# A Neural Network Architecture for Syntax Analysis

Chun-Hsien Chen & Vasant Honavar
Artificial Intelligence Research Group
Department of Computer Science
Iowa State University
Ames, Iowa 50011-1040
U.S.A
chen@cs.iastate.edu, honavar@cs.iastate.edu

ISU CS-TR 95-18, August 1995

# A Neural Network Architecture for Syntax Analysis

Chun-Hsien Chen & Vasant Honavar *
Artificial Intelligence Research Group
Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, IA 50011. U.S.A.
chen@cs.iastate.edu, honavar@cs.iastate.edu

August 9, 1995

**Abstract**

Artificial neural networks (ANNs), due to their inherent parallelism and potential fault tolerance, offer an attractive paradigm for robust and efficient implementations of syntax analyzers. This paper proposes a modular neural network architecture for syntax analysis on continuous input stream of characters. The components of the proposed architecture include neural network designs for a stack, a lexical analyzer, a grammar parser and a parse tree construction module. The proposed NN stack allows simulation of a stack of large depth, needs no training, and hence is not application-specific. The proposed NN lexical analyzer provides a relatively efficient and high performance alternative to current computer systems for lexical analysis especially in natural language processing applications. The proposed NN parser

*Vasant Honavar's research is partially supported by the National Science Foundation through the grant IRI-9409580. The authors are grateful to Dr. Lee Giles of NEC Research Institute, Princeton, for helpful suggestions on an earlier draft of this paper.

generates parse trees by parsing strings from widely used subsets of deterministic context-free languages (generated by LR grammars). The estimated performance of the proposed neural network architecture (based on current CMOS VLSI technology) for syntax analysis is compared with that of commonly used approaches to syntax analysis in current computer systems. The results of this performance comparison suggest that the proposed neural network architecture offers an attractive approach for syntax analysis in a wide range of practical applications such as programming language compilation and natural language processing.

# 1 Introduction

Artificial neural networks (ANNs) offer an attractive computational model for a variety of applications in computer science and engineering, artificial intelligence, and cognitive modeling for a variety of reasons including their inherent parallelism and potential for fault tolerance. Despite the success in the application of ANN to a broad range of tasks in pattern classification, control, function approximation, and system identification, their use in symbolic computing tasks (e.g., storage and retrieval of records in large databases and knowledge bases, language processing, etc.) is only beginning to be explored [Goonatilake and Khebbal, 1995; Honavar, 1994; Honavar and Uhr, 1994, 1995; Levine and Aparicio, 1994; Sun and Bookman, 1995; Uhr and Honavar, 1994].

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 [McCulloch and Pitts, 1943; Kleene, 1956, Minksy, 1967]. 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 [Allen, 1990; Chen and Honavar, 1994; Elman, 1990; Frasconi et al., 1993; Giles et al., 1992; Horne, Hush, and Abdallah, 1992; Mozer, 1990; Noda and Nagao, 1992; Omlin and Giles, 1994; Sanfeliu and Alquezar, 1992; Servan-Schreiber, Cleeremans and McClelland, 1994; Waltraus, 1992].

There has been considerable work on extending the computational capa-

bilities of recurrent neural network models by providing some form of external memory in the form of a tape [Williams and Zipser, 1989] or a stack [Berg, 1992; Fanty, 1986; Pollack, 1990; Das, Giles, and Sun, 1993; Giles, Horne and Lin, 1995; Hester, 1994; Miikkulainen, 1995; Schulenberg, 1992; Siegelman, 1991; Sun et al., 1993; Zeng et al., 1994].

To the best of our knowledge, to date, most of the research on neural architectures for syntax analysis — with the exception of [Siegelman, 1991; Pollack, 1987] which explore connectionist Turing machine models which simulate a stack using binary representations of a fractional number; and [Chen and Honavar, 1994; Omlin and Giles, 1995] which focus on neural network realizations of finite state automata (with the latter emphasizing fault tolerance aspects) — focus 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 or natural language sentences constructed using limited vocabulary). While language learning is an important research problem in its own right and deserves further investigation using a variety of approaches, the focus of this paper is on neural architectures for syntax analysis for languages with known grammars — a task that finds applications in natural language processing, program compilation, structural pattern recognition, analysis of mathematical expressions, analysis of logical expressions for automated reasoning, and connectionist expert systems.

The proposed modular neural architecture for syntax analysis consists of several components which include neural networks for stack, lexical analyzer, grammar parser and parse tree construction module. For simplicity, syntax error handling is not considered and it is assumed that the input stream consists of *ASCII*-coded characters. The rest of the paper is organized as follows:

Sections 2, 3, and 4 (respectively) propose modular neural network architectures for stack, lexical analysis, and parsing. Section 5 compares the performance of the proposed neural architecture for syntax analysis with that of conventional software implementations on current computer systems. Section 6 concludes with a summary.

# 2 Neural Network Design for a Stack (NN Stack)

`Pop` and `push` are the main actions of a stack whose mechanism can be simulated by a *dfa* (deterministic finite automaton) with memory to store stack symbols accessed sequentially by *stack top pointer* (SP) pointing to the top symbol of a stack. Pointer to the top of the stack can be simulated by current state of the *dfa*, and the current action of a stack by the input of the *dfa*.

## 2.1 Architecture of NN Stack

A design for an NN stack obtained by adding a *write control module* to the *NN DFA* (neural network for deterministic finite automata) [Chen and Honavar, 1994] is shown in Figure 1. (The addition of such a control module might at first glance appear to violate the commonly held view on neural networks. However, the operation of most existing neural networks implicitly assume at least some form of control and there is no reason not to build in a variety of control and co-ordination structures into neural networks [Honavar and Uhr, 1990]). The NN DFA, which can simulate the operation of any given DFA, is a partially recurrent neural network which essentially consists of a 3-layer Perceptron (with a single hidden layer) and recurrent links from its output neurons to some of the input neurons. The Perceptron is constructed using one-shot learning to realize the binary mapping function for a given set of ordered pairs of input/output binary vectors. Such a binary mapping function encodes the transition function of a DFA.

The NN stack 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 *NN* stack with the discrete time line, denoted by *0,1,...,t,t+1,....*
- *Action* code which is a 2-bit binary code so that
  - *01* denotes `push`.
  - *10* denotes `pop`.

– *00* denotes *no action.*

- *Stack symbol* which is an $n$-bit binary code for the symbol to be pushed into or popped off a stack during a stack operation.

