A Neural-Network Architecture for Syntax Analysis

Chun-Hsien Chen and Vasant Honavar, *Member, IEEE*

***Abstract*— Artificial neural networks (ANN's), due to their inherent parallelism, offer an attractive paradigm for implementation of symbol processing systems for applications in computer science and artificial intelligence. This paper explores systematic synthesis of modular neural-network architectures for syntax analysis using a prespecified grammar—a prototypical symbol processing task which finds applications in programming language interpretation, syntax analysis of symbolic expressions, and high-performance compilers. The proposed architecture is assembled from ANN components for lexical analysis, stack, parsing and parse tree construction. Each of these modules takes advantage of parallel content-based pattern matching using a neural associative memory. The proposed neural-network architecture for syntax analysis provides a relatively efficient and high performance alternative to current computer systems for applications that involve parsing of LR grammars which constitute a widely used subset of deterministic context-free grammars. Comparison of quantitatively estimated performance of such a system [implemented using current CMOS very large scale integration (VLSI) technology] with that of conventional computers demonstrates the benefits of massively parallel neural-network architectures for symbol processing applications.**

***Index Terms*—Lexical analysis, modular neural networks, neural associative processing, neural associative processor, neural parser, neural symbolic processing, parsing, syntax analysis.**

I. Introduction

IT is often suggested that traditionally serial symbol processing systems of artificial intelligence (AI) and inherently massively parallel artificial neural networks (ANN's) offer two radically, perhaps even irreconcilably different paradigms for modeling minds and brains—both artificial as well as natural [65], [83]. AI has been successful in applications such as theorem proving, knowledge–based expert systems, mathematical reasoning, syntax analysis, and related applications which involve systematic symbol manipulation. On the other hand, ANN's have been particularly successful in applications such as pattern recognition, function approximation, and nonlinear control [27], 76] which involve primarily numeric computation. However, as shown by Church, Kleene, McCulloch, Post, Turing, and others, both AI and ANN represent particular realizations of a universal (Turing-equivalent) model of computation [99]. Thus, despite assertions by some to the contrary, any task that can be realized by one can, in principle, be accomplished by the other. However, most AI systems have been traditionally programmed in languages that were influenced by Von Neumann's design of a serial stored program computer. ANN systems on the other hand, have been inspired by (albeit overly simplified) models of biological neural networks. They represent different commitments regarding the architecture and the primitive building blocks used to implement the necessary computations. Thus they occupy different regions characterized by possibly different cost–performance tradeoffs in a much larger space of potentially interesting designs for intelligent systems.

Given the reliance of both traditional AI and ANN on essentially equivalent formal models of computation, a central issue in design and analysis of intelligent systems has to do with the identification and implementation, under a variety of design, cost, and performance constraints, of a suitable subset of Turing-computable functions that adequately model the desired behaviors. Today's AI and ANN systems each demonstrate at least one way of performing a certain task (e.g., logical inference, pattern recognition, syntax analysis) naturally and thus pose the interesting problem for the other of doing the same task, perhaps more elegantly, efficiently, robustly, or cost-effectively than the other. In this context, it is beneficial to critically examine the often implicit and unstated assumptions on which current AI and ANN systems are based and to identify alternative (and potentially better) approaches to designing such systems. Massively parallel symbol processing architectures for AI systems or highly structured (as opposed to homogeneous fully connected) ANN's are just two examples of a wide range of approaches to designing intelligent systems [99], [34], [35]. Of particular interest are alternative designs (including synergistic hybrids of ANN and AI designs) for intelligent systems [20], [28], [32], [34], [35], [46], [92], [95], [99]. Examples of such systems include: neural architectures for database query processing [10], generation of context-free languages [100], rule-based inference [12], [72], [88], [94], computer vision [4], [58], natural language processing [6], [13], learning [19], [31] [89], and knowledge-based systems [43], [75]. We strongly believe that a judicious and systematic exploration of the design space of such systems is essential for understanding the nature of key cost–performance tradeoffs in the synthesis of intelligent systems.

Against this background, this paper explores the synthesis of a neural architecture for syntax analysis using prespecified grammars—a prototypical symbol processing task with applications in interactive programming environments (using interpreted languages such as LISP and JAVA), analysis of symbolic expressions (e.g., in real-time knowledge-based systems and database query processing), and high-performance compilers. This paper does not address machine learning of

Manuscript received August 18, 1995; revised July 12, 1996, Februry 20, 1997, and October 5, 1998. The work of V. Honavar was supported in part by the National Science Foundation under Grant IRI-9409580 and the John Deere Foundation. This work was performed when C.-H. Chen was a doctoral student at Iowa State University.

C.-H. Chen is with the Advanced Technology Center, Computer and Communication Laboratories, Industrial Technology Research Institute, Chutung, Hsinchu, Taiwan, R.O.C.

V. Honavar is with the Artificial Intelligence Research Group, Department of Computer Science, Iowa State University, Ames, IA 50011 USA.

Publisher Item Identifier S 1045-9227(99)00621-9.

unknown grammars (which finds applications in tasks such as natural language acquisition).

A more general goal of this paper is to explore the design of massively parallel architectures for symbol processing using neural associative memories (processors) [9] as key components. Information processing often entails a process of pattern matching and retrieval (pattern-directed associative inference) which is an essential part of most AI systems [24], [45], [97] and dominates the computational requirements of many AI applications [25] [45], [63].

The proposed high-performance neural architecture for syntax analysis is systematically (and provably correctly) synthesized through composition of the necessary symbolic functions using a set of component symbolic functions each of which is realized using a neural associative processor. It takes advantage of massively parallel pattern matching and retrieval capabilities of neural associative processors to speed up syntax analysis for real-time applications.

The rest of this paper is organized as follows: The remainder of Section I reviews related research on neural architectures for syntax analysis. Section II briefly reviews multilayer Perceptrons, binary mapping Perceptron module which is capable of arbitrary binary mapping and is used to realize the components of the proposed neural network architectures, symbolic functions realized by binary mappings, and composition of symbolic functions. Section III briefly reviews deterministic finite automata (DFA), and neural-network architecture for DFA (NN DFA). Sections IV, V, and VI, respectively, develop modular neural-network architectures for stack, lexical analysis, and parsing. Section VII compares the estimated performance of the proposed neural architecture for syntax analysis [based on current CMOS very large scale integration (VLSI) technology] with that of commonly used approaches to syntax analysis in conventional computer systems that rely on inherently sequential index or matrix structure for pattern matching. Section VIII concludes with a summary and discussion.

A. Review of Related Research on Neural Architectures for Syntax Analysis

The capabilities of neural-network models (in particular, recurrent networks of threshold logic units or McCulloch–Pitts neurons) in processing and generating sequences (strings defined over some finite alphabet) and hence their formal equivalence with finite state automata or regular language generators/recognizers have been known for several decades [40], [52], [56]. More recently, recurrent neural-network realizations of finite state automata for recognition and learning of finite state (regular) languages have been explored by numerous authors [3], [8], [14], [16]–[18], [37], [60], [64], [66], [67], [82], [87], [102]. There has been considerable work on extending the computational capabilities of recurrent neural-network models by providing some form of external memory in the form of a tape [103] or a stack [5], [11], [29], [55], [74], [84], [90], [93], [105].

To the best of our knowledge, to date, most of the research on neural architectures for syntax analysis has focused on the investigation of neural networks that are designed to *learn* to parse particular classes of syntactic structures (e.g., strings from deterministic context-free languages (DCFL) or natural language sentences constructed using limited vocabulary). Notable exceptions are: connectionist realizations of Turing Machines (wherein a stack is simulated using binary representation of a fractional number) [90], [73]; a few neural architectures designed for parsing based on a known grammar [15], [85]; and neural-network realizations of finite state automata [8], [67]. Nevertheless, it is informative to examine the various proposals for neural architectures for syntax analysis (regardless of whether the grammar is preprogrammed or learned). The remainder of this section explores the proposed architectures for syntax analysis in terms of how each of them addresses the key subtasks of syntax analysis.

Reference [15] proposes a neural network to parse input strings of fixed maximum length for known context-free grammars (CFG's). The whole input string is presented at one time to the neural parser which is a layered network of logical AND and OR nodes with connections set by an algorithm based on CYK algorithm [36].

PARSEC [39] is a modular neural parser consisting of six neural-network modules. It transforms a semantically rich and therefore fairly complex English sentence into three output representations produced by its respective output modules. The three output modules are *role labeler* which associates case-role labels with each phrase block in each clause, *inter-clause labeler* which indicates subordinate and relative clause relationships, and *mood labeler* which indicates the overall sentence mood (declarative or interrogative). Each neural module is trained individually by a variation of the backpropagation algorithm. The input is a sequence of syntactically as well as semantically tagged words in the form of binary vectors and is sequentially presented to PARSEC, one word at a time. PARSEC exploits generalization as well as noise tolerance capabilities of neural networks to reportedly attain 78% correct labeling on a test set of 117 sentences when trained with a training set of 240 sentences. Both the test and training sets were based on conference registration dialogs from a vocabulary of about 400 words.

SPEC [55] is a modular neural parser which parses variable-length sentences with embedded clauses and produces case-role representations as output. SPEC consists of a *parser* which is a simple recurrent network, a *stack* which is realized using a recursive autoassociative memory (RAAM) [74], and a *segmenter* which controls the *push/pop* operations of the *stack* using a 2-layer perceptron.

RAAM has been used by several researchers to implement stacks in connectionist designs for parsers [5], [29], [55]. A RAAM is a 2-layer perceptron with recurrent links from hidden neurons to part of input neurons and from part of output neurons to hidden neurons. The performance of a RAAM stack is known to degrade substantially with increase in depth of the stack, and the number of hidden neurons needed for encoding a stack of a given depth has to be determined through a process of trial and error [55]. A RAAM stack has to be trained for each application. See [93] for a discussion of some of the drawbacks associated with the use of RAAM as a stack.

Each module of SPEC is trained individually using the backpropagation algorithm to approximate a mapping function

as follows: Let Q be a finite nonempty set of *states*, Γ a finite nonempty *input alphabet*, V_{CRV} a finite nonempty set of case-role vectors, $A = \{output,\ push,\ pop\}$ the set of stack actions, and V_{stack} the set of compressed stack representations at the hidden layer of a RAAM. Then the first and second connection layers of the *parser* approximate the transition function of a DFA (see Section III-A) f_{P1}: $\Gamma \times Q \rightarrow Q$ and a symbolic mapping function f_{P2}: $Q \rightarrow V_{CRV}$, respectively; the *segmenter* approximates a symbolic function f_S: $\Gamma \times Q \rightarrow Q \times A$; and the first and second connection layers of the RAAM approximate the *push* function f_{push}: $V_{stack} \times Q \rightarrow V_{stack}$ and the *pop* function f_{pop}: $V_{stack} \rightarrow V_{stack} \times Q$ of a RAAM stack, respectively. The input string is sequentially presented to SPEC and is a sequence of syntactically untagged English words represented as fixed-length distributive vectors of gray-scale values between zero and one. The emphasis of SPEC was on exploring the generalization as well as noise tolerance capabilities of a neural parser. SPEC uses central control to integrate its different modules and reportedly achieves 100% generalization performance on a whole test set of 98 100 English relative clause sentences with up to four clauses. Since the words (terminals) in the CFG which generates the test sentences are not pretranslated by a lexical analyzer into syntactically tagged tokens, the number of production rules and terminals tend to increase linearly with the size of the vocabulary in the CFG. Augmenting SPEC with a lexical analyzer offers a way around this problem.

References [11], [93], [105] propose higher-order recurrent neural networks equipped with an external stack to learn to recognize deterministic CFG, i.e., to learn to simulate a deterministic pushdown automata (DPDA). References [11], [93] use an analog network coupled with a *continuous* stack and use a variant of a real-time recurrent network learning algorithm to train the network. Reference [105] uses a discrete network coupled with a *discrete* stack and employs a pseudo-gradient learning method to train the network. The input to the network is a sequentially presented, unary-coded string of variable length. Let Q be a finite nonempty set of *states*, Γ a finite nonempty *input alphabet*, Λ a finite nonempty *stack alphabet*, $A = push,\ pop,\ no\text{-}operation\}$ the set of stack actions, and *Boolean* the set $\{false,\ true\}$. These recurrent neural networks approximate the transition function of a DPDA, i.e., f_{DPDA}: $Q \times \Gamma \times \Lambda \rightarrow Q \times \Lambda \times A$. The networks are trained to approximate a language recognizer function f_L: $\Gamma^* \rightarrow Boolean$. Strings generated from CFG including *balanced parenthesis grammar*, $a^n b^n$, $a^{m+n} b^m c^n$, $a^n b^n c b^m a^m$, *postfix grammar*, and/or *palindrome grammar* were used to evaluate the generalization performance of the proposed networks. Hybrid systems consisting of a recurrent neural network and a stack have also been used to learn CFL [61].

The proposed neural architecture for syntax analysis is composed of neural-network modules for stack, lexical analysis, parsing, and parse tree construction. It differs from most of the neural-network realizations of parsers in that it is systematically assembled using neural associative processors (memories) as primary building blocks. It is able to exploit massively parallel content-based pattern matching and retrieval capabilities of neural associative processors. This offers an opportunity to explore the potential benefits of ANN's massive parallelism in the design of high-performance computing systems for real time symbol processing applications.

II. Neural Associative Processors and Symbolic Functions

This section reviews the design of a neural associative processor using a binary mapping perceptron [8], [9] and the representation of symbolic functions in terms of binary mapping.

A. Perceptrons

A 1-layer perceptron has m input neurons, n output neurons and one layer of connections. The output y_i of output neuron i, $1 \le i \le n$, is given by $y_i = f_H(\Sigma_{j=1}^{m} w_{ij} x_j - \theta_i)$, where w_{ij} denotes the weight on the connection from input neuron j to output neuron i, θ_i is the threshold of output neuron i, x_j is the value at input neuron j, and f_H is *binary hardlimiter* function, where

$$f_H(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise.} \end{cases}$$

It is well known that such a 1-layer perceptron can implement only *linearly separable* functions from $\boldsymbol{R}^m$ to $\{0,1\}^n$ [57]. We can see the connection weight vector $w_i = \langle w_{i1}, \cdots, w_{im} \rangle$ and the node threshold θ_i as defining a linear hyperplane H_i which partitions the m-dimensional pattern space into two half-spaces.

A 2-layer perceptron has one layer of k hidden neurons (and hence two layers of connections with each hidden neuron being connected to each of the input as well as output neurons). In this paper, every hidden neuron and output neuron in the 2-layer perceptron use binary hardlimiter function f_H as activation function and produce binary outputs; its weights are restricted to values from $\{-1, 0, 1\}$; and it uses integer thresholds. It is known that such a 2-layer perceptron can realize arbitrary binary mappings [9].

B. Binary Mapping Perceptron Module (BMP)

Let U be a set of k distinct input binary vectors $u_1, \cdots, u_k$ of dimension m, where $u_h = \langle u_{h1}, \cdots, u_{hm} \rangle$, $u_{hi} \in \{0,1\}$, $1 \le h \le k$ & $1 \le i \le m$. Let V be a set of k desired output binary vectors $v_1, \cdots, v_k$ of dimension n, where $v_h = \langle v_{h1}, \cdots, v_{hn} \rangle$, $v_{hj} \in \{0,1\}$, $1 \le h \le k$ & $1 \le j \le n$. Consider a binary mapping function g defined as follows:

$$\begin{aligned} &g: \boldsymbol{B}^m \rightarrow (V \cup \{\langle 0^n \rangle\}) \\ &g(u_h) = v_h, \quad \text{for} \quad 1 \le h \le k \\ &g(x) = \langle 0^n \rangle, \quad \text{for} \quad x \in (\boldsymbol{B}^m - U) \end{aligned}$$

where $\boldsymbol{B}^m$ is the m-dimensional binary space. A BMP module [8] for the desired binary mapping function g can be synthesized using a 2-layer perceptron as follows: The BMP module (see Fig. 1) has m input, k hidden and n output neurons. For each binary mapping ordered pair (u_h, v_h), where $1 \le h \le k$, we create a hidden neuron h with threshold $\Sigma_{i=1}^{m} u_{hi} - 1$. The connection weight from input neuron i to this hidden

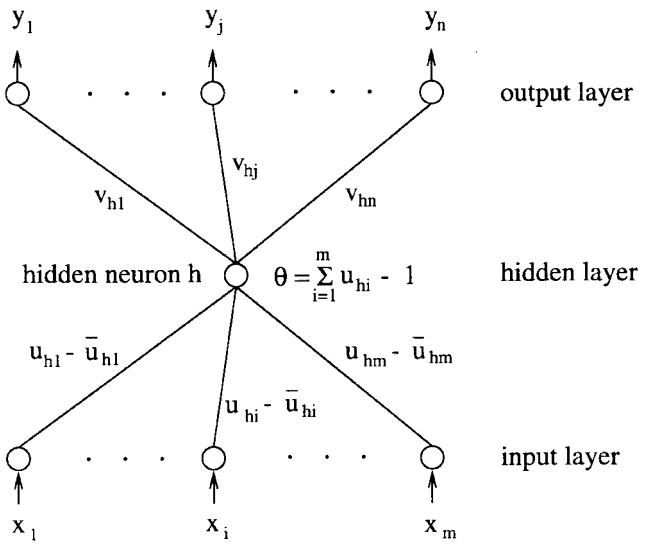

Fig. 1. The settings of connection weights and hidden node threshold in the proposed BMP module for an associated binary mapping ordered pair (u_h, v_h). The threshold for each of the output neurons is set to zero. The activation function at hidden and output neurons is hardlimiter function.

neuron is $u_{hi} - \overline{u}_{hi} = 2u_{hi} - 1$ where $1 \leq i \leq m$; and that from this hidden neuron to output neuron j is v_{hj}, where $1 \leq j \leq n$. The threshold for each of the output neurons is set to zero. Note that identity function I, i.e., $I(x) = x$, can be used as activation function at output neuron for more efficient hardware implementation.

