, it returns $last = false$.

The main program that calls *PARSE* is given below:

```
begin
    if there is a production in R(1,n) whose LHS = S then
        PARSE(S,1,n,FIRST)
    endif;
    while not done(1,n) do
        PARSE(S,1,n,NEXT);
    endwhile;
end.
```

One can verify that running the main program using the recognition matrix of Figure 2.1 outputs the parse trees of $w = abaa$ in the order shown in Figure 2.2.

For the time complexity, it is clear that constructing the recognition matrix takes $O(n^3)$ time. Each call to $PARSE(S,1,n,tag)$ in the main program takes $O(n^2)$ steps. This follows from the fact that since the grammar is in Chomsky normal form, a parse tree has $2n-1$ nodes (productions). For each production, at most one call to subroutine *MATCH* is performed to determine its children, and this takes $O(n)$ time. Moreover, all calls to *UNMARK* within *PARSE* takes at most $O(n^2)$ steps. Thus, the total running time is $O(n^3 + mn^2)$, where m is the number of distinct parse trees of the input string. Note that the second term dominates when $m = \Omega(n)$.

3. The Systolic Array Model

The systolic parsing algorithm is essentially a parallelization of the sequential algorithm described in the previous section. The systolic array that implements the algorithm is illustrated in Figure 3.1. It consists of two triangular arrays: the P-array (the square nodes) and the Q-array (the circular nodes). Both triangular arrays have n processors along each dimension, where n is the length of the input string to be parsed. The processors are assumed to be indexed as shown. For the P-array, $P(i,j)$ denotes the processor in the i-th leftmost column, of the j-th row. For the Q-array, $Q(i,j)$ denotes the processor in the i-th rightmost column, of the $(j-i+1)$-st row. For convenience, we call a processor of the P-array (Q-array) as a P-processor (Q-processor). The processors are interconnected as shown in the figure. All communication links are assumed to be bi-directional (i.e., data can travel in either direction).

The operation of the systolic array is synchronous, i.e., computations take place at distinct clock cycles. The input is the string $a_1 a_2 \cdots a_n$ to be parsed, followed by an end-of-input marker \$. This input is fed serially to processor $P(1,1)$ of the P-array; a_i is input at clock cycle i, $1 \le i \le n$, and \$ at clock cycle $n+1$. The parse trees (if any) of the input string are generated in "stages". At the end of each stage, a new parse tree would be stored "on-the-fly" in the Q-array; more precisely, if the parse tree contains a production from $R(i,j)$, then this production would be stored in processor $Q(i,j)$.

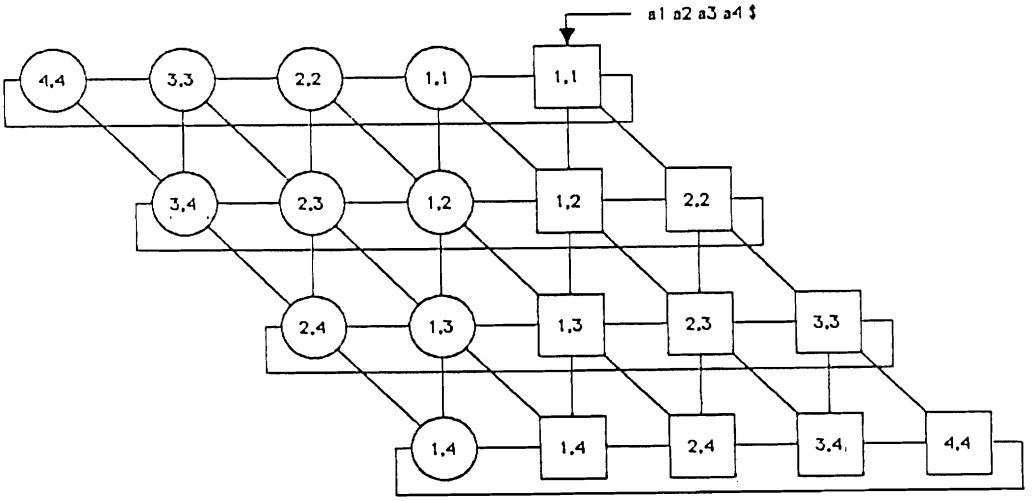


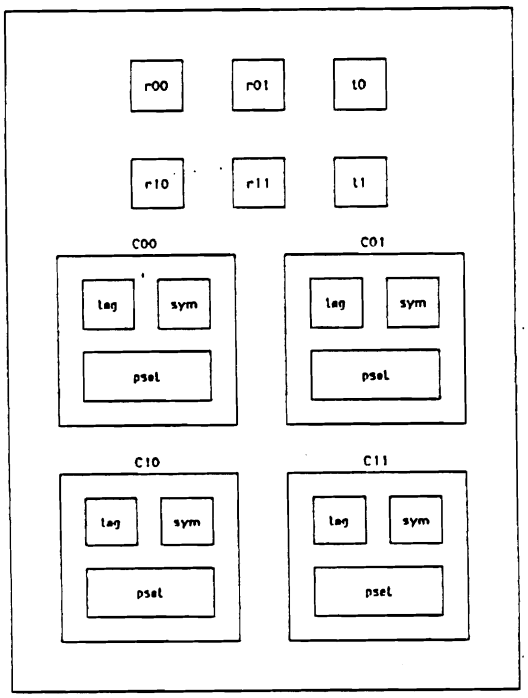
Figure 3.1. Systolic array model.

Each processor has a local memory consisting of fixed number of registers. In describing the systolic algorithm, it is convenient to give names to some of these registers, as shown in Figure 3.2. A P-processor has six registers r_{pq} and t_p ($p, q \in \{0,1\}$), each capable of holding an ordered subset of productions of the underlying grammar. In addition, it has four *cells*, C_{pq} ($p, q \in \{0,1\}$), where a cell is a collection of three registers: *tag*, *sym*, and *pset*. Register *tag* can hold a value from the set {*FIRST*, *CURRENT*, *NEXT*}, *sym* can hold a single nonterminal symbol, and *pset* can hold an ordered subset of productions. A Q-processor has five registers: p, *done*, *ldone*, *rdone*, *id* and *last_id*. Registers *done*, *ldone*, *rdone* and *last_id* can hold boolean values; p can hold a single production. Finally, *id* can hold values of the form (l,b) where l is an integer in the range $0 \le l \le n$ and $b \in \{0,1\}$. We shall explain the use of these registers in subsequent sections.

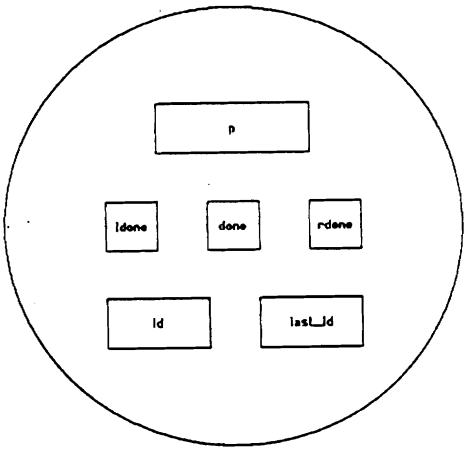
As in the sequential case, the systolic parsing algorithm consists of two phases: a *recognition* phase which computes the recognition matrix, and a *parse generation* phase which outputs the parse trees. The recognition phase is similar to the one described in [CHAN87]; the difference is that the algorithm in [CHAN87] was given in terms of a sequential machine characterization of the systolic array. The algorithm presented here is the "systolic version" of the sequential machine in [CHAN87]. Using the same sequential machine, [CHAN87] also describes how to output a *single* parse tree of the input string. Here, we present a parse generation phase that outputs all such parse trees with only a small increase in the space complexity.

4. The Systolic Recognition Phase

The systolic recognition phase computes the recognition matrix R and determines whether the input string $a_1 a_2 \cdots a_n$ is in the language generated by the grammar. During this phase, only the processors of the P-array take part in the computation; the Q-array is not used.



(a)



(b)

Figure 3.2. Memory organization of (a) a P-processor and (b) a Q-processor.

The recognition phase has the property that the movement of data in the P-array is only from lower-indexed to higher-indexed processors (i.e., from left to right and from top to bottom). We take advantage of the uniformity of the data flow by introducing the notion of a *forward sweep*, which simplifies the description of the computational steps involved. For a processor p of the P-array, let d_p be the rectilinear distance (i.e., counting only horizontal and vertical links) of p from processor $P(1,1)$. Then, p is said to be at *forward sweep* s iff it is at clock cycle $d_p + s$. For example, forward sweep 1 is clock cycle 1 for $P(1,1)$, clock cycle 2 for $P(1,2)$, clock cycle 3 for $P(2,2)$ and $P(1,3)$, etc. The important thing to note is that in a given forward sweep, a processor is "viewed" one clock cycle *earlier* than the neighboring processors to its right or below it. Thus, a computation that takes place in the former processor *can* affect the latter processors also at the same forward sweep.

Conceptually, the recognition phase starts, for all processors, at forward sweep 1 and ends at forward sweep $n+1$. With respect to processor $P(1,1)$, these correspond to the first $n+1$ clock cycles during which it reads the input $a_1 a_2 \cdots a_n \$$. During each forward sweep, the P-array computes a new portion of the recognition matrix; in particular, at forward sweep s, $1 \le s \le n$, only the set of entries $\{R(a,s) \mid 1 \le a \le s\}$ are computed.

Matrix entries are computed only at processors $P(j,j)$, $1 \le j \le n$, henceforth called *primary* processors. A primary processor computes one or more such entries but at different forward sweeps. More precisely, $P(j,j)$ computes $R(s-j+1,s)$ at forward sweep s, $j \le s \le n$. For example, $P(3,3)$ computes $R(1,3)$, $R(2,4)$, $\cdots$, $R(n-2,n)$ at forward sweeps 3, 4, $\cdots$, n.

The *secondary* processors $P(i,j)$, $1 \le i < j \le n$, play a different role. Suppose that primary processor $P(j,j)$ is assigned to compute entry $R(a,b)$ at some forward sweep. Then, at the *same* forward sweep, the secondary processors to the left of $P(j,j)$ would have stored in their local memory the set of "convolving pairs" $\{[R(a,c), R(c+1,b)] \mid a \le c < b\}$ which are needed to compute the value of $R(a,b)$. The mapping from convolving pairs to secondary processors is best explained by means of an example. Consider the case when processor $P(5,5)$ wishes to compute $R(2,6)$ at forward sweep 6. Then the required convolving pairs $\{[R(2,c), R(c+1,6)] \mid 2 \le c < 6\}$ would be stored in processors $P(1,5)$, $\cdots$, $P(4,5)$ as shown in Figure 4.1-(a). Intuitively, the mapping is obtained by first listing the convolving pairs $\{[R(a,c), R(c+1,b)]\}$ in increasing order of c, then "folding" the list about the middle as shown in Figure 4.1-(b). (As we shall see later, this "folded" mapping guarantees that data can be routed among processors using only nearest-neighbor connections.)

