CHAPTER 4

The Role of Visual Perception, Selective Attention, and Short-Term Memory for Symbol Manipulation: A Neural Network Model that Learns to Evaluate Simple LISP Expressions*

Rainer Goebel
Department of Psychology
University of Braunschweig

INTRODUCTION

Which computational processes allow us to accomplish complex cognitive tasks such as computer programming? Following Rumelhart, Smolensky, McClelland, and Hinton (1986), there are three abilities of human information processing that seem to be essential for understanding human intelligence:

- We are especially good at pattern matching. We are able to quickly "settle" on an interpretation of an input pattern. This ability of simply "perceiving" answers to problems seems to be central at all levels of cognition (e.g., perception, memory, and comprehension).
- We are good at modeling our world. This involves the anticipation of new environmental states.
- We are good at manipulating our environment. This involves a kind of serial processing mechanism. What sets human intellectual accomplishments apart from animals is that we are able to manipulate the

*I am grateful to Uli Frauenfelder, Klaus Hahn, Michael Moser, and Dirk Vorberg for helpful discussions. This work was supported by a grant from the Deutsche Forschungsgemeinschaft (Vo 212/2-4).
environment so that it comes to represent something (e.g., numbers or letters). We can manipulate these “symbols” in the environment or by mental simulations. This allows us to perform temporally extended reasoning and problem-solving tasks.

At present, it seems that pattern matching is best accomplished by parallel distributed processing (PDP) models and that symbol manipulation is best accomplished by symbol processing systems. Recurrent PDP models are good at pattern matching, associative memory, and pattern completion through a parallel, interactive relaxation process. Starting with external stimulation and an internal activity state, these networks relax to a stable state (fixed point attractor) corresponding to an “interpretation” of the stimulation. By applying appropriate learning rules (e.g., Ackley, Hinton, & Sejnowski, 1985; Hopfield, 1982, 1984; Pineda, 1989), attractors can be located at positions of the activity landscape that correspond to desired knowledge items.

The second ability—the anticipation of new environmental or internal states—may be considered as a special case of “pattern matching/completion” because parts of the (relaxed) state vector can be used to represent expected environmental changes.

In contrast to (physical) symbol systems (Newell, 1980), PDP models have difficulties with the third ability, that is, to perform algorithmic-like sequential tasks, especially those involving variables and explicit rule following (e.g., Norman, 1986). The ability to perform symbolic manipulations seems to be a specifically human form of information processing. Although pattern matching may be the more fundamental computational process of human cognition, the ability of creating and manipulating symbolic expressions must also be taken into account by connectionist models in order to achieve a more complete understanding of human cognition.

Consequently, this chapter describes a neural network model that integrates the advantages of homogenous PDP networks and symbol manipulation systems. The model should be considered as a first step toward the ultimate goal to establish a general modeling framework that provides a psychologically more adequate approach for explaining human cognition as symbol manipulation systems or current PDP models in isolation.

In order to obtain concrete results, the evaluation of simple LISP expressions was chosen as an example task because it is a paradigmatic symbolic manipulation task and a good starting point for building psychologically interesting models of complex cognitive processes such as computer programming.

The following contributions are made in this chapter:

- the difficulties of homogenous PDP models concerning symbol manipulation tasks are analyzed

- cognitive processes are identified, which are able to overcome these difficulties, most notably selective attention and short-term memory
- a modular neural network model is described in detail which is able to perform the symbolic example task by incorporating structures and mechanisms for translation-invariant perception, selective attention, and short-term memory.

**WHY ARE PDP MODELS POOR AT SYMBOL MANIPULATION?**

The goal of the first simulation experiments is to study the performance of simple PDP models on a purely symbolic task to gain insights into their difficulties with such tasks. Of course, the results of the study depend on the chosen networks and task. The used networks are small examples of what is called a PDP model throughout the chapter, namely homogenous feedforward or recurrent back-propagation networks.

**A Symbolic Task: Evaluating Simple LISP Expressions**

The evaluation of simple LISP expressions was chosen as the example task because it counts as a paradigmatic case of algorithmic symbol manipulation. In this section, the task is used in a highly simplified form in order to analyze two homogenous PDP networks. In the fourth section, a neural network is presented which overcomes the difficulties of the analyzed PDP networks. This network uses a more complex version of the LISP evaluation task.

The PDP networks of the first study are trained to learn two primitive LISP functions—QUOTE and SECOND (or CAR (CDR 'list')). The function QUOTE simply returns its argument, and the function SECOND selects the second element of a list. For present purposes it is sufficient to use lists of length 3 and only two elements (e.g., the letters A and B). Thus, only eight distinct sequences are possible as input:

A-A-A      B-A-A
A-A-B      B-A-B
A-B-A      B-B-A
A-B-B      B-B-B

There is a deeper reason for using this task which will become clear in the third section. Some related simulation experiments with recurrent networks
trained on a broader corpus of variable-length lists are reported in (Goebel, 1990a).

**Training procedure.** The training process runs through several stages. First, a network is trained on all eight lists to see whether it can learn the task to criterion. Then one list is excluded, the network learns the remaining seven lists, and generalization performance is tested for the missing case. All eight different training sets are tested this way. In the next stage the studied networks are trained on all combinations of six lists, and generalization performance is tested for the two excluded lists. Each individual simulation was replicated 10 times using each time a new set of random weights drawn from the interval $-1, 1$.

**The rationale behind this task.** Although only a small problem size is used this task has implications for the general case: There are $\alpha^l$ possible strings with $\alpha$ as the alphabet size and $l$ as the length of the longest considered string. Even with a rather small alphabet size (e.g., $\alpha = 24$) and a small maximal string length (e.g., $l = 7$), there are huge numbers of possible strings (more than $6 \times 10^9$). Thus, it is simply impossible to train a network on all possible sequences (lists). The success of the network critically depends on its generalization behavior: Is it possible to train only a reasonably small subset of all strings so that the network generalizes correctly to all other cases?

**Technical Details**

The networks are using the back-propagation learning rule (Rumelhart et al., 1986). In order to apply the generalized delta rule to the second studied, recurrent network (see later) the unfolding through time technique was adopted (Minsky & Papert, 1969; Rumelhart et al., 1986) and improved (Goebel, 1990a; Williams & Zipser, 1990). In all simulations, the learning criterion was set such that the activity value of each output unit should not deviate more than 0.1 from the supplied target value. The target values for the studied recurrent network could be specified as “don’t care values” (Jordan, 1986), meaning that an output unit is allowed to produce an arbitrary value at a certain time step.

