CuPit — A Parallel Language for Neural Algorithms: Language Reference and Tutorial

Lutz Prechelt (prechelt@ira.uka.de)
Institut für Programmstrukturen und Datenorganisation
Universität Karlsruhe
76128 Karlsruhe, Germany
++49/721/608-4068, Fax: ++49/721/694092

January 20, 1994

Technical Report 4/94

Abstract

CuPit is a parallel programming language with two main design goals:

1. to allow the simple, problem-adequate formulation of learning algorithms for neural networks with focus on algorithms that change the topology of the underlying neural network during the learning process and
2. to allow the generation of efficient code for massively parallel machines from a completely machine-independent program description, in particular to maximize both data locality and load balancing even for irregular neural networks.

The idea to achieve these goals lies in the programming model: CuPit programs are object-centered, with connections and nodes of a graph (which is the neural network) being the objects. Algorithms are based on parallel local computations in the nodes and connections and communication along the connections (plus broadcast and reduction operations). This report describes the design considerations and the resulting language definition and discusses in detail a tutorial example program.

Contents

List of Figures

List of Tables

Part I: Introduction

Section 1 sets the stage. Section 2 sketches why CuPit has been designed and what it is good for. Section 3 lists what the language design goals were and which aspects have deliberately been taken out of consideration. Section 4 gives a superficial overview of the design. Section 5 contains a few hints on how to use this report.

Section 6 contains a detailed discussion of a tutorial example program and can be read first if the reader wishes to skip sections 2 to 4.

The rest of the report describes the language design in detail and serves as (a) language reference manual, (b) language user manual, and (c) source program for the scanner and parser of the CuPit compiler. In order to fulfill purpose (c), the report contains a complete syntax specification of CuPit in the notation of the Eli [GHL+92] compiler construction system. This specification is typeset with FunnelWeb [Wil92] and can automatically be extracted and used to generate the front-end of the CuPit compiler.

1 Purpose and scope of the language

Before anything else, it is necessary to describe what the scope of CuPit is. The purpose of this description is merely to give an idea what the rest of this report is all about; I will thus give a fuzzy description here only and complement it with some counterexamples.

First of all, CuPit is a language for programming *neural algorithms*. The purpose of a neural algorithm in the sense of CuPit is *learning by example*. This means that a neural algorithm processes a *set of examples*; usually each example input is processed several times. Note that *using* the learned structure can be seen as a special mode of learning and is thus subsumed here. The goal of the learning process is to construct an automaton with certain properties, which are defined by the neural algorithm (e.g. approximate a certain mapping on the examples with a given precision).

The central data structure of a neural algorithm is the *(neural) network*, a directed graph of *nodes* and *connections*. The purpose of the neural algorithm is to modify this graph by adding or deleting nodes and connections and by changing the internal states of the nodes and connections. The result of the neural algorithm's computation is the structure of the graph and the internal states of its nodes and connections.

The operations in neural algorithms are triggered by a central agent (the main program) and can roughly be divided into the following classes:

1. local operations in a node or connection,
2. local operations in the central agent,
3. transportation of data from one node to another through an existing connection,
4. combining data from many nodes or connections into a single value (reduction operations),
5. creation or deletion of nodes or connections,
6. organizational stuff, e.g. accessing the examples.

The operations on nodes and connections can be applied in an object-parallel fashion and are the main source of parallelism.

The typical overall structure of a neural algorithm is

1. Look at n examples.
2. Modify the network.
3. Begin again (using the same or new examples), unless some stop criterion is satisfied.

The value of n can be 1 or the number of examples available or somewhere in between. If it is possible to look at all of those n examples independent of each other (i.e., $n > 1$), this is another important source of parallelism.

Among the things *not* possible in a neural algorithm are

1. Arbitrary interaction of data from different nodes or connections (including non-local pointers).
2. Arbitrary threads of parallel execution (including arbitrarily nested parallelism).
3. Arbitrary distributed data structures.

2 Motivation

While both, the field of parallel computation and the field of neural networks, are booming, relatively little is done in the field of parallel neural network simulation. Only a few specialized implementations of individual popular learning algorithms exist and even fewer attempts to provide parallel implementations of more flexible neural network simulators have been made. Even the existing programs are usually quite limited in scope. They are targeted at people that apply neural networks, but are not of much value for people that explore new neural network learning methods, since only well-known learning methods are implemented and the programs are not easily extendible.

The reason for this situation is the lack of compilers for programming languages that allow easy implementation of such software on parallel machines in a portable fashion. Most currently implemented parallel programming languages force the programmer to bother with the communication topology and number of processors of the machine at hand, requiring much work for the mapping of data and algorithm onto a certain machine in addition to the work required for the algorithms themselves.

Those languages that try to overcome these programming difficulties have inherent efficiency problems. Very sophisticated optimization methods, which have not been developed today, will be necessary to overcome these problems — if it is possible at all. Other languages do a somewhat halfhearted job by solving some aspects of the efficiency problem, but taking only part of the mapping work off the programmer.

It looks as if parallel machines suggest the use of domain-specific languages much more than sequential machines do: To exploit domain-specific constraints in order to improve the optimization capabilities of the compiler seems to be necessary if we want to compile problem-oriented languages into programs that run as fast as equivalent programs that are written in a machine-oriented language. At least it may be much *simpler* to write good optimizing compilers for a domain-specific language than for a general-purpose language.

Neural algorithms have two properties that make them a particularly rewarding target for a domain-specific parallel language:

- They usually have a strong and well-known computational structure (regularity) that can be exploited by a compiler.
- Their runtime on sequential machines is so long even for moderately sized problems that using parallel machines is an urgent need.

3 Language goals and non-goals

CuPit is designed and implemented as part of a Ph.D. thesis. The focus in that thesis is compiler technology: A data mapping for irregular neural networks on massively parallel machines that achieves load balancing and data locality at once. This means that the goals of the language design are chosen as to open new perspectives on the design of parallel languages that can be compiled efficiently and at the same time will neglect some important properties of programming languages in general. This is necessary in order to keep the design and implementation of the language feasible. Those aspects of a language that are usually considered important but are neglected here are explicitly listed as non-goals.

3.1 Goals

The following properties are central to the design of CuPit and mark the difference in comparison to most existing parallel programming languages in respect to the implementation of neural algorithms.