Note that for the input binary vector u_h, only the hidden neuron h outputs a one, and the rest of the hidden neurons output zero. Thus the output of the jth output neuron is v_{hj}, and so the binary output vector is $\langle v_{h1}, \cdots, v_{hn} \rangle = v_h$. While for an input vector $x \notin U$, no hidden neuron is activated and the output is $\langle 0^n \rangle$.

The computation of a mapping in a BMP module can be viewed as a two-stage associative process: *identification* and *recall*. Since input is binary, the weights in the first-layer connections of a BMP module are either one or -1. During identification, a bit of an input pattern that is wrongly on (with respect to a stored pattern), contributes -1 to the activation of the corresponding hidden neuron and a bit of an input pattern that is rightly on (with respect to a stored pattern) contributes $+1$ to the activation of the corresponding hidden neuron. A bit of an input pattern that is (rightly or wrongly) off (with respect to a stored pattern) contributes zero to the activation of the corresponding hidden neuron. Each hidden neuron sums up the contributions to its activation from its first-layer connections, compares the result with its threshold (which equals the number of "1's" in the stored memory pattern minus one), and produces output value one if its activation exceeds its threshold. If one of the hidden neurons is turned on, one of the stored memory patterns will be *recalled* by that hidden neuron. Note that an input pattern is matched against all the stored memory patterns *in parallel*. If the time delay for computing the activation at a neuron is fixed, the time complexity for such a pattern matching process is $O(1)$. Note that this is attained at the cost of a hidden neuron (and its connections) for each stored association.

C. Binary Mapping and Symbolic Functions

In general, most of simple, nonrecursive symbolic functions and table lookup functions can be viewed in terms of a binary random mapping f_B which is defined as follows: Let U be a set of k distinct binary input vectors $u_1, \cdots, u_k$ of dimension m; and V be a set of k binary output vectors $v_1, \cdots, v_k$ of dimension n. Then

$$f_B \colon U \to V$$
$$f_B(u_i) = v_i, \quad \text{for} \quad 1 \leq i \leq k.$$

Let $|Z|$ denote the cardinality of set Z. The binary vector u_i, where $1 \leq i \leq k$, represents an ordered set of r binary-coded symbols from symbol sets $\Gamma_1, \Gamma_2, \cdots, \Gamma_r$, respectively, (i.e., $\exists \alpha_1 \in \Gamma_1, \cdots, \alpha_r \in \Gamma_r \ s.t. \ u_i = \alpha_1 \cdot \alpha_2 \cdot \ \cdots \ \cdot \alpha_r$, where $\cdot$ denotes the concatenation of two binary codes). The binary vector v_i, where $1 \leq i \leq k$ represents an ordered set of s symbols from symbol sets $\Delta_1, \Delta_2, \cdots, \Delta_s$, respectively, and $|U| = |\Gamma_1||\Gamma_2| \cdots |\Gamma_r|$. f_B defines a symbolic function f such that

$$f \colon \Gamma_1 \times \cdots \times \Gamma_r \to \Delta_1 \times \cdots \times \Delta_s.$$

In this case, the mapping operations of f_B can be viewed in terms of the operations of f on its associated symbols.

Neural-network modules for symbol processing can be synthesized through a *composition* of appropriate primitive symbolic functions which are directly realized by suitable BMP modules. There are two of basic ways of recursively composing composite symbolic functions from component symbolic functions (which may themselves be composite functions or primitive functions). Suppose g is a symbolic function defined as follows:

$$g \colon \Delta_1 \times \cdots \times \Delta_s \to \Lambda_1 \times \cdots \times \Lambda_t.$$

The composition of f and g is denoted by $g \circ f$ such that

$$g \circ f \colon \Gamma_1 \times \cdots \times \Gamma_r \to \Lambda_1 \times \cdots \times \Lambda_t$$

and for every $(\alpha_1, \cdots, \alpha_r)$ in $\Gamma_1 \times \cdots \times \Gamma_r$

$$g \circ f(\alpha_1, \cdots, \alpha_r) = g(f(\alpha_1, \cdots, \alpha_r)).$$

Suppose f_i is a symbolic function such that

$$f_i \colon \Gamma_1 \times \cdots \times \Gamma_r \to \Delta_i \quad \text{for} \quad 1 \leq i \leq s.$$

The composition c of symbolic functions $g, f_1, \cdots, f_s$ is defined as

$$c \colon \Gamma_1 \times \cdots \times \Gamma_r \to \Lambda_1 \times \cdots \times \Lambda_t$$

and for every $(\alpha_1, \cdots, \alpha_r)$ in $\Gamma_1 \times \cdots \times \Gamma_r$

$$c(\alpha_1, \cdots, \alpha_r) = g(f_1(\alpha_1, \cdots, \alpha_r), \cdots, f_s(\alpha_1, \cdots, \alpha_r)).$$

The processing of input strings of variable length (of the sort needed in lexical analysis and parsing) can be handled by composite functions $\hat{f} \colon \Gamma^* \to \Delta^*$, $\hat{g} \colon \Lambda \times \Gamma^* \to \Lambda$, and $\hat{c} \colon \Lambda \times \Gamma^* \to \Lambda \times \Delta^*$ in the proposed modular neural architecture, where Γ^* (Δ^*) denotes the set of all strings over the alphabet Γ (Δ). Here, function $\hat{f}$ denotes the processing of input strings of variable length by a parser or a lexical

analyzer; function $\hat{g}$ denotes the recursive evaluation of input strings of variable length by the extended transition function of a DFA; and function $\hat{c}$ denotes the recursive parsing of syntactically tagged input tokens by the extended transition function of an LR(1) parser. The functions $\hat{f}, \hat{g}$, and $\hat{c}$ that process input strings of variable length can be composed using symbolic functions f, g, c, output selector function, and string concatenation function by recursion on the length of the input string (see Section III-A for an example). Other recursive symbolic functions can also be composed using composition and recursion. We do not delve into recursion any further. The interested reader is referred to [71], [80], and [104] for details.

The operation of a desired composite function on its symbolic input (string) is fully characterized analytically in terms of its component symbolic functions on their respective symbolic inputs and outputs. The component symbolic functions are either composite functions of other symbolic functions or primitive symbolic functions which are realized directly by BMP modules. This makes it possible to systematically (and probably correctly) synthesize any desired symbolic function using BMP modules. (Such designs often require recurrent links for realizing recursive functions such as the extended transition function $\hat{\delta}$ of a DFA or a more complex recursive function for the LR parser as we shall see later).

III. Neural-Network Automata

This section reviews the synthesis of a neural-network architecture for a finite state machine [8].

A. Deterministic Finite Automata (DFA)

A *deterministic finite automaton* (finite-state machine) is a 5-tuple $M_{DFA} = (Q, \Gamma, \delta, q_0, F)$ [36], where Q is a finite nonempty set of *states*, Γ is a finite nonempty *input alphabet*, $q_0 \in Q$ is the *initial state*, $F \subseteq Q$ is the set of *final* or *accepting states*, and δ: $Q \times \Gamma \rightarrow Q$ is the *transition function*. A finite automaton is deterministic if there is at most one transition that is applicable for each pair of state and input symbol.

The extended transition function $\hat{\delta}$ of a DFA with transition function δ is a mapping from $Q \times \Gamma^*$ to Q defined by recursion on the length of the input string as follows.

- **Basis**: $\hat{\delta}(q_i, \epsilon) = q_i$, where ϵ is empty string.
- **Recursive step**: $\hat{\delta}(q_i, ua) = \delta(\hat{\delta}(q_i, u), a)$ for all input symbols $a \in \Gamma$ and strings $u \in \Gamma^*$.

The computation of the machine M_{DFA} in state q_i with string w halts in state $\hat{\delta}(q_i, w)$. The evaluation of the function $\hat{\delta}(q_0, w)$ simulates the repeated application of the transition function δ required to process the string w from initial state q_0. A string w is accepted by M_{DFA} if $\hat{\delta}(q_0, w) \in F$; otherwise it is rejected. The set of strings accepted by M_{DFA} is denoted as $L(M_{DFA}) = \{w | \hat{\delta}(q_0, w) \in F\}$, called *the language* of M_{DFA}.

A *Mealy machine* is a DFA augmented with an output function. It is defined by a 6-tuple $M_{Mealy} = (Q, \Gamma, \Delta, \delta, \lambda, q_0)$ [36], where Q, Γ, δ, and q_0 are as in the DFA M_{DFA}, Δ is a finite nonempty *output alphabet*, and λ is output function mapping from $Q \times \Gamma$ to Δ. $\lambda(q, a)$ is the output associated with the transition from state q on input symbol a. The output of M_{Mealy} responding to input string $a_1 a_2 \cdots a_n$ is output string $\lambda(q_0, a_1)\lambda(q_1, a_2) \cdots \lambda(q_{n-1}, a_n)$, where $q_0, q_1, \cdots, q_n$ is the sequence of states such that $\delta(q_{i-1}, a_i) = q_i$ for $1 \leq i \leq n$.

B. NN Deterministic Finite Automata (NN DFA)

A partially recurrent neural-network architecture can be used to realize a DFA as shown in [8]. It uses a BMP module to implement the transition function of a DFA.

- In the BMP module, the input neurons are divided into two sets. One set of input neurons has no recurrent connections and receives the binary coded current input symbol. There are $n = \lceil \log_2 |\Gamma| \rceil$ such input neurons. (Here $\lceil \cdot \rceil$ denotes the integer ceiling of a real value). The second set has $m = \lceil \log_2(|Q| + 1) \rceil$ input neurons and holds the current state (coded in binary). Each input neuron in this set has a recurrent connection from the corresponding output neuron.
- The output neurons together hold the next state (coded in binary). There are $m = \lceil \log_2(|Q| + 1) \rceil$ output neurons.
- Every transition is represented as an ordered pair of binary codes. For each such ordered pair, a hidden neuron is used to realize the ordered pair in terms of binary mapping. Thus the number of required hidden neurons equals the number of valid transitions in the transition function. For example, suppose $p, q \in Q, a \in \Gamma, \delta(p, a) = q$ is a valid transition, and p, q as well as a are encoded as binary codes such that $p = \langle p_1, \cdots, p_m \rangle, q = \langle q_1, \cdots, q_m \rangle$ and $a = \langle a_1, \cdots, a_n \rangle$ where $p_i, q_i, a_j \in \{0, 1\}$ for $1 \leq i \leq m$ and $1 \leq j \leq n$. Then the transition $\delta(p, a) = q$ is represented as a binary mapping ordered pair $(\langle p_1, \cdots, p_m, a_1, \cdots, a_n \rangle, \langle q_1, \cdots, q_m \rangle)$ implemented by a BMP module (See Section II-B for details).
- An explicit synchronization mechanism is used to support the recursive evaluation of the extended transition function $\hat{\delta}$ on input string of variable length. Note that $\hat{\delta}$ maps from $Q \times \Gamma^*$ to Q.

The transitions of a DFA can be represented as a two-dimensional table with current state and current input symbol as indexes. The operation of such a DFA involves repetitive lookup of the next state from the table using current state and current input symbol at each move until an error state or an accepting state is reached. Such a repetitive table lookup process involves content-based pattern matching and retrieval wherein the indexes of the table are used as input patterns to retrieve the next state. This process can exploit the massively parallel associative processing capabilities of a neural associative memory.

IV. Neural-Network Design for a Stack (NNStack)

The capability of DFA is limited to recognition and production of the set of regular languages, the simplest class of languages in Chomsky hierarchy [36]. The capability of DFA can be extended by adding a *stack*. The resulting automata can recognize the set of DCFL, a more complex and widely used class of languages in Chomsky hierarchy [36]. A stack can be

coded as a string over a stack alphabet, with its top element at one end of the string and its bottom element at the other end. Pop and push are the main actions of a stack. These actions can be performed by a DFA which is augmented with memory to store stack symbols which are accessed sequentially using a *stack top pointer* (SP) which points to the top symbol of a stack. The stack top pointer is maintained by the current state of the DFA, and the current action of the stack by the input to the DFA.

A. Symbolic Representation of Stack

Let $A = \{\mathtt{pop}, \mathtt{push}, \mathtt{no-action}\}$ be the set of possible stack actions, C the set of possible stack configurations (contents), S the set of stack symbols, $P = \{0, 1, 2, \cdots, n\}$ the set of possible positions of stack top pointer, and n the maximal depth (capacity) of a given stack. Let $\perp$ be stack bottom symbol and $c \cdot s$ denote the stack configuration after a stack symbol s is pushed onto a stack which has the configuration c. An empty stack only contains the stack bottom symbol $\perp$. Note that $C = \{\alpha | \alpha \in \perp \cdot S^* \text{ and } |\alpha| \leq n\}$, where $|\alpha|$ denotes the number of stack symbols in the stack configuration α and $\cdot$ denotes the concatenation of two symbol strings. Assume that the value of stack top pointer does not change on a $\mathtt{no-action}$ action, and it is incremented on a push action and decremented on a pop action. The operation of a stack can be characterized by the symbolic function f_{Stack}, where

$$f_{Stack}\text{: } A \times S \times C \times P \rightarrow C \times P, \text{ defined by}$$

$$f_{Stack}(\mathtt{push}, s, c, p) = \begin{cases} (c \cdot s, p+1), & \text{if } s \in S; c \in C; \text{ and} \\ & \quad p \in P \text{ and } p \leq n-1 \\ \mathtt{error}, & \text{otherwise} \end{cases}$$

$$f_{Stack}(\mathtt{pop}, *, c, p) = \begin{cases} (c', p-1), & \text{if } c \in C \text{ and } c = c' \cdot s \text{ for} \\ & \quad \text{some } s \in S \text{ and some } c' \in C \\ & \quad \text{and } p \in P \text{ and } p \geq 1 \\ \mathtt{error}, & \text{otherwise} \end{cases}$$

$$f_{Stack}(\mathtt{no-action}, *, c, p) = \begin{cases} (c, p), & \text{if } c \in C \text{ and } p \in P \\ \mathtt{error}, & \text{otherwise} \end{cases}$$

where $*$ stands for a don't care. Suppose $S = \{\mathtt{a}, \mathtt{b}\}$. Then, $f_{Stack}\ (\mathtt{push}, \mathtt{a}, \perp\ \mathtt{ab}, 2) = (\perp\ \mathtt{aba}, 3)$, and $f_{Stack}\ (\mathtt{pop}, *, \perp\ \mathtt{aba}, 3) = (\perp\ \mathtt{ab}, 2)$. The retrieval of stack top symbol can be characterized by the symbolic function f_{Top}, where

$$f_{Top}\text{: } C \times P \rightarrow S \cup \{\perp\}, \text{ defined by}$$

$$f_{Top}(c, p) = \begin{cases} \perp & \text{if } c = \perp \text{ and } p = 0 \\ s & \text{if } c \in C \text{ and } c = c' \\ & \quad \cdot s \text{ for some } s \in S \\ & \quad \text{and some } c' \in C; \text{ and} \\ & \quad p \in P \text{ and } |c| = p \\ \mathtt{error} & \text{otherwise.} \end{cases}$$

For example, $f_{Top}\ (\perp \mathtt{ab}, 2) = \mathtt{b}$, and $f_{Top}\ (\perp, 0) = \perp$.