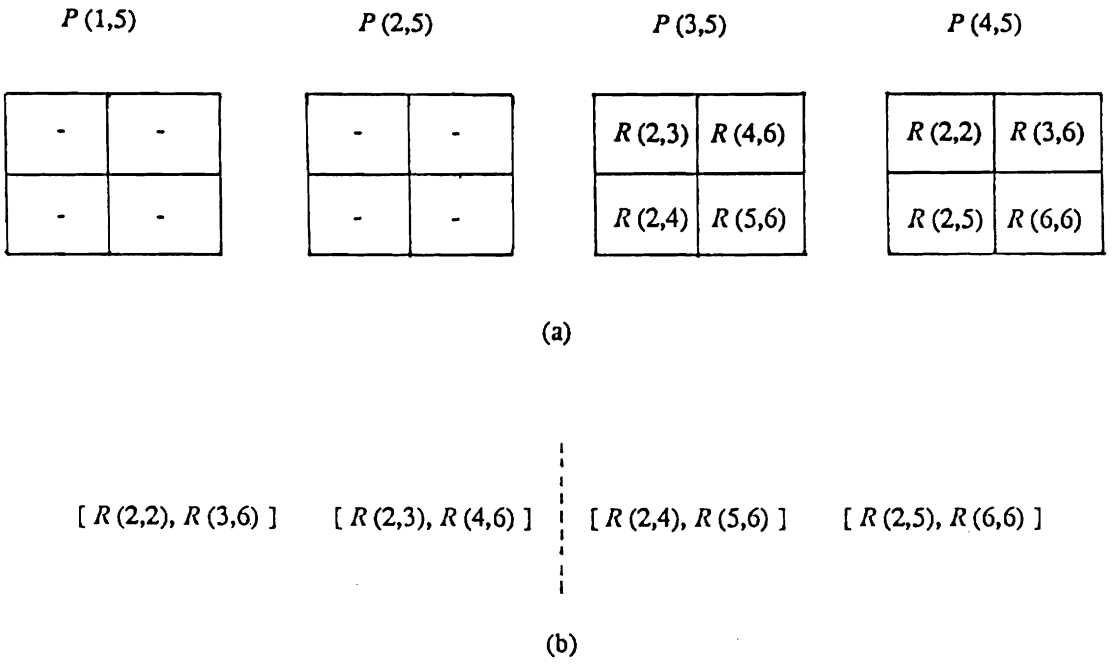


Figure 4.1. Mapping from convolving pairs of $R(2,6)$ to secondary processors.

The formal mapping is given by Invariants 4.1 and 4.2 below. The processors use the four r_{pq} registers to store the entries. The notation $r_{pq}(i,j,s)$ means the contents of register r_{pq} of processor $P(i,j)$ at forward sweep s.

Invariant 4.1. For $1 \le i < j \le s \le n$,

$$r_{00}(i,j,s) = \begin{cases} \varnothing & \text{if } 2i < j \\ R(s-j+1,s-i) & \text{otherwise} \end{cases}$$

$$r_{01}(i,j,s) = \begin{cases} \varnothing & \text{if } 2i < j \\ R(s-i+1,s) & \text{otherwise} \end{cases}$$

$$r_{10}(i,j,s) = \begin{cases} \varnothing & \text{if } 2i \le j \\ R(s-j+1,s-j+i) & \text{otherwise} \end{cases}$$

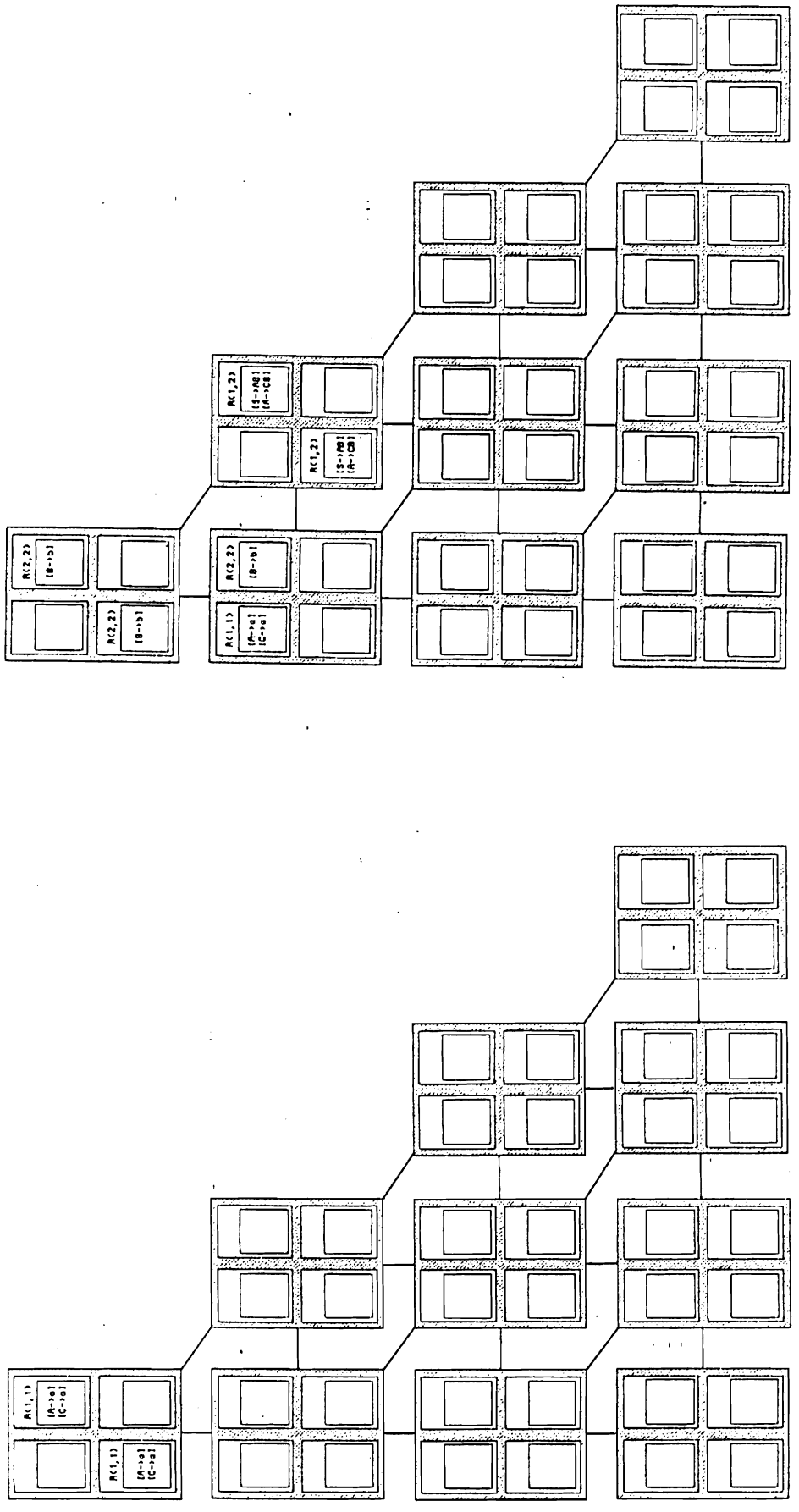
$$r_{11}(i,j,s) = \begin{cases} \varnothing & \text{if } 2i \le j \\ R(s-j+i+1,s) & \text{otherwise.} \end{cases}$$

Invariant 4.2. For $1 \le j \le s \le n$,

$$r_{00}(j,j,s) = r_{11}(j,j,s) = \varnothing,$$
$$r_{01}(j,j,s) = r_{10}(j,j,s) = R(s-j+1,s).$$

Invariants 4.1 and 4.2 specify the register values for secondary and primary processors, respectively. All registers are assumed to be initialized to the empty set $\varnothing$. Observe from Invariant 4.1 that some secondary processors may have some registers permanently set to $\varnothing$; this indicates that no matrix entry is mapped onto the register. Moreover, for primary processors (see Invariant 4.2), r_{00} and r_{11} are always $\varnothing$, and r_{01} and r_{10} hold the computed entry. Although one register should be sufficient, this mapping simplifies the routing of data (to be explained later). Finally, the invariants define the register values of $P(i,j)$ only for forward sweeps $s \ge j$. If $s < j$, the registers of $P(i,j)$ retain their initial values $\varnothing$. Figure 4.2 illustrates the register values for a 4×4 P-array at forward sweeps 1 through 4.

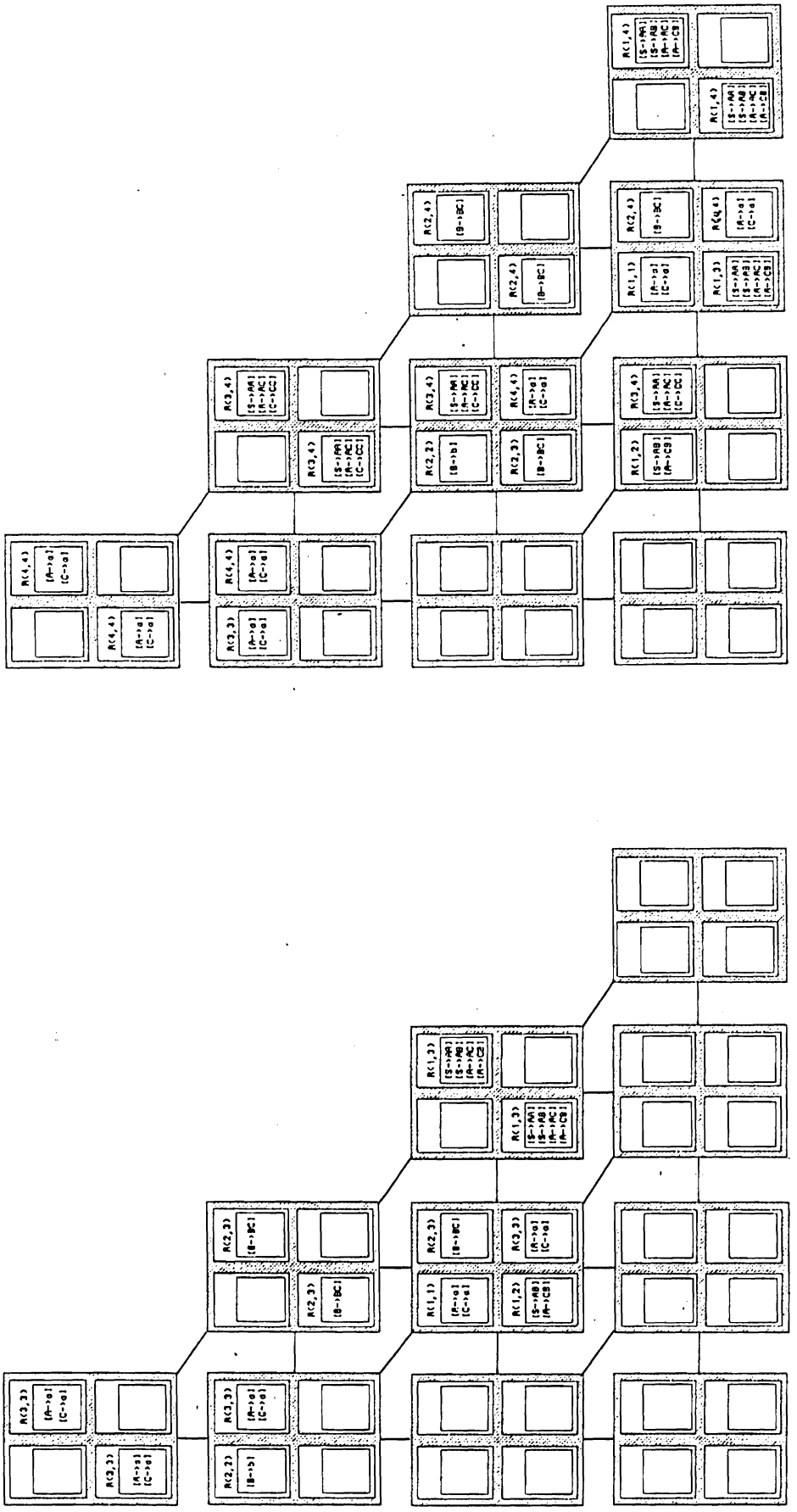
It is easy to see how Invariant 4.2 can be realized for every primary processor given that Invariant 4.1 holds for secondary processors. For a given forward sweep, Invariant 4.1 states that all the convolving pairs required to compute the entry at the primary processor are available in the secondary processors to its left. Thus, the desired value is simply the union, over all secondary processors, of $(r_{00}{}^*r_{01}) \cup (r_{10}{}^*r_{11})$. This value can be computed as follows: Each processor has a left input terminal IN_v and a right output terminal OUT_v (for a processor in the leftmost column other than $P(1,1)$, IN_v is assumed to be permanently set to $\varnothing$). At the start of each forward sweep, the processor receives a value from IN_v, computes $IN_v \cup (r_{00}{}^*r_{01}) \cup (r_{10}{}^*r_{11})$ then sends the result to OUT_v. The output from OUT_v then travels with unit-delay to



forward sweep 1

forward sweep 2

Figure 4.2. Contents of the r_{pq} registers at the end of forward sweeps 1 and 2.