**Feedforward Networks: The Buffer Approach**

In order to represent strings in feedforward nets one can use an input buffer large enough to hold the longest string: The input layer and output layer are divided in many parts ("slots"), each representing one element (letter) in successive order.

A feedforward network with three input and output slots was constructed.

---

![Diagram](https://via.placeholder.com/150)

**Figure 4.1.** The feedforward network: Examination of the buffer approach

Each slot consisted of two units, one representing the slot-filler A and one representing slot-filler B. Two additional input units—one for the QUOTE function and one for the SECOND function—were used. The input layer was totally connected to the hidden layer consisting of five units which in turn was totally connected to the output layer (see Figure 4.1). A task was presented to the network by clamping the appropriate units at the input layer. If, for example, the LISP expression (SECOND '(A A B)) should be evaluated, the input units representing SECOND, A (slot 1), A (slot 2), and B (slot 3) are activated. The desired output would be that the element A in slot 2 becomes active (see Figure 4.1).

**Results.** The network quickly learns to perform the QUOTE and SECOND function on all eight lists. The network also succeeded on the 7-list subtask generalizing well to the one excluded list. However, beginning with the 6-list subtask, the network didn’t generalize well.\(^1\)

The typical kind of “errors” produced by the network can be seen by considering the case where the two lists:

- B-B-A, and
- B-B-B

were excluded from the training set. In this case errors were produced in the first and second position, but not in the third one. This is due to the fact that

\(^1\)Similar results were obtained using more (6,7) or less than 5 (4) hidden units.
during training twice as many As than Bs occurred in the first and second position, resulting in a "bias" to produce an A in these positions, even if the letter B is actually present. This demonstrates the capability of PDP models to discover statistical regularities in the training set, which in this example corresponds to the greater frequency of occurrence of the element A in the first and second position.

Recurrent Networks: Sequential Processing

A disadvantage of feedforward networks using the buffer approach is that acquired knowledge is position dependent: If the network has learned to recognize an A in the first slot, it does not know how to recognize the A in the second slot because this knowledge is tied to the weights emerging from the first slot.

Recurrent networks seem a promising alternative for the symbolic task due to the ability for handling sequentially presented data. The advantage is that each input element enters the network through the same connections. However, this requires seriality: An input element enters through the same connections at different time steps as opposed to the feedforward case in which each element enters at different connections at the same time step. In general, sequences of varying length may be processed flexibly because there is no fixed buffer width. Recurrent connections provide the potential of a short-term memory because they allow one to retain information of past events over time which may establish a context for the processing of subsequent elements.

Note that the output layer of a recurrent network may be either a static buffer in which different slots are filled over time, or it may consist of one slot (like the input layer) in which a sequential output produces itself. Both types of output schemes were tested and did produce similar results. For simplicity, only the second case is considered here.

A simple recurrent network with 4 input units, 6 hidden units—fully interconnected—and 2 output units was constructed (see Figure 4.2). The network operates like the network proposed by Elman (1990) except that it uses back propagation through time (e.g., Williams & Zipser, 1990).

Consider a typical training example corresponding to the LISP expression (QUOTE (A B B)) (see Figure 4.3 which shows the network unfolded in time): At the first time step the first element of the list, A, is presented to the input layer. The activation flows to the hidden layer and then to the output layer as in feedforward networks. At this time step no special output is required (don't-care condition—Jordan, 1986). At the next time step both the activity of the second presented list element, B, together with the activity of the hidden layer at the previous time step flow to the hidden layer at this time step and then to the output layer. Then, the third input element, B, is presented and processed in the same way. Now, one of the units representing

![Goebel](112) Role of Visual Perception

Figure 4.2. The recurrent network: Examination of the sequential approach

![Goebel](113)

Figure 4.3. The "unfolded" recurrent network showing the network state at different time steps

A LISP function is activated. If, for example, the QUOTE unit is activated, the network must play back the list at the output layer. Therefore the first element of the list, A, is the required target value at the current time step for the output layer. At the next time step no input is presented, but the network has to produce the second element, B, and at the next time step the final element, which is B again. During training, the difference values between the required target values and the actually produced output values are back propagated through the network. All computed weight changes for the same weight at different time steps are added together.
Results. The task is much more difficult to solve than in the feedforward case because the recurrent network itself must learn to construct an appropriate compact internal representation of the sequentially presented elements which enables the network to convert this representation back to a correct output sequence. The network has to construct itself an appropriate internal "plan vector" (see Jordan, 1986). This is a difficult problem since the input lists are all similar to each other and the injected error values are propagated back through a deeply layered (unfolded) network.

The "generalization errors" produced by the recurrent network were qualitatively similar to the results obtained with the feedforward network. If the model did not generalize "correctly" to novel sequences, it responded in most cases according to the frequency of occurrence of an element in a certain position. Additionally, even if a subset of patterns was chosen in which in each position the number of elements was the same, the system exploits higher order regularities between positions for generating an output. If, for example, the lists A-B-A, A-B-B, B-A-A, B-A-B were used during training and the expression SECOND B-B-B was tested, the network produced in some replications an A as an answer and in other replications a B. A closer inspection revealed that this behavior depended on the internal representation constructed for the new sequence B-B-B. If this representation was more similar to the internal representation of the list A-B-B, the network produced a B as output; if this representation was more similar to the internal representation of the list B-A-B, the network produced an A as output. Thus, the response of the network is always based on the context in which the critical (second) element occurs, preventing the network to generalize correctly.

Evaluation of the Results: Generalization vs. Cross-Talk

The interpretation of the simulation experiments is two-fold. On the one hand, the results demonstrate a well-known strength of PDP models—to extract statistical regularities. On the other hand, they show that this property prevents the considered networks to learn the required symbol manipulation task, even if only a small subset is excluded from all possible sequences. How general is this result? Is it also valid for other LISP expressions and other symbol manipulation tasks?

To answer this question, an analysis of the network behavior is important: In order to find appropriate weights sufficient to accomplish the required input-output mappings, any regularities occurring in the input patterns are exploited during learning. The network responds to new input lists according to a similarity metric discovered between the learned input lists. The analysis of the network "generalization errors" reveal that the discovered similarity metric depends primarily on the frequency of elements in a specific position and the frequency of combinations between elements at specific positions.