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

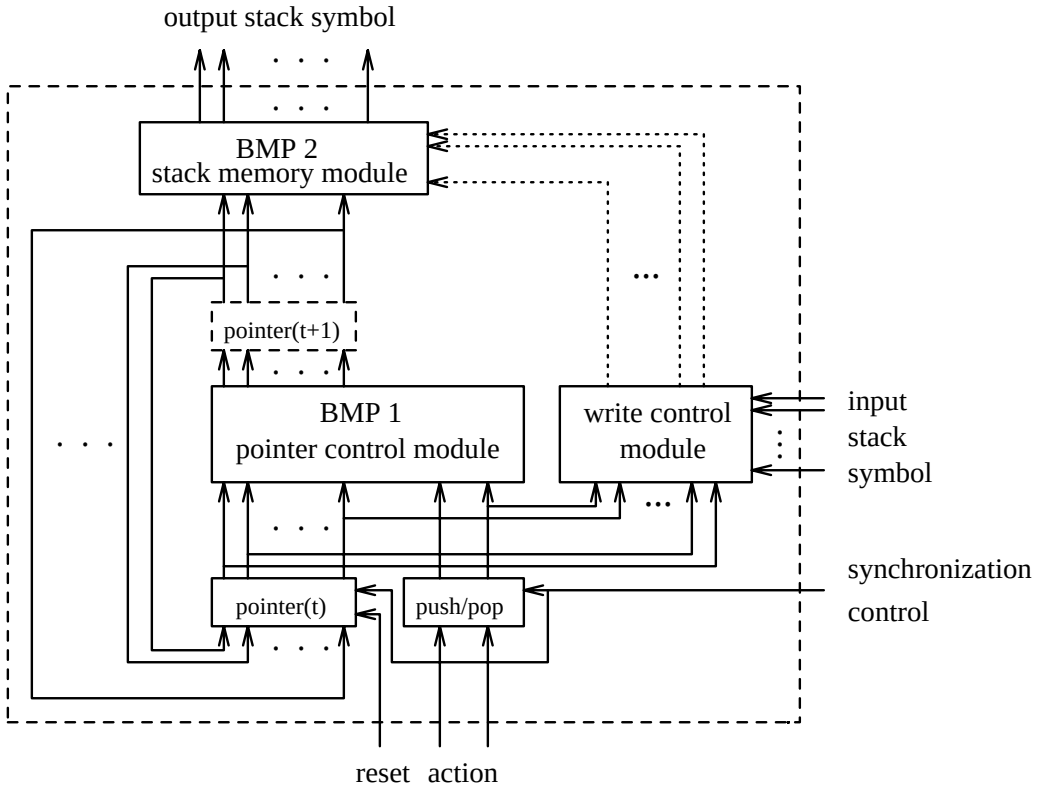


Figure 1. A neural network architecture for stack mechanism. The dotted box labeled with *pointer*(t+1) exists only logically but not physically. A `push` stack action enables the *write control module* to write stack symbol into the stack memory module.

The pointer control module (*BMP 1*) controls the movement of SP which is incremented on a `push` and decremented on a `pop`. BMP (Binary Map-

ping Perceptron) module is a multi-layer Perceptron with 2 layers of connections and allows automated random binary mapping (associations between arbitrary binary or bipolar patterns) with one-shot learning (see [Chen and Honavar, 1994] for details). The pointer control module uses $m + 2$ input, $3 \times 2^m$ hidden, and $m$ output neurons. $m$ of the input neurons together encode $2^m$ possible values of SP and the other 2 input neurons together encode the stack action. The $m$ output neurons represent next value of SP (*pointer*(t+1)) after a stack action. Every increment or decrement transition of SP can be viewed as a binary mapping which takes one hidden neuron to implement in a BMP module. Since "`no action`" is also reserved as one of legal stack actions, $3 \times 2^m$ hidden neurons are used in the pointer control module.

The stack memory module (BMP 2) uses $m$ input neurons, $n$ output neurons, and $2^m$ hidden neurons which together allow storing $2^m$ 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 module 2 uses its 2nd-layer connections associated with a hidden neuron to store a stack symbol [Chen and Honavar, 1994; 1995a].

The write control module has two sets of binary inputs. One has $m + 1$ input signals, $m$ of which are from current SP (*pointer*(t)) and 1 of which is from the second output line of `push/pop` buffer. The other is an $n$-bit binary input encoding the stack symbol to be pushed into the stack. The $n$ dotted output lines from the write control module writes the $n$-bit binary-coded stack symbol into $n$ of the 2nd-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 connections are located by using current SP value (*pointer*(t)). (The processing of stack *overflow* and *underflow* is not discussed here and is left to implementation of error handling mechanisms).

Note that the proposed design for NN stack shown in Figure 1 is based on an assumption that the write control module finishes updating the 2nd-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 to output stack symbol; 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 to output stack

symbol 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 passing between these two modules at a `push` operation.

## 2.2 Examples of NN Stack Operations

The following example illustrates the operation of the NN stack by considering several successive stack operations. Let $m = 6$ and stack symbols be encoded into 8-bit ASCII codes. Then there are 64 possible SP values and $n = 8$. Let $\dot{w}^1_{ji}$ and $\dot{w}^2_{kj}$ respectively denote the 1st-layer connection weight from input neuron $i$ to hidden neuron $j$ and the 2nd-layer connection weight from hidden neuron $j$ to output neuron $k$ in the pointer control module; then $1 \leq i \leq 8$, $1 \leq j \leq 192$ and $1 \leq k \leq 6$. Also let $\ddot{w}^1_{qp}$ and $\ddot{w}^2_{rq}$ respectively denote the 1st-layer connection weight from input neuron $p$ to hidden neuron $q$ and the 2nd-layer connection weight from hidden neuron $q$ to output neuron $r$ in the stack memory module; then $1 \leq p \leq 6$, $1 \leq q \leq 64$ and $1 \leq r \leq 8$.

1. **At time =** $t_1$, Suppose SP $= <0,0,0,1,0,0>$ and the stack action to be performed is a `push` on the stack symbol $A = 41_{16} = 0100\ 0001_2$. Let $\Phi = \phi_1\phi_2\phi_3\phi_4\ \phi_5\phi_6\phi_7\phi_8$ be an 8-bit binary code denoting the current stack symbol to be pushed. Then $\phi_2 = \phi_8 = 1$ and other $\phi_i$'s are 0. Before the execution of the `push` operation (encoded as $< 0,1 >$), $pointer(t) =< 0,0,0,1,0,0 >= 4$; and after the execution of `push`, $pointer(t+1) =< 0,0,0,1,0,1 >= 5$. The mapping from $< 0,0,0,1,0,0,0,1 >$ (the 2 rightmost bits of which together denote a stack action `push` and the 6 leftmost bits of which denote an SP value 4) to $< 0,0,0,1,0,1 >$ is done by the $13th$ ($13 = 1 + 3 \times$ *SP value at time* $t_1$) hidden neuron and its associated connections in the pointer control module. The write control module uses binary input $< 1,0,0,0,1,0,0 >$ (the leftmost bit of which denotes a stack action `push` and the 6 rightmost bits of which denote an SP value 4) to locate the $6th$ ($6 = 2 +$ *SP value at time* $t_1$) hidden neuron of the stack memory module and to update the weights of the eight 2nd-layer connections associated with the $6th$ hidden neuron according to expression $\ddot{w}^2_{r6} = \phi_r,\ 1 \leq r \leq 8$. When $pointer(t+1) =< 0,0,0,1,0,1 >= 5$ is passed to the stack memory module, its $6th$ ($6 = 1 + pointer(t+1)$) hidden neuron is turned on to recall the stack symbol $A = 0100\ 0001_2$ which

is stored by the 2nd-layer connections associated with the $6th$ hidden neuron. Note that in the stack memory module the first hidden neuron and its associated 2nd-layer connections are used to store the *stack start symbol* (stack bottom) which is pointed by SP $= <0,0,0,0,0,0>$.

2. **At time** $= t_1 + 1$, SP $= <0,0,1,0,0,1>$ and the stack action to be performed is a `push` on the stack symbol $B = 42_{16} = 0100\ 0010_2$. Then $\phi_2 = \phi_7 = 1$ and other $\phi_i$'s are 0. Before the execution of the `push` operation, $pointer(t) =< 0,0,0,1,0,1 >= 5$; and $pointer(t+1) =< 0,0,0,1,1,0 >= 6$ after the execution of `push`. The mapping from $<0,0,0,1,0,1,0,1>$ to $<0,0,0,1,1,0>$ is done by the $16th$ ($16 = 1 + 3 \times$ *SP value at time* $t_1$*+1*) hidden neuron and its associated connections in the pointer control module. The write control module uses binary input $< 1,0,0,0,1,0,1 >$ (the leftmost bit of which denotes a stack action `push` and the 6 rightmost bits of which denote an SP value 5) to locate the $7th$ ($7 = 2 +$ *SP value at time* $t_1$*+1*) hidden neuron in the stack memory module and to update the weights of the 2nd-layer connections associated with the $7th$ hidden neuron using current $\Phi$ as before. When $pointer(t+1) =< 0,0,0,1,1,0 >= 6$ is passed to the stack memory module, its $7th$ ($7 = 1 + pointer(t+1)$) hidden neuron is turned on to recall the stack symbol $B = 0100\ 0010_2$ which is stored by the 2nd-layer connections associated with the $7th$ hidden neuron.