forward sweep 3 forward sweep 4

Figure 4.2. (cont.) Contents of the r_{pq} registers at the end of forward sweeps 3 and 4.

the IN_v terminal of the next processor. It is clear that the value that arrives at the primary processor is the desired matrix entry. The primary processor then stores this value in its r_{01} and r_{10} registers. Processor $P(1,1)$ is a special case: we let IN_v be the terminal from which it receives the input string $a_1 a_2 \cdots a_n\$$. At forward sweep i, $1 \le i \le n$, $P(1,1)$ reads a_i from IN_v, computes the set $\{[A \rightarrow a_i] \in P\}$, then stores the result in its r_{01} and r_{10} registers.

Once computed by a primary processor, an entry is routed to various secondary processors to participate in the computation of new entries. Invariant 4.1 gives the desired mapping. We now specify the required data routing steps. Each processor has four input terminals IN_{pq} and four output terminals OUT_{pq} ($p, q \in \{0,1\}$) connected to neighboring processors as shown in Figure 4.3. More precisely, the IN_{00} and IN_{11} terminals of processor $P(i,j)$ receive data from the OUT_{00} and OUT_{11} terminals, respectively, of processor $P(i-1,j-1)$, and the IN_{01} and IN_{10} terminals receive data from the OUT_{01} and OUT_{10} terminals, respectively, of processor $P(i,j-1)$. (For processors with non-existent neighbors along the directions shown, the relevant inputs are assumed to be $\varnothing$.) Data items travel through the communication links at different speeds. In particular, outputs from terminals OUT_{00}, OUT_{01}, OUT_{10} and OUT_{11} reach their destinations 3, 1, 2, and 2 clock cycles later, respectively (indicated in the figure by the number of black squares in each link).

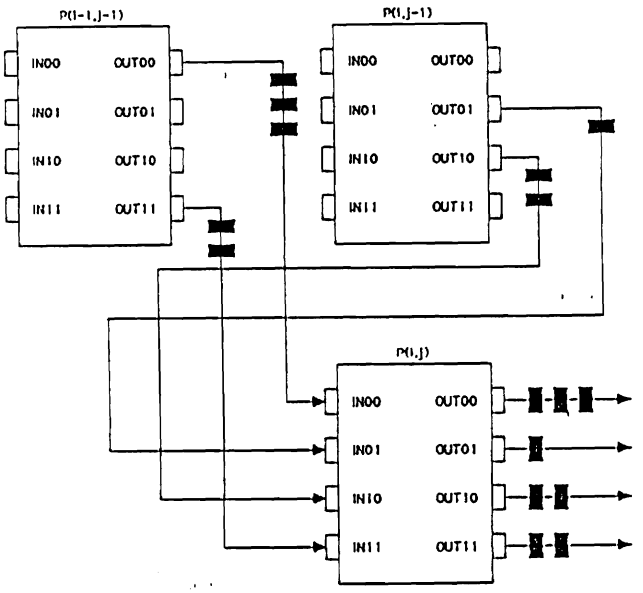


Figure 4.3. The IN_{pq} and OUT_{pq} terminals of a P-processor and their interconnections.

For a secondary processor, data arriving at the IN_{pq} terminals are used to update its local registers, as depicted in Figure 4.4. For processors $P(i,j)$ satisfying $2i \neq j$, register r_{pq} is updated to the value received from IN_{pq}; similarly, OUT_{pq} gets the value of r_{pq}. For processors $P(i,j)$ satisfying $2i = j$, the input terminals are switched for r_{00} and r_{10}, and the output terminals are switched for r_{01} and r_{11}. For a

primary processor, inputs (if any) arriving at the IN_{pq} terminals are ignored. After storing the newly computed entry in its registers, the processor routes the register contents to the associated output terminals the same way as described.

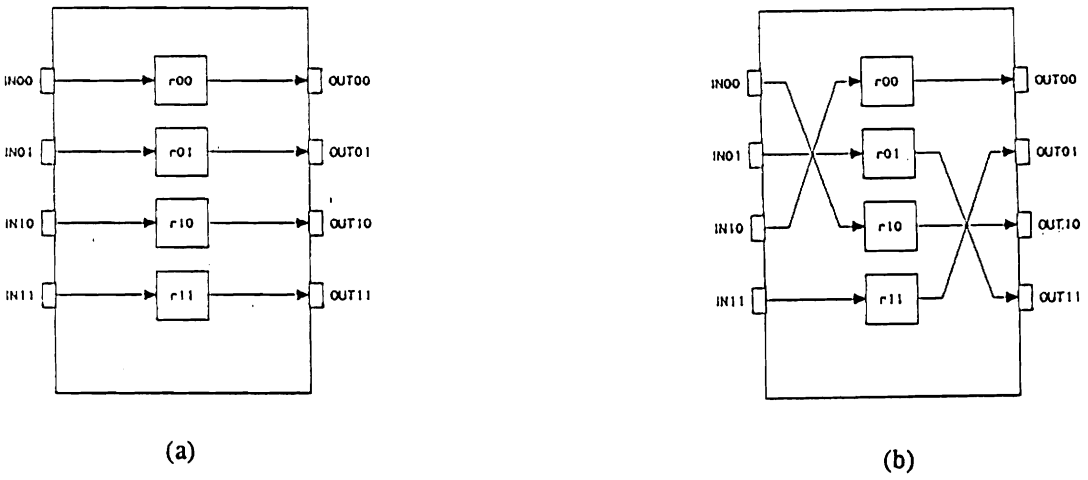


Figure 4.4. Updating the r_{pq} registers of processor $P(i,j)$ for the case (a) $2i \neq j$ and (b) $2i = j$.

For processor $P(i,j)$, the above data routing step (and the associated computational step which computes the convolutions) is performed at every forward sweep $s \geq j$. For forward sweeps $s < j$, the processor is "inactive". The processors can be activated at the right forward sweeps as follows: At clock cycle 1 (when the first input symbol is read), processor $P(1,1)$ generates a "start" control signal which travels downwards with 2-delay (i.e., hops from processor to processor every 2 clock cycles) and to the right with unit-delay. One can easily verify that the "start" signal reaches processor $P(i,j)$ at forward sweep $s = j$.

At this point, we explain the use of registers t_0 and t_1 in each processor (see Figure 3.2). At the clock cycle when a processor receives the "start" signal, it also copies into its t_0 and t_1 registers, the updated contents of its r_{01} and r_{11} registers, respectively. In subsequent clock cycles, the contents of t_0 and t_1 are left unchanged. The information stored in these registers will be used later in the parse generation phase.

The computational and data routing steps previously described guarantee that Invariants 4.1 and 4.2 hold for all processors of the P-array. In particular, at the end of forward sweep n, processor $P(n,n)$ would have computed the value of $R(1,n)$. The proof is straightforward induction (on the sweep number and processor index) and is left to the reader (see also [CHAN87]).

Forward sweep $n+1$ (at which processor $P(1,1)$ reads the end-of-input marker $\$$) is used to terminate the recognition phase for all processors. When $\$$ is read, processor $P(1,1)$ issues a "halt" signal which travels downwards and to the right with unit-delay. When received by a processor other that $P(n,n)$, the processor terminates its computation. For processor $P(n,n)$, it checks if $R(1,n)$ (which is stored in its r_{01} and

r_{10} registers) contains a production whose LHS is the start symbol S. If there is no such production, it sends a "reject" signal back to processor $P(1,1)$ and the systolic array halts. Otherwise, $P(n,n)$ initiates the *parse generation* phase described in the next section.

Remark 4.1. We have some final remarks about the recognition phase. If one wishes only to determine whether the input string is in the language generated by the grammar, then the systolic array need not execute the next phase. In this case, one gets the answer from processor $P(n,n)$ at the end of forward sweep $n+1$, which corresponds to clock cycle $3n-1$. Furthermore, observe that every processor stores in its registers, values which are dependent only on the size of the grammar and not on the length of the input (i.e., the processor is finite-state). It is also a simple exercise to modify the systolic algorithm just described so that each processor does not need to know its index (e.g., as is required to distinguish processors $P(i,j)$ such that $2i = j$).

5. The Systolic Parse Generation Phase

The systolic parse generation phase is essentially a parallelization of procedure *PARSE* described in Section 2. During this phase, both P-array and Q-array take part in the computation. Conceptually, the phase is divided into m stages, where m is the number of distinct parse trees of the input string. At the end of each stage, a new parse tree is stored "on-the-fly" in the Q-array; more precisely, if the parse tree contains a production from $R(i,j)$, then this production would be stored in processor $Q(i,j)$.

Every stage begins with processor $P(n,n)$ issuing a "begin-parse" control signal which reaches all other processors of the P-array and Q-array by moving upwards and to the left with unit-delay. Thus, a processor a (rectilinear) distance d away from $P(n,n)$ receives the signal d clock cycles later. For a processor, let *reverse sweep* 1 (of the current stage) be the clock cycle at which it receives the "begin-parse" signal. Then, reverse sweep 2 is the next clock cycle, reverse sweep 3 the clock cycle after reverse sweep 2, etc. A reverse sweep is just like a forward sweep, the only difference being that in a given reverse sweep, a processor is "viewed" one clock cycle earlier that the neighboring processors to its *left* and *above* it.

The parse tree that is eventually stored in the Q-array at the end of each stage is output from the P-array. Informally, the P-array identifies and "marks" the productions making up the parse tree from the recognition matrix entries stored in its primary processors. The mapping described in the previous section is especially suited for carrying this "marking" process since at every forward sweep, the convolving pairs of the entry computed at a primary processor are all stored in the secondary processors to its left. Thus, if a production, say $[A \rightarrow BC]$, has already been identified as part of the parse tree at some primary processor, then the children of this production in the parse tree can be obtained by performing a "search" of the convolving pairs stored in the secondary processors (i.e., find a register-pair $[r_{p0}, r_{p1}]$ such that B is the LHS of some production in r_{p0} and C is the LHS of some production in r_{p1}). To do this, however, the flow of information should now be from right-to-left (rather than from left-to-right as is the case for a forward sweep). Moreover, since at the end of forward sweep n the primary processors only hold the set of entries $\{R(a,n) \mid 1 \leq a \leq n\}$, the "lost" entries should somehow be recovered.