1. Adequateness. The language has constructs that allow to express those operations that are usually needed to specify neural algorithms easily and in a compact and understandable form.
2. Efficiency. Properly written CuPit programs contain enough structure for a compiler to exploit in order to compile into efficient code for various kinds of parallel machines, even — and this is the most important point — for irregular networks.
3. Flexibility. While the scope of CuPit places strong restrictions on what can be implemented with it, no arbitrary additional constraints are enforced *within* this scope.
4. Portability. Those properties of the target machine that can not be expected to be identical or very similar on other possible target machines shall be completely hidden in CuPit. These are number of processors, interconnection topology, SIMD/MIMD mode, message routing, memory management, etc.

3.2 Non-Goals

The following properties are explicitly *not* targeted by the language design presented in this report. These properties are usually considered necessary in order to achieve maximum ease of writing, reading, porting, and debugging programs. On the other hand, they are relatively well-understood after decades of research in sequential programming languages and can later be added to the design if the overall design proves to be useful in respect to its main goals. The reason for leaving these aspects out of the design is that they would result in much more additional work to design and implement them properly than additional benefit to the language for my research purposes.

1. Fine-tune portability. For perfect portability, a language must provide exact definitions of the behavior of arithmetical operations, sizes and layout of data objects, etc. CuPit takes a "we do just what this machine usually does" approach instead.
2. Handyness in all aspects. All aspects of a language should be elegant, compact, handy, and convenient to use. CuPit allows itself to be a bit clumsy and inconvenient in some respects, where these are merely technical deficiencies that are not too difficult to to repair and that do not impair the central goals of the language design. For example, CuPit forces "defined before applied".
3. Debugging support. CuPit does not allow for easy observation or manipulation of what is going on while a program is running.
4. Object-orientedness. In procedural languages, facilities for object-oriented modeling with inheritance are becoming more and more standard. While CuPit has some constructs that support an object-centered way of programming (and these are in fact closely related to the key ideas of CuPit's design), mechanisms for inheritance and dynamic dispatch have been left out of the language completely.
5. Genetic algorithm support. No attempts have been made to support working with populations of networks in a genetic algorithm fashion.
6. Hybridism. There is no special support in CuPit for combination of neural algorithms with other algorithms that do not obey the neural algorithm program model (e.g. that require direct access to arbitrary data). It is quite difficult to define such capabilities without losing much of the domain-specific constraints that are so useful in CuPit for efficient implementation.

3.3 General purpose languages versus domain specific languages

The general requirements for general purpose languages differ somewhat from those for domain-specific languages. For a programming language to be useful, it must make it easy for programmers to learn and use it. For high-level general purpose programming languages, the following overall properties are required to achieve ease of use:

1. Orthogonality of language constructs, i.e., the ability to combine all constructs that are combinable conceptually.
2. Consistency in spirit of language constructs, i.e., the presence of one and only one "programming model" that can explain all constructs[1].
3. Simplicity of language constructs and their interaction.

For domain specific languages, orthogonality and simplicity are not absolutely needed, because the domain metaphor guides the programmer and can make lateral restrictions or complicated semantics of constructs appear quite natural, and thus acceptable. Consistency in spirit, on the other hand, is extremely important in this case, in order to make the implementation of the domain metaphor clear to the programmer.

4 CuPit Overview

This section will give an overview of how the individual goals of the design are realized: what kinds of language constructs are present and how they are integrated into an overall structure.

4.1 Overall language structure

The overall structure of CuPit is mostly the same as for any procedural language: There are definitions of types and data objects, definitions of procedures and functions, and statements that define actions and control flow. What is different in CuPit is

1. There are special kinds of data types (nodes, connections, networks) that exploit the restrictions CuPit imposes on programs. This allows to make the neural network data structure that underlies the program more explicit.
2. There are some statments and expressions doing unusual things; especially calling a procedure or function for a group of objects in parallel. This allows to restrict the forms of parallel operations that can occur, which in turn allows for better optimization of the program on a parallel machine.

4.2 Special declarations

In addition to the usual records and arrays, there are three special categories of data types in CuPit that are used to describe parallelism and allow problem-oriented programming of neural algorithms:

1. Network types that describe arrangements of nodes and connections.
2. Node types that describe the structure of what is usually called units or neurons.
3. Connection types that describe the structure of what is usually called connections or weights.

All Connection, Node, or Network data types consist of a record data type plus some additional functionality; this makes nodes and connections in CuPit much more general than they are in most other neural network simulation software. These three kinds of data types are used to introduce three kinds of parallelism into CuPit: parallel execution of operations on sets of connections (including reduction), parallel execution of operations on sets of nodes, and parallel execution of operations on replicated exemplars of a whole network. Most of the special statements and expressions that CuPit introduces are related to these three categories of data types.

These special categories of data types also motivate the name of the language: CuPit is named after Warren McCulloch and Walter Pitts, who published the first description of a formal neuron in 1943 [MP43].

[1] Or at least: that explains most constructs and is not broken by the others

4.3 Program structure and parallelism

The overall structure of a CuPit program is as follows: The main program consists of a procedure that is executed sequentially. It may call other sequential procedures or functions, too. This part of the program is called the *central agent*, because it activates all parallel operations that can occur. The central agent may declare one or more variables of network type and call procedures that operate on them. These procedure calls can activate one of the four levels of parallelism that are possible in CuPit:

1. A network procedure can call a node procedure for a whole group of nodes of the network in parallel.
2. This can also be done for several groups (with different node types) in parallel.
3. A node procedure can call a connection procedure for a whole group of connections coming into the node or going out of the node in parallel.
4. The whole network can be replicated to execute on several input examples in parallel.

In addition, there are a number of special statements that allow dynamic changes in the topology of a network. These statements imply similar kinds of parallelism.

4.4 Input and output

Since there is no agreement today about how parallel I/O should be defined on various kinds of parallel machines, it has been left out of CuPit: I/O has to be performed by calling external procedures. This means that in order to achieve full portability of CuPit programs, a standard I/O library has to be defined. Since the data distribution to be used for the actual data of the neural network is the secret of the compiler and may change often, we need an additional interface structure: CuPit defines a sort of variables called I/O objects that have a well-defined memory layout. These variables are used for communication between the CuPit program and external program parts.