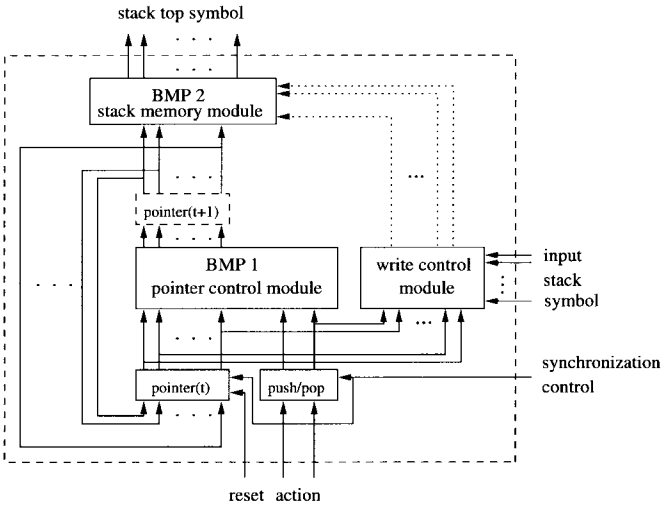

Fig. 2. A neural-network architecture for stack mechanism. The dotted box labeled with *pointer*$(t+1)$ exists only logically but not physically. A push stack action enables the *write control module* to write stack symbol into the stack memory module.

B. Architecture of NNStack

This section discusses the neural-network realization of a stack in terms of symbolic functions f_{Stack} and f_{Top}. A design for NNStack obtained by adding a *write control module* to an NN DFA is shown in Fig. 2. (The use of such a circuit might be considered by some to be somewhat unconventional given the implicit assumption of lack of explicit control in many neural-network models. However, the operation of most existing neural networks implicitly assumes at least some form of control. Given the rich panoply of controls found in biological neural networks, there is no reason not to build in a variety of control and coordination structures into neural networks whenever it is beneficial to do so [33]). NNStack has an n-bit binary output corresponding to the element popped from the stack, and four sets of binary inputs:

- *Reset* which is a 1-bit signal which resets *pointer*(t) (current SP) to point to the bottom of the stack at the beginning.
- *Synchronization control* which is a 1-bit signal that synchronizes NNStack with the discrete-time line, denoted by $0, 1, \cdots, t, t+1, \cdots$.
- *Action* code which is a 2-bit binary code so that
  a) 01 denotes push.
  b) 10 denotes pop.
  c) 00 denotes no action.
- *Stack symbol* which is an n-bit binary code for the symbol to be pushed onto or popped off a stack during a stack operation.

NNStack consists of a *pointer control module*, a *stack memory module*, a *write control module* and two *buffers*. The first buffer stores current SP value (*pointer*(t)) and the second stores the current stack action (push/pop). In Fig. 2, the dotted box labeled with *pointer* (t+ 1) exists only logically but not physically, and *pointer*(t) and *pointer*(t + 1), respectively, denote SP before and after a stack action. SP is coded into an m-bit binary number.

Pointer Control Module: The pointer control module (BMP 1) realizes a symbolic function $f_{PControl}$:$A \times P \rightarrow P$ and controls the movement of SP which is incremented on a push and decremented on a pop. For example, $f_{PControl}(\mathtt{pop}, 2) = 1$, and $f_{PControl}(\mathtt{push}, 2) = 3$. The pointer control module uses $m + 2$ input, 3×2^m hidden, and m output neurons. m of the input neurons represent $pointer(t)$ (current SP value), and the remaining two input neurons encodes the stack action. There are 2^m possible SP values. The m output neurons represent $pointer$(t + 1) (the SP value after a stack action). Each change in SP value can be realized by a binary mapping (with one hidden neuron per change). Since "no action" is one of legal stack actions, 3×2^m hidden neurons are used in the pointer control module.

Stack Memory Module: The stack memory module (BMP 2) realizes the symbolic function f_{Top}. It uses m input neurons, n output neurons, and 2^m hidden neurons which together allow storage of 2^m stack symbols at 2^m SP positions. The stack symbols stored in stack memory module are accessed through $pointer$(t + 1) (the output of the pointer control module). Note that the BMP 2 module uses its second-layer connections associated with a hidden neuron to store a symbol [8], [9].

Write Control Module: The write control module (plus BMP 2) realizes a symbolic function f_{SWrite}: $A \times S \times C \times P \rightarrow C$. An example for the computation of f_{SWrite} is f_{SWrite} $(\mathtt{push}, \mathtt{a}, \perp\mathtt{ab}, 2) = \perp\mathtt{aba}$. Physically, it receives m binary inputs from the buffer labeled with $pointer$(t) (denoting current SP), 1 binary input from the second output line of the buffer labeled with push/pop (denoting current stack action), and n binary inputs (denoting the stack symbol to be pushed onto the stack) from environment. BMP 2 (stack memory module) is used to store current stack configuration. The module does nothing when a pop is performed. The n dotted output lines from the write control module write the n-bit binary-coded stack symbol into n of the second-layer connections associated with a corresponding hidden neuron in the stack memory module when a push is performed. The hidden neuron and its n associated second-layer connections are located by using current SP value ($pointer$(t)). (The processing of stack *overflow* and *underflow* is not discussed here. It has to be taken care of by appropriate error handling mechanisms).

Timing Considerations: The proposed design for NNStack shown in Fig. 2 is based on the assumption that the write control module finishes updating the second-layer connection weights associated with a hidden neuron of BMP 2 before the signals from BMP 1 are passed to BMP 2 during a push stack action. If this assumption fails to hold, the original design needs to be modified by adding: n links from input stack symbol (buffer) to output stack symbol (buffer); an inhibition latch, which is activated by the leftmost output line of the push/pop buffer, on the links to inhibit signal passing from input stack symbol (buffer) to output stack symbol (buffer) at a pop operation; a second inhibition latch, which is activated by the rightmost output line of the push/pop buffer, between BMP 1 and BMP 2 to inhibit signal transmission between these two modules at a push operation.

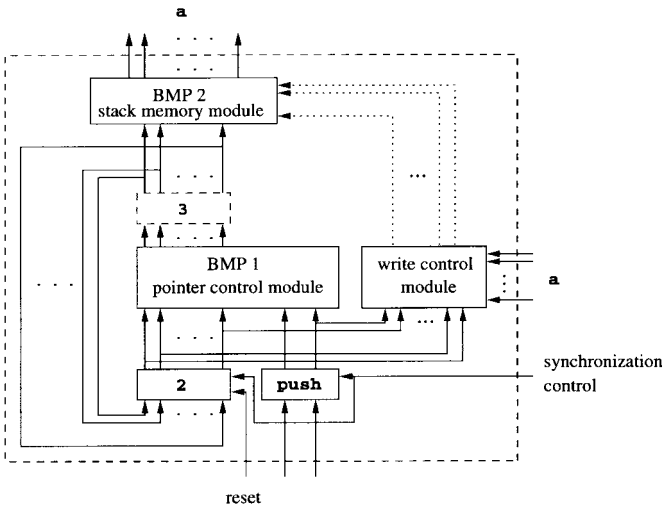

Fig. 3. The inputs and outputs of the neural modules in the NN Stack which computes $f_{Stack}(\mathtt{push}, \mathtt{a}, \perp\mathtt{ab}, 2) = (\perp\mathtt{aba}, 3)$ and $f_{Top}(\perp\mathtt{aba}, 3) = \mathtt{a}$.

C. NNStack in Action

This section symbolically illustrates how the modules of NNStack together realize a stack by considering several successive stack actions. The Appendix shows how the modules realize a stack in terms of binary codings. Symbolic function f_{Stack} is a composition of symbolic functions $f_{PControl}$ and f_{SWrite} s.t. $\forall(a, s, c, p) \in A \times S \times C \times P$, $f_{Stack}(a, s, c, p) = (f_{SWrite}(a, s, c, p), f_{PControl}(a, p))$. Fig. 3 shows the inputs and outputs of the neural modules in an NN Stack which computes f_{Stack} $(\mathtt{push}, \mathtt{a}, \perp\mathtt{ab}, 2) = (\perp\mathtt{aba}, 3)$ and f_{Top} $(\perp\mathtt{aba}, 3) = \mathtt{a}$. Consider the following sequence of stack operations:

1) At time $= t_1$, suppose the value of stack top pointer (current SP value) is four and the stack action to be performed is a push on a stack symbol a. Let c_{t_1} be current stack configuration. Then, the new stack configuration after this push action is $c_{t_1} \cdot \mathtt{a}$, and the new stack top symbol is a. At this time step, NNStack computes $f_{Stack}(\mathtt{push}, \mathtt{a}, c_{t_1}, 4) = (c_{t_1} \cdot \mathtt{a}, 5)$ and $f_{Top}(c_{t_1} \cdot \mathtt{a}, 5) = \mathtt{a}$, i.e., we have the following.
2) The pointer control module computes $f_{PControl}(\mathtt{push}, 4) = 5$.
3) The write control module (plus stack memory module) computes $f_{SWrite}(\mathtt{push}, \mathtt{a}, c_{t_1}, 4) = c_{t_1} \cdot \mathtt{a}$.
4) The stack memory module computes $f_{Top}(c_{t_1} \cdot \mathtt{a}, 5) = \mathtt{a}$.
5) At time $= t_1 + 1$, suppose the stack action to be performed is a push on a stack symbol b. Then, the new stack configuration after this push action is $c_{t_1} \cdot \mathtt{a} \cdot \mathtt{b}$, and the new stack top symbol is b. At this time step, NNStack computes $f_{Stack}(\mathtt{push}, \mathtt{b}, c_{t_1} \cdot \mathtt{a}, 5) = (c_{t_1} \cdot \mathtt{a} \cdot \mathtt{b}, 6)$ and $f_{Top}(c_{t_1} \cdot \mathtt{a} \cdot \mathtt{b}, 6) = \mathtt{b}$, i.e.,
   A) the pointer control module computes $f_{PControl}(\mathtt{push}, 5) = 6$,
   B) the write control module (plus stack memory module) computes $f_{SWrite}(\mathtt{push}, \mathtt{b}, c_{t_1} \cdot \mathtt{a}, 5) = c_{t_1} \cdot \mathtt{a} \cdot \mathtt{b}$, and
   C) the stack memory module computes $f_{Top}(c_{t_1} \cdot \mathtt{a} \cdot \mathtt{b}, 6) = \mathtt{b}$.

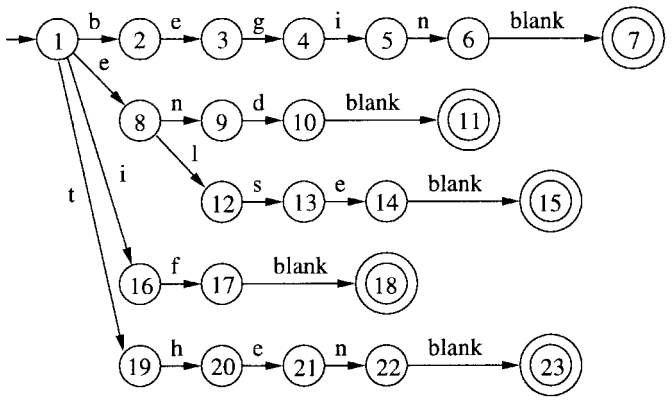

Fig. 4. The simplified state diagram of a DFA which recognizes keywords: `begin`, `end`, `if`, `then`, and `else`.

6) At time $= t_1 + 2$, suppose the stack action to be performed is a `pop`. Then, the new stack configuration after this `pop` action is $c_{t_1} \cdot \mathtt{a}$, and the new stack top symbol is `a`. At this time step, NNStack computes $f_{Stack}(\mathtt{pop}, *, c_{t_1} \cdot \mathtt{a} \cdot \mathtt{b}, 6) = (c_{t_1} \cdot \mathtt{a}, 5)$ and $f_{Top}(c_{t_1} \cdot \mathtt{a}, 5) = \mathtt{a}$, i.e., we have the following.
   A) the pointer control module computes $f_{PControl}(\mathtt{pop}, 6) = 5$.
   B) the write control module does nothing.
   C) the stack memory module computes $f_{Top}(c_{t_1} \cdot \mathtt{a}, 5) = \mathtt{a}$.

V. Neural-Network Design for a Lexical Analyzer (NNLexAn)

A lexical analyzer carves a string of characters into a string of words and translates the words into syntactically tagged lexical tokens. The computation of a lexical analyzer can be defined by a recursive symbolic function $\hat{f}_{LexAn}$: $\Gamma^*\$ \rightarrow \Delta^*\$$. Γ is the input alphabet, \$ is a special symbol denoting "*end of input*," and Δ is the set of lexical tokens. $\Gamma^*\$$ (or $\Delta^*\$$) denote the set of strings obtained by adding the suffix \$ to each of the strings over the alphabets Γ (or Δ). The syntactically tagged tokens are to be used as single logical units in parsing. Typically, the tokens are of fixed length to simplify the implementation of parsing algorithms and to enhance the performance of the implemented parsers. The conventional approach to implementing a lexical analyzer using a DFA (in particular, a Mealy machine) can be realized quite simply using an NN DFA [8].

However, a major drawback of this approach is that all legal transitions have to be exhaustively specified in the DFA. For example, Fig. 4 shows a simplified state diagram without all legal transitions specified for a lexical analyzer which recognizes keywords of a programming language: `begin`, `end`, `if`, `then`, and `else`. Suppose the lexical analyzer is in a state that corresponds to the end of a keyword. Then its current state would be state *7*, *11*, *15*, *18*, or *23*. If the next input character is *b*, there should be legal transitions defined from those states to state *2*. That is the same for states *8*, *16* and *19* in order to handle the next input characters e, i, and t. Thus, this extremely simple lexical analyzer with 22 explicitly defined legal transitions has 20 unspecified transitions. The realization of such a simple five-word (23-state) lexical analyzer by an NN DFA requires $20 + 22 = 42$ hidden neurons. Additional transitions have to be defined in order to allow multiple blanks between two consecutive words in the input stream, and for error handling. These drawbacks are further exacerbated in applications involving languages with large vocabularies.

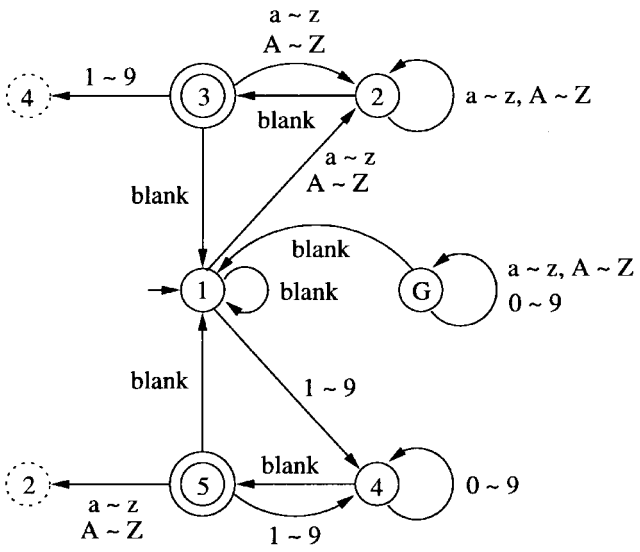

Fig. 5. The state diagram of a DFA which simulates a simple word segmenter carving continuous input stream of characters into words including integer constants, keywords and identifiers. Both the keywords and identifiers are strings of English characters.

A better alternative is to use a dictionary (or a database) to serve as a lexicon. The proposed design for NNLexAn consists of a *word segmenter* for carving an input stream of characters into a stream of words, and a *word lookup table* for translating the carved words of variable length into syntactically tagged tokens of fixed length. Such a translation can be realized by a simple query to a database using a key. Such database query processing can be efficiently implemented using neural associative memories [10].

A. Neural-Network Design for a Word Segmenter (NNSeg)

In program translation, the primary function of a word segmenter is to identify *illegal words* and to group input stream into *legal words* including keywords, identifiers, constants, operators, and punctuation symbols. A word segmenter can be defined by a recursive symbolic function $\hat{f}_{WordSeg}$: $\Gamma^*\$ \rightarrow \Lambda^*\$$, where Γ is the input alphabet, \$ is a special symbol denoting "*end of input*," and Λ is the set of legal words. $\Gamma^*\$$ (or $\Lambda^*\$$) denotes the set of strings obtained by adding the suffix \$ to each of the strings over the alphabets Γ (or Λ).