The trick is to be able to "reconfigure" the P-array such that at reverse sweep 1, 2, ..., n, every processor holds the same memory contents that it had at forward sweep n, $n-1$, ..., 1, respectively. That this can be accomplished follows from the observation that during the recognition phase, every newly computed entry starts from an r_{pq} register of a primary processor then follows a unique directed path through the P-array. Moreover, the path always ends either at a t_p register at some forward sweep $s \leq n$ (after which the t_p register is no longer changed) or at an r_{pq} register at forward sweep n. Thus, the r_{pq} and t_p registers at the end of the recognition phase contain all the entries computed in all n forward sweeps; in n reverse sweeps these entries can be sent back to their previous locations by routing them along the paths opposite to what they took during the recognition phase.

In order not to lose the information stored in the r_{pq} and t_p registers at the end of the recognition phase (they will be required at the start of each new stage), we instead use the cells of the P-array for storing and routing the data (see Figure 3.2). In particular, we let register *pset* of cell C_{pq} (or $pset(C_{pq})$ for short) take the place of register r_{pq}. For example, for $n = 4$, the contents of the *pset* registers of the P-array at reverse sweeps 1 through 4 would be identical to those shown in Figure 4.2, except that reverse sweep 1 corresponds to forward sweep 4, reverse sweep 2 corresponds to forward sweep 3, etc.

The "routing scheme" for cells is essentially the reverse of that shown in Figure 4.4: simply replace "r_{pq}" by "C_{pq}" and reverse the directions of all the arrows. The delays associated with the links (see Figure 4.3) remain the same. (To route a cell we mean to route the contents of the three registers *tag*, *sym* and *pset* that make up the cell.) Processor $P(i,j)$ performs the routing step for its cells at every reverse sweep. There are two exceptions: The first is reverse sweep 1, when processor $P(i,j)$ updates $pset(C_{00})$ and $pset(C_{10})$ to r_{00} and r_{10}, respectively, instead of getting the data as inputs (which turn out to be nonexistent at reverse sweep 1). The second exception is reverse sweep $n-j+1$, when processor $P(i,j)$ instead updates $pset(C_{01})$ and $pset(C_{11})$ to t_0 and t_1, respectively; this has the opposite effect of copying r_{01} and r_{11} into t_0 and t_1, respectively, at forward sweep j. (We shall explain later how processor $P(i,j)$ would know when it is at reverse sweep $n-j+1$).

We now describe the computational steps performed by the P-array. At each reverse sweep, a processor carries out the computational steps only after it has updated its cells. The heart of the computation is an "instruction" called *MATCH* which is issued by a primary processor to all secondary processors to its left. *MATCH* can be thought of the "systolic equivalent" of subroutine *MATCH* in procedure *PARSE*. In general, this instruction has the form $MATCH(\pi, (tag_1,tag_2), id, last\_id)$ where

- π is a production in P,
- $tag_1, tag_2 \in \{FIRST, CURRENT, NEXT, NULL\}$,
- $id = (l,b)$ where l is an integer such that $0 \leq l \leq n$ and $b \in \{0,1\}$, and
- $last\_id \in \{true, false\}$.

For a primary processor $P(j,j)$, the cells of the secondary processors to its left can be thought of as a "chain" of cell-pairs, as depicted in Figure 5.1. When $P(j,j)$ issues a *MATCH* instruction, say, $MATCH([A \rightarrow BC], (tag_1,tag_2), (l,b), last\_id)$, the cell-pairs are "searched" in the order shown in the figure *starting* at cell-pair $[C_{b0},C_{b1}]$ of processor $P(l,j)$ (processors prior to $P(l,j)$ simply propagate the instruction unchanged to the next processor with unit-delay). Now, let $[C_1,C_2]$ be the first cell-pair

satisfying the property that

(*) there is a production in $pset(C_1)$ whose LHS = B and
there is a production in $pset(C_2)$ whose LHS = C.

If $[C_1,C_2]$ is cell-pair $[C_{b'0},C_{b'1}]$ of processor $P(l',j)$ then:

(1) $sym(C_{b'0})$ and $sym(C_{b'1})$ are set to nonterminals B and C, respectively,

(2) $tag(C_{b'0})$ and $tag(C_{b'1})$ are set to tag_1 and tag_2, respectively, and

(3) processor $P(l',j)$ modifies the instruction it sends to its left to $MATCH([A \rightarrow BC], (\text{NULL}, \text{NULL}), (l',b'), last\_id)$.

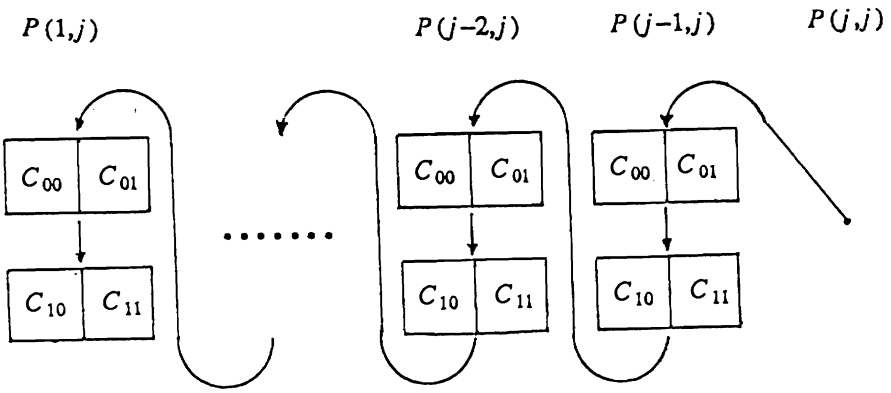


Figure 5.1. The cells of secondary processors to the left of $P(j,j)$ depicted as a "chain" of cell-pairs.

Updating (tag_1,tag_2) to $(NULL,NULL)$ indicates that a match has already been found; the place where the match occurred is given in the new $id = (l',b')$. The rest of the cells-pairs following the one where the match occurred continue to be tested for property (*), this time to determine whether *last_id* needs to be updated. If another match occurs, then *last_id* is updated to *false*; otherwise, it retains its old value.

The *MATCH* instructions leaving the leftmost column of the P-array serve as input to the Q-array. The steps performed by a Q-processor are simple: at each reverse sweep, it shifts the contents of its local registers p, *id* and *last_id* into the corresponding registers of the processor to its left, then updates its own registers to those it receives from the processor to its right. For a Q-processor in the rightmost column, the new contents of its p, *id*, and *last_id* registers are obtained from the π, *id*, and *last_id* arguments, respectively, of the *MATCH* instruction (if any) it receives from the corresponding processor in the P-array. (If no *MATCH* instruction is received, the Q-processor simply clears the three registers.)

For processors of both the P-array and Q-array, the data routing steps and the computational steps associated with the *MATCH* instruction are executed at every reverse sweep starting at reverse sweep 1 (which is when they receive the "begin-parse" signal). For all processors on the j-th row (from the top), reverse sweep $n-j+1$ is the last reverse sweep when these steps are performed. A processor on the j-th row can know when it is at reverse sweep $n-j+1$ as follows: At reverse sweep 1, processor $P(n,n)$ issues

an "end-parse" control signal which travels upwards with 2-delay and to the left with unit-delay. A processor receives this signal at reverse sweep $n-j+1$.

We now explain how the *MATCH* instructions are used to generate a parse tree of the input string. The actions performed by the systolic array for the first stage are slightly different from those of the succeeding stages. We first describe what happens during stage 1.

Stage 1. Stage 1 begins when the "halt" signal indicating the end of the recognition phase reaches processor $P(n,n)$. At this clock cycle, processor $P(n,n)$ issues the "begin-parse" signal to all other processors to start reverse sweep 1 of the stage. At the start of reverse sweep 1, the data routing steps described earlier would place the value of $R(1,n)$ into registers $pset(C_{01})$ and $pset(C_{10})$ of primary processor $P(n,n)$. Moreover, the cells of the secondary processors to its left would hold the convolving pairs of $R(1,n)$. Suppose that $pset(C_{01})$ (or $pset(C_{10})$) has a production whose LHS is the start symbol S. Then, $P(n,n)$ first sets $sym(C_{01})$ to nonterminal symbol S and $tag(C_{01})$ to *FIRST*, then does the following:

(1) Locate the first production $\pi \in pset(C_{01})$ such that LHS(π) = $sym(C_{01})$. Moreover, if π is the last such production, distinguish π by some special symbol, say $\bar{\pi}$ (this information will be used later in the Q-array);

(2) Send *MATCH*(π (or $\bar{\pi}$), (*FIRST*,*FIRST*), ($n-1$,0), *true*) to the processor to its left.

The *MATCH* instruction would search for the first cell-pair $[C_1,C_2]$ which contains a pair of productions that match the right-hand side of π. The cell-pair is then "marked" by updating their *sym* and *tag* registers. In addition, new *id* and *last_id* values would be computed and, together with production π (or $\bar{\pi}$), shifted into the Q-array. Now, the routing scheme for cells would eventually bring the marked cells to primary processors (either as C_{01} or C_{10}) at some reverse sweep. When this happens, the secondary processors to the left of the primary processor would again hold the convolving pairs of the entry stored in the *pset* register of the marked cell. The process then repeats. More precisely, a primary processor $P(j,j)$ receiving a marked cell C (at most one marked cell would arrive at any reverse sweep) checks $tag(C)$ and $sym(C)$. If $tag(C)$ = *FIRST*, then it performs step (1) above for C, and issues a *MATCH* instruction as in step (2), except that the fourth argument is ($j-1$,0). For primary processors not receiving a marked cell, no *MATCH* instruction is generated.

The marking process continues until the processors receive the "end-parse" signal. For the sample grammar and input string given in Figure 2.1, the configurations of the P-array and Q-array for the n reverse sweeps (n = 4) is shown in Figure 5.2. In particular, at the end of reverse sweep n, the Q-array would have stored in its p registers a parse tree of the input string. This parse tree can then be read off directly from the Q-array or pipelined out of the Q-array to a host computer. We omit the steps involved as they are relatively straightforward.