Both PDP networks constructed a distributed representation of the whole input list in the hidden layer. The developed holistic representation expresses the parts—the individual elements and their serial order—in an implicit way. As opposed to the concatenative or spatial compositional representation used in the buffer approach and generally in symbolic systems, the holistic representation developed within the hidden layer is functionally compositional (van Gelder, 1990). Although this representation is sufficient to produce the parts of trained sequences as output, the obtained results show that the representation is not able to faithfully represent sequences that are constructed out of parts in a new way that goes against the learned regularities. This happens because the construction of the holistic representation in a homogenous PDP model is sensitive to the whole context in which each individual element is embedded.

This leads to the following conclusion: The construction of a holistic, functionally compositional representation exploiting statistical regularities may be a positive or negative property depending on the task at hand. As a positive effect it appears as similarity-based generalization, as a negative effect it appears as cross-talk. The chosen symbolic task requires a kind of processing without cross-talk or, equivalently, without similarity-based generalization: The system should be able to respond to elements in isolation independent of the surrounding context. It is then not surprising that one of the most important characteristics of symbolic systems corresponds exactly to this property and is called structure-sensitive processing (see Fodor & Pylyshyn, 1988). Structure-sensitive operations play an important role for the philosophy of physical symbol systems since they promise the capability for information processing based solely on structure, without any regard to content.

In summary, there are two different kinds of processing—structure-sensitive vs. context-sensitive—which may both be desirable depending on the task at hand. Furthermore, it seems impossible to have both properties at once within a single module because these properties are mutually exclusive. Context-sensitive processing is the basic or "easy" mode of PDP models explaining their parallel pattern matching abilities, whereas structure-sensitive processing is the basic mode of symbol systems explaining their capabilities for symbolic manipulations.

It seems that a major aspect of the strength of human cognition is that we are able to do both kinds of processing. Thus, it is interesting to consider how human programmers accomplish the LISP evaluation task.

SELECTIVE ATTENTION AND SHORT-TERM MEMORY

How do human programmers evaluate simple LISP expressions? Obviously, they use selective attention to segregate relevant parts out of the whole
information currently available. Selective attention allows them to serially scan the available information controlling readout of specific information for further processing.

Thus, selective attention is an important mechanism for reducing cross-talk by restricting the flow of information into central processing stages to only a subset of the available information. Treisman and Schmidt (1982), among others, demonstrate psychophysical evidence for this role of attention in visual perception. They found that when attention is distracted while a display containing several colored letters is presented, subjects sometimes reported "illusory conjunctions," wrong combinations of presented colors and letters.

Selective attention is not restricted to perceptual processes. For example, James (1890) distinguished between external and internal attention. While external attention influences the flow of sensory information, internal attention selects information currently present in short-term memory. Internal attention is thus important for allowing symbolic manipulations to occur "in our minds."

The important role of selective attention and short-term memory (plus external memories) for allowing "human symbol manipulation" becomes evident if they are related to the main property of symbol systems (Newell, 1980), that is, the fundamental distinction between processes and data. As attention selects parts of the available information, symbol systems can get and operate on data stored in a large, potentially unbounded, memory store.

Now the underlying reasons for choosing the LISP functions QUOTE and SECOND as a paradigmatic example becomes evident: If a PDP network could learn the function QUOTE on a small subset and would be able to generalize to all other possible lists (of a restricted length), then the network would have learned to behave as a simple short-term memory. Furthermore, if the network would also be able to learn the function SECOND, then the system would have learned to behave as a simple attention mechanism. Unfortunately, the homogenous PDP networks described above were unable to learn these tasks.

The simulation studies and theoretical observations lead to the following conclusions for building a neural network model possessing symbol manipulation abilities.

A more complex, modular network is required which combines elements of both types of network. As in the case of the feedforward network, at least one (visual) buffer is necessary to hold input over time. This could be accomplished by having a static external input corresponding to a piece of paper containing the LISP evaluation task. As the recurrent network, a network able to process information sequentially is required (e.g., located on top of the input buffer). The network should be able to scan various locations of the input buffer selecting partial information for further processing. Such a network would possess an attention mechanism. It could learn not only to perform direct input-output mappings, but also to learn to control the focus of attention.

SIMULATING PERCEPTION, ATTENTION, AND SHORT-TERM MEMORY WITH A NEURAL NETWORK MODEL

A neural network model called "Neuro-LISP" (NLISP) was constructed which possesses a visual attention mechanism, the ability of invariant letter and word recognition, and a simple short-term memory. The construction of these mechanisms was inspired by psychological and neurobiological findings. NLISP is a simplified version of a model specifically constructed to explain and predict various attentional phenomena (Goebel, 1993).

The Architecture of Neuro-LISP

Figure 4.4 shows the general architecture of NLISP. The input to the system is represented similarly as in the studied feedforward network: The input layer is divided into several slots, each able to hold several slot fillers. The input representation can be viewed as a simplified representation of an early visual processing stage while the eyes are fixated on a written line of text containing a LISP expression.

In contrast to the previously presented networks, NLISP is designed to handle more complex LISP expressions. A presented LISP expressions may be of various length up to maximally 25 elements. It may contain the LISP primitives, CAR, CDR, CONS, and LIST. Function calls may be embedded; however, argument lists must be flat in the current version of the model.

In order to be able to allow the representation of LISP expressions, the input module consists of 25 slots, each able to hold one of the slot fillers A, B, C, D, I, L, N, O, R, S, T, (,), and the QUOTE sign. Each individual slot filler is thus represented within a one-dimensional "retinotopic map" (see Figure 4.4). An input pattern is presented to the network by clamping on the appropriate letter units at the desired locations.

The input module projects to both the central-recognition module via the focus-of-attention layer and the absolute-position module. These two pathways are inspired by the "where" and "what" pathways found in the visual system of the brain (e.g., Ungerleider & Mishkin, 1982): The central-recognition module is able to recognize a letter or LISP primitive, but it does not know its spatial position, whereas, the absolute-position module knows the position of an object, but it does not know the shape of that object.