Fig. 5 shows the state diagram of a DFA simulating a simple word segmenter which carves continuous input stream of characters into integer constants, keywords, and identifiers. Both the keywords and identifiers are defined as strings of English characters. For simplicity, the handling of *end-of-input* is not shown in the figure. The word segmenter terminates processing upon encountering the end-of-input symbol. Each time when the word segmenter goes into an accepting state, it

instructs the word lookup table to look up a word that has been extracted from the input stream and stored in a buffer. Any unspecified transition goes to state G which identifies illegal words. State 1 is initial state. The state 2 in dotted circle is identical to state 2 in solid-line circle and that is same for state 4 (it is drawn in this way to avoid clutter).

Since syntax error handling is not discussed here, it may be assumed that any illegal word is discarded by the word segmenter and is also discarded from the buffer which temporarily stores the illegal word being extracted from the input stream. Such a word segmenter can also be realized by an NN DFA. Since any undefined (unimplemented) transition moves into a binary-coded state of all zeros automatically in an NN DFA, it would be expedient to encode the garbage state (state G in Fig. 5) using a string of all zeros. Although the most straightforward implementation of NN DFA [8] (also see Section III-B) uses one hidden neuron per transition, one can do better. In Fig. 5 the ten transitions from state 4 on ASCII-coded input symbols $0, 1, \cdots, 9$ can be realized by only two hidden neurons in an NN DFA using *partial pattern recognition* [9], [10]. Other transitions on input symbols $0, 1, \cdots, 9, a, b, \cdots, z$, and A, B, $\cdots$, Z can be handled in a similar fashion.

B. Neural-Network Design for a Word Lookup Table (NNLTab)

During lexical analysis in program compilation or similar applications, each word of variable length (extracted by the word segmenter) is translated into a *token* of fixed length. Each such token is treated as a single logical entity: an identifier, a keyword, a constant, an operator or a punctuation symbol. Such a translation can be defined by a simple symbolic function $f_{WordTran}$: $\Lambda \cup \{\$\} \rightarrow \Delta \cup \{\$\}$. Here, $\Lambda, \$$, and Δ denote the same entities as in the definition of $\hat{f}_{WordSeg}$ and $\hat{f}_{LexAn}$ above. For example, $f_{WordTran}$ (f) = keyword − token, where if $\in \Lambda$ and keyword-token $\in \Delta$. Note that the function $f_{WordTran}$ can be realized by a BMP module by way of exact match and partial match [9]. In other lexical analysis applications, a word may be translated into a token having two subparts: category code denoting the syntactic category of a word and feature code denoting the syntactic features of a word.

Conventional approach to doing such translation (dictionary lookup) is to perform a simple query on a suitably organized database (with the segmented word being used as the key). This content-based pattern matching and retrieval process can be efficiently and effectively realized by neural associative memories. Database query processing using neural associative memories is discussed in detail in [10] and is summarized briefly in what follows. Each word and its corresponding token are stored as an association pair in a neural associative memory. Each such association is implemented by a hidden neuron and its associated connections. A query is processed in two steps: *identification* and *recall*. During the identification step, a given word is compared to all stored words in parallel by the hidden neurons and their associated first-layer connections in the memory. Once a match is found, one of the hidden neurons is activated to recall the corresponding token using the second-layer connections associated with the activated hidden neuron.

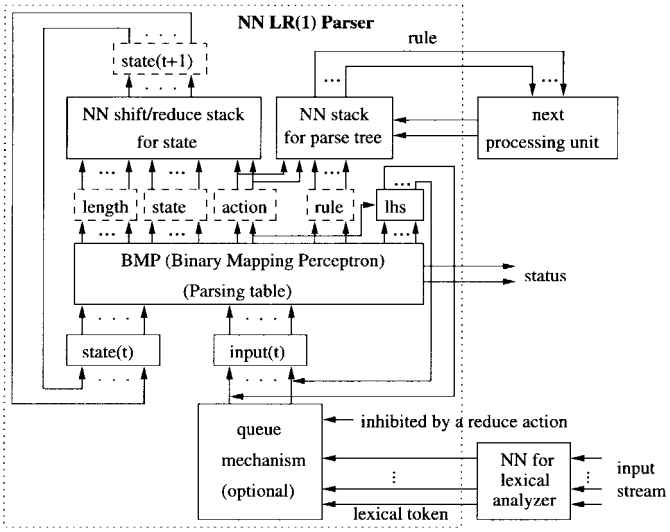

Fig. 6. Neural networks for LR(1) parser. The dotted boxes *length*, *state*, *action*, *rule*, and *state*$(t+1)$ exist only logically but not physically. See text for further explanation.

The time required for processing such a query is of the order of 20 ns (at best) to 100 ns (at worst) given the current CMOS technology for implementation of artificial neural networks. (The interested reader is referred to [10] for details).

VI. A Modular Neural Architecture for LR Parser (NNLR Parser)

LR(k) grammars generate the so-called deterministic context-free languages which can be accepted by deterministic push-down automata [36]. Such grammars find extensive applications in programming languages and compilers. LR parsing is a linear time table-driven algorithm which is widely used for syntax analysis of computer programs [1], [7], [91]. This algorithm involves extensive pattern matching which suggests the consideration of a neural-network implementation using associative memories. This section proposes a modular neural-network architecture (Fig. 6) for parsing LR(1) grammars. LR(k) parsers scan input from left to right and produce a rightmost derivation tree by using lookahead of k unscanned input symbols. Since any LR(k) grammar for k ≥ 1 can be transformed into an LR(1) grammar [91], LR(1) parsers are sufficient for practical applications [36].

An LR(1) grammar can be defined as $G_{LR(1)} = (V, T, \Upsilon, \Theta)$ [36], where V and T are finite sets of variables (nonterminals) and terminals respectively, Υ is a finite set of production rules, and $\Theta \in V$ is a special variable called the *start symbol*. V and T are disjoint. Each production rule is of the form $A \rightarrow \alpha$, where $A \in V$ and $\alpha \in (V \cup T)^*$. An LR(1) parser can be defined by a recursive symbolic function $\hat{f}_{LRParser}$: $\Delta^*\$ \rightarrow \Upsilon^*$, where Δ ($\Delta = T$ in the context), \$ and Δ^* are as in $\hat{f}_{LexAn}$, and Υ^* denotes the set of all sequences of production rules over the rule alphabet Υ. Although $\hat{f}_{LRPaser}$ corresponds in form to the recursive symbolic function $\hat{f}_{LexAn}$ in Section V, it can not be realized simply by a Mealy machine which implements $\hat{f}_{LexAn}$. This is due to the fact that the one-to-one mapping relationship

TABLE I
THE PARSE TABLE OF THE LR(1) PARSER FOR GRAMMAR G_1. THE SYMBOLS IN THE TOP ROW ARE GRAMMAR SYMBOLS. THE SYMBOL $*$ DENOTES A `don't care`

State	$	E	T	F
q_0		(s,q_1,*,*,*,i)	(s,q_2,*,*,*,i)	(s,q_3,*,*,*,i)
q_1	(*,*,*,*,*,a)			
q_2	(r,*,p_2,1,E,i)			
q_3	(r,*,p_4,1,T,i)			
q_4		(s,q_8,*,*,*,i)	(s,q_2,*,*,*,i)	(s,q_3,*,*,*,i)
q_5	(r.*,p_6,1,F,i)			
q_6			(s,q_9,*,*,*,i)	(s,q_3,*,*,*,i)
q_7				(s,q_{10},*,*,*,i)
q_8				
q_9	(r,*,p_1,3,E,i)			
q_{10}	(r,*,p_3,3,T,i)			
q_{11}	(r,*,p_5,3,F,i)			

between every input symbol of the input string and the output symbol of the output string at corresponding position in a Mealy machine does not hold for $\hat{f}_{LRParser}$. A stack is required to store intermediate results of the parsing process in order to realize an LR(1) parser which is characterized by $\hat{f}_{LRParser}$.

A. Representation of Parse Table

Logically, an LR parser consists of two parts: a driver routine which is the same for all LR parsers and a parse table which is grammar-dependent [1]. LR parsing algorithm precompiles an LR grammar into a parse table which is referred by the driver routine for deterministically parsing input string of lexical tokens by `shift/reduce` moves [1], [7]. Such a parsing mechanism can be simulated by a DPDA (deterministic pushdown automata) with ϵ-moves [36]. An ϵ-move does not consume the input symbol, and the input head is not advanced after the move. This enables a DPDA to manipulate a stack without reading input symbols. The neural-network architecture for DPDA (NN DPDA) proposed in [8], augmented with an NNStack (see Section IV above), is able to parse DCFL. However, the proposed NN DPDA architecture cannot efficiently handle ϵ-moves because of the need to check for the possibility of an ϵ-move at every state. Therefore, a modified design for LR(1) parsing is discussed below.

Parse table and stack are the two main components of an LR(1) parser. Parse table access can be defined by the symbolic function $f_{ParseTable}$: $Q \times (\Delta \cup V \cup \{\$\}) \rightarrow A \times Q \cup \{*\} \times \Upsilon \cup \{*\} \times N \cup \{*\} \times V \cup \{*\} \times Z$ in terms of binary mapping. Here, Q is the finite set of states; $\Delta, V, \$$, and Υ have the same meaning as in the definition of $G_{LR(1)}$ and $\hat{f}_{LRParser}$ given above; $A = \{$`shift,reduce`$\}$ is the set of parsing actions; $*$ denotes a `don't care`; N is the set of natural numbers; and $Z = \{$`error, in − progress, accept`$\}$ is the set of possible parsing status values. Table I in Section VI-D is such an example for a parse table.

A parse table can be realized using a BMP module as described in Sections II-B and II-C in terms of binary mapping. The next move of the parsing automaton is determined by current input symbol a and the state q that is stored at the top of the stack. It is given by the parse table entry corresponding to $[q, a]$. Each such two-dimensional parse table entry *action*$[q, a]$ is implemented as a 6-tuple binary code ⟨*action, state, rule, length, lhs, status*⟩ in the BMP for parse table where

- *action* is a 2-bit binary code denoting one of two possible actions, 01 (`shift`) or 10 (`reduce`);
- *state* is an S-bit binary number denoting "*the next state*;"
- *rule* is an R-bit binary number denoting the grammar production rule p to be applied if the consulted *action* is a `reduce`;
- *length* is an L-bit binary number denoting the length of the right-hand side of the grammar production rule p to be applied if the consulted *action* is a `reduce`;
- *lhs* is an H-bit binary code encoding the grammar nonterminal symbol at the left-hand side of the grammar production rule p to be applied if the consulted *action* is a `reduce` and
- *status* is a 2-bit binary code denoting one of three possible *parsing status* codes, 00: *error*, 01: `in progress`, or 10: `accept` (used by higher-level control to acknowledge the success or failure of a parsing).

Note that the order of the tuple's elements arranged in Fig. 6 is different from above. A canonical LR(1) parse table is relatively large and would typically have several thousand states

for a programming language like C. SLR(1) and LALR(1) tables, which are far smaller than LR(1) table, typically have several hundreds of states for the same size of language, and they always have the same number of states for a given grammar [1]. (The interested readers are referred to [7] for a discussion of differences among LR, SLR, and LALR parsers). The number of states in the parse table of LALR(1) parsers for most programming languages is between about 200 and 450, and the number of symbols (lexical tokens) is around 50 [7], i.e., the number of table entries is between about 10 000 and 22 500.

Typically a parse table is realized as a two-dimensional array in current computer systems. Memory is allocated for every entry of the parse table, and the access of an entry is via its offset in the memory, which is computed efficiently by the size of the fixed memory space for each entry and the indexes of an entry in the array. However, it is much more natural to retrieve an entry in a table using content-based pattern matching on the indexes of the entry. As described in Sections II-B and II-C a BMP module can effectively and efficiently realize such content-based table lookup.

LR grammars used in practical applications typically produce parse tables with between 80 and 95% undefined error entries [7]. The size of the table is reduced by using lists which can result in a significant performance penalty. The use of a BMP module for such table lookup help overcome this problem since undefined mappings are naturally realized by a BMP module without the need for extra space and without incurring any performance penalties. Thus, LALR(1) parsing (which is generally the technique of choice for parsing computer programs) table can be realized using at most about $22500 \times 20\% = 4500$ hidden neurons.

B. Representation of Parse Tree and Parsing Moves

An LR parser scans input string from left to right and performs bottom-up parsing resulting in a rightmost derivation tree in reverse. Thus, a stack can be used to store the *parse tree* (derivation tree) which is a sequence of grammar production rules (in reverse order) applied in the derivation of the scanned input string. The rule on top of the final stack which stores a successfully parsed derivation tree is a grammar production rule with the *start symbol* of an LR grammar at its left-hand side. Note that each rule is represented by an R-bit binary number and the mapping from a binary-coded rule to the rule itself can be realized by a BMP module.

A *configuration* of an LR parser is an ordered pair whose first component corresponds to the stack contents and whose second component is the part of the input that remains to be parsed. A configuration can be denoted by $(q_0 q_1 \cdots q_i, a_j a_{j+1} \cdots a_n \$)$, where q_i is the state on top of the stack (current state), q_0 is the stack bottom symbol, a_j is current input symbol, and \$ is a special symbol denoting "*end of input*." The initial configuration is $(q_0, a_1 a_2 \cdots a_n \$)$. In the following, we use the example grammar G_1 and the input lexical token string I $\times$I + I \$ $(= S_1)$ in Section VI-D for illustration, and binary and symbolic codes are used interchangeably. Then, the initial configuration for the input lexical string S_1 is $(q_0,$ I $\times$ I + I \$). Let 0^k be a k-bit binary number (code) of all zeros denoting a value of `don't care` for $k \geq 1$. In the proposed NNLR Parser, the configurations resulting from one of four types of *moves* on parsing an input lexical token are as follows.

- If *action*$[q_i, a_j] = \langle 01, q, 0^R, 0^L, 0^H, 01 \rangle$, the parser performs a `shift` move and enters the configuration $(q_0 q_1 \cdots q_i q, a_{j+1} \cdots a_n \$)$. For example, for the parser of G_1 with a current configuration $(q_0,$ I $\times$ I + I \$), the next configuration is $(q_0 q_5, \times$ I + I \$) since *action* $[q_0,$ I$] = (\mathrm{s}, q_5, *, *, *, \mathrm{i})$ according to G_1's parse table (Table I in Section VI-D), where s denotes `shift` (coded as 01) and i denotes `in progress` (coded as 01). Such a `shift` move is realized in one cycle in the proposed NNLR Parser.
- If *action*$[q_i, a_j] = \langle 10, 0^S, p, l, h, 01 \rangle$, the parser performs a `reduce` by producing a binary number p (which denotes a grammar production rule $A \rightarrow \beta$ being applied, where the grammar nonterminal A is denoted by the binary code h, and l is the number of nonempty grammar symbols in β) as part of the parse tree, popping l symbols off the stack, consulting parse table entry $[q_{i-l}, h]$ and entering the configuration $(q_0 q_1 \cdots q_{i-l} q, a_j \cdots a_n \$)$ where *action*$[q_{i-l}, h] = \langle 01, q, 0^R, 0^L, 0^H, 01 \rangle$. For example, for the parser of G_1 with a current configuration $(q_0 q_5, \times$ I + I \$), the parser first consults G_1's parse table for *action*$[q_5, \times] = (\mathrm{r}, *, p_6, 1, \mathrm{F}, \mathrm{i})$, where r denotes `reduce` (coded as 10). Then, the parser performs a `reduce` move, pushes production rule p_6 onto the stack which stores parse tree, pops one state (which is q_5) off the stack which stores states, and consults the parse table for *action*$[q_0, \mathrm{F}]$, where F is the left-hand side of production rule p_6. Then, since *action*$[q_0, \mathrm{F}] = (\mathrm{s}, q_3, *, *, *, \mathrm{i})$, the parser performs a `shift` move and enters the new configuration $(q_0 q_3, \times$ I + I \$). Such a `reduce` move is realized in two cycles in the proposed NNLR Parser since the parse table is consulted twice for simulating the move.
- If *action*$[q_i, a_j] = \langle 0^2, 0^S, 0^R, 0^L, 0^H, 10 \rangle$, parsing is completed.
- If *action*$[q_i, a_j] = \langle 0^2, 0^S, 0^R, 0^L, 0^H, 00 \rangle$, an error is discovered and the parser stops. Note that such an entry is a binary code of all zeros. (We do not discuss error handling any further in this paper).