The clock cycle at which the "end-parse" signal is received represents the end of the stage for each processor of the P-array. On the other hand, the Q-array performs another step which involves the update of the *ldone*, *done* and *rdone* registers of its processors. This is accomplished as follows: At reverse sweep n (which is also when it receives the "end-parse" signal), processor $Q(n,n)$ sends out an "update" control signal to all other processors of the Q-array, this signal traveling diagonally downwards with 2-

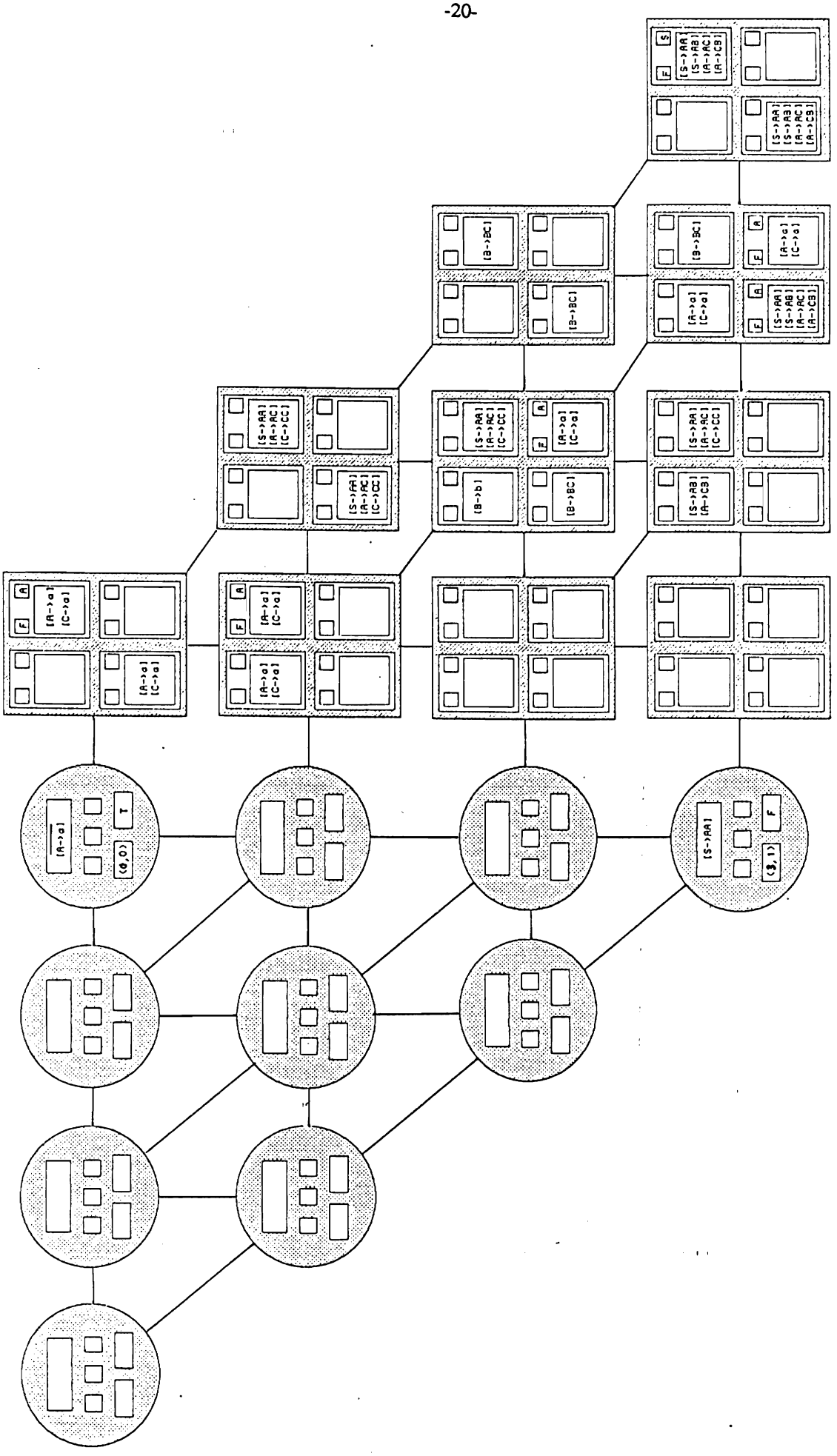


Figure 5.2. Register contents at the end of reverse sweep 1, stage 1.

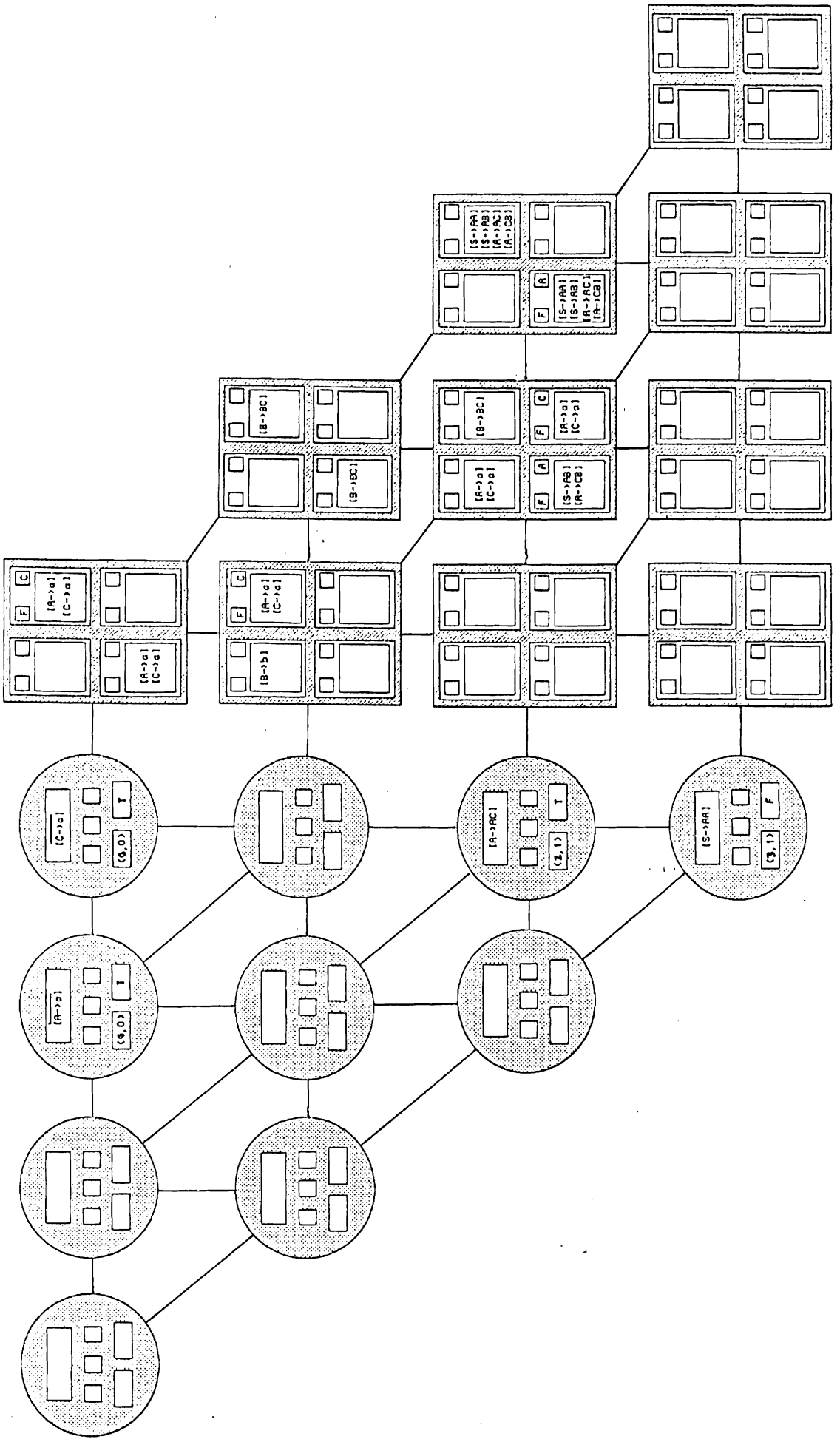


Figure 5.2. (cont.) Register contents at the end of reverse sweep 2, stage 1.

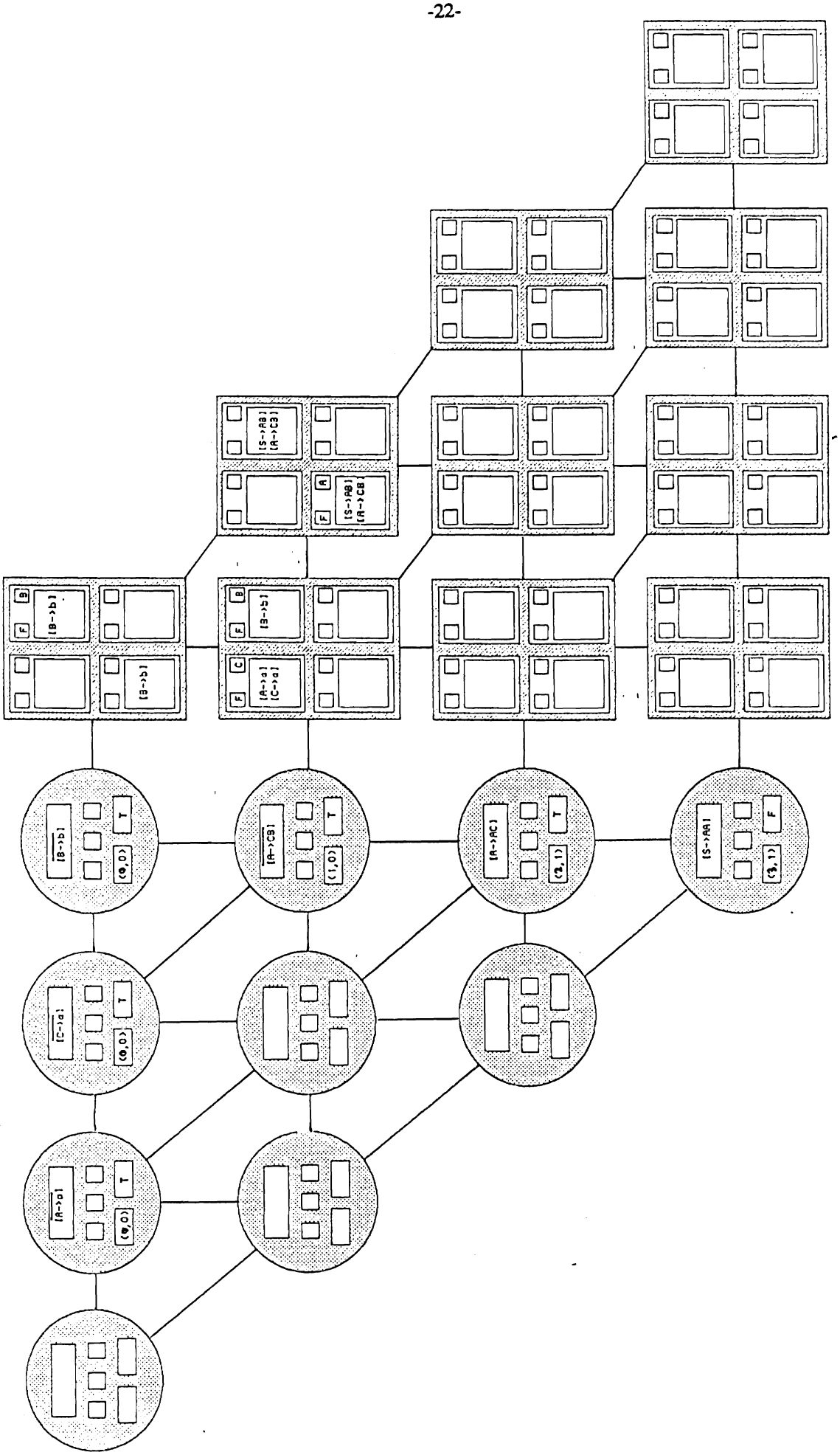


Figure 5.2. (cont.) Register contents at the end of reverse sweep 3, stage 1.