3. **At time** $= t_1 + 2$, SP $= <0,0,0,1,1,0>$ and the stack action to be performed is a `pop`. Before the execution of the `pop` operation (encoded as $<1,0>$), $pointer(t) =< 0,0,0,1,1,0 >= 6$; and after the execution of the `pop` operation, $pointer(t+1) =< 0,0,0,1,0,1 >= 5$. When $pointer(t+1) =< 0,0,0,1,0,1 >= 5$ is passed to the stack memory module, its $6th$ hidden neuron is turned on to recall the stack symbol $0100\ 0001_2 = A$ which is stored by the 2nd-layer connections associated with the $6th$ hidden neuron.

## 2.3 Comparison with Other Neural Network Models Using Stacks

RAAM (Recursive Auto-Associative Memory) [Pollack, 1990] is a relatively popular neural network model which has been used by several researchers

to implement stacks in connectionist designs for parsers [Berg, 1992; Hester et al., 1994; Miikkulainen, 1995]. A RAAM is a 3-layer Perceptron with recurrent links from hidden layer to part of input layer and from part of output layer to hidden layer. It is known that the deeper the stack structure encoded by a RAAM is, the greater is the degradation in accuracy of decoding [Miikkulainen, 1995]. Other problems associated with the use of RAAM as stacks are:

- **application dependency**: Every different application needs to train a new RAAM to fit its purpose since the encoding used by a RAAM is application-specific.
- **local minima**: The RAAM is trained using some variant of generalized delta rule — a gradient-descent algorithm that is susceptible to local minima which might interfere with learning of the desired input-output mapping.
- **training overhead** : Gradient descent algorithms such as generalized delta rule can be excruciatingly slow and their performance is very sensitive to the choice of parameters such as the learning rate.
- **design overhead** : The number of hidden neurons needed to realize the desired stack depth given a set of stack symbols is generally decided through a process of trial and error.

Several other researchers have made use of a hybrid system consisting of a recurrent neural network and a stack [Giles et al., 1990; Sun et al., 1993; Das, Giles, and Sun, 1992; Mozer and Das, 1993]. However, the primary focus of this work was on learning of context-free languages, typically using some variant of inductive learning using gradient descent on a suitably defined differentiable objective function and not on the design of a neural network architecture for parsing a known language. Since the proposed model does not involve learning, it circumvents the drawbacks of RAAM and related stack models — of course, at the expense of requiring that the langugage be known a-priori. This requirement is naturally met in many syntax analysis applications (e.g., compilers, knowledge-based systems, etc.)

# 3 Neural Network Design for a Lexical Analyzer (NN Lexical Analyzer)

Conventionally, a lexical analyer is simulated by a *dfa* which can be realized quite simply using an NN DFA [Chen and Honavar, 1994]. One drawback with such a simulation, especially for natural language processing, is that all legal transitions have to be exhaustively specified in the *dfa*. For example, Figure 2 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*.

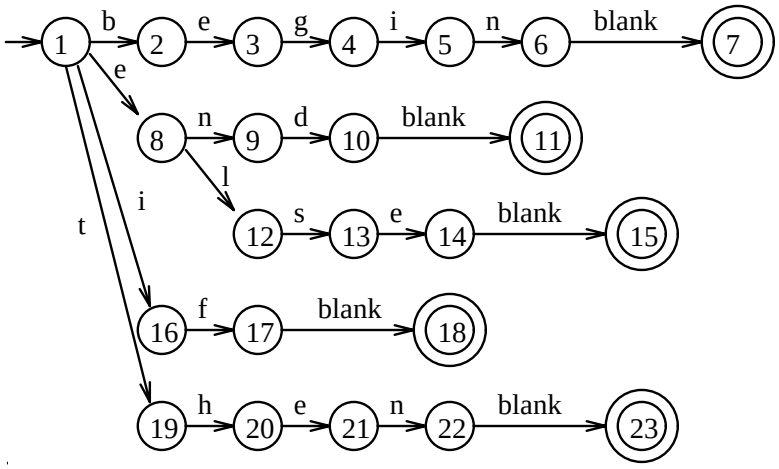


Figure 2. The simplified state diagram of a *dfa* which recognizes keywords : *begin, end, if, then*, and *else*.

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*. In all, there are 20 unspecified legal transitions for such a simple lexical analyzer with 22 transitions explicitly defined. The realization of such a simple 5-word (23-state) lexical analyzer by an NN DFA is wasteful since the NN DFA needs to use 42 (= 20 + 22) hidden neurons (every transition takes one hidden neuron to realize). Furthermore, more transitions have to be defined in order to allow multiple blanks between two consecutive words in the input stream,

and error handling requires the definition of additional states and transitions. These drawbacks are further exacerbated in natural language processing due to the large vocabulary involved.

A better alternative is to use a Dictionary (or a database) to serve as a lexicon. The proposed NN lexical analyzer consists of a *word segmenter* for carving an input stream into words, and a *word lookup table* for looking up the words in a dictionary or lexicon. Looking a word up in a dictionary 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 [Chen and Honavar, 1995b].

## 3.1 Neural Network Design for a Word Segmenter (NN Word Segmenter)

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. In natural language processing, the main function of a word segmenter is to identify illegal words and to group input stream into legal words which may need to be further categorized. Figure 3 shows the state diagram of a *dfa* simulating a simple word segmenter which carves continuous input stream 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. After the word segmenter encounters end-of-input, it terminates processing. Every time when the word segmenter goes into accepting state, it instructs the word lookup table to look up a word extracted from the input stream and stored in a buffer. 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 [Chen and Honavar, 1994]. It is suggested that the garbage state (state G in Figure 3) be encoded into a binary code of all zeros since any undefined (un-implemented) transition moves into a binary-coded state of all zeros automatically in an NN DFA. Every transition takes one hidden neuron to implement in an NN DFA according to [Chen and Honavar, 1994],

but that is not necessary. In Figure 3 the 10 transitions from state 4 on input symbols 0,1,...,9 can be realized by only two hidden neurons in an NN DFA using *don't-care technique* or *technique of partial pattern recognition* [Chen and Honavar, 1995a, 1995b]. That is similar for other transitions on input symbols 1,2,...,9, a,b,...,z, and A,B,...,Z.

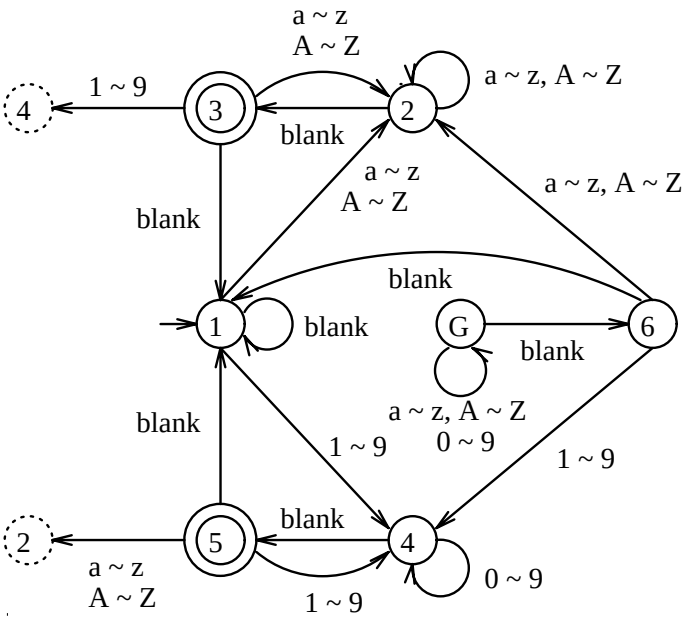


Figure 3. The state diagram of a *dfa* which simulates a simple word segmenter carving continuous input stream of characters into words including integer constants, keywords and identifiers. Both the keywords and identifiers are strings of English characters. For simplicity, the handling of *end-of-input* is not shown in the figure. After the word segmenter encounters *end-of-input*, it stops. 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 real-line circle and that is same for state 4 (it is drawn in this way to avoid clutter).