C. Architecture of an NNLR Parser

Fig. 6 shows the architecture of a modular neural-network design for an LR(1) parser which takes advantage of the efficient `shift/reduce` technique. The NNLR Parser uses an optional queue handler module and an NN stack which stores the parse tree (derivation tree). The queue handler stores lexical tokens extracted by the NN lexical analyzer and facilitates the operation of lexical analyzer and parser in parallel. To extract the binary-coded grammar production rules in derivation order sequentially out of the NN stack which stores parse tree, the next processing unit connected to the NN stack sends binary-coded stack `pop` actions to the stack in an appropriate order.

Modules of the NNLR Parser The proposed NNLR Parser consists of a BMP module implementing the parse ta-

ble, an NN shift/reduce stack storing states during shift/reduce simulation, a buffer (state) storing the current state (from the top of the NN shift/reduce stack), and a buffer (input(t)) storing either current input lexical token or a grammar nonterminal symbol produced by last consulted parsing action which is a reduce. When the last consulted parsing action is a reduce encoded as 10; the grammar production rule to be reduced is pushed onto the stack for parse tree, the transmission of input(t) is from the latched buffer *lhs*, and the input from the queue mechanism is inhibited by the leftmost bit of the binary-coded reduce action. When the last consulted parsing action is a shift encoded as 01, the transmission of input(t) is from the queue mechanism and the input from the latched buffer *lhs* is inhibited by the rightmost bit of the binary-coded shift action.

Parsing is initiated by reset signals to the NN shift/reduce stack and the NN stack storing parse tree. The signals reset the SP's of these two stacks to stack bottom and hence state(t) is reset to initial state. To avoid clutter, the reset signal lines are not shown in Fig. 6. The current state buffer state(t) and the current input buffer input(t) need to be synchronized but the necessary synchronization circuit is omitted from Fig. 6.

The working of an LR parser can be viewed in terms of a sequence of transitions from an initial configuration to a final configuration. The transition from one configuration to another can be divided into two steps: the first involves consulting the parse table for next action using current input symbol and current state on top of the stack; the second step involves execution of the action—either a shift or a reduce—as specified by the parse table. In the NNLR Parser, the first step is realized by a BMP module which implements the parse table lookup; and the second step is executed by a combination of an NN shift/reduce stack which stores states, and an NN stack which stores the parse tree (and a BMP module when the next action is a reduce).

Complexity of the BMP Module for Parse Table: Let M be the number of defined *action* entries in the parse table. All grammar symbols are encoded into H-bit binary codes. The BMP module for parse table uses $S+H$ input neurons, M hidden neurons, and $4+S+R+L+H$ output neurons. Note that the BMP module produces a binary output of all zeros, denoting a parsing error (see previous description of status code in an *action* entry of the parse table), for any undefined *action* entry in the parse table. The R-bit binary-coded grammar production rule is used as the stack symbol for the NN stack which stores the parse tree.

Complexity of the NN Stack for Parse Tree: Assume the pointer control module of the NN stack for parse tree use m_p bits to encode its SP values. Then the pointer control module of the NN stack for parse tree uses m_p+2 input neurons, 3×2^{m_p} hidden neurons, and m_p output neurons. The stack memory module uses m_p input neurons, 2^{m_p} hidden neurons, and R output neurons. The write control module receives m_p+1 binary inputs (the stack pointer + push/pop signal), and R binary inputs (the grammar production rule).

Complexity of the Shift/Reduce NN Stack: To efficiently implement the reduce action in LR parsing, the NN shift/reduce stack can be slightly modified from the NN stack described in Section IV to allow multiple stack pops in one operation cycle of the NNLR Parser. The number of pops is coded as an L-bit binary number and equals the number of nonempty grammar symbols at the right-hand side of the grammar production rule being reduced. It is used as input to the pointer control module and write control module in the NN shift/reduce stack. Thus, the pointer control module uses L additional input neurons in the NN shift/reduce stack as compared to the NNstack proposed in Section IV. The S-bit output from the NN parse table, namely, the S-bit binary code for state, is used as the stack symbol to the NN shift/reduce stack. Let the maximum number of nonempty grammar symbols that appear in the right-hand side of a production rule in the LR grammar being parsed be L_m. Then k multiple pops are implemented in the NN shift/reduce stack in a manner similar to a single pop in the NN stack proposed in Section IV except that the SP value is decreased by k instead of 1, where $1\leq k\leq L_m$. Hence for each SP value, L_m-1 additional hidden neurons are required to allow multiple pops in the pointer control module.

D. NNLR Parser in Action

This section illustrates the operation of the proposed NNLR Parser to parse a given LR(1) grammar.

The Example Grammar: The example of LR(1) grammar (G_1) used here is taken from [1]. The BNF (Backus–Naur Form) description of the grammar G_1 is as follows:

$$expression \rightarrow expression + term | term$$
$$term \rightarrow term \times factor | factor$$
$$factor \rightarrow (expression) | identifier.$$

Using E, T, F, and I to denote expression, term, factor, and identifier (respectively), these rules can be rewritten in the form of production rules p_1 through p_6

$$\begin{aligned}
&\text{Production rule } p_1\text{:} \quad \mathrm{E}\rightarrow \mathrm{E}+\mathrm{T}\\
&\text{Production rule } p_2\text{:} \quad \mathrm{E}\rightarrow \mathrm{T}\\
&\text{Production rule } p_3\text{:} \quad \mathrm{T}\rightarrow \mathrm{T}\times \mathrm{F}\\
&\text{Production rule } p_4\text{:} \quad \mathrm{T}\rightarrow \mathrm{F}\\
&\text{Production rule } p_5\text{:} \quad \mathrm{F}\rightarrow (\mathrm{E})\\
&\text{Production rule } p_6\text{:} \quad \mathrm{F}\rightarrow \mathrm{I}.
\end{aligned}$$

Then $\{\mathrm{I},+,\times,(,)\}$ is the set of terminals (i.e., the set of possible lexical tokens from the lexical analyzer), {E, T, F} is the set of nonterminals, $\{p_1,p_2,p_3,p_4,p_5,p_6\}$ is the set of production rules, and E is the start symbol of the grammar G_1.

The operation of the parser is shown in terms of symbolic codes (instead of the binary codes used by the NN implementation) to make it easy to understand. Note however that the transformation of symbolic codes into binary form used by NNLR Parser is rather straightforward and has been explained in the preceding sections.

The Example Parse Table: Let s and r denote the parsing actions shift and reduce and a, e, and i the parsing status values accept, error, and in progress, respectively.

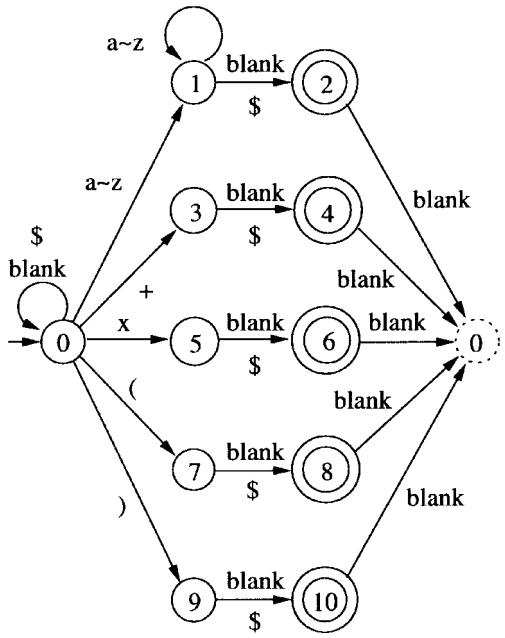

Fig. 7. The state diagram of the DFA M_{L_1} for the lexical analyzer L_1. When the DFA receives a \$, it stops the processing of the input string.

TABLE II
THE TRANSITION FUNCTION δ_{L_1} OF THE DFA M_{L_1}. THE ENTRIES THAT ARE LEFT BLANK ARE UNDEFINED. THESE CORRESPOND TO THE ERROR STATE

State	a, b, ..., z	+	×	(	)	blank	$
q_0	q_1	q_3	q_5	q_7	q_9	q_0	q_0
q_1	q_1					q_2	q_2
q_2	q_1	q_3	q_5	q_7	q_9	q_0	q_0
q_3						q_4	q_4
q_4	q_1	q_3	q_5	q_7	q_9	q_0	q_0
q_5						q_6	q_6
q_6	q_1	q_3	q_5	q_7	q_9	q_0	q_0
q_7						q_8	q_8
q_8	q_1	q_3	q_5	q_7	q_9	q_0	q_0
q_9						q_{10}	q_{10}
q_{10}	q_1	q_3	q_5	q_7	q_9	q_0	q_0

The parse table of the LR(1) parser (more specifically, SLR(1) parser) for grammar G_1 is shown in Table I.

The implementation and operations of the NN `shift/reduce` stack and the NN stack for parse tree follow the discussion and examples in Section IV and they are not discussed here. The parse table can be represented by a binary mapping which in turn can be easily realized by a neural associative processor (see Section II for details). Following the notation introduced in Section VI for the NN realization of the parse table, we have: $M = 45$ since there are 45 defined entries in the parse table; $S = \lceil log_2 12 \rceil = 4$ since there are 12 states; $H = \lceil log_2(8+2) \rceil = 4$ since there are 8 grammar symbols plus a *null symbol* ϵ and an additional *end-of-input* symbol \$; $R = \lceil log_2 6 \rceil = 3$ since there are 6 production rules; and $L = \lceil log_2 3 \rceil = 2$ since the maximum number of nonempty grammar symbols that appear in the right-hand side of a production rule in the LR grammar G_1 is 3. Therefore, the BMP for parse table of G_1 has $S + H = 8$ input, 45 hidden, and $4+S+R+L+H = 17$ output neurons.

The Example Lexical Analyzer: Assume every identifier I is translated from a string of lower case English characters. The lexical analyzer L_1 which translates input strings of $+, \times, (,), \$$, *blank*, and lower case English characters into strings of lexical tokens can be realized by an NN DFA. Fig. 7 shows the simplified state diagram of the DFA M_{L_1} for L_1. Note that additional machinery needed for error handling is not included in the DFA M_{L_1}; and when the DFA sees a \$, it stops the processing of the input string and appends a \$ at the end of the output string. The state 0 in the dotted circle is identical to the state 0 in solid circle. To avoid clutter, some transitions are not shown in the figure. For example, there are transition from states 2, 4, 6, 8, and 10 to state 1 on current input symbols a, b, $\cdots$, z. Similarly, transitions from states 2, 4, 6, 8, and 10 to state 3 (on $+$), state 5 (on $\times$), state 7 (on '(') and state 9 (on ')') are not shown in the figure.

The transition function δ_{L_1} of the DFA M_{L_1} is shown in Table II. This function can be expressed as a binary mapping which in turn can be easily realized by a neural associative processor (see Section II for details). In the NN DFA, BMP module 1 realizes the transition function $\delta_{L_1}: Q \times \Gamma \rightarrow Q$ and BMP module 2 realizes a translation function $\lambda': Q \rightarrow \Delta$ s.t. $\lambda'(q_2) = \mathrm{I}, \lambda'(q_4) = +$, $\lambda'(q_6) = \times$, $\lambda'(q_8) = (, \lambda'(q_{10}) =)$, and $\lambda'(q) = \epsilon$ (null symbol, which is discarded) for other $q \in Q$, where $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}\}$ is the set of states, $\Gamma = \{a, b, \cdots, z, +, \times, (,)\$, \text{ blank}\}$ is the input alphabet, and $\Delta = \{\mathrm{I}, +, \times, (,)\}$ is the output alphabet (i.e., the set of lexical tokens). The symbolic functions δ_{L_1} and λ' can be expressed as binary mappings which in turn can be realized by neural associative processors (see Section II for details).

The Operations of the Example NNLR Parser: Let us now consider the operation of the LR(1) parser when it is presented with the input string $aa \times bbb + cccc\$$. This string is first translated by the lexical analyzer L_1 into a string of lexical tokens I × I + I \$ which is then provided to the LR(1) parser. This translation is quite straightforward, given the state diagram and transition function δ_{L_1} (Table II) of M_{L_1} and its translation function λ'. Note that there is a `space` between each pair of consecutive words in the character string, and there is no `space` token between each pair of consecutive lexical tokens in the string of lexical tokens.

The string of lexical tokens is parsed by the LR(1) parser whose moves are shown in Fig. 8. At step 1, the parse table entry corresponding to (q_0, I) is consulted. Its value is (s, $q_5, *, *, *$, i). This results in shifting I and pushing state

Step	Content of shift/reduce stack	Remaining input	Referred entries of parse table	Content of parse tree stack
(1)	q_0	I×I+I$	(q_0,I)	$\perp$
(2)	q_0q_5	×I+I$	(q_5,×), (q_0,F)	$\perp$
(3)	q_0q_3	×I+I$	(q_3,×), (q_0,T)	$\perp p_6$
(4)	q_0q_2	×I+I$	(q_2,×)	$\perp p_6p_4$
(5)	$q_0q_2q_7$	I+I$	(q_7,I)	$\perp p_6p_4$
(6)	$q_0q_2q_7q_5$	+I$	(q_5,+), (q_7,F)	$\perp p_6p_4$
(7)	$q_0q_2q_7q_{10}$	+I$	(q_{10},+), (q_0,T)	$\perp p_6p_4p_6$
(8)	q_0q_2	+I$	(q_2,+), (q_0,E)	$\perp p_6p_4p_6p_3$
(9)	q_0q_1	+I$	(q_1,+)	$\perp p_6p_4p_6p_3p_2$
(10)	$q_0q_1q_6$	I$	(q_6,I)	$\perp p_6p_4p_6p_3p_2$
(11)	$q_0q_1q_6q_5$	$	(q_5,$), ($q_6$,F)	$\perp p_6p_4p_6p_3p_2$
(12)	$q_0q_1q_6q_3$	$	(q_3,$), ($q_6$,T)	$\perp p_6p_4p_6p_3p_2p_6$
(13)	$q_0q_1q_6q_9$	$	(q_9,$), ($q_0$,E)	$\perp p_6p_4p_6p_3p_2p_6p_4$
(14)	q_0q_1	$	(q_1,$)	$\perp p_6p_4p_6p_3p_2p_6p_4p_1$

Fig. 8. Moves of the LR(1) parser for grammar G_1 on input string I × I + I. Note that the parse table is accessed twice for each `reduce` action.

q_5 onto the `shift/reduce` stack. At step 2, the table entry corresponding to $(q_5, \times)$ is consulted first. Its value is $(\mathrm{r}, *, p_6, 1, \mathrm{F}, \mathrm{i})$ which indicates a `reduce` on production rule p_6. Therefore, production rule p_6 is pushed onto the stack which stores parse tree, state q_5 is popped off the stack which stores states, and table entry corresponding to $(q_0$, F) is consulted next. The entry is $(\mathrm{s}, q_3, *, *, *, \mathrm{i})$ which means shifting F and pushing state q_3 onto the stack. Fig. 9 shows the inputs and outputs of the neural modules in the NNLR Parser when the NNLR Parser sees an input lexical token × at state q_5. The remaining steps are executed in a similar fashion. At the end of the moves (step 14), the sequence of production rules stored on the stack for parse tree can be applied in reverse order to derive the string I × I + I from grammar start symbol E.

VII. Performance Analysis

This section explores the performance advantages of the proposed neural-network architecture for syntax analysis in comparison with that of current computer systems that employ inherently sequential index or matrix structure for pattern matching. The performance estimates for the NNLR Parser assume hardware realization based on current CMOS VLSI technology. In the analysis that follows, it is assumed that the two systems have comparable input–output (I/O) performance and error handling capabilities.

A. Performance of Hardware Realization of the NNLR Parser

Electronic hardware realizations of ANN have been explored by several authors [22], [23], [26], [47], [0], [51], [59], [78], [98], [101]. Such implementations typically employ CMOS analog, digital, or hybrid (analog/digital) electronic circuits. Analog circuits typically consist of processing elements for multiplication, summation and thresholding. Analog CMOS technology is attractive for realization of ANN because it can yield compact circuits that are capable of high-speed asynchronous operation [21]. Reference [98] reports a measured propagation delay of 104 ns in a digital circuit with each synapse containing an 8-bit memory, an 8-bit subtractor, and an 8-bit adder. Reference [98] reports throughput at the rate of 10MHz (or equivalently, delay of 100ns) in a Hamming Net pattern classifier using analog circuits. Reference [23] describes a hybrid analog-digital design with 5-bit (4 bits + sign) binary synapse weight values and current-summing circuits that is used to realize a 2-layer feedforward ANN with a network computation delay of less than 20 ns.