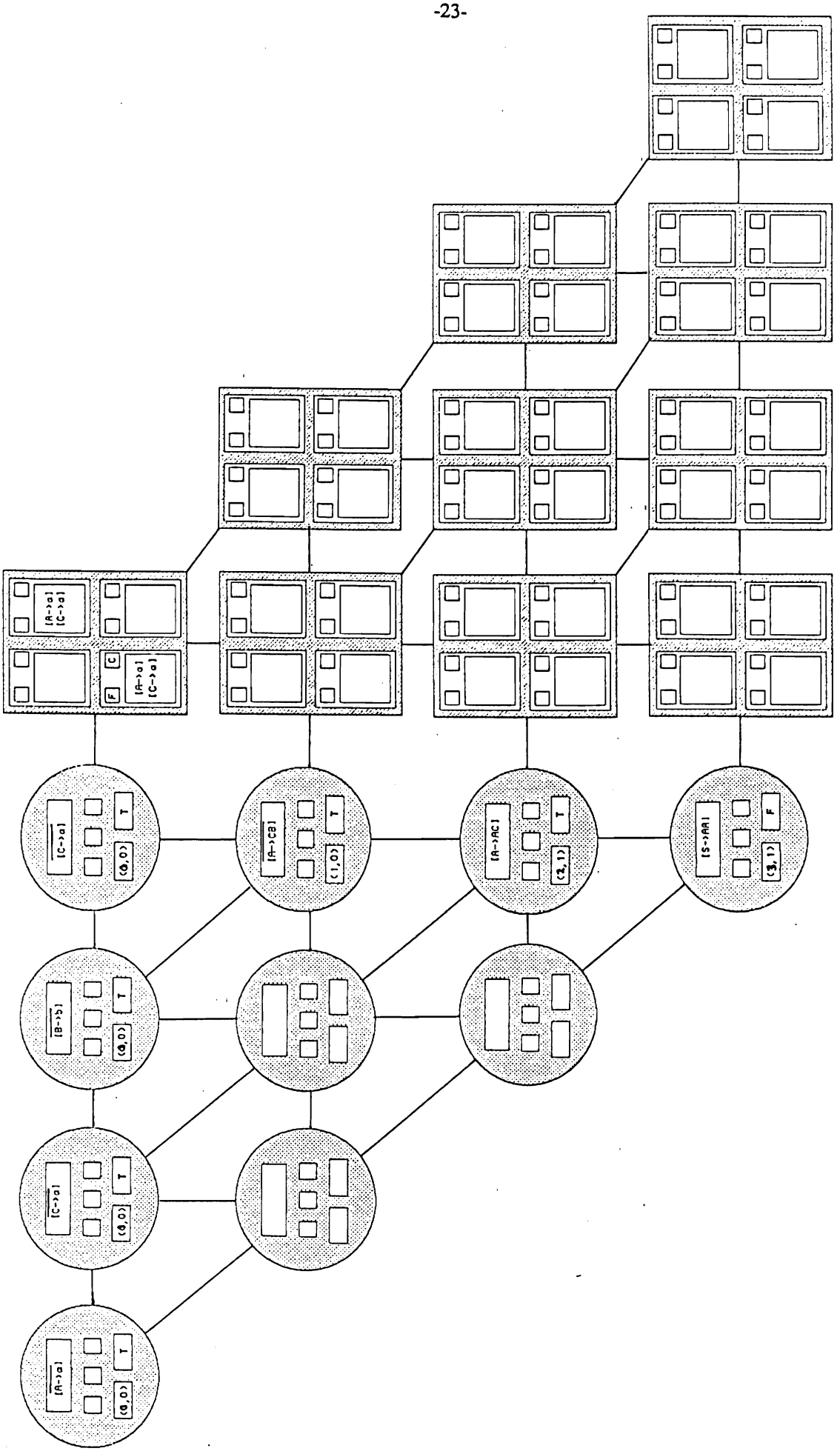


Figure 5.2. (cont.) Register contents at the end of reverse sweep 4, stage 1.

delay and to the right with unit-delay. For processors on the top row of the Q-array (i.e., processors $Q(j,j)$, $1 \leq j \leq n$), the following is performed when they receive the "update" signal: set *ldone* = *rdone* = *done* = *true* and send the contents of *done* diagonally downwards with 2-delay and vertically downwards with unit-delay. For a processor in a lower row, one diagonal input and one vertical input would arrive at the time it receives the "update" signal. The processor then does the following:

(1) If its p register does not contain a production, then clear its *done*, *ldone* and *rdone* registers and route the vertical and diagonal inputs to the next vertical and diagonal processors below it, respectively.

(2) If its p register contains a production, then set *ldone* to the value of the vertical input and *rdone* to the value of the diagonal input. Set *rdone* to *true* iff (*i*) *ldone* = *rdone* = *true*, (*ii*) *last_id* = *true*, and (*iii*) the p register contains a distinguished production $\bar{\pi}$. Otherwise, set *done* to *false*. Route the contents of *done* vertically and diagonally downwards.

For example, after the update step, the Q-array would have the configuration shown in Figure 5.3. After the update step for processor $Q(1,n)$, it sends the contents of all of its local registers to processor $P(n,n)$ of the P-array to begin the next stage. In addition, processor $Q(1,n)$ sends a signal to all processors of the Q-array, this signal traveling upwards and to the left with unit-delay. When received by a Q-processor, it sends the contents of all its local registers to the processor to its left and receives the updated values from the processor to its right. The effect is that the entire parse tree is shifted out of the Q-array and pipelined into the primary processors of the P-array (using the toroidal connections; see Figure 3.1).

Stage k > 1. Each subsequent stage after stage 1 effectively starts at the clock cycle when processor $P(n,n)$ receives an input from processor $Q(1,n)$. At this clock cycle, processor $P(n,n)$ sends the "begin-parse" to all other processors to start reverse sweep 1 of the new stage. The data routing and computational steps performed during the stage are identical to those in stage 1, except for primary processors which now receive inputs from the Q-array. For convenience, we assume that the input to a primary processor is of the form I = (*p*, *ldone*, *done*, *rdone*, *id*, *last_id*). The *MATCH* instruction issued by a primary processor now depends on this input. The main thing to note is that if a primary processor holds an entry $R(a,b)$ then input I represents the register contents of processor $Q(a,b)$ after the update step of the preceding stage. In particular, if argument p of the input holds a production π, then π is in $R(a,b)$ and is part of the parse tree last generated. The rest of the arguments of the input are used by the primary processor to determine how the next parse tree would be generated, in a manner similar to that performed by procedure *PARSE*.

The steps executed by a primary processor are as follows. (It is instructive to compare these steps with procedure *PARSE*). At reverse sweep 1, processor $P(n,n)$ sets $sym(C_{01}) = S$ as before. This time, however, it checks the value of *done* from input I. If *done* = *true*, then the parse tree from the previous stage is the last one and $P(n,n)$ sends a signal to all processors to halt all computation. If *done* = *false*, then there is a next parse tree, in which case processor $P(n,n)$ sets $tag(C_{01})$ to *NEXT*.

The following steps are also performed by every primary processor $P(j,j)$ that has a marked cell C (C is C_{01} for processor $P(n,n)$):

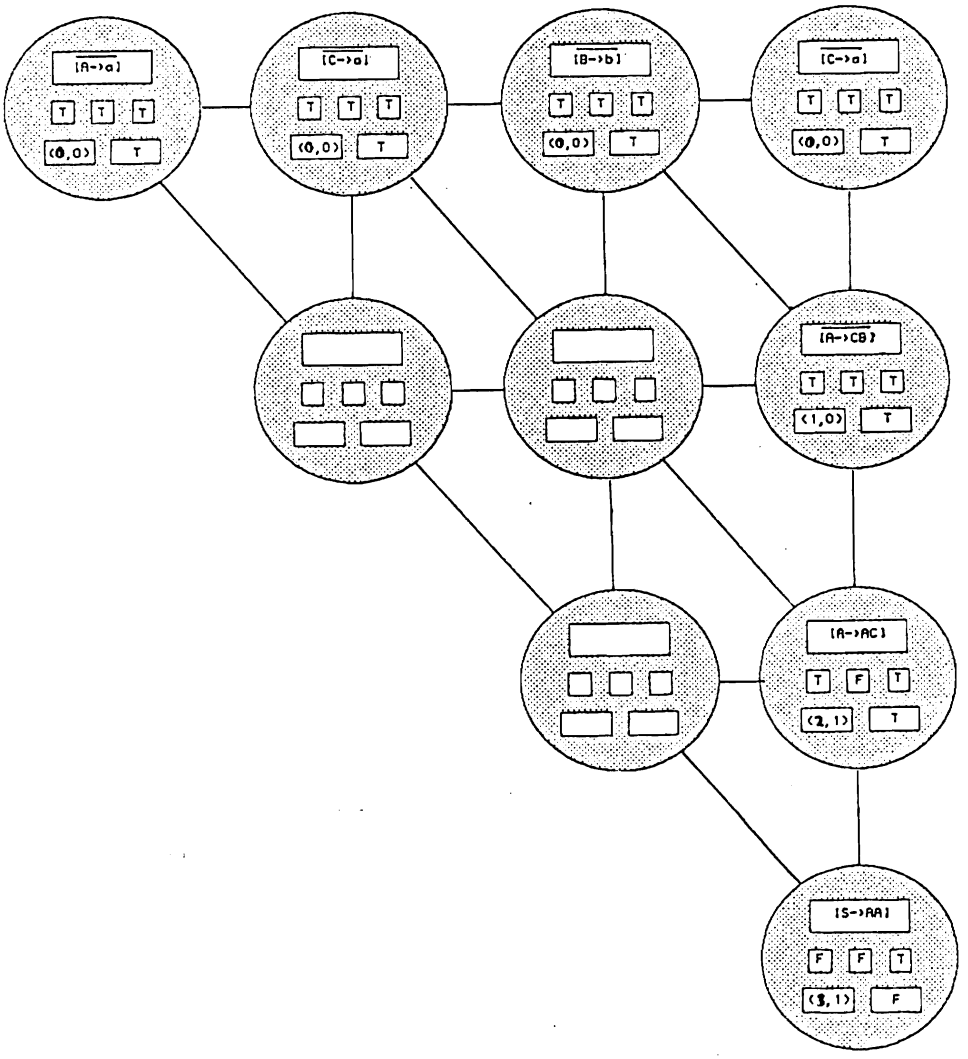


Figure 5.3. Register contents of the Q-array after the update step of stage 1.

(1) If $tag(C) = CURRENT$, then output $MATCH(p, (CURRENT, CURRENT), id, last\_id)$, where p, id, and $last\_id$ are from input I.

(2) If $tag(C) = FIRST$, then locate the first production $\pi \in pset(C)$ such that $\mathrm{LHS}(\pi) = sym(C)$. If this also the last such production, distinguish π as $\overline{\pi}$. Output $MATCH(\pi$ (or $\overline{\pi})$, $(FIRST, FIRST)$, $(j-1,0)$, $true)$.

(3) If $tag(C) = NEXT$, check input I and do the following:

(a) If $rdone = false$, output $MATCH(p, (CURRENT, NEXT), id, last\_id)$.

(b) If $rdone = true$ and $ldone = false$, output $MATCH(p, (NEXT, FIRST), id, last\_id)$.

(c) If *rdone* = *true* and *ldone* = *true*, check *last_id*. If *last_id* = *false*, then output *MATCH*(p, (*FIRST*,*FIRST*), id', *true*) where id' is defined as follows: if $id = (l,0)$ then $id' = (l,1)$; if $id = (l,1)$ then $id' = (l-1,0)$ (this simply moves the starting point of the search to the next cell-pair). If *last_id* = *true*, then locate the next production π in *pset*(C), *after* the one stored in p, such that LHS(π) = *sym*(C). If this is also the last such production, distinguish π as $\bar{\pi}$. Output *MATCH* (π (or $\bar{\pi}$), (*FIRST*,*FIRST*), ($j-1$,0), *true*).

If a primary processor does not receive a marked cell, then it ignores input I and does not issue a *MATCH* instruction; this produces the same effect as subroutine *UNMARK* in procedure *PARSE*.