## 3.2 Neural Network Design for a Word Lookup Table (NN Word Lookup Table)

For the purpose of grammar parsing which follows lexical analysis, in the lexical analysis of programming languages an extracted word is translated

into a *token*. Each such token is treated as a single logical entity, which may be an identifier, a keyword, a constant, an operator or a punctuation symbol. In the lexical analysis of natural languages, a word is translated into a token having two sub-parts: category code encoding grammatical category of a word (e.g., *noun*, *determiner*, etc.) and feature code encoding syntactic feature of a word (e.g., singular, masculine, etc.) [Gazdar, 1989; Sanamrad, 1987].

Such a word translation can be viewed in terms of *binary mapping* which can be realized by a $BMP$ module [Chen and Honavar 1994]. The number of input and output neurons used in a BMP module depends on the problem to be dealt with. Such a translation (dictionary lookup) is conveniently performed in the form of a simple database query using key-based organizations (with the segmented word being used as the key). This can be efficiently realized using neural associative memory designs for database query processing [see Chen and Honavar, 1995b for details]. The time required for processing such a query is of the order of 20 ns (at best) to 100 nanoseconds (at worst) given the current CMOS technology for implementation of artificial neural networks. (The interested reader is referred to section 5.1 and [Chen and Honavar, 1995b] for details).

# 4 A Modular Neural Architecture for LR Parser (NN LR Parser)

LR($k$) grammars generate the so-called deterministic context-free languages which can be accepted by deterministic push-down automata [Hopcroft and Ullman, 1979]. Such grammars find extensive applications in programming languages and compilers. LR parsing which parses LR grammars in linear time is a widely used method for syntax analysis, e.g. syntax analysis of programming languages [Aho, Sethi, and Ullman, 1986; Chapman, 1987; Sippu and Soisalon-Soininen, 1990]. An architecture of modular neural networks (Figure 4) for parsing LR(1) grammars is proposed in this section. 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, where k $\geq$ 1, can be transformed into an LR(1) grammar [Sippu and Soisalon-Soininen, 1990]; LR parsers with k = 1 are sufficient for practical

purposes [Hopcroft and Ullman, 1979]. In the following section, the construction of parse tree as the result of parsing is also addressed, but syntax error handling is not considered.

## 4.1 Representation of Parsing Table

Logically, an LR parser consists of two parts: a driver routine which is the same for all LR parsers and a parsing table which changes from one parser to another [Aho, Sethi, and Ullman, 1986]. LR parsing algorithm pre-compiles an LR grammar into a parsing table which guides deterministically the parsing on input string of lexical tokens by `shift/reduce` moves [Aho, Sethi, and Ullman, 1986; Chapman, 1987]. Such a parsing mechanism can be simulated by a *dpda* (deterministic pushdown automata) with $\epsilon$-moves [Hopcroft and Ullman, 1979]. An $\epsilon$-move does not consume the input symbol, and the input head is not advanced after the move. This type of move facilitates a DPDA to manipulate a stack without reading input symbols. Combined with the NN stack proposed in section 3.1, the neural network architecture for *dpda* (NN *DPDA*) proposed in [Chen and Honavar, 1994] is able to parse *DCFL* (deterministic context-free languages). However, the NN *DPDA* architecture proposed in [Chen and Honavar, 1994] cannot efficiently handle $\epsilon$-moves due to the requirement of checking the possibility of an $\epsilon$-move at every state, a modified design for LR(1) parsing is discussed below. A parsing table and a stack are two main components of an LR(1) parser. The next move of the parsing automaton is determined by the current input symbol $a$ and the state $s$ that is on top of the stack. It is given by the parsing table entry corresponding to $[s, a]$. Each such 2D table entry ACTION$[s, a]$ in the parsing table is implemented as a 6-tuple binary code $< action, state, rule, length, lhs, status >$ in the NN LR(1) parser, 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 to go to*";
- *rule* is an $R$-bit binary number denoting *the grammar production rule r to be reduced* 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 $r$ to be used 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 $r$ to be used if the consulted *action* is a `reduce` ($H$ is assumed to be large enough to encode all symbols of the grammar into $H$-bit binary codes); and
- *status* is a 2-bit binary code denoting one of three possible *parsing statuses*, 00: *error*, 01: *in progress*, or 10: *accept* (used for a higher-level control to acknowledge the success or failure of a parsing).

Note that the order of the tuple's elements arranged in Figure 4 is different from above.

## 4.2 Representation of Parsing Configuration and Parse Tree

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 $(s_0 s_1 \cdots s_i,\ a_j a_{j+1} \cdots a_n\$)$, where $s_i$ is the state on top of the stack, $s_0$ is the stack start symbol, $a_j$ is current input symbol, and \$ is a special symbol denoting "*end of input*". Let $0^k$ be a $k$-bit binary number of all zeros and denote a value of *don't care* for $k \geq 1$. In the proposed NN LR(1) parser, the configurations resulting from one of four types of *move*s on parsing an input lexical token are as follows:

- If ACTION$[s_i, a_j] =< 01, s, 0^R, 0^L, 0^H, 01 >$, the parser performs a `shift` move and enters the configuration $(s_0 s_1 \cdots s_i s,\ a_{j+1} \cdots a_n\$)$. Such a `shift` move can be realized in one cycle in the proposed NN parser.
- If ACTION$[s_i, a_j] =< 10, 0^S, r, l, h, 01 >$, the parser performs a `reduce` by producing a binary number $r$ (denoting certain 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 non-empty grammar symbols in $\beta$) as part of the parse tree, popping the $l$ top symbols off the stack, consulting parsing table entry $[s_{i-l}, h]$ and entering the configuration $(s_0 s_1 \cdots s_{i-l} s,\ a_{j+1} \cdots a_n\$)$ where ACTION$[s_{i-l}, h] =< 01, s, 0^R, 0^L, 0^H, 01 >$. Such a `reduce` move is realized in two cycles in the proposed NN parser since the parsing table is consulted twice

for simulating the move. In the simulation of a `reduce` move, the first access of the parsing table and the execution of the consulted action (`reduce`) are realized in the first cycle; and the second access of the parsing table due to a `reduce` action and the execution of the consulted action (`shift`) are realized in the second cycle.

- If ACTION$[s_i, a_j] =< 0^2, 0^S, 0^R, 0^L, 0^H, 10 >$, parsing is completed.
- If ACTION$[s_i, a_j] =< 0^2, 0^S, 0^R, 0^L, 0^H, 00 >$, an error is discovered and the parser stops. Note that such an entry is a binary code of all zeros.

Since an LR parser scans input string from left to right and performs bottom-up parsing which constructs rightmost derivation tree in reverse, a stack is 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. So the rule on top of the final stack which stores a successfully parsed derivation tree should be a grammar production rule with the *start symbol* of an LR grammar at the left hand side of it. Note that every rule is represented by an $R$-bit binary number and the mapping from a binary-coded rule to the rule itself can be easily realized by a binary mapping perceptron module [Chen and Honavar, 1994, 1995a].