In addition to the bottom-up input from the input module, the focus-of-attention layer also receives top-down activation from the absolute-position
The relative-position layer is involved in specifying the relative spatial arrangement of the information currently in the focus of attention. It allows the system to copy the currently focused (selected) input region to the central-recognition module so that the relative spatial arrangement of the attended information is retained. This process establishes position-invariant recognition of letters and words. If, for example, the input CAR is present at an arbitrary input location and attention is focused on that region, the central-recognition module gets CAR as input independent of the specific position of this string at the input module (see Figure 4.4).

The relative-position layer feeds to the central-recognition module. During this computation, the information about the absolute position of attended information is lost which is, however, represented within the absolute-position module. The bottom part of the central-recognition module consists of several maps organized in the same way as the input module, but consisting of only three slots. The bottom part of the central-recognition module feeds to the letter/word-recognition layer and receives recurrent connections from that layer (see Figure 4.4). This recurrent loop is responsible for the recognition of the primitive LISP functions and individual elements.

A recognized LISP function or letter can be stored in the short-term memory which is part of the controlled-processing module. The short-term memory consists of several slots for holding suspended LISP functions and arguments. The short-term memory of NLISP (used as a "stack") is essential for its ability of recursively evaluating embedded LISP expressions.

A subpart of the controlled-processing module—the current-goal layer—plays a specific role in controlling the flow of processing by modifying its own state, by storing and retrieving information in short-term memory, and by sending commands to the attention-shift layer which determines the top-down guidance of visual attention. The current-goal layer can be regarded as the internal focus of attention holding the subtask that is currently executed.

The controlled-processing module also controls the flow of information to the output layer producing sequentially intermediate and final results. The information produced as output is supposed to be "written on a paper." Principally, the same visual processes as described earlier could be used to access the produced information for further processing. However, in the present version of the model, all intermediate results are also held in short-term memory and can be read out allowing NLISP to perform symbolic manipulations "mentally."

In the next section the performance of the network is described on a functional level emphasizing the interplay between the submodules. Then, each submodule is presented in detail, focusing on the algorithmic level, that is, how each module computes its subtask. Thereafter it is described how NLISP is able to learn some of its knowledge.
Simulation Results

The global sequence of operations performed by Neuro-LISP can be expressed as the following "algorithm":

1. The current-goal slot of short-term memory is initialized to evaluate a LISP expression which is presented at an arbitrary position of the input module.
2. Spatial attention is directed to the left side of the LISP expression to get the first LISP function.
3. The LISP function is identified and stored in the current-goal slot of short-term memory.
4. The number of arguments (one or two) of the current LISP function is retrieved from long-term memory.
5. The attention spotlight is moved to the right for evaluating each argument in turn.
6. If a QUOTE sign is found, the next LISP expression is assigned as the value of the current argument.
7. If instead of a QUOTE sign an opening brace is found, the current LISP function is stored ("pushed") in STM and steps 2-8 are executed for evaluating the embedded LISP expression.
8. If all arguments of the current LISP function are evaluated, the function is applied to the arguments. The function is either applied directly to the visual input or to arguments held in short-term memory. The resulting expressions are produced sequentially at the output layer and are also stored in the arguments-slot of short-term memory.
9. When the evaluation of the current LISP function has finished and if there are suspended functions and arguments held in short-term memory, they are now retrieved in backwards order ("popped") and applied to arguments held in short-term memory or to not-yet evaluated arguments of the visual input.
10. If the function in the current-goal slot is executed and there are no more suspended functions held in short-term memory, NLISP has finished the task.

Tables 4.1-4.3 show three examples of the major information processing steps performed by NLISP. The left column shows the presented LISP expression as well as the currently attended information (darkened region). In the next column the response of the recognition layer is shown. The next three columns show the content of short-term memory. In the current-goal slot, one unit or a set of units is activated controlling the flow of further processing. The function-stack column shows suspended LISP primitives which are

Table 4.1. The Evaluation of the LISP Expression (CAR '(A B C)).

<table>
<thead>
<tr>
<th>focus of attention</th>
<th>recognition</th>
<th>short-term memory</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>(CAR '(A B C))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(CAR (A B C))</td>
<td>car</td>
<td>eval</td>
<td></td>
</tr>
<tr>
<td>(CAR (A B C))</td>
<td>quote</td>
<td>car-apply</td>
<td></td>
</tr>
<tr>
<td>(CAR (B C))</td>
<td>a</td>
<td>car-output</td>
<td>a</td>
</tr>
<tr>
<td>(CAR (B C))</td>
<td></td>
<td>car-pop</td>
<td>a</td>
</tr>
<tr>
<td>(CAR (B C))</td>
<td></td>
<td>ready</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2. The Evaluation of the LISP Expression (CAR (CDR '(A B C))).

<table>
<thead>
<tr>
<th>focus of attention</th>
<th>recognition</th>
<th>short-term memory</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>(CAR (CDR '(A B C)))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(CAR (CDR (A B C)))</td>
<td>car</td>
<td>car-apply</td>
<td></td>
</tr>
<tr>
<td>(CAR (CDR (A B C)))</td>
<td></td>
<td>car-push</td>
<td>car</td>
</tr>
<tr>
<td>(CAR (CDR (A B C)))</td>
<td>c</td>
<td>cdr</td>
<td>c</td>
</tr>
<tr>
<td>(CAR (CDR (A B C)))</td>
<td></td>
<td>cdr-apply</td>
<td>c</td>
</tr>
<tr>
<td>(CAR (CDR (A B C)))</td>
<td></td>
<td>cdr-apply</td>
<td>c</td>
</tr>
<tr>
<td>(CAR (CDR (A B C)))</td>
<td></td>
<td>cdr-apply</td>
<td>c</td>
</tr>
<tr>
<td>(CAR (CDR (A B C)))</td>
<td></td>
<td>cdr-apply</td>
<td>c</td>
</tr>
<tr>
<td>(CAR (CDR (A B C)))</td>
<td></td>
<td>cdr-apply</td>
<td>c</td>
</tr>
</tbody>
</table>

waiting until an embedded evaluation process has finished. The argument-stack column shows the stored evaluated arguments which can be accessed by suspended functions. The final column shows the sequentially produced output of intermediate and final results.

Example 1. The first example (see Table 4.1) shows the essential steps the system performs while evaluating the LISP expression (CAR (A B C)). The simulation time proceeds from the top to the bottom of the table. At the first line the current-goal slot is initialized with the task to evaluate the presented LISP expression. To achieve this goal, the system moves the focus of spatial attention to the left side of the expression to get the first LISP
Table 4.3. The Evaluation of the LISP Expression (CONS 'A '(B C)).