The first- and second-layer subnetworks of the proposed BMP neural architecture are very similar to the first-layer subnetwork of a Hamming Net. BMP architecture with two layers of connection weights is same as that implemented by [50] except that the latter uses integer input values, 5-bit synaptic weights, one output neuron, and sigmoid-like activation function whereas BMP uses binary input values, synaptic weights from $\{-1, 0, 1\}$, multiple output neurons, and binary hardlimiter as activation function. Hence the computation delay of BMP module implemented using current CMOS technology can be expected to be at best of the order of 20 ns. It is worth noting that development of specialized hardware for implementation of ANN is still in its infancy. Conventional CMOS technology which is probably the most common choice for VLSI implementation of ANN at present, is very likely to be improved by newer technologies such as BiCMOS, NCMOS [42], pseudo-NMOS logic, standard N-P domino logic, and quasi N-P domino logic [48].

B. Performance of Syntax Analysis Using Conventional Computers

To simplify the comparison, it is assumed that each instruction on a conventional computer takes τ ns on an average. For instance, on a relatively cost-effective 100 MIPS processor, a typical instruction would take 10 ns to complete. (The MIPS measure for speed combines clock speed, effect of caching, pipelining and superscalar design into a single figure for speed of a microprocessor). Similarly, we will assume that a single identification and recall operation by a neural associative memory takes α ns. Assuming hardware implementation based on current CMOS VLSI technology, $\alpha = 20$ ns.

Syntax analysis in a conventional computer typically involves: lexical analysis, grammar parsing, parse tree construction and error handling. These four processes are generally coded into two modules [1]. Error handling is usually embedded in grammar parsing and lexical analysis respectively, and parse tree construction is often embedded in grammar parsing. The procedure for grammar parsing is the main module. In single-CPU computer systems, even assuming negligible overhead for parameter passing, a procedure call entails, at the very minimum, (1) saving the context of the caller procedure and activation of the callee procedure

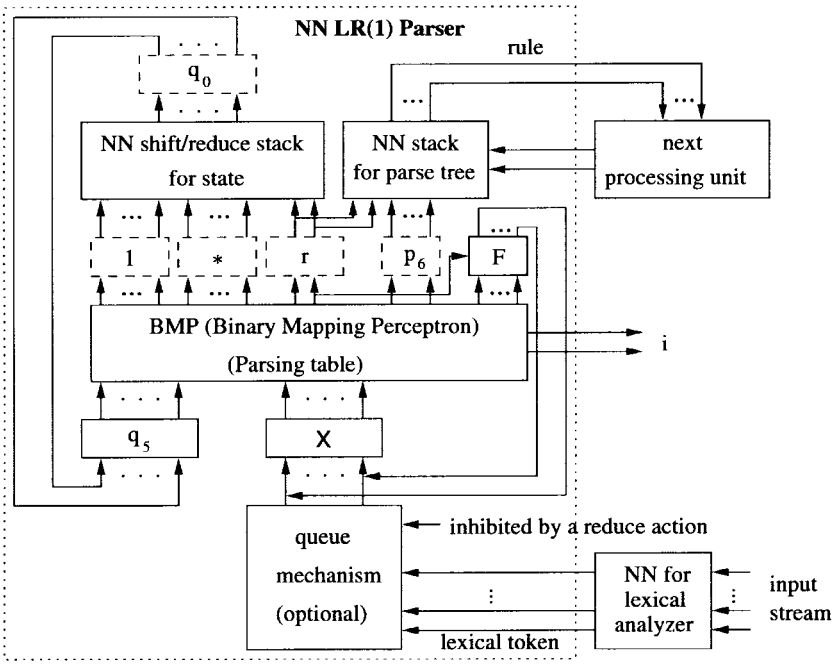

Fig. 9. The inputs and outputs of the neural modules in the NNLR Parser when the NNLR Parser sees an input lexical token $\times$ at state q_5.

which typically requires six instructions [49]; and (2) context restoration and resumption of caller procedure upon the return (exit) of the callee procedure, which typically requires at least three instructions [49]). Thus, a procedure call entails a penalty of nine instructions or about 9τ ns.

Performance Analysis of Lexical Analyzer: Lexical analysis can be performed by a DFA whose transitions can be represented as a two-dimensional table with current state and current input symbol as indexes. The continuous transition moves of such a DFA involve repetitive lookup of the next state from the table using current state and current input symbol at each move until an error state or an accepting state is reached. Such a repetitive table lookup involves content-based pattern matching and retrieval which can be performed more efficiently by neural memories.

Each entry of the DFA transition table implemented on conventional computers contains three parts: the next state; a code which tells whether the next state is an accepting state, an error state, or neither; and the lexical token to use if the next state is an accepting state. Implementing such a repetitive table lookup on conventional computers requires, at a minimum, six instructions: one (or two) multiplication and one addition to compute the offset in the transition table (to access the location where next state is stored), one memory access to fetch the next state from the table, one addition to compute the offset of the second part in the transition table (based on the known offset of the first part), one memory access to fetch the second part from the table, and one branch-on-comparison instruction to jump back to the first instruction of the loop if the next state is neither an error state nor an accepting state. (Note that this analysis ignores I/O processing requirements). Thus, each state transition takes six instructions or 6τ ns.

In contrast, the proposed NN architecture for lexical analyzer computes the next state using associative (content-addressed) pattern *matching-and-retrieval* in a single *identification-and-recall* step of a BMP module. In the two-dimensional table, the values of the two indexes for an entry provide a unique pattern—the *index pattern*, for accessing the table entry. In the BMP module, each index pattern and the corresponding entry are stored as an association pair by a hidden neuron and its associated connections. The BMP module performs a table lookup in two steps: *identification* and *recall*. In the identification step, a given index pattern is compared to all stored index patterns in parallel by the hidden neurons and their associated first-layer connections. Once a match is found, one of the hidden neurons will be activated to recall the associated entry value using the second-layer connections associated with the activated hidden neuron.

In program compilation, a segmented word is translated into a syntactically tagged token when the DFA for lexical analysis enters an accepting state. On conventional computers, this translation step costs, at the very minimum, three instructions (or 3τ ns): one addition to compute the offset of the third part in the transition table (based on the known offset of the first part), one memory access to fetch the lexical token from the table, and one branch instruction to jump back to the first instruction of the loop for carving next word.

In other syntax analysis applications that involve large vocabularies, a database lookup is typically used to translate a word into a syntactically tagged token. In this case, depending on the size of the vocabulary and the organization of the database, it would generally take more than several hundred nanoseconds (ns) to perform this translation. (The interested reader is referred to [8] for a comparison of database query processing using neural associative memories as opposed to conventional computers).

A BMP module is capable of translating a carved word into a token as described in Section II-B in a single cycle of

identification-and-recall with a time delay of α ns. Note that this step can be pipelined (see the NNLR Parser in action in Section VI-D).

In summary, if we assume the average length of words in input string being W symbols and we ignore I/O, error handling and the overhead associated with procedure calls, it would take $(6W + 3)\tau$ ns on average to perform lexical analysis of a word on a conventional computer. In contrast, it would take $(W + 1)\alpha$ ns using the proposed NN lexical analyzer. This analysis ignores I/O and error handling. For example, assuming a 100 MIPS conventional computer ($\tau =$ 10 ns), and current CMOS VLSI implementation of neural associative memories ($\alpha = 20$ ns), with $W = 5$, then the former takes 330 ns and the latter 120 ns.

Performance Analysis of LR Parser: LR parsing also involves repetitive table lookup which can be performed efficiently by neural associative memories. LR parser is driven by a two-dimensional table (parse table) with current state and current input symbol as indexes. Once a next state is retrieved, it is stored on a stack and is used as the current state for the next move. Parsing involves repetitive application of a sequence of `shift` and `reduce` moves. A `shift` move would take at least six instructions, or equivalently 6τ ns on a conventional computer. This includes three instructions to consult the parse table, one instruction to push the next state onto the stack, one instruction to increment the stack pointer, and one instruction to go back to the first instruction of the repetitive loop for next move. A `reduce` move involves a parse table lookup, a `pop` of the state stack, a `push` to store a rule onto the stack for parse tree, and a `shift` move. Thus, a `reduce` would take at least $3+1+2+6 = 12$ instructions, or equivalently 12τ ns, on a conventional computer.

In the proposed NNLR Parser, the computation delay consists of the delays contributed by the operation of the two NNStacks and the BMP module which stores the parse table. An NNStack consists of two BMP modules, one of which is augmented with a write control module. Assuming that the computation delay of an NN stack is roughly equal to that of two sequentially linked BMP modules (2α ns), a `shift` move (which takes one operation cycle of the NNLR Parser) and a `reduce` move (which takes two operation cycles of the NNLR Parser) would consume 3α ns and 6α ns, respectively. (This analysis ignores the effect of queuing between the NNLR Parser and the NN lexical analyzer).

Assuming that the average length of words in input string be W symbols, and ignoring I/O, error handling and the overhead associated with procedure calls, parsing a word (a word has to be translated into a lexical token by lexical analysis first) by `shift` and `reduce` moves would take $(6W+9)\tau$ ns and $(6W + 15)\tau$ ns, respectively, on a conventional computer.

In contrast, because the NNLR Parser and NN lexical analyzer can operate in parallel, `shift` and `reduce` moves take 3α ns or $(W+1)\alpha$ ns (whichever is larger) and 6α ns or $(W+1)\alpha$ ns (whichever is larger) respectively on the NNLR Parser.

Thus, as shown in Table III, for typical values of α, τ, and W, the proposed NNLR Parser offers a potentially attractive alternative to conventional computers for syntax analysis.

TABLE III
A COMPARISON OF THE ESTIMATED PERFORMANCE OF THE PROPOSED NNLR PARSER WITH THAT OF CONVENTIONAL COMPUTER SYSTEMS FOR SYNTAX ANALYSIS. W IS THE AVERAGE NUMBER OF SYMBOLS IN A WORD, α IS THE COMPUTATION DELAY OF A BMP MODULE, AND τ IS THE AVERAGE TIME DELAY FOR EXECUTION OF AN INSTRUCTION IN CONVENTIONAL COMPUTER SYSTEMS

	NN LR Parser	Conventional computers
time for lexical analysis of a word	$(W+1)\alpha$	$(6W+3)\tau$
time for a `shift` move of parsing	max $[3\alpha, (W+1)\alpha]$	$(6W+9)\tau$
time for a `reduce` move of parsing	max $[6\alpha, (W+1)\alpha]$	$(6W+15)\tau$

It should be noted that the preceding performance comparison has not considered alternative hardware realizations of syntax analyzers. These include hardware implementations of parsers using conventional building blocks used for building today's serial computers. We are not aware of any such implementations although clearly, they can be built. In this context it is worth noting that the neural architecture for syntax analysis proposed in this paper makes extensive use of massively parallel processing capabilities of neural associative processors (memories). It is quite possible that other parallel (possibly nonneural-network) hardware realizations of syntax analyzers offer performance that compares favorably with that of the proposed neural-network realization. We can only speculate as to why there appears to have been little research on parallel architectures for syntax analysis. Conventional computer systems employ inherently sequential indexes or matrix structures for the purpose of table lookup during syntax analysis. A possible hardware implementation of syntax analyzers which avoids the burden of using sequential indexes and matrix structures for table lookup would be to use content-addressable memories for table lookup. Such an implementation would be similar to the proposed neural architecture which is synthesized from neural associative memories. Historically, research in high performance computing has focused primarily on speeding up the execution of numeric computations, typically performed by programs written in compiled languages such as C and FORTRAN. In such applications, syntax analysis is done during program compilation which is relatively infrequently compared to program execution. The situation is quite different in symbol processing (e.g., knowledge based systems of AI, analysis of mathematical expressions in software designed for symbolic integration, algebraic simplification, theorem proving) and interactive programming environments based on interpreted programming languages (e.g., LISP, JAVA). Massively parallel architectures for such tasks are only beginning to be explored.

VIII. SUMMARY AND DISCUSSION

Traditional AI and ANN's offer two apparently disparate approaches to modeling and synthesis of intelligent systems. Each provides natural ways of performing certain tasks (e.g., logical inference in the case of AI systems, pattern recognition in the case of ANN) and thus poses a challenge for the other of doing the same task perhaps more efficiently, elegantly,

robustly, or cost-effectively than the other. Each of them relies on essentially equivalent formal models of computation. They differ from each other in terms of their (implicit and often unstated) assumptions regarding the computational architecture and the primitive building blocks used to realize the desired behaviors. This places the two approaches in different regions of a vast (and as yet, unexplored) space of possible designs for intelligent systems. We believe that a judicious and systematic exploration of this design space, by (among other things) examining alternative designs for specific tasks (e.g., inference, syntax analysis, pattern recognition) is central to the enterprise of analysis and synthesis of intelligent systems.

Against this background, this paper explores the design of a neural architecture for syntax analysis of languages with known (*a priori* specified) grammars. Syntax analysis is a prototypical symbol processing task with a diverse range of applications in artificial intelligence, cognitive modeling, and computer science. Examples of such applications include: language interpreters for interactive programming environments using interpreted languages (e.g., LISP, JAVA), parsing of symbolic expressions (e.g., in real-time knowledge based systems, database query processing, and mathematical problem solving environments), syntactic or structural analysis of large collections of data (e.g., molecular structures, engineering drawings, etc.), and high-performance compilers for program compilation and behavior-based robotics. Indeed, one would be hard-pressed to find a computing application that does not rely on syntax analysis at some level. The need for syntax analysis in real time calls for novel solutions that can deliver the desired performance at an affordable cost. Artificial neural networks, due to their potential advantages for real-time applications on account of their inherent parallelism [81], offer an attractive approach to the design of high performance syntax analyzers.

The proposed neural architecture for syntax analysis is obtained through systematic and provably correct composition of a suitable set of component symbolic functions which are ultimately realized using neural associative processor modules. The neural associative processor is essentially a 2-layer perceptron which can store and retrieve arbitrary binary pattern associations [9]. Since each component in the proposed neural architecture computes a well-defined symbolic function, it facilitates the systematic synthesis as well as analysis of the desired computation at a fairly abstract (symbolic) level. Realization of the component symbolic functions using neural associative processors allows one to exploit massive parallelism to support applications that require syntax analysis to be performed in real time.

The proposed neural network for syntax analysis is capable of handling sequentially presented character strings of variable length, and it is assembled from neural-network modules for lexical analysis, stack processing, parsing, and parse tree construction. The neural-network stack can realize stacks of arbitrary depths (limited only by the number of neurons available). The parser outputs the parse tree resulting from syntax analysis of strings from widely used subsets of deterministic context-free languages (i.e., those generated by LR grammars). Since logically an LR parser consists of two parts: a driver routine which is the same for all LR parsers, and a parse table which varies from one application to the next [2], the proposed NNLR Parser can be used as a general-purpose neural architecture for LR parsing.

It should be noted that the paper's primary focus was on taking advantage of massive parallelism and associative pattern storage, matching, and recall properties of a particular class of neural associative memories in designing high performance syntax analyzers for *a priori* specified grammars. Consequently, it has not addressed several other potential advantages of neural-network architectures for intelligent systems. Notable among these are inductive learning and fault tolerance.

Machine learning of grammars or grammar inference is a major research topic which has been, and continues to be, the subject of investigation by a large number of researchers in artificial intelligence, machine learning, syntactic pattern recognition, neural networks, computational learning theory, natural language processing, and related areas. The interested reader is referred to [34], [44], [54], and [70] for surveys of grammar inference in general and to [3], [5], [11], [16], [18], [17], [29], [37], [39], [55], [60], [61], [64], [66], [82], [84], [87], [93], and [102]–[104] for recent results on grammar inference using neural networks. The neural architecture for syntax analysis that is proposed in this paper does not appear to lend itself to use in grammar inference using conventional neural-network learning algorithms. However, its use in efficient parallel implementations of recently developed symbol processing algorithms for regular grammar inference and related problems [68], [69] is currently under investigation.