Figure 5.4 illustrates the configurations of the systolic array for the n reverse sweeps of the second stage. At the end of reverse sweep n, a new parse tree would be stored in the Q-array. As in stage 1, an update step is performed for the *done*, *ldone* and *rdone* registers of the Q-array; the result is shown in Figure 5.5. After this update step, the next stage is ready to begin.

Remark 5.1. In general, the systolic algorithm generates the parse trees in an order different from procedure *PARSE*. The reason is that, because of the "folded" mapping from convolving pairs to secondary processors, the pairs are considered in a different order. Nevertheless, each stage always generates a new parse tree.

6. Complexity Analysis

Since the underlying context-free grammar is in Chomsky normal form, every parse tree has size (number of nodes) $2n-1$, where n is the length of the input string. We show that the systolic parsing algorithm runs in time $O(m \cdot n)$, where m is the number of distinct parse trees of the input string. The recognition phase is completed after $3n-1$ clock cycles (see Remark 4.1). One can also verify that the "begin-parse" signal from processor $P(n,n)$ that starts each stage occurs every $6n-3$ clock cycles. Thus, the running time of the systolic array is $3n-1 + m \cdot (6n-3) = O(m \cdot n)$.

The systolic array has $O(n^2)$ processors. Each processor requires at most $O(\log n)$ space. Thus, the total space complexity is $O(n^2 \log n)$. This is considerably more space-efficient than the systolic parsing algorithm given in [LANG87], which uses $O(n^3 \log n)$ space. In fact, one can do better for certain special cases. As mentioned in Remark 4.1, each processor uses only constant space if only the recognition phase is performed. This is in fact also true if only one parse tree is required as output. The *id* registers of the Q-processors, which are the only registers that hold $\log n$ bits, are not necessary since the information stored in these registers are only used after stage 1. Thus, to output the first parse tree, $O(n^2)$ total space is sufficient.

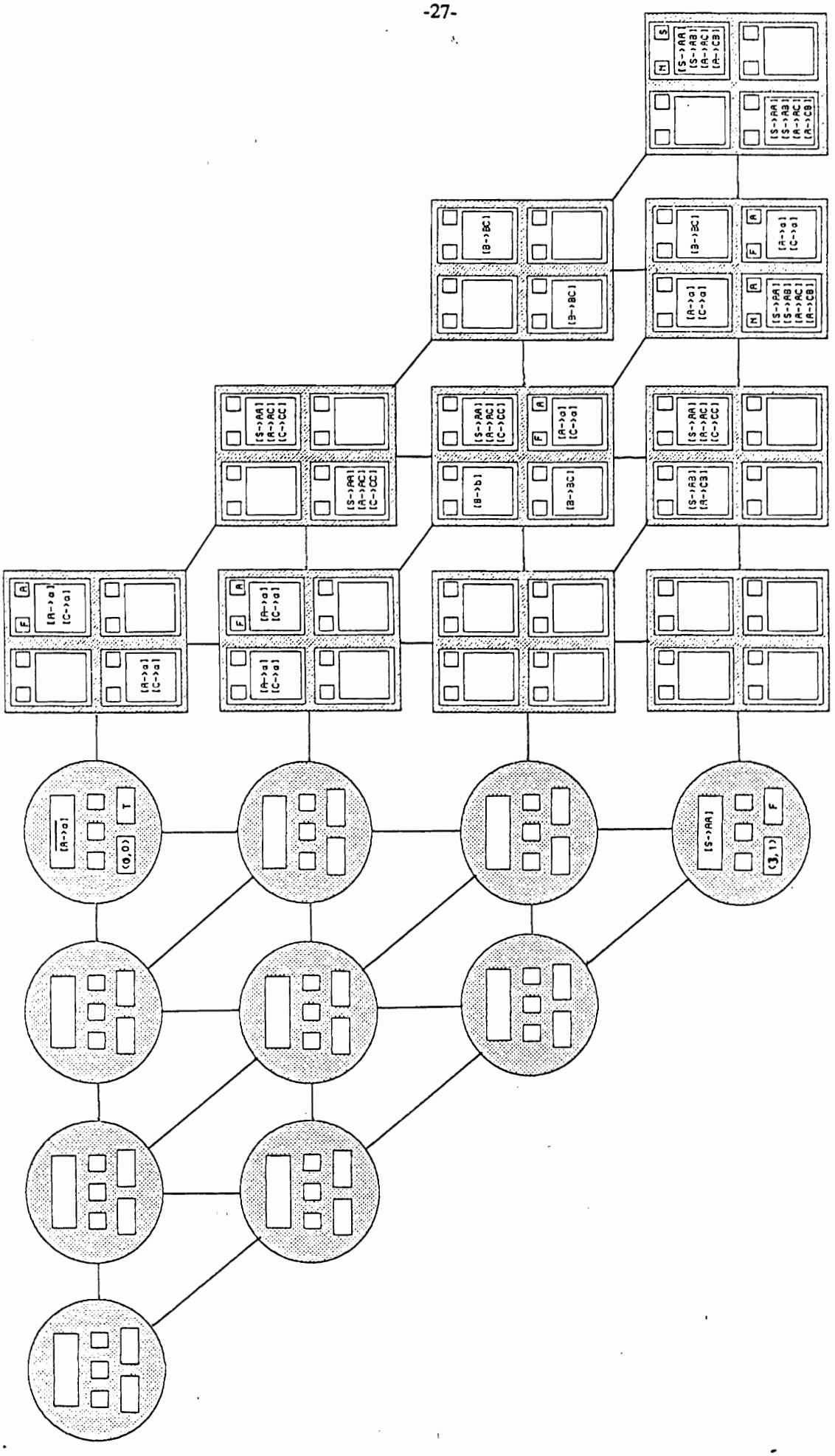


Figure 5.4. Register contents at the end of reverse sweep 1, stage 2.

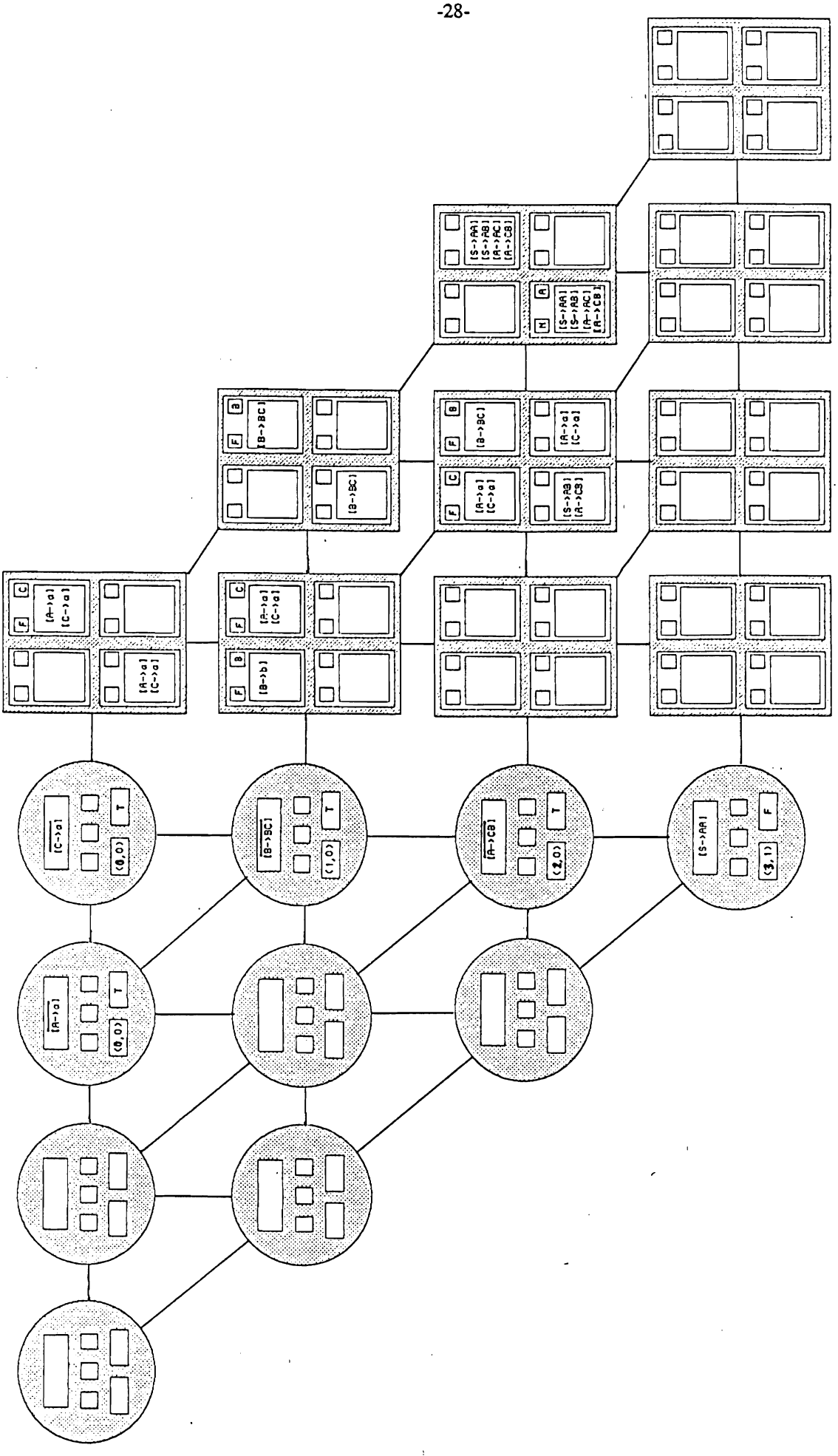


Figure 5.4. (cont.) Register contents at the end of reverse sweep 2, stage 2.