4.5 Kinds of semantic errors

There are three kinds of semantic errors in CuPit:

1. Errors in static semantics that will be detected by the compiler. Anything that is said to be *illegal, not allowed*, or the like and is not explicitly tagged to belong to (2) or (3), belongs into this category.
2. Run-time errors. The program may or may not detect the occurrence of a run-time error when the program executes. The compiler may provide options to turn checking for certain kinds of run-time errors on or off. It is allowed for the compiler to flag a run-time error already at compile-time, if it will occur in every run of the program where the statement or expression in question is executed. Typical kinds of run-time errors are arithmetic overflows, out-of-memory errors, out-of-range array indexing, and so on.
3. Undetected semantic errors. The language manual defines certain behaviors of a program that are illegal, but will not necessarily be detected by the compiler (usually, because it is not possible to do so). For instance, a program may not rely on a certain kind of implementation for a language construct to be used by the compiler unless the manual explicitly states that this implementation will be used. A program that contains this kind of error is called *erroneous*. The behavior of an erroneous program is completely undefined.

This definition of error categories is similar to that used in Ada [Uni81].

5 About this report

This report does not contain any kind of complete formal description of the CuPit semantics. Instead, it partly relies on the intuitive understanding of the CuPit semantics, based on the reader's (assumed) familiarity with other programming languages.

Various techniques are used in order to informally describe the semantics of language constructs on different levels of detail, preciseness, and "intuitiveness"; these techniques are not orthogonal:

1. natural language description
2. example application
3. case discrimination
4. equivalent CuPit program fragments
5. reference to well-known other programming languages, especially Modula-2 [Wir85], Ada [Uni81], and C [KR77]

Given these techniques of description, it is impossible to guarantee that the language definition is precise enough and complete enough to be unambiguous for practical purposes. If you find any aspects of the language that are not made sufficiently clear in this report, please contact the author.

Most of the language definition is given in bottom-up order. This approach was chosen, because those aspects of CuPit that discriminate it most from ordinary procedural languages are based on the connection, node, and network types; thus these kinds of type definitions have to be introduced first.

A tip for reading: You may find it useful to consult the index at the end of this report in order to find certain aspects of the definition more quickly. Index entries that have a page number in *italic* type indicate pages that contain a definition or description of the entry.

Wta.

6 Tutorial

This section aims to introduce the concepts and constructs of CuPit using a tutorial example. This introduction serves two purposes: First, an aspiring CuPit programmer shall learn what language constructs are available and how to use them. Second, the example shall explain what the major design decisions in the design of CuPit were and why they have been made the way they have.

We will proceed in three steps: First the example CuPit programming problem will be introduced; we will use the well-known backpropagation algorithm with two extensions as the example. Second, we will discuss what properties a description of a solution to this problem (i.e. a program for it) should have in order to be problem-adequate and what the major design decisions in CuPit were in order to introduce adequate language constructs. Third, we will step by step introduce the actual CuPit program for the backpropagation algorithm.

In this section, I assume some (but only little) basic knowledge about neural network terminology and about the way the backpropagation learning algorithm works.

6.1 The problem

6.1.1 Overview

The problem we will use to discuss the concepts and constructs of CuPit is a neural network learning algorithm known as "the back-propagation of error", usually just called backpropagation or backprop. The original reference to backpropagation is probably [Wer74], the most often-cited one is chapter 8 of [RM86], which also contains a quite nice mathematical derivation of the algorithm. We will extend the original algorithm by the RPROP rule for the computation of learning step sizes at each weight [RB93] and by a simple connection elimination scheme.

Backprop is a so-called supervised learning algorithm. This means that it learns from examples. Each example consists of n input values and m output values, all in the range $-1 \ldots 1$. The learning task is to find a mapping that describes with good precision for all examples the relation between the input and

the corresponding output values. Such a mapping for instance allows to compute approximate output values for a set of input values for which the real output values are not available ("generalization").

The overall structure of the algorithm is as follows:

1. The process starts with a random mapping that is defined by a number of parameters in the network called the "weights".
2. Each example is processed once (see below).
3. After all examples have been processed, the algorithm computes for each weight how it must be modified to maximally improve the accuracy of the mapping and modifies the weights accordingly. This is done by computing a gradient descent for a global error function.[2]
4. With this improved mapping the same process-examples-then-modify-weights step (called an "epoch") is started again.
5. This process is iterated until the quality of the mapping is satisfying or until we somehow decide to give up.

The operations with which each individual example is processed are the following:

1. The input values of the example are fed into the network.
2. The network processes these input values and produces output values.
3. These output values are compared to the original output values (i.e. those given in the example) and the differences are computed.
4. From the differences, an error measure is computed for each output value.
5. These error measures are fed backwards through the network.
6. During this backward pass through the network, the network uses the error measures to compute changes that should be made to each weight.
7. These changes are accumulated (averaged) over all the examples.

This algorithm works for a class of neural networks called "multi layer perceptrons" that will be described in the next section.