<table>
<thead>
<tr>
<th>focus of attention</th>
<th>recognition</th>
<th>short-term memory</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>(CONS 'A '(B C))</td>
<td>cons</td>
<td>eval</td>
<td></td>
</tr>
<tr>
<td></td>
<td>quote</td>
<td>cons-apply-1</td>
<td></td>
</tr>
<tr>
<td>(CONS 'A 'B C)</td>
<td>a</td>
<td>cons-apply-1</td>
<td></td>
</tr>
<tr>
<td>(CONS 'A 'B C)</td>
<td>quote</td>
<td>cons-apply-1</td>
<td></td>
</tr>
<tr>
<td>(CONS 'A 'B C)</td>
<td>quote</td>
<td>cons-apply-1</td>
<td></td>
</tr>
<tr>
<td>(CONS 'A 'B C)</td>
<td>quote</td>
<td>cons-apply-1</td>
<td></td>
</tr>
<tr>
<td>(CONS 'A 'B C)</td>
<td>b</td>
<td>cons-apply-2</td>
<td></td>
</tr>
<tr>
<td>(CONS 'A (C))</td>
<td>black</td>
<td>cons-apply-2</td>
<td></td>
</tr>
<tr>
<td>(CONS 'A 'B C)</td>
<td>c</td>
<td>cons-apply-2</td>
<td></td>
</tr>
<tr>
<td>(CONS 'A 'B C)</td>
<td>)</td>
<td>cons-apply-2</td>
<td></td>
</tr>
<tr>
<td>(CONS 'A 'B C)</td>
<td>cons-pop</td>
<td>(b, c)</td>
<td></td>
</tr>
<tr>
<td>(CONS 'A 'B C)</td>
<td>ready</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

function. The system retrieves the knowledge about where to direct attention from the absolute-position module which also sets the width of the attention-spotlight initially to three elements. At the attended location, NLISP recognizes the function CAR. This information reaches short-term memory leading to the new goal to evaluate the argument of the function CAR. The system retrieves from long-term memory the fact that the attention-spotlight has to be shifted to the right and shrinked to a width of length 1 because at this position either a QUOTE sign or an opening brace may be found. In this example a QUOTE sign is recognized leading to the new current-goal to apply the CAR function to the argument. From long-term memory it is retrieved that the application of the function CAR consists of returning the first element of the input list. The first element is retrieved by moving attention two positions to the right resulting in the recognition of the Element A. According to the new current-goal, this element is produced as output and also stored temporarily in the arguments stack of short-term memory. The system has then finished the application of the function CAR. This activates the goal to "pop" possibly waiting LISP functions. Because there exist no formerly "pushed" LISP functions, the system has completed the posed task.

Example 2. The second example (see Table 4.2) shows the evaluation process of the compound LISP expression (CAR (CDR '(A B C))). As in the previous example, the first LISP function is retrieved, which is again the function CAR. Attention is then moved to the right. Instead of finding a QUOTE sign, an opening brace is recognized, indicating that an embedded expression is to be evaluated. Therefore, the current LISP function is stored ("pushed") in short-term memory, waiting there until the embedded expression is evaluated. Then, attention is focused to the right side of the opening brace resulting in the recognition of the function CDR. For evaluating this function the spotlight is again moved to the right. Next, a QUOTE sign is recognized, activating the goal to apply the function CDR to its argument and to produce as output. The first element produced as output is an opening brace because the output must be a list, namely, the argument list without its first element. Attention is moved next to the second element of the list, and the Element B is produced as the next output. The system proceeds in moving attention sequentially to the right producing each recognized element as output until the closing brace of the argument list is found. Then the goal to apply the current function is completed. A "pop" operation is initiated next which retrieves the suspended function CAR. Because a pop operation was executed, the next goal is to apply the retrieved function not to the visual input but to the list held in the arguments slot of short-term memory. Thus, the first element—B—of the stored list (B C) is retrieved and finally produced as output.

Example 3. The third example (see Table 4.3) shows the processing of the expression (CONS 'A '(B C)). The system starts again by retrieving the LISP function on the left side of the presented LISP expression. Although the spotlight embraces only the three elements C,O,N, this is sufficient to recognize the function CONS. From long-term memory it is retrieved that this function has two arguments that must be evaluated in turn. The system starts to evaluate the first argument, moving attention to the right. Because it finds a QUOTE sign, it pushes the first argument, the Element A, into short-term memory. In order to evaluate the second argument, the focus of attention is moved to the right where again a QUOTE sign is found. Since at this point both arguments of the function CONS are determined, the function can be applied to these arguments. In order to construct the first part of the resulting list, the first argument, A, held in short-term memory is retrieved, and the first three elements of the output list (A, and )—(specifying a blank position)—are produced. Then the second argument at the input module (B C) is used to determine the final part of the output list. Therefore, the system sequentially scans the elements of that list until its closing brace is reached. During this process each encountered element is produced as output. Finally, the CDR function is "popped" completing the evaluation task since no additional functions are waiting in short-term memory.
Computational Details

After having described what subtask each module of NLISP performs, the internal structure of the modules are now described explaining how they compute the desired input-output mappings.

*The input module.* The architecture of the input module was described earlier. The units of the input module do not compute anything but are clamped simply to the values 0 or 1 indicating the absence or presence of a letter at a certain absolute “retinal” position (see Figure 4.5). The values of the input units do not change while performing a LISP evaluation task.

*The focus of attention layer.* The focus of attention layer is a one-dimensional array of linear units that simply add the incoming bottom-up and top-down activation at each position (see Figure 4.5). For present purposes, no interactions of the units are necessary. The units have a linear activation function clipped by 0 and 1. Each unit gets bottom-up activation from all units of the input module at the same position (slot). The weights are set to 0.1 so that the attention layer units respond with this small value if an input unit at the corresponding input position is active.

Additionally, each unit eventually gets top-down activation if the higher level system “decides” to attend to certain positions. There are 1-to-1 connections with strength 1.0 from the selected-positions layer to the focus layer. Thus, an activated selected-positions unit forces the corresponding focus-of-attention layer unit to be fully activated.