## 4.3 Architecture of an NN LR(1) Parser

Figure 4 shows the architecture of a modular neural network design for an LR(1) parser which takes advantage of efficient `shift/reduce` technique. The NN LR(1) parser uses an optional queue handler module and an NN stack storing parse tree (derivation tree) to interface with an NN lexical analyzer and a next processing unit respectively. The queue handler stores lexical tokens extracted by the NN lexical analyzer and facilitates the working of lexical analyzer and parser in parallel. To get the binary-coded grammar production rules in derivation order sequentially out of the NN stack which stores parse tree, the next processing unit sends binary-coded stack `pop` actions to the stack one by one.

### Modules of the NN LR(1) Parser

The proposed NN LR(1) parser consists of one BMP (Binary Mapping Perceptron, see [Chen and Honavar, 1994; 1995a] for details) module imple-

menting parsing table, one NN `shift/reduce` stack storing states during `shift/reduce` simulation, a buffer storing current state `state(t)` which is on top of the NN `shift/reduce` stack, and a buffer 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 feeding of `input(t)` is from the latched buffer *lhs* and the feeding of `input(t)` from the queue handler 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 feeding of `input(t)` is from the queue handler and the feeding of `input(t)` from the latched buffer *lhs* is inhibited by the rightmost bit of the binary-coded `shift` action.

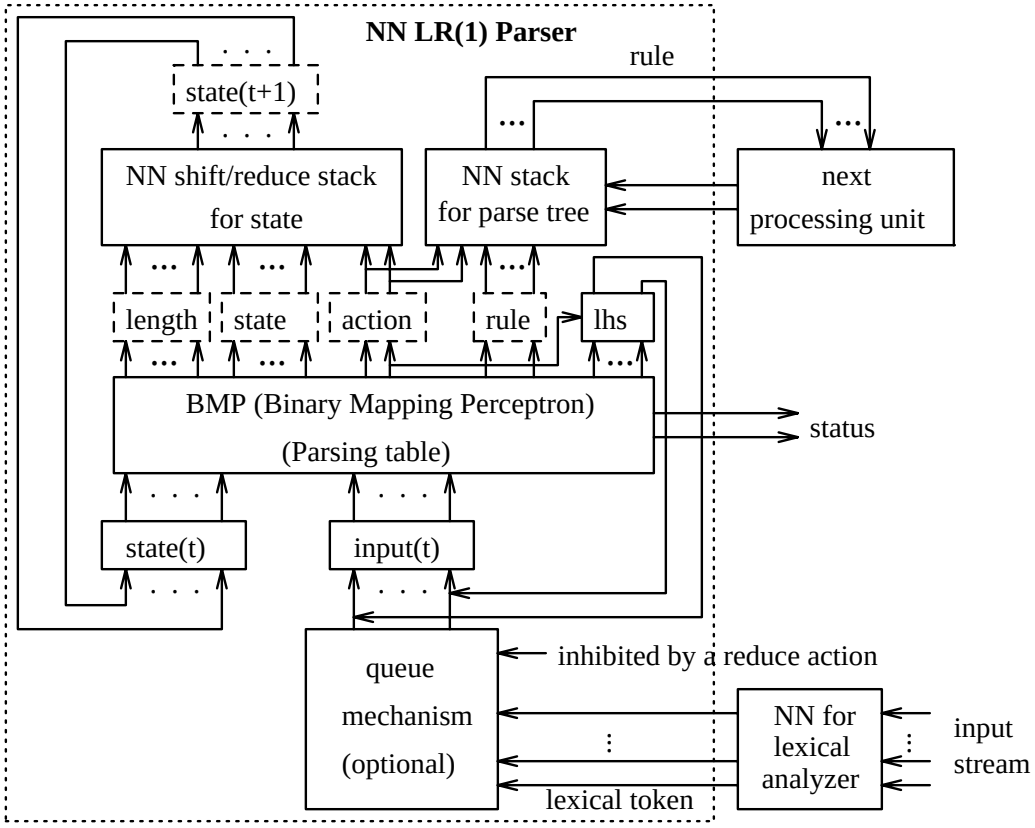


Figure 4. Neural networks for LR(1) parser. The dotted boxes *length*, *state*, *action*, *rule*, and $state(t+1)$ exist only logically but not physically. The state on top of the `shift/reduce` stack is denoted as `state(t)`, and `input(t)` denotes either current input token or a grammar nonterminal symbol produced

by last parsing action which is a `reduce`. When the last consulted parsing action is a `reduce` encoded as 10, the feeding of `input(t)` is from the latched buffer *lhs* and the feeding of `input(t)` 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 feeding of `input(t)` is from the queue mechanism and the feeding of `input(t)` from the latched buffer *lhs* is inhibited by the rightmost bit of the binary-coded `shift` action.

The start of a parsing is triggered by a reset signal to the NN `shift/reduce` stack and the NN stack storing parse tree, which resets the SPs of these two stacks to stack bottom and hence resets `state(t)` to initial state. To avoid clutter, the reset signal lines are not shown in Figure 4. The current state buffer `state(t)` and the current input buffer `input(t)` are under synchronization control which is also omitted in Figure 4. In Figure 4, the dotted boxes *length*, *state*, *action*, *rule*, and *state*(t+1) exists only logically but not physically.

The parsing of an LR parser can be viewed in terms of transitions from an initial configuration to a final configuration, which are made of a sequence of `shift` and `reduce` actions. The transition from one configuration to another can be divided into two steps. The first step is to consult the parsing table for next action using current input symbol and current state on top of the stack. The look-up of an entry in the parsing table can be viewed generally in terms of binary mapping. The second step is to execute the next action which is either a `shift` or a `reduce` consulted from the parsing table. In the NN LR(1) parser, the first step is simulated by a BMP module which implements the consultation of the parsing table; and the second step is executed by the NN `shift/reduce` stack which stores states, the NN stack which stores parse tree, and sometimes the BMP module again when the next action is a `reduce`.

### Number of Neurons Used in the Modules of the NN LR(1) Parser

Let $M$ be the number of defined ACTION entries in the parsing table. Note that all grammar symbols of the grammar which covers all input lexical tokens are encoded into $H$-bit binary codes. Then the BMP module for parsing 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 about `status` in an ACTION entry of the parsing table), for any undefined ACTION entry in the parsing table. The $R$-bit binary-coded grammar production rule is used as the stack symbol for the NN stack which stores parse tree.

Assume the pointer control module of the NN stack for parse tree use $m_p$ bits to encode its SP values. Then in the NN stack for parse tree, the pointer control module uses $m_p + 2$ input neurons, $3 \times 2^{m_p}$ 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; and the write control module has two sets of input signals, one of which has $m_p + 1$ signals and the other of which has R signals that encode the grammar production rules.

To be able to efficiently implement the `reduce` action in LR parsing, the NN `shift/reduce` stack is slightly modified from the NN stack described in Section 2 to allow multiple stack `pop`s in one operation cycle of the NN Parser. The number of `pop`s, which is an $L$-bit binary number and equals the number of non-empty grammar symbols at the right hand side of the grammar production rule being reduced, is also used as input to the pointer control module and write control module in the NN `shift/reduce` stack. So each of these modules use $L$ additional input neurons in the NN `shift/reduce` stack as compared to the NN stack proposed in Section 2. The $S$-bit binary-coded state emitted from the NN parsing table is used as the stack symbol to the NN `shift/reduce` stack. Let $L_m$ be the maximum number of non-empty grammar symbols at the right hand side of all grammar production rules in the LR grammar being parsed. A $k$ multiple `pop`s is implemented in the NN `shift/reduce` stack the same way as a single `pop` in the NN stack proposed in Section 2 except that the SP value is decreased by $k$ instead of 1, where $1 \leq k \leq L_m$. Hence at every SP value, $L_m - 1$ more hidden neurons are required to realize multiple `pop`s in a range between 1 and $L_m$ `pop`s in the pointer control module.