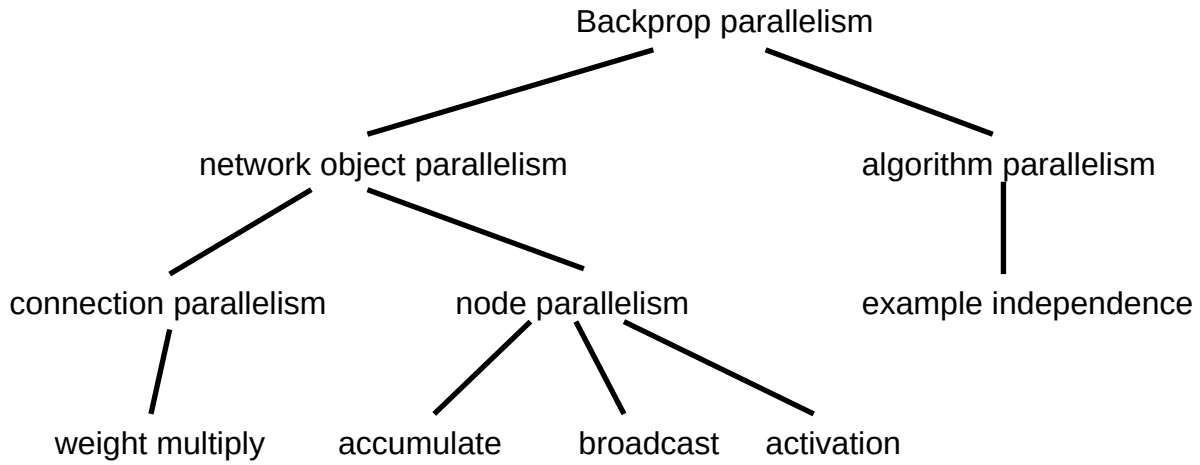
6.1.2 The network

The central object for any neural network learning algorithm is the network itself. Thus, it is also the central data structure in any program that implements this learning algorithm. The kind of network that is usually used with the backpropagation algorithm is called a "fully connected multi layer perceptron". In the following, we assume a 3 layer perceptron. It consists of three groups of units and two groups of weights. Both kinds of groups (unit groups as well as weight groups) are sometimes called "layers" which makes the term a bit confusing (and has the effect that some people would call our network a 2 layer perceptron). The groups of units are the input units, the hidden units, and the output units. The groups of weights are the hidden weights and the output weights. A weight is a weighted connection between two units. The set of hidden weights contains exactly one connection from each input unit to all of the hidden units. The set of output weights contains exactly one connection from each hidden unit to all of the output units. There is one additional unit, called the bias unit, that constantly outputs the value 1 and that has exactly one connection to each hidden and output unit. In our implementation of the network, we will make this bias unit a member of the input unit group.

6.1.3 The algorithm

The input units do nothing special, they just broadcast the input value they receive (from the example) to all the connections they have to the hidden units. Connections deliver a value they receive at one end to the other end after multiplying it with the weight that is stored in the connection. Hidden and output units accumulate the values delivered by the connections that they receive and apply a so-called

[2] Actually, the RPROP learning rule does not perform gradient descent, but uses the gradient only as a hint to what the best direction to move within the parameter space is.



This figure shows the taxonomy of the main kinds of parallelism that are embedded in the backpropagation programming problem. Similar kinds of parallelism apply to most neural network learning algorithms.

Figure 1: **Kinds of parallelism inherent in the backpropagation problem**

activation function to the result in order to compute their output value. In the case of hidden units this output value is again broadcasted to the outgoing connections. In the case of output units the output value is not broadcasted but instead compared with the correct output value given in the current example. The difference of these two values is squared and the result used as the error value for this example at any single unit. The process so far is called the *forward pass* through the network.

The error values are then propagated backwards through the network. This process is called the *backward pass* and gives the whole algorithm its name. The purpose of the backward propagation of errors is that each weight can compute its contribution to the overall error for this example. From this contribution, each weight can compute a *delta* which describes how the weight should change in order to reduce the error most. The output weights multiply the errors by their weight values and propagate the product to the hidden units. The hidden units accumulate these weighted errors and multiply the sum with the derivative of the activation function of the value they received as input in the forward pass. This hidden unit error value is further propagated backwards to the hidden weights, which operate on them just like the output weights did on the output errors.

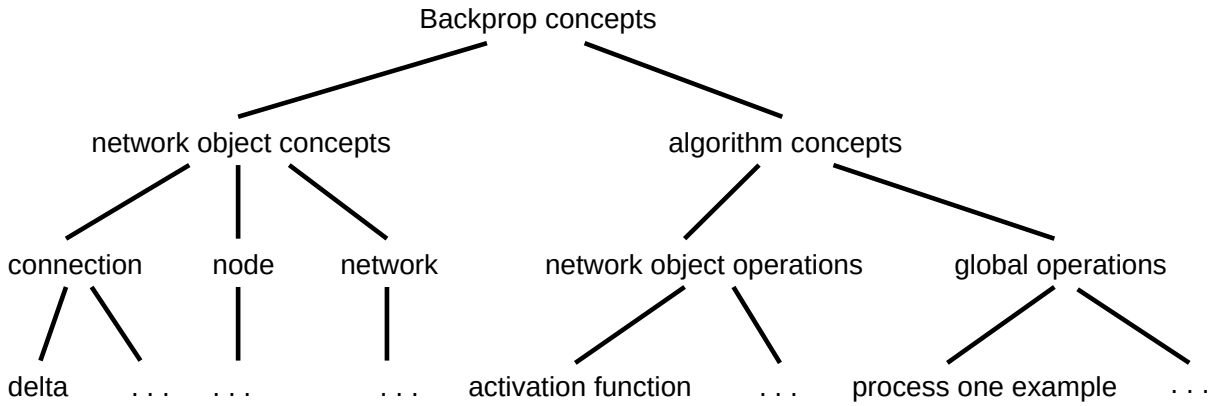
This combination of forward pass and backward pass is repeated for each example. The connections accumulate the delta values they compute in each backward pass. When all examples are processed, each connection weight adapts by adding the product of its accumulated deltas and a parameter called the *learning rate* to its current weight value.

This example processing and weight adaption is repeated until some custom termination criterion is fulfilled.

6.1.4 Inherent parallelism

One of the often-cited properties of neural networks is that they employ "massive parallelism". And indeed there is a lot of parallelism inherent in the backpropagation algorithm for multi layer perceptrons. This parallelism comes in two different forms: Parallelism that is coupled to objects in the network and parallelism that stems from a global property of the learning algorithm. The taxonomy of the kinds of parallelism present in the backpropagation algorithm is shown in figure 1

The object parallelism is attached to either the connections or the units. For the connections, all the weight multiplications of connections that connect the same pair of layers are independent of each other and can thus be executed in parallel. For the units, the computations of the activation functions in all units of a layer are independent of each other and can be executed in parallel. The reduction that is used to accumulate the inputs of a unit and the broadcast that is used to send the output of a unit to



This figure shows the first two levels of concepts that are embedded in the backpropagation programming problem. Only two of the concepts from the third level are given as examples, the others are left out. Similar concept hierarchies occur in most neural network learning algorithms.

Figure 2: **Part of backpropagation concept taxonomy**

all connections are parallelizable operations that can be executed in time logarithmic in the number of connections (for broadcast even faster, depending on the actual hardware).

The parallelism inherent in the algorithm, on the other hand, has nothing to do with individual objects in the network but stems from the independence of the computations for each individual example: Since the result of the processing of an example is nothing more than to compute one delta value per connection, and since these delta values are just accumulated over all the examples, all the examples can just as well be computed in parallel. The results of such independent example computations can be reduced to the result that sequential computation had achieved in logarithmic time.