*The relative-position layer.* The relative-position layer plays an important role for achieving position-invariant letter and word recognition. A simple way to achieve position-invariant letter recognition would be to have a set of higher level units with each responding to one letter present at an arbitrary position of the input module. Thus, these units would have a large receptive field covering the whole input space. This would imply that the higher level units would lose the knowledge of the absolute position of the recognized letter.

However, if one extends this approach to strings of letters, a severe problem would be that the spatial arrangement of the elements within the spotlight of attention would be lost. As an example, consider the case where the word CAR is presented somewhere at the input module and attention is focused on that word. According to the outlined approach, the higher level units C, A, and R would be active, but the knowledge about the serial order of these letters would be lost. To overcome this problem, the task of the relative-position layer is to retain the spatial arrangement of the information within the spotlight of attention while abstracting at the same time the absolute position of the attended information.

This is performed by representing the position of an attended letter relative to the focus of attention. Therefore, the relative-position layer uses two units for representing each letter in a specific absolute position (see Figure 4.6). The left unit of each pair represents that an element possesses a location on the left side within the attended region, the right unit represents that an element falls on the right side. The weights from the input module and from the focus-of-attention layer copy an attended element to the left or right unit of the corresponding slot of the relative-position module. If an element falls in the center of the attention spotlight, both the left and right unit of the corresponding slot of the relative-position module are activated (see Figure 4.6). Overall, the relative-position layer computes an intermediate result which contains both the spatial arrangement of the elements within the attended region and also its absolute position. At the next processing step, the absolute position information is discarded: The input layer of the central-recognition module consists of three slots with large receptive fields. Each unit of a slot detects a specific element at any absolute position, thus discarding absolute position information. However, a unit responds only to a specific relative position represented at the relative-position layer, thus retaining the information of spatial arrangement that finally allows the position-invariant...
recognition layer back to the input layer allow completion of missing information and the suppression of noise (produced, e.g., by slightly activated units outside the focus of attention).

The absolute-position module. During the computation of an invariant representation of the attended information, the knowledge about its absolute position is lost. The absolute-position module (see Figure 4.7) is responsible for retaining the absolute position of a presented LISP expression and the absolute position of the focus of attention. Based on this knowledge, the absolute-position module performs appropriate relative movements of the attention spotlight which are demanded by the currently active goal.

Each unit of the bottom layer—the active-positions layer—gets input from all units of the corresponding slot of the input module. Because all weights from all units of an input slot to the corresponding unit of the active-positions layer are set to 1, a unit of the active-positions layer is fully activated if any unit of the corresponding input slot is active. Thus, the units of this layer represent the information where an element exists, but they do not know what kind of information is present at that position.

Figure 4.6. The relative-position layer: Translation-invariant perception

recognition of the attended information. An extension of the outlined approach to invariant perception which allows position- and scale-invariant perception of two-dimensional input stimuli is described in (Goebel, 1993).

The central-recognition module. Through the process described earlier, the central-recognition module gets as input the elements currently in the focus of attention (see Figure 4.6). This information feeds to the recognition layer which has a local unit responding to a single letter or a LISP primitive (e.g., CAR, CDR, CONS, QUOTE, A, B, etc.). Recurrent connections from the

Figure 4.7. The absolute-position module: Moving the focus of attention
The weights to the next layer of the absolute-position module—the selected-positions layer—compute the leftmost active unit of the presented expression, which then serves as the anchor point for subsequent relative movements of the focus of attention. The leftmost unit is determined by a simple wiring scheme from the active-positions layer to the selected-positions layer. As described earlier, the activity of the selected-positions layer determines directly which units of the focus-of-attention layer will be active since the selected-positions layer projects to the focus-of-attention layer.

Starting with the anchor unit, the next desired position and size of the spotlight is determined by the activity of the attention-shift layer which itself receives its commands from the current-goal layer of the controlled-processing module (see later).

The attention-shift layer consists of two parts: one part representing the desired width of the attention spotlight, and one part representing the desired number of positions for shifting the focus of attention relative to its old position. This part consists of a linearly ordered set of units (see Figure 4.7). If a unit on the right or left side relative to the central unit is active, the focus of attention will be shifted for the appropriate number of positions in the right or left direction, respectively. This is achieved by first computing the leftmost active unit of the current spotlight and then performing a summation-like operation which adds the desired number of position changes to the absolute current position of the attention focus (see Figure 4.8a). This sum is represented as an analogue (thermometer) code at the compute-focus layer. Recurrent connections from this layer to the selected-positions layer transform the analogue code back to a local code activating the unit of the new position of the attention spotlight. Finally, depending on the desired width of the attention focus, the appropriate number of additional selected-positions units are activated (see Figure 4.8b).

The controlled-processing module is responsible for storing and retrieving information in short-term memory and for guiding the flow of further processing determined by the currently active goal unit(s).

Short-term memory. The short-term memory of NLISP consists of distinct parts for storing LISP functions and arguments. Because these stores have a similar internal organization, only the short-term memory for LISP functions is described in detail (see Figure 4.9 and 4.10).

Short-term memory is represented as sustained activity over time (cf. Goldman-Rakic, Funahashi, & Bruce, 1990). Once activated, a current-goal unit as well as a short-term memory unit will stay active over time through a self-connection with weight 1. The part of short-term memory which stores LISP functions operates like a stack: A given LISP function can be either pushed into memory or the last previously stored function can be retrieved (popped) from memory. There are two major subparts of the memory store. The left subpart, containing the content-units, is responsible for holding stored information, and the right subpart, containing the retrieval-unit, is used during the retrieval of stored items. The short-term memory is used either by activating the push-unit (see Figure 4.9) or the pop-unit (see Figure 4.10). When the push-unit is activated, a LISP function is supplied to the function-entry layer, the LISP function is stored into the next free slot of the memory that is provided by the currently active pointer-unit depicted on the left side of the short-term memory (see Figure 4.9a and 4.9b). In terms of symbol systems, the pointer units can be regarded as providing memory addresses, whereas the corresponding content-units represent the content stored at a particular address.