# 5 Performance Comparison with Syntax Analysis in Current Computer Systems

This section compares the anticipated performance of syntax analysis by the proposed neural architecture with the estimated performance of syntax

analysis by current computer systems. This analysis takes into account the performance of hardware that is used in these two systems and the process used for syntax analysis. First, the performance of the proposed neural network is estimated, based on current CMOS technology for realizing neural networks. Next, syntax analysis using current computer systems is examined, and its performance is estimated and compared with that of the proposed neural architecture. In the comparison, error handling is not considered and it is assumed that the two systems have comparable I/O performance.

## 5.1 Performance of Current Electronic Realization for Neural Networks

Many electronic realizations of artificial neural networks (ANNs) have been reported [Graf and Henderson, 1990; Grant et al., 1994; Hamilton et al., 1992; Lont and Guggenbühl, 1992; Masa et al., 1994; Massengill and Mundie, 1992; Moon et al., 1992; Robinson et al., 1992; Uchimura et al., 1992; Watanabe et al., 1993]. ANNs are implemented mostly using CMOS-based analog, digital, and hybrid electronic circuits. Analog circuits which mainly consist of processing elements for multiplication, summation and thresholding is popular for the realization of ANNs since compact circuits capable of high-speed asynchronous operation can be achieved [Gowda et al., 1993]. [Uchimura et al., 1992] reports a measured propagation delay of 104 *ns* from the *START* signal until the result is latched by using digital synapse circuit containing an 8-bit memory, an 8-bit subtractor and an 8-bit adder. [Masa et al., 1994] adopts a hybrid analog-digital design with 5-bit (4 bits + sign) binary synapse weight values and current-summing circuits to implement a feed-forward neural network with 2 connection layers, and a network computation time of less than 20 *ns* is reported. [Grant et al., 1994] achieves the throughput at the rate of 10MHz (delay = 100 *ns*) in a Hamming Net pattern classifier using analog circuits. The 1st-layer and 2nd-layer subnetworks of the proposed BMP neural architecture are very similar to the 1st-layer subnetwork of a Hamming Net respectively, and the neural architecture with 2 connection layers in the proposed BMP is exactly same as that implemented by [Masa et al., 1994] except that [Masa et al., 1994] uses discretized input values, 5-bit synaptic weights, one output neuron, and sigmoid-like activation function. The proposed BMP uses binary input values, bipolar synaptic weights, multiple

output neurons, and binary hardlimiter as activation function. Hence the computation delay of the proposed BMP can be expected to be at worst of the order of 100 nanoseconds and at best 20 nanoseconds given the current CMOS technology for realizing ANN.

The development of specialized hardware for implementation of ANNs is still in its early stages. Conventional CMOS technology that is currently the main technology for VLSI implementation of ANN is known to be slow [Kumar, 1994; Lu and Samuleli, 1993]. Other technologies, such as BiCMOS, NCMOS [Kumar, 1994], pseudo-NMOS logic, standard N-P domino logic, and quasi N-P domino logic [Lu and Samuleli, 1993], may provide better performance for the realization of ANNs. Thus, the performance of the hardware implementation of ANNs is likely to improve with technological advances in VLSI.

## 5.2 Performance Analysis of Syntax Analysis in Current Computer Systems

In the following analysis, it is assumed that all program and data codes for analyzing syntax using current computer systems are pre-loaded from main memory to caches (instruction, data and secondary caches) since accessing program instructions and data from main memory, the cost-effective access time of which is around 60 ns currently, slows down the performance of whole process. To simplify the comparison, it is assumed that all program instructions run on a currently cost-effective 32-bit 100 MIPS (million instructions per second, the effects of processor clock, caches, pipeline and superscalar design are combined into this simple measure for the speed of a microprocessor) processor and that every instruction takes 10ns on average to access and execute although some instructions like those involving multiplication would take two or more times.

Usually, a syntax analysis system mainly integrates four processes including lexical analysis, grammar parsing, parse tree construction and error handling. In current computer systems, these four processes are generally coded into two main procedures. 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 procedure which often evokes the other procedure as needed. The two

procedures may evoke other sub-procedures to maximize modularity within themselves. In single-CPU computer systems which have extensive users, if global variables are used as often as possible, no value is passed between procedures, no variable of caller and its super procedures is accessed in the callee procedure, and static chain is used for the implementation of procedure calls; the penalty associated with modularity due to a procedure call at least includes (1) state saving of the caller procedure, activation of the callee procedure and entering of the callee procedure at a procedure call, which together costs at least 6 instructions (see Figure 6.3 of [MacLennan, 1987]); and (2) context restoration and resumption of caller procedure at the return (exit) of the callee procedure, which together costs at least 3 instructions (see Figure 6.4 of [MacLennan, 1987]). Such penalty due to a procedure call totally takes 9 instructions to implement, which in turn takes about 90ns .

### Performance Analysis of Lexical Analyzer

Usually lexical analysis can be decomposed into two steps: word segmentation and translation of words. Word segmentation is simulated by a *dfa* whose transitions can be represented as a 2D table with current state and current input character as indices. So the transitions of such a $DFA$ can be characterized as a repetitive table look-up process for next state using current state and current input symbol until an error state or an accepting state is reached. In current computer systems, such a repetitive table look-up process can be simply implemented by four instructions which include one multiplication and one addition to compute for next state the offset in the transition table, one memory access to fetch the next state from the table using the offset, and one branch-on-comparison instruction to jump back to the first instruction of the repetitive loop if the next state is neither an error state nor an accepting state. (Note that this analysis ignores I/O processing requirements). According to previous assumption, every transition takes 4 instructions which together takes 40ns to execute; in contrast, the computation delay for a transition implemented by the proposed NN lexical analyzer is estimated to be at best 20 ns and at worst 100 ns.

For programming language compilation, the translation of a segmented word into a token is usually implemented directly using a `case` statement or a series of `if` statements. For most programming languages, the number of different tokens used is typically about 50 [Chapman, 1987]. In this case, it

takes 25 `if` statements on average to translate a word into its corresponding token if these `if` statements are not efficiently organized. So this step would take 250ns in the compilation of programming languages. For natural language processing, a simple database would be used to translate the meaning of a word due to the large vocabulary in natural languages. In this case, it would generally take more than 250ns, depending on the size of the vocabulary, to translate a word into its corresponding desired meaning using a database (The interested reader is referred to [Chen and Honavar, 1995b] for details about the anticipated performance of database query processing in current computer systems); in contrast, the computation delay for such a translation using the proposed BMP module is estimated to be at best 20ns and at worst 100ns.

Assume the average length of words in the syntax is $W$. Then, without considering I/O, error handling and the penalty of modularity, it would take $(40W + 250)$ ns on average to analyze a word using the virtual 100 MIPS processor and takes at best $20W + 20$ ns using the proposed NN lexical analyzer. If $W = 5$, then the former takes 450ns and the latter takes at best 120ns to analyze a word without considering I/O, error handling and the penalty of modularity in programming languages.