Fault tolerance capabilities of neural architectures under different fault models (neuron faults, connection faults, etc) have been the topic of considerable research [9], [86], [96] and are beyond the scope of this paper. However, it is worth noting that the proposed neural-network design for syntax analysis inherits some of the fault tolerance capabilities of its primary building block, the neural associative processor. The interested reader is referred to [9] for details.

It is relatively straightforward to estimate the cost and performance of the proposed neural architecture for syntax analysis based on the known computation delays associated with the component modules (using known facts or a suitable set of assumptions regarding current VLSI technology for implementing the component modules). Our estimates suggest that the proposed system offers a systematic and provably correct approach to designing cost-effective high-performance syntax analyzers for real-time syntax analysis using known (*a priori* specified) grammars.

The choice of the neural associative processors as the primary building blocks for the synthesis of the proposed neural architecture for syntax analysis was influenced, among other things, by the fact that they find use in a wide range of systems in computer science, artificial intelligence, and cognitive modeling. This is because associative pattern matching and recall is central to pattern-directed processing which is at the heart of many problem solving paradigms in AI (e.g., knowledge-based expert systems, case based reasoning) as well as computer science (e.g., database query processing, in-

formation retrieval). As a result, design, VLSI implementation, and applications of associative processors have been studied extensively in the literature [9], [10], [30], [38], [41], [45], [53], [62], [63], [77], [79]. Conventional computer systems employ inherently sequential index or matrix structure for the purpose of table lookup which entails a process pattern matching. The proposed neural associative memory [9] can compare a given input pattern with all stored memory patterns in parallel. Therefore, it can serve as a cost-effective SIMD (single instruction, multiple data) computer system which is dedicated to massively parallel pattern matching and retrieval. The neural-network architecture for syntax analysis proposed in this paper demonstrates the versatility of neural associative processors (memories) as generic building blocks for systematic synthesis of modular massively parallel architectures for symbol processing applications which involves extensive table lookup. A more general goal of this paper is to explore the design of massively parallel architectures for symbol processing using neural associative memories (processors) as key components. This paper takes a small step in this direction and adds to the growing body of literature [20], [34], [46], [95] that demonstrates the potential benefits of integrated neural-symbolic architectures that overcome some of the limitations of today's ANN and AI systems.

APPENDIX

This Appendix illustrates how the modules of an NNStack together realize a stack by considering several successive stack actions in terms of binary codings. The notations used here follow Section IV. Let $m = 6$ and stack symbols be encoded into 8-bit ASCII codes. Then there are 64 possible SP values and $n = 8$. Let $\ddot{w}^1_{ji}$ and $\ddot{w}^2_{kj}$, respectively, denote the first-layer connection weight from input neuron i to hidden neuron j and the second-layer connection weight from hidden neuron j to output neuron k in the pointer control module. Then $1 \le i \le 8$, $1 \le j \le 192$ and $1 \le k \le 6$. Let $\ddot{w}^1_{qp}$ and $\ddot{w}^2_{rq}$, respectively, denote the first-layer connection weight from input neuron p to hidden neuron q and the second-layer connection weight from hidden neuron q to output neuron r in the stack memory module. Then $1 \le p \le 6$, $1 \le q \le 64$ and $1 \le r \le 8$. Let $SP(t)$ denote SP value at time t.

1) At time $= t_1$, suppose $SP(t_1) = 4 = \langle 000\,100 \rangle$ and the stack action to be performed is a push on a stack symbol $\mathtt{a} = 61_{16} = 0110\ 0001_2$. Let $\Phi = \phi_1\phi_2\phi_3\phi_4\ \phi_5\phi_6\phi_7\phi_8$ be the 8-bit binary code denoting the current stack symbol to be pushed. Then $\phi_2 = \phi_3 = \phi_8 = 1$ and other ϕ_i's are 0. Let c_{t_1} be the stack configuration at time t_1. This time step computes a stack push action with $f_{Stack}(\mathtt{push}, \mathtt{a}, c_{t_1}, 4) = (c_{t_1} \cdot \mathtt{a}, 5)$ and $f_{Top}(c_{t_1} \cdot \mathtt{a}, 5) = \mathtt{a}$. Before the execution of the push action (encoded as $\langle 01 \rangle$), $pointer(t) = SP(t_1) = \langle 000\,100 \rangle = 4$; and after the push action, $pointer(t+1) = \langle 000\,101 \rangle = 5$.
   a) Symbolically, the pointer control module computes $f_{PControl}(\mathtt{push}, 4) = 5$. In the pointer control module, the mapping from binary input $\langle 000100, 01 \rangle$ to binary output $\langle 000\,101 \rangle$ is done by the 13th $(13 = 1 + 3\times\ 4)$ hidden neuron and its associated connections. Note that the two rightmost bits of the binary input together denote a stack action push, the six leftmost bits together denote an SP value 4, three hidden neurons are used for three legal stack actions at each SP value, and the first of the three neurons is reserved for push action.
   b) Symbolically, the write control module (plus stack memory module) computes $f_{SWrite}(\mathtt{push}, \mathtt{a}, c_{t_1}, 4) = c_{t_1} \cdot \mathtt{a}$. The write control module uses binary input $\langle 1\,000\,100 \rangle$ to locate the sixth $(6 = 2 + 4)$ hidden neuron of the stack memory module and to update the weights of the eight second-layer connections associated with the sixth hidden neuron according to expression $\ddot{w}^2_{r6} = \phi_r, 1 \le r \le 8$. Note that the leftmost bit of the binary input denotes a stack action push and the six rightmost bits together denote an SP value 4.
   c) Symbolically, the stack memory module computes $f_{Top}(c_{t_1} \cdot \mathtt{a}, 5) = \mathtt{a}$. When $pointer(t+1) = \langle 000\,101 \rangle = 5$ is passed to the stack memory module, its sixth $(6 = 1 + 5)$ hidden neuron is turned on to recall the stack symbol $\mathtt{a} = 0110\ 0001_2$ which is stored by the second-layer connections associated with the sixth hidden neuron. Note that in the stack memory module the first hidden neuron and its associated second-layer connections are used to store the *stack start symbol* (stack bottom) which is pointed by SP $= \langle 000\,000 \rangle$.
2) At time $= t_1 + 1, SP(t_1 + 1) = 5 = \langle 001\,001 \rangle$ and the stack action to be performed is a push on a stack symbol $\mathtt{b} = 62_{16} = 0110\ 0010_2$. Then $\phi_2 = \phi_3 = \phi_7 = 1$ and other ϕ_i's are zero. Symbolically, this time step computes a stack push action with $f_{Stack}(\mathtt{push}, \mathtt{b}, c_{t_1} \cdot \mathtt{a}, 5) = (c_{t_1} \cdot \mathtt{a} \cdot \mathtt{b}, 6)$ and $f_{Top}(c_{t_1} \cdot \mathtt{a} \cdot \mathtt{b}, 6) = \mathtt{b}$. Before the execution of the push action, $pointer(t) = \langle 000\,101 \rangle = 5$; and $pointer(t+1) = \langle 000\,110 \rangle = 6$ after the push action.
   a. Symbolically, the pointer control module computes $f_{PControl}(\mathtt{push}, 5) = 6$. In the pointer control module, the mapping from binary input $\langle 000101, 01 \rangle$ to binary output $\langle 000\,110 \rangle$ is done by the 16th $(16 = 1 + 3 \times 5)$ hidden neuron and its associated connections.
   b. Symbolically, the write control module (plus stack memory module) computes $f_{SWrite}(\mathtt{push}, \mathtt{b}, c_{t_1} \cdot \mathtt{a}, 5) = c_{t_1} \cdot \mathtt{a} \cdot \mathtt{b}$. The write control module uses binary input $\langle 1\,000\,101 \rangle$ to locate the seventh $(7 = 2 + 5)$ hidden neuron in the stack memory module and to update the weights of the second-layer connections associated with the seventh hidden neuron using current Φ as before.
   c. Symbolically, the stack memory module computes $f_{Top}(c_{t_1} \cdot \mathtt{a} \cdot \mathtt{b}, 6) = \mathtt{b}$. When $pointer(t+1) =$

$\langle 000\,110\rangle = 6$ is passed to the stack memory module, its seventh $(7 = 1 + 6)$ hidden neuron is turned on to recall the stack symbol $\mathtt{b} = 0110\ 0010_2$ which is stored by the second-layer connections associated with the seventh hidden neuron.

3) At time $= t_1 + 2, SP(t_1 + 2) = 6 = \langle 000\,110\rangle$ and the stack action to be performed is a pop. Symbolically, this time step computes a stack pop action with $f_{Stack}(\mathtt{pop}, *, c_{t_1} \cdot \mathtt{a} \cdot \mathtt{b}, 6) = (c_{t_1} \cdot \mathtt{a}, 5)$ and $f_{Top}(c_{t_1} \cdot \mathtt{a}, 5) = \mathtt{a}$. Before the execution of the pop action (encoded as $\langle 10\rangle$), $pointer(t) = \langle 000\,110\rangle = 6$; and after the pop action, $pointer(t+1) = \langle 000\,101\rangle = 5$.

   a) Symbolically, the pointer control module computes $f_{PControl}(\mathtt{pop}, 6) = 5$. In the pointer control module, the mapping from binary input $\langle 000110, 10\rangle$ to binary output $\langle 000\,101\rangle$ is done by the 20th $(20 = 2 + 3 \times 6)$ hidden neuron and its associated connections. Note that three hidden neurons are used for three legal stack actions respectively at every SP value, and the second of the three neurons is used for pop action.

   b) Symbolically, the stack memory module computes $f_{Top}(c_{t_1} \cdot \mathtt{a}, 5) = \mathtt{a}$. When $pointer(t+1) = \langle 000\,101\rangle = 5$ is passed to the stack memory module, its sixth hidden neuron is turned on to recall the stack symbol $0110\ 0001_2 = \mathtt{a}$ which is stored by the second-layer connections associated with the sixth hidden neuron.

ACKNOWLEDGMENT

The authors are grateful to Dr. L. Giles of NEC Research Institute, Princeton, NJ, and anonymous referees for helpful suggestions on an earlier draft of this paper.

REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.
[2] A. V. Aho and J. D. Ullman, *Principles of Compiler Design*. Reading, MA: Addison-Wesley, 1977.
[3] R. B. Allen, "Connectionist language users," *Connection Sci.*, vol. 2, no. 4, p. 279, 1990.
[4] M. Arbib, "Schema theory: Cooperative computation for brain theory and distributed AI," in *Artificial Intelligence and Neural Networks: Steps Toward Principled Integration*, V. Honavar and L. Uhr, Eds. San Diego, CA: Academic, 1994, pp. 51–74.
[5] G. Berg, "A connectionist Parser with recursive sentence structure and lexical disambiguation," in *Proc. 10th Nat. Conf. Artificial Intell.*, 1992, pp. 32–37.
[6] L. A. Bookman, "A framework for integrating relational and associational knowledge for comprehension, in *Computational Architectures Integrating Neural and Symbolic Processes: A Perspective on the State of the Art*, R. Sun and L. Bookman, Eds. Norwell, MA: Kluwer Academic Publishers, 1995, ch. 9, pp. 283–318.
[7] N. P. Chapman, *LR Parsing: Theory and Practice*.Cambridge. U.K.: Cambridge Univ. Press, 1987.
[8] C. Chen and V. Honavar, "Neural-network automata," in *Proc. World Congr. Neural Networks*, San Diego, CA, vol. 4, pp. 470–477, June 1994.
[9] C. Chen and V. Honavar, "A neural architecture for content as well as address-based storage and recall: Theory and applications," *Connection Sci.*, vol. 7, nos. 3/4, pp. 281–300, 1995.
[10] ——, "A neural-network architecture for high-speed database query processing system," *Microcomputer Applicat.*, vol. 15, no. 1, pp. 7–13, 1996.
[11] S. Das, C. L. Giles, and G. Z. Sun, "Using prior knowledge in a NNDPA to learn context-free languages," in *Advances in Neural Information Processing Systems 5*, S. J. Hanson, J. D. Cowan, and C. L. Giles, Eds. San Mateo, CA: Morgan Kaufmann, 1993, pp. 65–72.
[12] C. P. Dolan and P. Smolensky, "Tensor product production system: A modular architecture and representation," *Connection Sci.*, vol. 1, pp. 53–58, 1989.
[13] M. G. Dyer, "Connectionist natural language processing: A status report," in *Computational Architectures Integrating Neural and Symbolic Processes: A Perspective on the State of the Art*, R. Sun and L. Bookman, Eds. Norwell, MA: Kluwer, 1995, ch. 12, pp. 389–429.
[14] J. L. Elman, "Finding structure in time," *Cognitive Sci.*, vol. 14., pp. 179–211, 1990.
[15] M. A. Fanty, "Context-free parsing with connectionist networks," *Proc. AIP Neural Networks for Computing, Conf. 151*, Snowbird, UT, 1986, pp. 140–145.
[16] P. Frasconi, M. Gori, M. Maggini, and G. Soda, "Unified integration of explicit rules and learning by example in recurrent networks," *IEEE Trans. Knowledge Data Eng.*, 1993.
[17] C. L. Giles, B. W. Horne, and T. Lin, "Learning a class of large finite state machines with a recurrent neural network," *Neural Networks*, vol. 8, no. 9, pp. 1359–1365, 1995.
[18] C. L. Giles, C. G. Miller, D. Chen, G. Z. Sun, and Y. C. Lee, "Learning and extracting finite state automata with second-order recurrent neural networks," *Neural Comput.*, vol. 4., no. 3., p. 380, 1992.
[19] L. Goldfarb and S. Nigam, "The unified learning paradigm: A foundation for AI," in *Artificial Intelligence and Neural Networks: Steps Toward Principled Integration*, V. Honavar and L. Uhr, Eds. San Diego, CA: Academic, 1994, pp. 533–559.
[20] S. Goonatilake and S. Khebbal, Eds., *Intelligent hybrid systems*. London, U.K.: Wiley, 1995.
[21] S. M. Gowda *et al.*, "Design and characterization of analog VLSI neural-network modules," *IEEE J. Solid-State Circuits*, vol. 28, pp. 301–313, Mar. 1993.
[22] H. P. Graf and D. Henderson, "A reconfigurable CMOS neural network," in *ISSCC Dig. Tech. Papers*, San Francisco, CA, 1990, pp. 144–145.
[23] D. Grant *et al.*, "Design, implementation and evaluation of a high-speed integrated hamming neural classifier," *IEEE J. Solid-State Circuits*, vol. 29, pp. 1154–1157, Sept. 1994.
[24] K. E. Grosspietsch, "Intelligent systems by means of associative processing," in *Fuzzy, Holographic, and Parallel Intelligence*, B. Souček and the IRIS Group, Eds. New York: Wiley, 1992, pp. 179–214.
[25] A. Gupta, "Parallelism in Production Systems," Ph.D. dissertation, Carnegie-Mellon Univ., Pittsburgh, PA, Mar. 1986.
[26] A. Hamilton *et al.*, "Integrated pulse stream neural networks: Results, issues, and pointers," *IEEE Trans. Neural Networks*, vol. 3, pp. 385–393, May 1992.
[27] M. Hassoun, *Fundamentals of Artificial Neural Networks*. Cambridge, MA: MIT Press, 1995.
[28] J. Hendler, "Beyond the fifth generation: Parallel AI research in Japan," *IEEE Expert*, Feb. 1994, pp. 2–7.
[29] K. A. Hester *et al.*, "The predictive RAAM: A RAAM that can learn to distinguish sequences from a continuous input stream," in *Proc. World Congr. Neural Networks*, San Diego, CA, vol. 4, June 1994 pp. 97–103.
[30] G. E. Hinton, "Implementing semantic networks in parallel hardware," in *Parallel Models of Associative Memory*, G. E. Hinton and J. A. Anderson, Eds. Hillsdale, NJ: Lawrence Erlbaum, 1989.
[31] V. Honavar, "Toward learning systems that integrate different strategies and representations," in *Artificial Intelligence and Neural Networks: Steps Toward Principled Integration*, V. Honavar and L. Uhr, Eds. San Diego, CA: Academic, 1994, pp. 615–644.
[32] ——, "Symbolic artificial intelligence and numeric artificial neural networks: Toward a resolution of the dichotomy," in *Computational Architectures Integrating Symbolic and Neural Processes*, R. Sun and L. Bookman, Eds. Norwell, MA: Kluwer, 1995, pp. 351–388.
[33] V. Honavar and L. Uhr, "Coordination and control structures and processes: Possibilities for connectionist networks," *J. Experimental Theoretical Artificial Intell.*, vol. 2, pp. 277–302, 1990.
[34] ——, Eds., *Artificial Intelligence and Neural Networks: Steps Toward Principled Integration*. San Diego, CA: Academic, 1994.
[35] ——, "Integrating symbol processing systems and connectionist networks," in *Intelligent Hybrid Systems*, S. Goonatilake and S. Khebbal, Eds. London, U.K.: Wiley, 1995, pp. 177–208.
[36] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Reading, MA: Addison-Wesley, 1979.