The thresholds of the content-units are set so that only the combined activity of the active push unit, the active pointer unit, and the active function-entry unit is sufficient to exceed its threshold. Once activated, the self-excitation of a content-unit keeps it active over time. A push operation is
completed by "incrementing the pointer" (see Figures 4.9b and 4.9c) through the combined activity of the push-unit and the formerly active pointer unit. When the pop-unit is activated, the decrement-units located on the left side come into play (see Figure 4.10). The combined activity of the popunit and the active pointer unit is sufficient to exceed the threshold of the appropriate decrement-unit (see Figure 4.10a). The activated decrement-unit serves three purposes (see Figure 4.10b). First, it gates the flow of activation from the content-units of the current address to the corresponding retrieval-units. Second, it deletes the information held by the content-units of the current address, and finally, it decrements the pointer. The information represented in the activated retrieval units is finally copied to the function-exit layer (see Figure 4.10c) from where it can be accessed for further processing.

Long-Term Memory. The current-goal layer possesses a central position within NLISP because it receives from and projects to many subcomponents of the system (see Figure 4.11). It receives connections from the central-recognition module, it receives recurrent connections from itself, it receives and projects to all short-term memory stores, and it projects to the output layer and to the attention-shift layer. The weights of these connections represent the task-specific long-term memory knowledge of NLISP.

The operation of the controlled-processing module can be described in symbolic terms as a goal-oriented production system (e.g., Anderson, 1983; Pirolli, 1986) executing rules of the form

\[
\text{IF} \quad (<\text{current-goal}> \text{ and } <\text{perception}>)
\text{THEN} \quad (<\text{new-current-goal}> \text{ plus}
\text{attention-commands} \text{ plus}
\text{short-term-memory commands} \text{ plus}
\text{output}>
\]

The whole "rule-set" of NLISP is shown in Table 4.4. The condition part of the rules are "matched" in parallel since all current-goal units receive at the same time the activity from the central-recognition module and other current-goal units. The current-goal unit receiving the highest activation becomes active; "a rule fires." The active current-goal unit then triggers all commands of the action side simultaneously by the parallel flow of activation to the different subsystems. This "recognize-act cycle" is started with the initialization goal "evaluate expression" and proceeds until the presented LISP expression is evaluated completely.

The next section describes how NLISP can acquire itself some of its knowledge, especially the weight values defining the described rule set.

Figure 4.11. The controlled-processing module: Controlling the flow of processing

Learning

NLISP is a combination of a prewired and adaptive system. The general design principle of NLISP is that the overall architecture as well as the lower processing levels are handwired, constituting "innate" knowledge. As one moves on to higher levels, the system becomes more and more flexible, able to learn by experience.

The system is able to learn within the central-recognition module, within the absolute-position module, and within the controlled-processing module. The weights for producing translation-invariant processing and the weights of short-term memory are prewired. The learning process was performed hierarchically; the central-recognition module and the absolute-position module must have finished learning before the task-specific knowledge within the controlled-processing module can be acquired.
Table 4.4 The "Rule-Set" of NLISP Controlling the Flow of Processing.

<table>
<thead>
<tr>
<th>current-goal</th>
<th>recognition</th>
<th>new current-goal</th>
<th>attention commands</th>
<th>STM commands</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>eval-expression</td>
<td>car</td>
<td>car</td>
<td>right 1: range 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval-expression</td>
<td>quote</td>
<td>quote</td>
<td>right 4: range 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval-expression</td>
<td>car-eval</td>
<td>car-eval</td>
<td>right 2: range 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval-expression</td>
<td>car-apply</td>
<td>car-apply</td>
<td>right 3: range 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval-expression</td>
<td>car-pred</td>
<td>car-pred</td>
<td>right 3: range 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval-expression</td>
<td>(</td>
<td>eval-expression</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval-expression</td>
<td>cdr</td>
<td>cdr</td>
<td>right 4: range 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval-expression</td>
<td>quote</td>
<td>quote</td>
<td>right 5: range 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval-expression</td>
<td>car-eval</td>
<td>car-eval</td>
<td>right 6: range 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval-expression</td>
<td>car-apply</td>
<td>car-apply</td>
<td>right 7: range 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval-expression</td>
<td>(</td>
<td>eval-expression</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval-expression</td>
<td>cons</td>
<td>cons</td>
<td>right 8: range 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval-expression</td>
<td>car-eval-arg1</td>
<td>car-eval-arg1</td>
<td>right 9: range 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval-expression</td>
<td>cons-apply-arg1</td>
<td>cons-apply-arg1</td>
<td>right 10: range 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval-expression</td>
<td>cons-apply-arg2</td>
<td>cons-apply-arg2</td>
<td>right 11: range 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval-expression</td>
<td>cons-apply-arg3</td>
<td>cons-apply-arg3</td>
<td>right 12: range 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval-expression</td>
<td>eval-expression</td>
<td>eval-expression</td>
<td>right 13: range 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval-expression</td>
<td>list</td>
<td>list</td>
<td>right 14: range 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval-expression</td>
<td>quote</td>
<td>quote</td>
<td>right 15: range 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval-expression</td>
<td>car-eval-arg1</td>
<td>car-eval-arg1</td>
<td>right 16: range 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval-expression</td>
<td>car-eval-arg2</td>
<td>car-eval-arg2</td>
<td>right 17: range 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval-expression</td>
<td>car-eval-arg3</td>
<td>car-eval-arg3</td>
<td>right 18: range 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval-expression</td>
<td>list</td>
<td>list</td>
<td>right 19: range 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval-expression</td>
<td>quote</td>
<td>quote</td>
<td>right 20: range 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval-expression</td>
<td>car-eval-arg1</td>
<td>car-eval-arg1</td>
<td>right 21: range 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval-expression</td>
<td>car-eval-arg2</td>
<td>car-eval-arg2</td>
<td>right 22: range 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval-expression</td>
<td>car-eval-arg3</td>
<td>car-eval-arg3</td>
<td>right 23: range 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval-expression</td>
<td>list</td>
<td>list</td>
<td>right 24: range 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval-expression</td>
<td>quote</td>
<td>quote</td>
<td>right 25: range 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval-expression</td>
<td>car-eval-arg1</td>
<td>car-eval-arg1</td>
<td>right 26: range 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval-expression</td>
<td>car-eval-arg2</td>
<td>car-eval-arg2</td>
<td>right 27: range 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval-expression</td>
<td>car-eval-arg3</td>
<td>car-eval-arg3</td>
<td>right 28: range 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval-expression</td>
<td>list</td>
<td>list</td>
<td>right 29: range 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Learning to recognize letters and words. The central-recognition module was trained to respond to the relevant letters and LISP primitives. Therefore, a desired input is presented (and attended to), and the appropriate target values are injected at the recognition layer. The system learns the weights in both directions of the two layers of the central-recognition module using the back-propagation learning rule (see earlier in the second section).