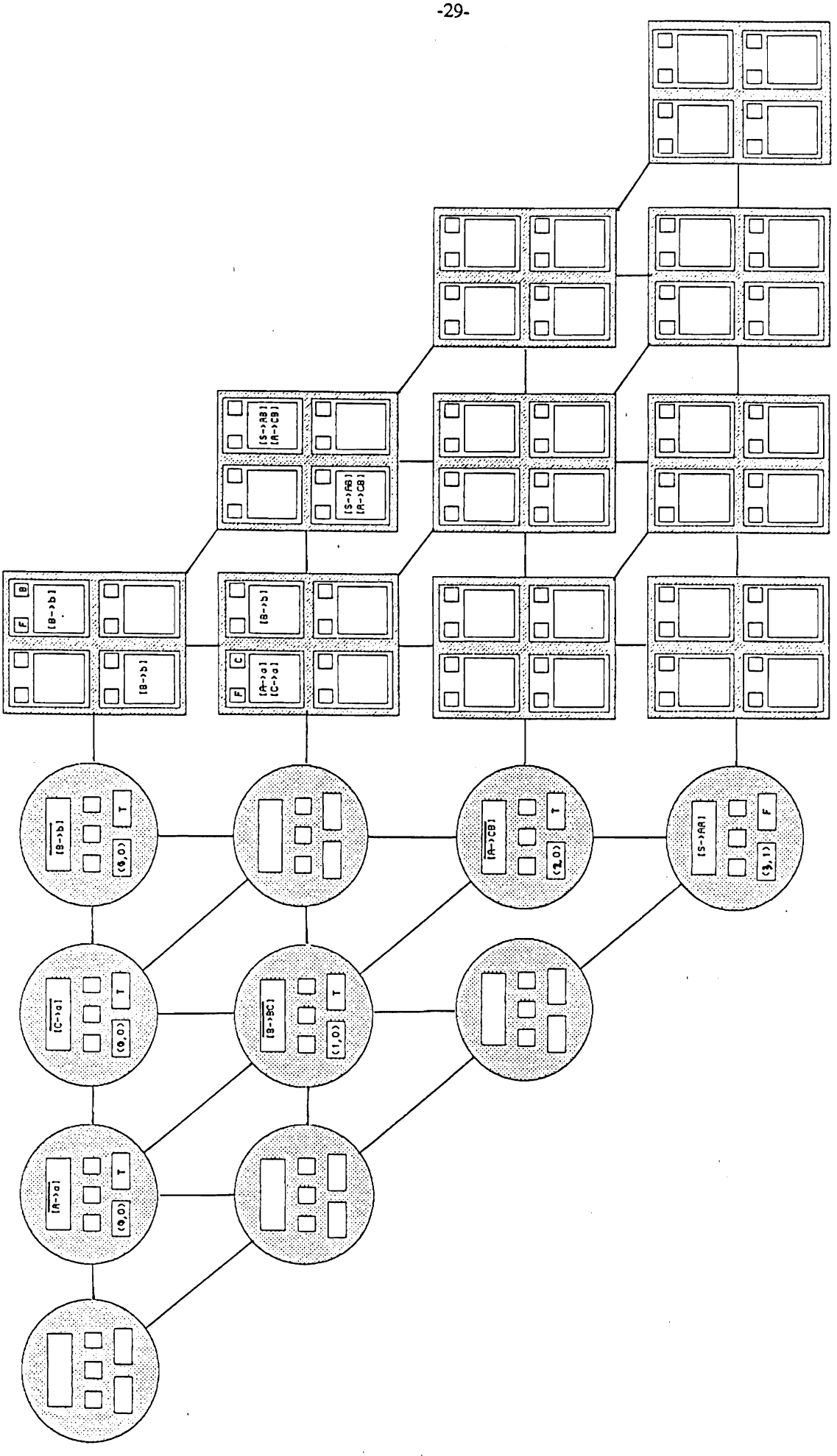


Figure 5.4. (cont.) Register contents at the end of reverse sweep 3, stage 2.

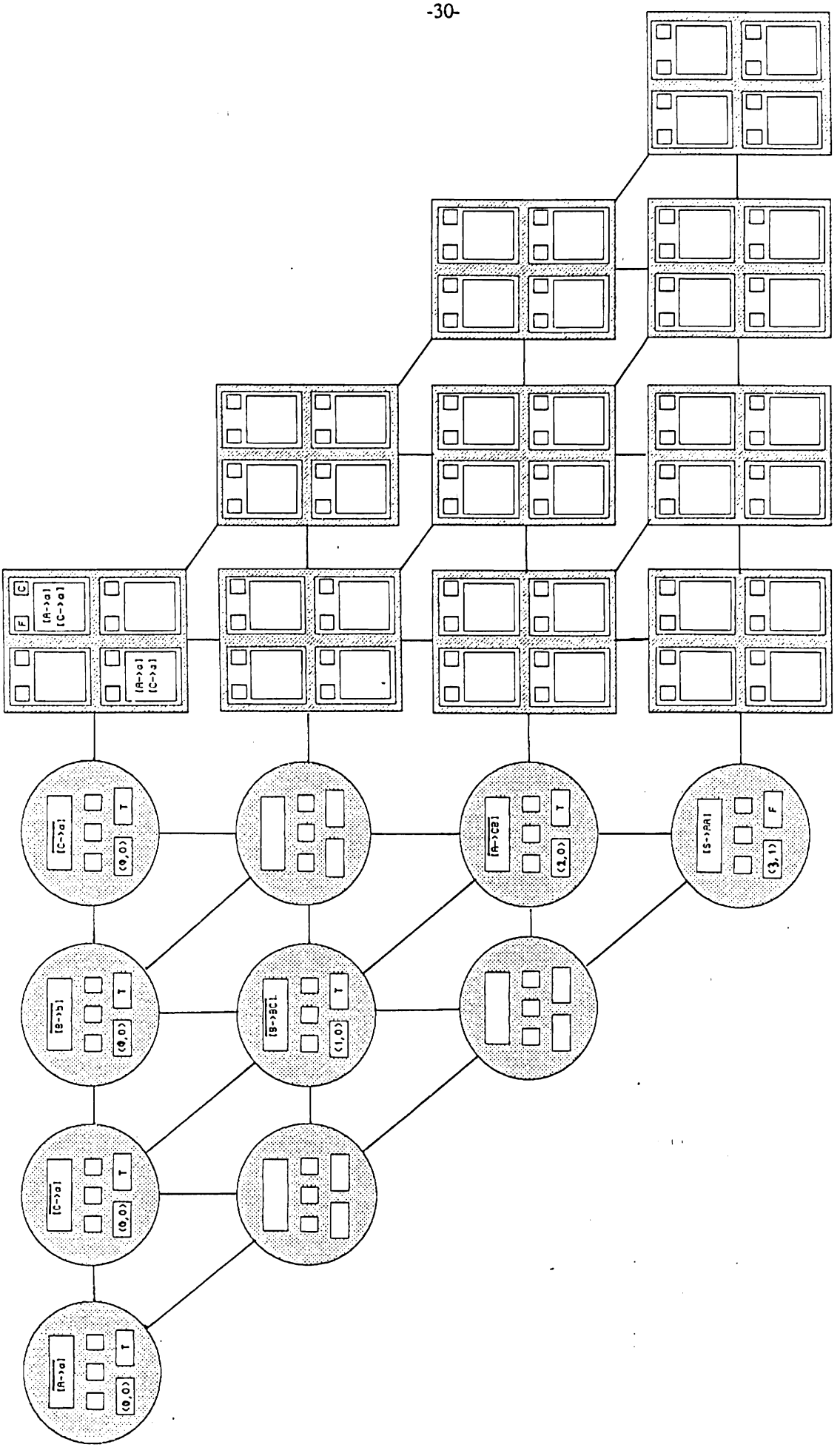


Figure 5.4. (cont.) Register contents at the end of reverse sweep 4, stage 2.

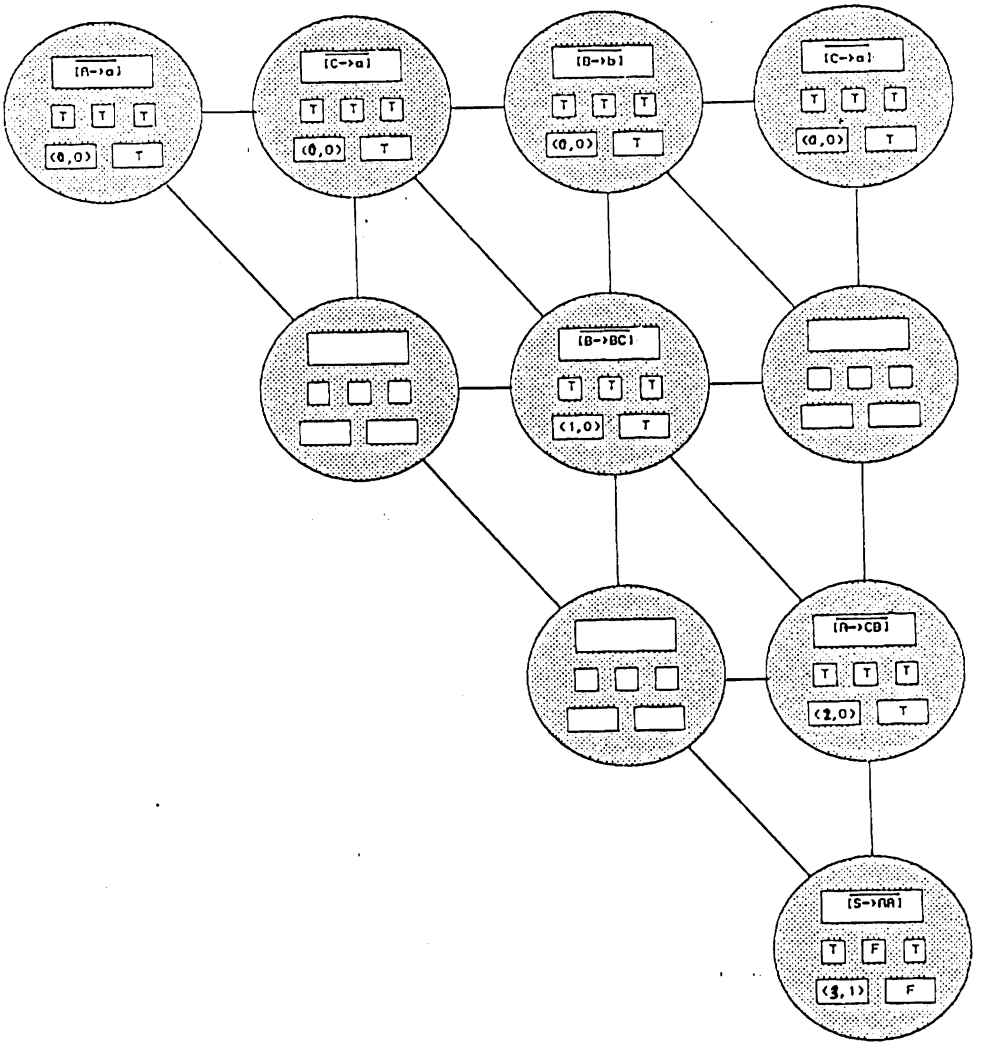


Figure 5.5. Register contents of the Q-array after the update step of stage 2.

References

[AHO72] Aho, A. V. and J. D. Ullman, *The Theory of Parsing, Translation and Compiling*, Vol. 1, *Parsing*, Prentice-Hall, Englewood Cliffs, N.J., 1972.

[CHAN87] Chang, J. H., O. H. Ibarra, and M. A. Palis, "Parallel parsing on a one-way array of finite-state machines", *IEEE Transactions on Computers*, 36:1 (1987), 64-75.

[CHIA84] Chiang, Y. T. and K. S. Fu, "Parallel parsing algorithms and VLSI implementations for syntactic pattern recognition", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6:3 (1984), 302-314.

[EARL70] Earley, J., "An efficient context-free parsing algorithm", *Communications of the ACM*, 13:2 (1970), 94-102.

[GUIB79] Guibas, L. J., H.-T. Kung, and C. D. Thompson, "Direct VLSI implementation of combinatorial algorithms", *Proceedings Caltech Conference on VLSI*, 1979, 509-525.

[KOSA75] Kosaraju, S. R., "Speed of recognition of context-free languages by array automata", *SIAM Journal on Computing*, 4:3 (1975), 331-340.

[LANG86] Langlois, L., "Parallel parsing on an array of processors", *Internal Report CSR-200-86, Department of Computer Science, University of Edinburgh*, July, 1986.

[RYTT85] Rytter, W., The complexity of two-way pushdown automata and recursive programs, in *Combinatorial Algorithms on Words*, A. Apostolico and Z. Galil (eds.), NATO ASI Series F:12, Springer-Verlag: New York/Berlin.

[VALI75] Valiant, L., "General context-free recognition in less than cubic time", *Journal of Computer and Systems Sciences*, 10:2 (1975), 308-315.

[YOUN67] Younger, D. H., "Recognition and parsing of context-free languages in time n^3", *Information and Control*, 10:2 (1967), 189-208.