[37] B. Horne, D. R. Hush, and C. Abdallah, "The state-space recurrent neural network with application to regular grammatical inference," Dept. Electr. Comput. Eng., Univ. New Mexico, Albuquerque, NM, UNM Tech. Rep. EECE 92-002, 1992.
[38] D. B. Howe and K. Asanović, "SPACE: Symbolic processing in associative computing elements," in *VLSI for Neural Networks and Artificial Intelligence*, J. G. Delgado-Frias, Ed. New York: Plenum, 1994, pp. 243–252.
[39] A. N. Jain, A. Waibel, and D. S. Touretzky, "PARSEC: A structured connectionist parsing system for spoken language," in *IEEE Proc. Int. Conf. Acoust., Speech, Signal Processing*, San Francisco, CA, Mar. 1992, pp. 205–208.
[40] S. C. Kleene, "Representation of events in nerve nets and finite automata," in *Automata Studies*, C. E. Shannon and J. McCarthy, Eds. Princeton, NJ: Princeton Univ., 1956, pp. 3–42.
[41] P. Kogge, J. Oldfield, M. Brule, and C. Stormon, "VLSI and rule-based systems," in *VLSI for Artificial Intelligence*, J. G. Delgado-Frias and W. R. Moore, Eds. Norwell, MA: Kluwer, 1989, pp. 95–108.
[42] R. Kumar, "NCMOS: A high performance CMOS logic," *IEEE J. Solid-State Circuits*, vol. 29, pp. 631–633, 1994.
[43] R. C. Lacher and K. D. Nguyen, "Hierarchical architectures for reasoning," in *Computational Architectures Integrating Neural and Symbolic Processes: A Perspective on the State of the Art*, R. Sun and L. Bookman, Eds. Norwell, MA: Kluwer, ch. 4, pp. 117–150, 1995.
[44] P. Langley, *Elements of Machine Learning*. Palo Alto, CA: Morgan Kaufmann, 1995.
[45] S. H. Lavington, C. J. Wang, N. Kasabov, and S. Lin, "Hardware support for data parallelism in production systems," in *VLSI for Neural Networks and Artificial Intelligence*, J. G. Delgado-Frias, Ed. New York: Plenum, 1994, pp. 231–242.
[46] D. Levine and M. Aparicio IV, Eds., *Neural Networks for Knowledge Representation and Inference*. Hillsdale, NJ: Lawrence Erlbaum, 1994.
[47] J. B. Lont and W. Guggenbühl, "Analog CMOS implementation of a multilayer perceptron with nonlinear synapses," *IEEE Trans. Neural Networks*, vol. 3, pp. 457–465, 1992.
[48] F. Lu and H. Samueli, "A 200-MHz CMOS pipelined multiplier-accumulator using a quasidomino dynamic full-adder cell design," *IEEE J. Solid-State Circuits*, vol. 28, pp. 123–132, 1993.
[49] B. J. MacLennan, *Principles of Programming Languages: Design, Evaluation, and Implementation*, 2nd ed. New York: CBS College, 1987.
[50] P. Masa, K. Hoen, and H. Wallinga, "70 input, 20 nanosecond pattern classifier," *IEEE Int. Joint Conf. Neural Networks*, Orlando, FL, vol. 3, 1994.
[51] L. W. Massengill and D. B. Mundie, "An analog neural-network hardware implementation using charge-injection multipliers and neuron-specific gain control, *IEEE Trans. Neural Networks*, vol. 3, pp. 354–362, May 1992.
[52] W. S. McCulloch and W. Pitts, "A logical calculus of ideas immanent in nervous activity," *Bull. Math. Biophys.*, vol. 5, pp. 115–133, 1943.
[53] D. McGregor, S. McInnes, and M. Henning, "An architecture for associative processing of large knowledge bases (LKB's)," *Computer J.*, vol. 30, no. 5, pp. 404–412, Oct. 1987.
[54] L. Miclet, *Structural Methods in Pattern Recognition*. New York: Springer-Verlag, 1986.
[55] B. Miikkulainen, "Subsymbolic parsing of embedded structures," in *Computational Architectures Integrating Neural and Symbolic Processes: A Perspective on the State of the Art*, R. Sun and L. Bookman, Eds. Norwell, MA: Kluwer., ch. 5, pp. 153–186, 1995.
[56] M. Minsky, *Computation: Finite and Infinite Machines*. Englewood Cliffs, NJ: Prentice-Hall, 1967.
[57] M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA: MIT Press, 1969.
[58] E. Mjolsness, "Connectionist grammars for high-level vision," in *Artificial Intelligence and Neural Networks: Steps Toward Principled Integration*, V. Honavar and L. Uhr, Eds. San Diego, CA: Academic, 1994, pp. 423–451.
[59] G. Moon *et al.*, "VLSI implementation of synaptic weighting and summing in pulse coded neural-type cells," *IEEE Trans. Neural Networks*, vol. 3, pp. 394–403, May 1992.
[60] M. C. Mozer and J. Bachrach, "Discovering the structure of a reactive environment by exploration," *Neural Comput.*, vol. 2, no. 4, p. 447, 1990.
[61] M. C. Mozer and S. Das, "A connectionist symbol manipulator that discovers the structure of context-free languages," *Advances in Neural Inform. Processing Syst*. San Mateo, CA: Morgan Kaufmann, vol. 5, p. 863, 1993.
[62] J. Naganuma, T. Ogura, S. I. Yamada, and T. Kimura, "High-speed CAM-based architecture for a prolog machine (ASCA)," *IEEE Trans. Comput.*, vol. 37, pp. 1375–1383, Nov. 1988.
[63] Y. H. Ng, R. J. Glover, and C. L. Chng, "Unify with active memory," in *VLSI for Artificial Intell.*, J. G. Delgado-Frias and W. R. Moore, Eds. Norwell, MA: Kluwer, 1989, pp. 109–118.
[64] I. Noda and M. Nagao, "A learning method for recurrent neural networks based on minimization of finite automata," in *Proc. Int. Joint Conf. Neural Networks*. Piscataway, NJ: IEEE Press, vol. 1, 1992, pp. 27–32.
[65] D. A. Norman, "Reflections on cognition and parallel distributed processing," in *Parallel Distributed Processing*, J. McClelland, D. Rumelhart, and the PDP Research Group, Eds. Cambridge, MA: MIT Press, 1986.
[66] C. Omlin and C. L. Giles, "Extraction and insertion of symbolic information in recurrent neural networks," in *Artificial Intelligence and Neural Networks: Steps Toward Principled Integration*, V. Honavar and L. Uhr, Eds. San Diego, CA: Academic, 1994, pp. 271–299.
[67] ———, "Stable encoding of large finite-state automata in recurrent neural networks with sigmoid discriminants," *Neural Computat.*, vol. 8, no. 4, pp. 675–696, May 1996.
[68] R. G. Parekh and V. Honavar, "Learning regular languages from simple examples," Dept. Comput. Sci., Iowa State Univ., Tech. Rep. ISU-CS-TR 97-06, 1997.
[69] R. G. Parekh, C. Nichitiu, and V. Honavar, "A polynomial time incremental algorithm for regular grammar inference," Dept. Comput. Sci., Iowa State Univ., Tech. Rep. ISU-CS-TR 97-03, 1997.
[70] R. G. Parekh and V. Honavar, "Automata induction, grammar inference, and language acquisition," in *Handbook of Natural Language Processing*, H. Moisl, R. Dale, and H. Somers, Eds. New York: Marcel Dekker, 1998.
[71] B. Péter, *Recursive Functions in Computer Theory*. New York: Halsted, 1981.
[72] G. Pinkas, "A fault-tolerant connectionist architecture for construction of logic proofs, in *Artificial Intelligence and Neural Networks: Steps Toward Principled Integration*, V. Honavar and L. Uhr, Eds. San Diego, CA: Academic, 1994, pp. 321–340.
[73] J. B. Pollack, *On Connectionist Models of Language Processing*, Ph.D. dissertation, Comput. Sci. Dept., Univ. Illinois, Urbana-Champaign, 1987.
[74] ———, "Recursive distributed representations," *Artificial Intell.*, vol. 46, pp. 77–105, 1990.
[75] I. Popescu, "Hierarchical neural networks for rules control in knowledge-based expert systems," *Neural, Parallel Sci. Comput.*, vol. 3, pp. 379–392, 1995.
[76] B. D. Ripley, *Pattern Recognition and Neural Networks*. New York: Cambridge Univ., 1996.
[77] I. Robinson, "The pattern addressable memory: Hardware for associative processing," in *VLSI for Artificial Intelligence*, J. G. Delgado-Frias and W. R. Moore, Eds. Norwell, MA: Kluwer Academic Publishers, 1989, pp. 119–129.
[78] M. E. Robinson *et al.*, "A modular CMOS design of a Hamming network," *IEEE Trans. Neural Networks*, vol. 3, pp. 444–456, 1992.
[79] D. Rodohan and R. Glover, "A distributed parallel associative processor (DPAP) for the execution of logic programs," in *VLSI for Neural Networks and Artificial Intelligence*, J. G. Delgado-Frias, Ed. New York: Plenum, 1994, pp. 265–273.
[80] H. Rogers, Jr., *Theory of Recursive Functions and Effective Computability*. Cambridge, MA: MIT Press, 1987.
[81] D. E. Rumelhart, J. L. McClelland, and the PDP Research Group, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*. Cambridge, MA: MIT Press, 1986.
[82] A. Sanfeliu and R. Alquezar, "Understanding neural networks for grammatical inference and recognition," in *Advances in Structural and Syntactic Pattern Recognition*, H. Bunke, Ed. Singapore: World, 1992.
[83] W. Schneider, "Connectionism: Is it a paradigm shift for psychology?," *Behavior Res. Methods, Instruments, Comput.*, vol. 19, pp. 73–83, 1987.
[84] D. Schulenburg, "Sentence processing with realistic feedback, in *IEEE/INNS Int. Joint Conf. Neural Networks*, Baltimore, MD, vol. IV, pp. 661–666, 1992.
[85] B. Selman and G. Hirst, "A rule-based connectionist parsing system," in *Proc. 7th Annu. Conf. Cognitive Sci. Soc.*, Irvine, CA, 1985.
[86] C. H. Séquin and R. D. Clay, "Fault tolerance in artificial neural networks," in *Proc. IJCNN*, San Diego, CA, vol. 1, pp. 703–708, 1990.
[87] D. Servan-Schreiber, A. Cleeremans, and J. L. McClelland, "Graded state machines: The representation of temporal contingencies in simple recurrent neural networks," in *Artificial Intell. Neural Networks: Steps Toward Principled Integration*, V. Honavar and L. Uhr, Eds. San Diego, CA: Academic., 1994, pp. 241–269.
[88] L. Shastri and V. Ajjanagadde, "Connectionist system for rule based reasoning with multiplace predicates and variables," Comput. Inform.

Sci. Dept., Univ. Pennsylvania, Philadelphia, PA, Tech. Rep. MS-CIS-8906, 1989.
[89] J. W. Shavlik, "A framework for combining symbolic and neural learning," in *Artificial Intelligence and Neural Networks: Steps Toward Principled Integration*, V. Honavar and L. Uhr, Eds. San Diego, CA: Academic, 1994, pp. 561–580.
[90] H. T. Siegelman and E. D. Sontag, "Turing-computability with neural nets," *Appl. Math. Lett.*, vol. 4, no. 6, pp. 77–80, 1991.
[91] S. Sippu and E. Soisalon-Soininen, "Parsing theory, vol. II: LR(k) nad LL(k) parsing." Berlin, Germany: Springer-Verlag, 1990.
[92] B. Souček and the IRIS Group, Eds., "Neural and intelligent systems integrations: Fifth and sixth generation integrated reasoning information systems." New York: Wiley, 1991.
[93] G. Z. Sun, C. L. Giles, H. H. Chen, and Y. C., The Neural Network Pushdown Automation: Model, Stack and Learning Simulations, UMIA CS-TR-93-77, Univ. Maryland, College Park, MD, Aug. 1993.
[94] R. Sun, "Logics and variables in connectionist models: A brief overview," in *Artificial Intelligence and Neural Networks: Steps Toward Principled Integration*, V. Honavar, and I. Uhr, Eds. San Diego, CA: Academic, 1994, pp. 301–320.
[95] R. Sun and L. Bookman, Eds., *Computational Architectures Integrating Symbolic and Neural Processes*. Norwell, MA: Kluwer, 1995.
[96] G. Swaminathan, S. Srinivasan, S. Mitra, J. Minnix, B. Johnson, and R. Iñigo, "Fault tolerance of neural networks," *Proc. IJCNN*, Washington, D.C., vol. 2, pp. 699–702, 1990.
[97] K. J. Thurber and L. D. Wald, "Associative and parallel processors," *Comput. Surveys*, vol. 7, no. 4, pp. 215–255, 1975.
[98] K. Uchimura *et al.*, "An 8G connection-per-second 54 mW digital neural network with low-power chain-reaction architecture, *ISSCC Dig. Tech. Papers*, San Francisco, CA, 1992, pp. 134–135.
[99] L. Uhr and V. Honavar, "Artificial intelligence and neural networks: Steps toward principled integration," in *Artificial Intelligence and Neural Networks: Steps Toward Principled Integration*, V. Honavar and L. Uhr, Eds. San Diego, CA: Academic, 1994, pp. xvii–xxxii.
[100] F. Van der Velde, "Symbol manipulation with neural networks: Production of a context-free language using a modifiable working memory," *Connection Sci.*, vol. 7, no. 3/4, pp. 247–280, 1995.
[101] T. Watanabe *et al.*, "A single 1.5-V digital chip for a 10^6 synapse neural network," *IEEE Trans. Neural Networks*, vol. 4, pp. 387–393, May 1993.
[102] R. L. Watrous and G. M. Kuhn, "Induction of finite-state languages using second-order recurrent neural networks," *Neural Comput.*, vol. 4, no. 3, p. 406, 1992.
[103] R. J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural Comput.*, vol. 1, pp. 270–280, 1989.
[104] D. Wood, *Theory of Computation*. New York: Wiley, 1987.
[105] Z. Zeng, R. M. Goodman, and P. Smyth, "Discrete recurrent neural networks for grammatical inference," *IEEE Trans. Neural Networks*, vol. 5, pp. 320–330, Mar. 1994.

Chun-Hsien Chen received the B.E. degree in chemical engineering from National Tsing-Hua University, Taiwan, and the M.S. and Ph.D. degrees in computer science from Iowa State University, Ames.

He is currently with the Institute for Information Technology, Taiwan. His research interests include artificial intelligence and neural networks. His thesis research emphasizes neural architectures for knowledge representation and inference.

Vasant Honavar (M'98) received the B.E. degree in electronics engineering from Bangalore University, India, the M.S. degree in electrical and computer engineering from Drexel University, Philadelphia, PA, and the M.S. and Ph.D. degree in computer science from the University of Wisconsin, Madison.

He directs the Artificial Intelligence Research Laboratory in the Department of Computer Science at Iowa State University where he is currently an Associate Professor. His research and teaching interests include Artificial Intelligence, Artificial Neural Networks, Machine Learning, Adaptive Systems, Bioinformatics and Computational Biology. Evolutionary Computing, Grammatical Inference, Intelligent Agents and Multiagent systems, Neural and Cognitive Modeling, Distributed Artificial Intelligence, Data Mining and Knowledge Discovery, Evolutionary Robotics, Parallel and Distributed Artificial Intelligence, Knowledge-Based Systems, Distributed Knowledge Networks, and Applied Artificial Intelligence. He has published more than 80 research papers in refereed journals, books, and conferences and has coedited three books.

He is a coeditor-in-chief of the *Journal of Cognitive Systems Research*. His research has been partially funded by grants from the National Science Foundation, the John Deere Foundation, the National Security Agency, and IBM. Prof. Honavar is a member of ACM, AAAI, and the New York Academy of Sciences.