Due to the (handwired) translation invariant processing ability, the system automatically generalizes the knowledge acquired at one input position to all other input locations. Thus, if the system has learned to recognize the LISP function CAR at a specific location, it will recognize CAR also at all other possible input positions.

Learning to move the attention spotlight. In the presented version of the model, the absolute-position module uses a hardwired connection scheme. However, in an alternative model, it was tested whether appropriate weights could be learned using the back-propagation learning rule. As input, a desired relative movement and width of the attentional focus was supplied to the attention-shift layer. Additionally, a randomly selected absolute position of the attention spotlight was supplied to the selected-positions layer. At the next processing cycle, target values, defining the desired new absolute position and width of the attention spotlight were injected at the selected-positions layer. Using this procedure, the network developed appropriate weights to compute the new focus of attention. However, generalization performance was rather poor so that almost all possible patterns had to be included in the training corpus.

Learning task-specific knowledge. The prewired and learned weights described so far constitute the fundamental knowledge of NLISP. Based on this knowledge, the task-specific knowledge within the controlled-processing module can be acquired.

The "rules" described earlier are acquired by associating the currently perceived information and the actual state of the current-goal layer with the next state of the current-goal units, with short-term memory commands, with commands for moving the focus of attention, and with appropriate output values. In order to have a more flexible system, the units of the current-goal layer should be allowed to develop its own representation, but for simplifying the learning task, arbitrary units were selected for locally representing specific (sub-) goals.

At each time step, appropriate target values (including "don't care values") were supplied to the current-goal units, the attention-shift units, the push and pop units, and the output units. The specification of these target-values is assumed to be derived by reading verbal instructions and by analyzing examples. Although NLISP does not include these understanding processes, it makes specific assumptions about the required results of such processes for successful learning behavior. If, for example, a student learns what the LISP function CAR is doing, an instruction might be: "The function CAR returns the first element of its (evaluated) argument list." In terms of NLISP, the student must transform this sentence into appropriate movement commands for his (covert and overt) attention system in order to find and select the first element of the argument list. Similarly, if he detects an embedded LISP
expression instead of a QUOTE sign, he has to store in short-term memory that the CAR function has to be applied later.

This analysis seems also to reveal one reason why being told by a teacher is more helpful than by being told by a textbook alone: If a teacher gives instructions and presents examples, he also points to relevant positions of written material. Thus, the student gets helpful cues about where to focus his attention identifying the currently relevant information (e.g., the first element of a LISP list). If the student gets only instructions without examples (cf. Schmalhofer, Kühn, & Boschert, this volume), he must derive appropriate attention commands on his own. If he gets only examples with only some general instructions, he has the task to infer which parts of the examples are relevant, that is, again where he has to focus spatial attention. For example, a student might discover that in the example — (CAR '(A B C)) — A — the element on the right side is the same as the element at the first position of the argument list. The student then could test this hypothesis by looking at other examples.

DISCUSSION

Through analyzing homogenous feedforward and recurrent PDP networks, the lack of selective attention and short-term memory was identified as the major reason for the difficulties of PDP networks with symbol manipulation tasks. Selective attention allows the segregation of parts for further processing out of the whole information currently available. In terms of the concepts used to characterize symbol systems (cf. Fodor & Pylyshyn, 1988), selective attention establishes structure-sensitive processing, whereas spatial buffers, and short-term memory establish (concatenative) compositionality. Since selective attention — in combination with appropriate invariant perceptual mechanisms — allows one to segregate arbitrary parts of a presented input which may then be processed by central processing stages, the fundamental distinction, inherent in symbol systems, between data and processes (Derrick & Plaut, 1986) is achieved.

Based on these conclusions, a hierarchical, modular neural network model (NLISP) was developed in order to prove the viability of these considerations. Simulation studies demonstrated that the developed system is able to perform a typical symbol manipulation task — the evaluation of simple LISP expressions. The model performs this task by shifting attention to relevant positions of a presented LISP expression segregating specific parts (e.g., the first element of an argument list). It was also described how the model can learn some knowledge on its own, especially task-specific LISP knowledge. As a psychological model, NLISP sheds light on the important role of basic cognitive processes involved in elementary programming behavior. In contrast to other models of elementary programming, NLISP is an attempt to ground higher level cognition in perceptual processes such as visual perception and spatial attention. This seems to be an important goal because there is much evidence that high-level cognitive processes use some of the same mechanisms involved in perception (e.g., Farah, 1988; Lakoff & Johnson, 1980).

Although the developed model focuses on establishing symbol manipulation capabilities, it is not just an implementation of symbol processing systems because the information within the focus of attention is processed in parallel according to the principles of PDP models. Besides establishing symbol manipulation, selective attention adds a powerful mechanism to PDP models which allows pattern matching, pattern completion, and learning to occur within an arbitrary subset of the whole presented information. Therefore, selective attention also improves generalization performance of PDP models because it helps to detect known patterns within the whole presented input which otherwise may lead to uninterpretable responses due to cross-talk effects.

Although NLISP succeeded in producing the desired behavior it has several drawbacks in its present state. For example, most modules use local representations, and short-term memory is constructed in a way specific for the LISP evaluation task. Although short-term memory is able to store distributed representations, the "symbols" are stored in special areas totally disconnected from the rest of the system. A more complex but very flexible approach to short-term memory exploiting distributed representations (Goebel, 1999b) will be incorporated into the model as well as a more elaborate two-dimensional visual attention system (Goebel, 1993). Then, the model will be tested on more complex problem-solving tasks requiring both symbol manipulation and PDP-like capabilities.

Selective attention is often integrated into "box-and-arrow" models as a "homunculus" controlling the operation of information processing modules (Neumann, 1990). The presented model, as well as other connectionist models (Mozer, 1991; Sandon, 1990), show that (spatial and internal) selective attention can be established with the same simple network interactions that are used for other kinds of information processing (e.g., pattern recognition). The presented model even demonstrates that these simple "associational" processes are sufficient to provide structure-sensitive processing.

In summary, the proposed modeling approach may be helpful for understanding human intelligence by assigning a key role to selective attention, namely, to allow — together with translation-invariant perception and short-term memory — the combination of sequential structure-sensitive operations with parallel distributed processing.
REFERENCES