6.2 Properties of a problem-adequate solution

Several important concepts turn up in the above description of the backpropagation learning method. These are for example: input unit, hidden unit, output unit, connection, weight, activation function, error, forward pass, delta, example, and epoch.

Trying to classify the above concepts we find that some have to do with the objects from which the network is built and others with the learning algorithm itself. Among the concepts that describe objects we can identify the connections, the units, and the network as a whole. Among the algorithmic concepts we find some that are tightly connected to parts of the network and others that relate to global algorithmic control.

We want to define the requirements that a description of the solution of the backprop programming problem must satisfy in order to be considered problem-adequate. To guide this requirements definition process we use the concept taxonomy, which is shown in figure 2, plus the following general idea: A programming language should allow to model a problem and its solution in natural terms in order to ease understanding of the program (see e.g. [Hoa83, Wir83]). Thus, a program that defines a problem-adequate solution to the backprop learning problem must explicitly map the above concepts onto program entities and should provide appropriate abstractions that allow to use these concepts in a problem-oriented way.

We will now discuss what an adequate description should look like, first for the network objects, then for the operations (the algorithm itself), and last for the embedded parallelism. In these discussions, we list what language constructs we would like to have in a language to solve our programming problem.

6.2.1 Description of the network

The description of the network consists of several parts:

1. The data structure needed to represent a weight.

2. The data structure needed to represent an input/hidden/output unit.
3. The description of the layers of units (which and how big).
4. The description of which connections between units actually exist.

Obviously the data structures can be defined as data types. The data types used to describe connections and units are records. The layers can be described by arrays of units.

Since certain operations can be performed only on connections and others only on units, and in particular since connections have to be attached to nodes and nodes have to be part of a network, it is sensible to define *type categories* for these kinds of data structures, i.e., not to say for a connection type X that "X is a record type", but instead to say that "X is a connection type" etc.

To describe which connections actually exist may be more complicated. In simple cases, such as the fully connected multi layer perceptron, we can describe the set of existing connections by just giving a few cross products: "Each unit in layer A is connected to each unit in layer B by a connection of type C" etc. In more complicated, irregular cases, no compact declarative description saying which connections exist is possible; the easiest way to define which connections exist is to have statements to create them (individually or in groups).

There should be an additional type category that covers complete networks (i.e. several groups of units plus the connections among them).

These language features allow to express all the network object concepts in the concept taxonomy (figure 2) in a sufficiently explicit and problem-oriented way.

6.2.2 Description of the algorithm

For the program description of the actual algorithm to reflect the distinction between network object operations and global operations (figure 2), it is best to attach the network object operations to the object types they work on. For example the weight multiply operation that computes a connection's output from its input is best given as a procedure that is declared in the context of the connection type definition. This approach is in the spirit of object oriented languages, where each method declaration is attached to a class declaration. Using the same approach, the unit activation computation operation that computes a unit's output by applying the activation function to the unit's input will be declared in the context of the type declaration of the unit type and so on.

Connection procedures are only allowed to access the local data elements of the connection they are applied to (reading and writing). Unit procedures can access the local data elements of the unit they are applied to and can also call the connection procedures of the connections that connect into or out of the unit.

When the input of a node must be computed, though, these procedures do not suffice: A collection of values (one from each connection that goes into the node) has to be combined into a single value; this single value must then be delivered to the unit as an input value. This problem is most elegantly solved by declaring a *reduction function*. A reduction function combines two values into one by applying a binary operator to them. In the case of backpropagation, this operator is simply addition: $\mathrm{Reduce}(a, b) = a + b$. To reduce more than two values, this operator can be applied recursively if it is associative, e.g. $\mathrm{Reduce}(a, b, c) = \mathrm{Reduce}(\mathrm{Reduce}(a, b), c)$.
To compute the input of a unit, we can define that it is allowed to call a reduction function for values residing in the whole set of connections ending at that unit and that such a call shall mean the recursive application of the reduction function to all the connection's values resulting in the unit input value.

Global operations are simply global procedures just like in any other programming language that has a procedure concept.

6.2.3 Description of parallelism

The kinds of parallelism in neural network learning algorithms can best be divided according to the second level of figure 1 into

1. parallelism of operations on connections,
2. parallelism of operations on units or concerned with units, and
3. parallelism of examples.

Parallelism of examples ("example independence" in figure 1) can be described transparently: If the network is modeled as an explicit entity, we can couple operations to this network object. Such operations can then be applied to a single instance of this network or to several instances at once without any change. We might then have a statement that says "Now take this network and make several copies of it. Whenever I call a network operation, apply it to all of the copies in parallel". All that is left to do then, is to fetch several examples at once whenever a new example is needed. After each epoch, all the copies of the network have to be united to a single network again, in order to make the proper weight change. This recombination can be described by a call to a procedure that merges two networks into one (another kind of reduction operation). This call has to apply the merge function recursively to all the copies of the network in order to reduce the whole set of copies to a single copy. In the case of backpropagation, all this merge procedure has to do is to sum the deltas at each weight. Since the merging process is a local process at each unit and at each weight, we can declare the merge procedure for the network as a whole implicitly by declaring a merge procedure for each connection type and for each unit type.

Parallelism of operations on units (e.g. "activation" in figure 1) is best described by coupling procedures to each unit type as described in the previous section. Given such unit procedures, we can operate on units in parallel by just calling such a unit procedure not only for a single unit, but for a whole group of units at once. Thus, the same feature of the language serves two important abstraction purposes at once.

Parallelism of operations on connections (e.g. "weight multiply" in figure 1) is also best described by coupling procedures to connection types as described in the previous section. Calls to these connection procedures are made from unit procedures, achieving nested parallelism.

Broadcast operations from units ("broadcast" in figure 1) are described most simply by the passing of parameters to connection procedures that are called from unit procedures. The parallelism inherent in such a call can then be handled in any appropriate way when the program is compiled.

Reduction operations at units (i.e. the operations that reduce the values delivered by all the connections that go into the unit into a single value, e.g. "accumulate" in figure 1) are most appropriately described by defining reduction functions as described in the previous section. The parallelism inherent in a call to a reduction function can then be handled in any appropriate way when the program is compiled.