### Performance Analysis of LR Parser

The LR parser is also driven by a 2D table (parsing table) with current state and current input symbol as indices. Once a next state is looked up, it is used as the current state for next move and is maintained in a stack. During parsing, `shift` and `reduce` are the two main types of moves applied repetitively. A `shift` move at least would takes 6 instructions and hence 60 ns totally to implement, which include 3 instructions to implement a consultation of the parsing table, 1 instruction to store the next state into the stack memory, 1 instruction to increment the stack pointer, and 1 instruction to go back to the first instruction of the repetitive loop for next move. A `reduce` move basically consists of a consultation of the parsing table, a `pop` of the state stack, a `push` to store a rule into the stack for parse tree, and a `shift` move. So a `reduce` would take 12 $(= 3 + 1 + 2 + 6)$ instructions and hence 120ns to implement using the virtual 100 MIPS processor.

In the proposed NN parser, the computation delay has two parts. One is due to the operation of the BMP for parsing table, and the other is due

to the operation of the two stacks. The NN stack basically consists of one BMP and one BMP extended with write control module whose computation delay is unknown. Assuming that the computation delay of an NN stack is at best equal to that of two sequentially linked BMPs, which is at best 40ns, a `shift` move which takes an operation cycle of the NN parser and a `reduce` move which takes two operation cycles of the NN parser would respectively take at best 60 (= 20 + 40) ns and 120 (= 2 × 60) ns without considering the effect of queuing between the NN parser and the NN lexical analyzer.

Assuming the average length of words in the syntax is $W$. In single-CPU computer systems, the parsing procedure calls the procedure for lexical analysis once for every word. Therefore, ignoring I/O, error handling and penalty for modularity, it would respectively take $(40W+310)$ ns and $(40W+370)$ ns on average to parse a word by a `shift` move and by a `reduce` move using the virtual 100 MIPS processor. Correspondingly, because the NN parser and NN lexical analyzer operate in parallel, it would take at best 60 ns or $20W + 20$ ns, whichever is larger, to parse a word by a `shift` move; and it would take at best 120 ns or $20W+20$ ns, whichever is larger, to parse a word by a `reduce` move. Thus, the proposed NN LR parser provides an attractive alternative to current computer systems for parsing LR grammars.

# 6 Summary

Artificial neural networks, due to their inherent parallelism and potential fault tolerance, offer an attractive paradigm for robust and efficient implementations of syntax analysis. This paper has proposed a modular neural network architecture for syntax analysis on continuous input stream of characters. The components of the proposed architecture include neural networks (*NN*s) acting as a stack, a lexical analyzer, a grammar parser and a parse tree construction module. The proposed NN stack allows the simulation of a stack of large depth, needs no training, and hence is not application-specific. Such a general-purpose stack provides a good alternative to RAAM networks as a stack in integrated neural systems for symbol processing. The proposed NN lexical analyzer offers high performance and is an efficient alternative for lexical analysis, especially in natural language processing applications involving large vocabularies. The proposed NN parser can produce a parse tree as a result of parsing strings from widely used subsets of deterministic

context-free languages (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 parsing table which changes from one parser to another [Aho and Ullman, 1977], the proposed NN LR parser can be used as a general-purpose neural network architecture for LR parsing. Performance comparison of the proposed neural architecture for syntax analysis with that of current computer systems suggests that the former offer an attractive and efficient alternative for practical applications involving syntax analysis.

# References


Aho A. V., and Ullman J. D., Principles of Compiler Design, *Addison-Wesley*, Reading, MA. 1977.

Aho A. V., Sethi R., and Ullman J. D., Compilers: Principles, techniques, and Tools, *Addison-Wesley*, Reading, MA. 1986.

Allen, R. B., Connectionist Language Users, *Connection Science* vol. 2, no. 4., pp. 279-., 1990.

Berg G., A Connectionist Parser with Recursive Sentence Structure and Lexical Disambiguation, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pp. 32-37, MIT Press, Massachusetts, 1992.

Chapman N. P., LR Parsing : Theory and Practice, *Cambridge University Press*, Cambridge, 1987.

Chen C. and Honavar V., Neural Network Automata, *Proceedings of World Congress on Neural Networks*, vol. 4, pp. 470-477, San Diego, June 1994.

Chen C. and Honavar V., A Neural Architecture for Content as well as Address-Based Storage and Recall: Theory and Applications, *paper under review* (Draft available as: Iowa State University Tech. Rep. ISU CS-TR 95-03), 1995a.

Chen C. and Honavar V., A Neural Network Architecture for High-Speed Database Query Processing System, *paper under review* (Draft available as: Iowa State University Tech. Rep. ISU CS-TR 95-11), 1995b.

Das, S., Giles, L. and Sun, G-H., Using Hints to Successfully Learn Context-Free Grammars with a Neural Network Pushdown Automanton, *Advances in Neural Information Processing Systems*, vol. 5., pp. 65-, Morgan Kaufmann, San Mateo, CA., 1993.

Dyer M. G., Connectionist Natural Language Processing: A Status Report, *Computational Architectures Integrating Neural and Symbolic Processes : A Perspective on the State of the Art*, Sun R. and Bookman L. A. (Ed.), Chapter 12, Kluwer Academic Publishers, Massachusetts, 1995.

Elman, J. L., Finding Structure in Time, *Cognitive Science*, vol. 14., pp. 179-211, 1990.

Fanty M. A., Context-free Parsing with Connectionist Networks, *Proceedings of AIP Neural Networks for Computing, Conference No. 151*, pp. 140-145, Snowbird, Utah, 1986.

Frasconi, P., Gori, M., Maggini, M. and Soda, G., Unified Integration of Explicit Rules and Learning by Example in Recurrent Networks, *IEEE Transactions on Knowledge and Data Engineering*, 1993.

Gazdar G. and Mellish C., Natural Language Processing in POP-11: An Introduction to Computational Linguistics, *Addison-Wesley*, 1989.

Giles, C. L., Miller, C. B., Chen, D., Sun, G. Z., Lee, Y. C., Learning and Extracting Finite State Automata with Second-Order Recurrent Neural Networks, *Neural Computation*, vol. 4., no. 3., pp. 380-, 1992.

Giles, L., Horne, B. W., and Lin, T. Learning a Class of Large Finite State Machines With a Recurrent Neural Network. UMIACS TR 94-94. To appear in *Neural Networks*, 1995.

Goonatilake S. and Khebbal S. (Ed.), Intelligent Hybrid Systems. *Wiley*, London, 1995.

Gowda S. M. et al., Design and Characterization of Analog VLSI Neural Network Modules, *IEEE Journal of Solid-State Circuits*, vol. 28, no. 3, pp. 301-313, 1993.

Graf H. P. and Henderson D., A Reconfigurable CMOS Neural Network, *ISSCC Dig. Tech. Papers*, pp. 144-145, San Francisco, CA, 1990.

Grant D. et al., Design, Implementation and Evaluation of a High-Speed Integrated Hamming Neural Classifier, *IEEE Journal of Solid-State Circuits*, vol. 29, no. 9, pp. 1154-1157, September 1994.

Hamilton A. et al., Integrated Pulse Stream Neural Networks: Results, Issues, and Pointers, *IEEE Transactions on Neural Networks*, vol. 3, no. 3, pp. 385-393, May 1992.

Hester K. A. et al., The Predictive RAAM: A RAAM That Can Learn to Distinguish Sequences from a Continuous Input Stream, *Proceedings of World Congress on Neural Networks*, vol. 4, pp. 97-103, San Diego, June 1994.

Honavar V., Symbolic Artificial Intelligence and Numeric Artificial Neural Networks: Toward A Resolution of the Dichotomy. In: R. Sun and L. Bookman (Ed.), *Computational Architectures Integrating Symbolic and Neural Processes*. pp. 351-388. Kluwer, New York, 1994.

Honavar V. and Uhr L., Coordination and Control structures and Processes: Possibilities for Connectionist Networks, *Journal of Experimental and Theoretical Artificial Intelligence 2: 277-302*, 1990.