6.3 The solution

This section shows how the "properties of an adequate solution" as described in the previous section are achieved in a CuPit program for backpropagation by using appropriate CuPit language constructs. In each case we discuss why this language construct is present in CuPit and how it can be used. During these discussions, we step by step derive a complete CuPit program for the backpropagation task.

In fact, we do not derive a program for "vanilla" backpropagation, but for the RPROP Backpropagation variant [RB93]. In this variant, the learning rate is not fixed for the whole network, but instead each weight possesses its own learning rate and adapts it at each learning step using a simple learning rate adaption rule: The learning step at each weight is not proportional to the delta value of the weight, but instead starts at a certain fixed value `InitialEta` and is increased by multiplication with a factor `EtaPlus` (here: 1.1) as long as the sign of the weight's delta values does not change. When the sign changes, the learning step size is decreased by multiplication with a factor `EtaMinus` (here: 0.5). This change speeds up the backpropagation learning dramatically for most problems; often by an order of magnitude. Another extension to plain backpropagation is present in the program derived here: once in a while we delete all connections whose weight is below a certain threshold value. Such weight elimination can improve the generalization performance of a network by reducing the number of free parameters. We include it in order to show how one form of topology change works in CuPit.

6.3.1 Overview

CuPit introduces special categories of types to catch the important properties of units (called *nodes*), connections, and networks (see sections 6.1.2 and 6.2.1). In each type declaration T the operations for this type T of network objects are included (see sections 6.2.2 and 6.2.3). Reduction functions (see sections 6.2.2 and 6.2.3) are declared separately and the same reduction function can be used in operations of several different types. Example-level parallelism is achieved in the transparent fashion outlined in section 6.2.3: A `REPLICATE` statement is used to create multiple copies (called *replicates* in CuPit) of a network and `MERGE` procedures are declared for each connection type, node type, and network type to describe how such replicates a recombined into a single network again.

Reading the examples is done via special buffers: An external procedure written in any appropriate language fills the examples into a special buffer. A CuPit statement exists to transfer the examples from this buffer into the nodes of the network replicates (or vice versa).

A main procedure (called the *central agent* in CuPit) calls the appropriate procedures to read examples, apply them to the network, and update the weights. This procedure also controls the call of the weight elimination scheme and the termination of the whole learning program.

So much for the global picture, now let's look at the details. We begin with the network object type declarations, then discuss the network object procedures, and last describe external parts of the program and the central agent. If you want to take a look at the whole program at once, see appendix E on page 68.

6.3.2 The "weights"

For the backpropagation learning problem, only a single connection type is needed. We call this type `Weight`:

```
1   example connection type definition[1] ≡
    {
    TYPE Weight IS CONNECTION
      Real     in      := 0.0,
               out     := 0.0,
               weight  := RANDOM (-0.5...0.5),
               olddelta:= 0.0,
               eta     := initialEta,
               delta   := 0.0;
      example forward pass connection operation[8]
      example backward pass connection operation[10]
      example connection adapt procedure[12]
      example connection elimination procedure[16]
      example connection merge procedure[2]
    END TYPE;
    }
```

This macro is invoked in definition 30.

As we see, there are six data elements in a connection: `in` and `out` store the input and output value of the connection for the current example. `weight` stores the weight value itself and is initialized to a different random value in each connection object. `olddelta` stores the weight's delta of one epoch ago; this value is used to detect the change of delta's sign as needed by the RPROP rule. `eta` is the current local learning step size of this connection and `delta` is the variable that accumulates the error values over the backward passes of all examples.

How these values are used in the forward and backward pass and in the weight update is shown in sections 6.3.5, 6.3.6, and 6.3.7.

This is how corresponding connections of several replicates of a network are recombined into a single connection:

example connection merge procedure[2] ≡ 2

```
{
 MERGE IS
   ME.delta += YOU.delta;
 END MERGE;
 }
```

This macro is invoked in definition 1.

In this procedure, `ME` is the name of the connection object that is the result of the merge operation and `YOU` is the name of the connection object that is being merged into `ME` by this procedure. The merge procedure is recursively applied to replicates and merged replicates until only a single merged exemplar of the connection object is left. In this respect, merging is much like reduction. The only difference is that merging works on objects and is formulated in the form of procedures, while reduction works on values and is formulated in the form of functions.

6.3.3 The "units"

Despite the differences in functionality for the input, hidden, and output nodes in the multi layer perceptron, we can implement all three sorts by the same type. Some of the data and functionality of this type is not used in all three layers, but here we prefer to declare only a single type instead of saving a few bytes of storage. We call this type `SigmoidNode`, after the kind of activation function used in it:

example node type definition[3] ≡ 3

```
{
 TYPE SigmoidNode IS NODE
   Real        inData;     (* forward:  net input,
                              backward: delta *)
   Real        outData;    (* forward:  sigmoid(in),
                              backward: sum(deltas*weights) or teachinput *)
   IN Weight   in;
   OUT Weight  out;
   Real        cumErr := 0.0;  (* cumulated error (output units only) *)
   Int         errN  := 0;     (* number of error bits (output units only) *)
   Int         consN;          (* number of input connections *)
   example forward pass node operation[9]
   example backward pass node operation[11]
   example node adapt procedure[13]
   example node merge procedure[4]
   example resetErrors procedure[29]
   example node connection elimination procedure[17]
 END TYPE;
 }
```

This macro is invoked in definition 30.

There are five data elements and two connection interface elements in a node: `inData` and `outData` store the input and output value of the node for the current example. `cumErr` and `errN` are used to accumulate error statistics over all the examples if the node is used as an output node. `consN` is used to hold the number of connections attached to the input interface of the node when weight elimination is carried out.

`in` and `out` are the connection interface elements of the node type. Connections must always end at at an interface element that has interface mode `IN`. Such connections are called inputs to the node to which this interface element belongs. Connections must always originate at an interface element that has interface mode `OUT`. Such connections are called outputs of the node to which this interface element belongs. All connections attached to a particular interface element must have the same type. The functionality of

input interfaces and output interfaces is identical.[3] In the case of the multi layer perceptrons, all nodes have but a single input interface (called `in`) and a single output interface (called `out`). Both of these interfaces are declared to accept connections of type `Weight`.

How these data elements and interfaces are used in the various node operations is shown in sections 6.3.5, 6.3.6, 6.3.7, and 6.3.13.

This is how corresponding connections of several replicates of a network are recombined into a single connection:

4 *example node merge procedure*[4] ≡

```
{
MERGE IS
  ME.cumErr += YOU.cumErr;
  ME.errN   += YOU.errN;
END MERGE;
}
```

This macro is invoked in definition 3.

Only the error statistics have to be retained, all other data in a node is dispensable. The merge procedure for the connections does not have to be called here, because all merge procedures are called implicitly when merging of networks is performed.

6.3.4 The multi-layer perceptron

It is possible to represent multiple networks in a single CuPit program. Each such network is a variable of a particular network type. To declare the network type for the multi layer perceptron (`Mlp`), we first have to declare types for the groups of nodes that make up the layers in that network.

5 *example layer type definitions*[5] ≡

```
{
TYPE Layer  IS GROUP OF SigmoidNode END;
}
```

This macro is invoked in definition 30.

We declare `Layer` to be a dynamic array of `SigmoidNode`s. Such a dynamic array has initial size 0. Since we can adjust the number of nodes used for each layer at run time, only a single type is needed for all three layers of our network.

6 *example network type definition*[6] ≡

```
{
TYPE Mlp IS NETWORK
  Layer   in, hid, out;
  Real    totError;
  Int     errorsN;
  Int     consN;
  example create network[7]
  example processing one example[15]
  example network adapt procedure[14]
  example compute total error procedure[28]
  example network count connections procedure[19]
  example network eliminate connections procedure[18]
END TYPE;
}
```

This macro is invoked in definition 30.

[3] The discrimination is necessary for technical reasons: in order to know where to place the actual connection data

An `Mlp` object consists of one element for each layer of nodes, two elements to store the global error measures, one element to store an accumulated number of connections (computed after connection elimination), and six network procedures.

The connections of a network have to be created dynamically. In our case, the same is true for the nodes as well. We define a procedure that performs this node and connection creation below. An `EXTEND` statement adds a certain number of nodes to a node group. The first `CONNECT` statement says that one connection shall be created from each input node's `out` interface to each hidden node's `in` interface. Such a connection will have the type `Weight` because that is the type of the participating connection interfaces. Analogously, the second `CONNECT` statement says that one connection shall be created from each hidden node's `out` interface to each output node's `in` interface. The third `CONNECT` statement makes connections from the first node of the `in` layer to all nodes of the `out` layer. The purpose of these connections is to provide the bias input to the output nodes (see section 6.1.2) — we assume that input node 0 will receive the same input value for all examples so that it can act as the bias node.

example create network[7] ≡ 7
```
{
 PROCEDURE createNet (Int CONST inputs, hidden, outputs) IS
   EXTEND ME.in BY inputs;
   EXTEND ME.hid BY hidden;
   EXTEND ME.out BY outputs;
   CONNECT ME.in[].out  TO ME.hid[].in;
   CONNECT ME.hid[].out TO ME.out[].in;
   (* ...and bias for the output nodes: *)
   CONNECT ME.in[0...0].out TO ME.out[].in;
 END;
 }
```
This macro is invoked in definition 6.

6.3.5 The forward pass

For the forward pass, we have one operation in the connection type and one in the node type. The node operation calls the connection operation, thus achieving nested parallelism.

This definition is part of the connection type definition:

example forward pass connection operation[8] ≡ 8
```
{
 PROCEDURE transport (Real CONST val) IS
   ME.in  := val;
   ME.out := val*ME.weight;
 END PROCEDURE;
 }
```
This macro is invoked in definition 1.

The value that is passed as a parameter from the node that calls this procedure to all the nodes that execute it is stored as the input value of the connection. The value will be used again in the backward pass. From this input value the output value is computed by multiplying with the weight stored in the connection. Note that the parameter passing from the nodes into the connection procedures implements the broadcast mentioned in section 6.2.3.

This definition is part of the node type definition:

example forward pass node operation[9] ≡ 9
```
{
 PROCEDURE forward (Bool CONST doIn, doOut) IS
   IF doIn
```

```
    THEN REDUCTION ME.in[].out:sum INTO ME.inData;    END;
    IF doOut THEN
      ME.outData := sigmoid (ME.inData);
      ME.out[].transport(ME.outData);
    END;
  END PROCEDURE;
  }
```

This macro is invoked in definition 3.

The forward pass procedure of the nodes is parameterized in order to implement the different functionality of input nodes, hidden nodes, and output nodes (see section 6.3.8 to see how the parameters are used). First the input value is computed in `ME.inData` (this step is skipped if the node is an input node) The computation is carried out by calling the reduction function `sum` (see section 6.3.10) for the `out` elements of all connections that are attached to the interface `in`. Then the output value is computed by applying the activation function `sigmoid` to the input value. In the end, the output value is sent to the output connections by calling their `transport` procedure (as shown above). These last two steps are skipped if the node is an output node.

6.3.6 The backward pass

For the backward pass, most of the technical points are analogous to what happens in the forward pass. The algorithmic details, on the other hand, are much different. The backward pass is where the main idea of the backpropagation algorithm comes into view: computing the gradient of the error at each weight by propagating error values backwards through the network.

This definition is part of the connection type definition:

10 *example backward pass connection operation*[10] ≡

```
  {
  PROCEDURE btransport (Real CONST errval) IS
    ME.out   := errval;
    ME.delta += errval*ME.in;
    ME.in    := errval*ME.weight;
  END PROCEDURE;
  }
```

This macro is invoked in definition 1.

The backward transportation operation of the connections first stores the value that shall be propagated backwards in the `out` field of the connection object. This assignment is not really needed but it would for example enable to recall the error values propagated through each weight after the backward pass, if we wanted to. In the second step, the local `delta` value that accumulates the weight changes induced by the examples is changed: The product of the connection's input from the forward pass and the error propagated through the connection in the backward pass is added to the delta value. Note that both `errval` and `ME.in` may be negative. In the third step the actual backward propagation value of the connection is computed; this is the connection's contribution to the error in the node the connection originates at, i.e., in the node that supplied the forward propagation value to the connection.