Honavar V. and Uhr L. (Ed.), Artificial Intelligence and Neural Networks: Steps Toward Principled Integration, *Academic Press*, New York, 1994.

Honavar V. and Uhr L., Integrating Symbol Processing Systems and Connectionist Networks. In: Goonatilake, S. and Khebbal, S. (Ed.), *Intelligent Hybrid Systems*. pp. 177-208. Wiley, London, 1995.

Hopcroft J. E. and Ullman J. D., Introduction to Automata Theory, Languages, and Computation, *Addison-Wesley*, 1979.

Horne, B., Hush, D. R. and Abdallah, C., A State Space Recurrent Neural Network with Application to Regular Grammatical Inference, UNM Tech. Rep. No. EECE 92-002, Department of Electrical and Computer Engineering, University of New Mexico, Albuquerque, New Mexico, 1992.

Kleene, S. C., Representation of Events in Nerve Nets and Finite Automata. In: *Automata Studies*, Shannon, C. E., and McCarthy, J. (Ed.), pp. 3-42, Princeton University Press, Princeton, New Jersey, 1956.

Kumar R., NCMOS: A High Performance CMOS Logic, *IEEE Journal of Solid-State Circuits*, vol. 29, no. 5, pp. 631-633, 1994.

Levine D. and Aparicioiv M. (Ed.), Neural Networks for Knowledge Representation and Inference. *Lawrence Erlbaum*, Hillsdale, NJ, 1994.

Lont J. B. and Guggenbühl W., Analog CMOS Implementation of a Multilayer Perceptron with Nonlinear Synapses, *IEEE Transactions on Neural Networks*, vol. 3, no. 3, pp. 457-465, 1992.

Lu F. and Samueli H., A 200-MHz CMOS Pipelined Multiplier-Accumulator Using a Quasi-Domino Dynamic Full-Adder Cell Design, *IEEE Journal of Solid-State Circuits*, vol. 28, no. 2, pp. 123-132, 1993.

MacLennan B. J., Principles of Programming Languages : Design, Evaluation, and Implementation, 2nd edition, *CBS College Publishing*, 1987.

Masa P., Hoen K. and Wallinga H., 70 Input, 20 Nanosecond Pattern Classifier, *IEEE International Joint Conference on Neural Networks*, vol. 3, Orlando, FL, 1994.

Massengill L. W. and Mundie D. B., An Analog Neural Network Hardware Implementation Using Charge-Injection Multipliers and Neuron-Specific Gain Control, *IEEE Transactions on Neural Networks*, vol. 3, no. 3, pp. 354-362, May 1992.

McCulloch, W. S. and Pitts, W., A Logical Calculus of Ideas Immanent in Nervous Activity, *Bulletin of Mathematical Biophysics*, vol. 5, pp. 115-133, 1943.

Miikkulainen R, Subsymbolic Parsing of Embedded Structures, *Computational Architectures Integrating Neural and Symbolic Processes : A Perspective on the State of the Art*, Sun R. and Bookman L. A. (Ed.), Chapter 5, Kluwer Academic Publishers, Massachusetts, 1995.

Minsky, M. *Computation: Finite and Infinite Machines*, Prentice-Hall, Englewood Cliffs, New Jersey, 1967.

Moon G. et al., VLSI Implementation of Synaptic Weighting and Summing in Pulse Coded Neural-Type Cells, *IEEE Transactions on Neural Networks*, vol. 3, no. 3, pp. 394-403, May 1992.

Mozer, M. C. and Bachrach, J., Discovering the Structure of a Reactive Environment by Exploration, *Neural Computation*, vol. 2., no. 4., pp. 447-, 1990.

Mozer, M. C. and Das, S., A Connectionist Symbol Manipulator that Discovers the Structure of Context-Free Languages, *Advances in Neural Information Processing Systems* 5, pp. 863-, Morgan Kaufmann, San Mateo, CA., 1993.

Noda, I., and Nagao, M., A Learning Method for Recurrent Neural Networks Based on Minimization of Finite Automata, *Proceedings of International Joint Conference on Neural Networks* vol. 1., pp. 27-32, IEEE Press, Piscataway, New Jersey, 1992.

Omlin, C. and Giles, L., Extraction and Insertion of Symbolic Information in Recurrent Neural Networks, In: *Artificial Intelligence and Neural Networks: Steps Toward Principled Integration*, Honavar, V. and Uhr, L. (Ed.), Academic Press, New York, 1994.

Omlin, C. and Giles, L. Fault-Tolerant Implementation of Finite State Automata in Recurrent Neural Networks. RPI Computer Science Tech. Rep. 95-3, 1993.

Pollack, J. B., On Connectionist Models of Language Processing, Ph.D. Dissertation, Computer Science Department, University of Illinois, Urbana-Champaign, 1987.

Pollack J. B., Recursive Distributed Representations, *Artificial Intelligence* **46**, pp. 77-105, 1990.

Robinson M. E. et al., A Modular CMOS Design of a Hamming Network, *IEEE Transactions on Neural Networks*, vol. 3, no. 3, pp. 444-456, 1992.

Sanfeliu, A. and Alquezar, R., Understanding Neural Networks for Grammatical Inference and Recognition, *Advances in Structural and Syntactic Pattern Recognition*, Bunke, H. (Ed.), World Scientific, Singapore, 1992.

Sanamrad M. A., Wada K. and Matsumoto H., A Hardware Syntactic Analysis Processor, *IEEE Micro* 7, pp. 73-80, August 1987.

Servan-Screiber, D., Cleeremans, A., and McClelland, J. L., Graded State Machines: The Representation of Temporal Contingencies in Simple Recurrent Neural Networks, In: *Artificial Intelligence and Neural Networks: Steps Toward Principled Integration*, Honavar, V. and Uhr, L. (Ed.), Academic Press, New York, 1992.

Schulenburg D., Sentence Processing with Realistic Feedback, *IEEE/INNS International Joint Conference on Neural Networks*, vol. IV, pp. 661-666, Baltimore, 1992.

Siegelman, H. T. and Sontag, E. D., Turing-Computability with Neural Nets, *Applied Mathematics Letters*, vol. 4., no. 6., pp. 77-80, 1991.

Sippu S. and Soisalon-Soininen E., Parsing Theory, vol. II : LR(k) nad LL(k) Parsing, *Springer-Verlag*, 1990.

Sun R. and Bookman L. (Ed.), Computational Architectures Integrating Symbolic and Neural Processes. *Kluwer*, New York, 1994.

Uchimura K. et al., An 8G Connection-per-second 54mW Digital Neural Network with Low-power Chain-Reaction Architecture, *ISSCC Dig. Tech. Papers*, pp. 134-135, San Francisco, CA, 1992.

Uhr L. and Honavar V., Artificial Intelligence and Neural Networks: Steps Toward Principled Integration. In: Honavar, V. and Uhr, L. (Ed.), *Artificial Intelligence and Neural Networks: Steps Toward Principled Integration.* pp. xvii-xxxii. Academic Press, Boston, MA, 1994.

Waltraus, R. L., and Kuhn, G. M., Induction of Finite-State Languages Using Second-Order Recurrent Neural Networks, *Neural Computation*, Vol. 4., No. 3., pp. 406-, 1992.

Watanabe T. et al., A Single 1.5-V Digital Chip for a $10^6$ Synapse Neural Network, *IEEE Transactions on Neural Networks*, vol. 4, no. 3, pp. 387-393, May 1993.

Zeng Z., Goodman R. M. and Smyth P., Discrete Recurrent Neural Networks for Grammatical Inference, *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 320-330, March 1994.