This definition is part of the node type definition:

11 *example backward pass node operation*[11] ≡

```
  {
  PROCEDURE backward (Bool CONST doIn, doOut) IS
    (* (for output nodes, out is the teaching input)
       for all nodes, set  in := delta of the node
       for all nodes that are not input nodes, back-transport the delta
    *)
    Real VAR sout, diff;
```

```
    IF doIn  (* i.e. is not an output node *)
    THEN REDUCTION ME.out[].in:sum INTO ME.outData;
         ME.inData := ME.outData * sigmoidPrime(ME.inData);
    ELSE (* output node: *)
         (* error function:  E (target, actual) = 1/2*(target-actual)**2  *)
         sout := sigmoid (ME.inData);
         diff := ME.outData - sout;
         ME.inData := diff * sigmoidPrime(ME.inData);
         ME.outData := sout;
         ME.cumErr += 0.5 * diff * diff;
         IF absReal (diff) > errBitThreshold THEN ME.errN += 1;  END;
    END;
    IF doOut (* i.e. is not an input node *)
    THEN ME.in[].btransport (ME.inData);  END;
  END PROCEDURE;
  }
```

This macro is invoked in definition 3.

The backward pass is fundamentally different for output nodes and for non-output nodes. For non-output nodes the error values that come backwards through the output connections of the node are accumulated. The result is multiplied with the derivation of the activation function at the point used in the forward pass. The derivation of the activation function is called `sigmoidPrime` here.

For output nodes, the network error for this example and this node is computed. These error values serve as the basis of the whole backward propagation process. The error E_i of an output o_i given the correct output t_i ("teaching" input) is defined to be $E_i = 1/2(o_i - t_i)^2$, known as the *squared error* function. In our case, t_i is stored as `ME.outData` and o_i is available as `sigmoid(ME.inData)`. The error value that has to be propagated backwards through the network is the derivation of the error function in respect to the node's input. This derivation is $(o_i - t_i)*$`sigmoidPrime(ME.inData)` in our case. The pure error value is accumulated in the `cumErr` error statistics data element. A threshold-based error count is accumulated in `nrErr`.

Unless the node is an input node, the error derivative is then broadcasted (back-propagated) to the input connections of the node.

Note also that connection input reduction can just as well be called "against" the formal direction of connections, i.e., for `OUT` interface elements of a node. Analogously, broadcast into connections can just as well be done for input connections, i.e., connections that are attached to an interface element with mode `IN`. Connections are formally directed but can be used in both directions for all operations.

For the items appearing in the above piece of source code that have not yet been declared, see sections 6.3.11 and 6.3.12.

6.3.7 The weight update

After each epoch, i.e., when all examples have been processed once, the weights have to be updated. This is done according to the RPROP learning rule as described in section 6.3.

This definition is part of the connection type definition:

example connection adapt procedure[12] ≡ 12

```
  {
   PROCEDURE adapt () IS
     Real VAR newstep,
              deltaproduct := ME.olddelta * ME.delta,
              deltasign := signReal (ME.delta);
     IF deltaproduct > 0.0
     THEN ME.eta     := minReal (ME.eta * etaPlus, maxEta);
```

```
    ELSIF deltaproduct < 0.0
    THEN ME.eta     := maxReal (ME.eta * etaMinus, minEta);
    (* ELSE deltaproduct is 0, don't change ME.eta *)
    END;
    newstep     := deltasign * ME.eta - ME.weight * decayterm;
    ME.weight   += newstep;
    ME.olddelta := ME.delta;
    ME.delta    := 0.0;
  END PROCEDURE;
  }
```

This macro is invoked in definition 1.

This procedure is where the difference is located between vanilla backpropagation and RPROP backpropagation. In vanilla backpropagation, a learning rate is used that is global and fixed. Weight adaption always adds `delta` times this learning to the current weight value. In RPROP, the learning step is computed according to the following rule: Initially, the learning step size is some fixed value `initialEta`. In each weight adaption step, if the sign of `delta` is the same as in the last step, the local learning step size `eta` is multiplied by `etaPlus` (which in this example is 1.1). If, on the other hand, the sign of `delta` is not the same as in the last weight adaption step, the local learning step size is multiplied by `etaMinus` (which in this example is 0.5). The weight is then changed in the direction indicated by the sign of `delta` by the new local learning step size. Finally, the `delta` is reset to zero for the next epoch. This behavior is implemented in the above procedure.

This definition is part of the node type definition:

13 *example node adapt procedure*[13] ≡

```
  {
  PROCEDURE adapt () IS
    ME.in[].adapt ();
  END PROCEDURE;
  }
```

This macro is invoked in definition 3.

There is nothing to do for a node in the weight adaption, so this procedure just calls the weight adaption of the input connections.

This definition is part of the network type definition:

14 *example network adapt procedure*[14] ≡

```
  {
  PROCEDURE adapt () IS
    ME.out[].adapt ();
    ME.hid[].adapt ();
  END PROCEDURE;
  }
```

This macro is invoked in definition 6.

This procedure triggers the adaption of the weights for the whole network by calling the weight adaption of the hidden and the output layer. Since the layers adapt their input connection's weights and there are no input connections to the input nodes, no such call for the input layer is necessary.

6.3.8 Processing one example

After we have loaded the next example (not shown here), all we have to do to process the example is to call the `forward` and `backward` propagation procedures of the nodes with the appropriate parameters. This is done in the following procedure which is part of the network type definition:

15 *example processing one example*[15] ≡

```
{
 PROCEDURE  example () IS
   ME.in[].forward   (false, true);
   ME.hid[].forward  (true,  true);
   ME.out[].forward  (true,  false);
   ME.out[].backward (false, true);
   ME.hid[].backward (true,  true);
   (* ME.in[].backward  (true,  false); *) (* not needed *)
 END PROCEDURE;
 }
```

This macro is invoked in definition 6.

6.3.9 Eliminating weights

As mentioned in section 6.3 we want to build into our example program the capability to eliminate connections whose weights are below a certain threshold. Such an operation is straightforward in CuPit.

The following procedure has to be included in the connection type definition:

example connection elimination procedure[16] ≡ 16

```
{
 PROCEDURE eliminate (Real CONST p) IS
   ME.eta := initialEta;
   IF absReal(ME.weight) < p
   THEN REPLICATE ME INTO 0; END;
 END PROCEDURE;
 }
```

This macro is invoked in definition 1.

It makes all connections whose weights have an absolute value of less than `p` self-delete. Since such deletions momentarily destabilize the learning process, the learning step size of all weights is reset to the initial value.

The following procedure is part of the node type definition:

example node connection elimination procedure[17] ≡ 17

```
{
 PROCEDURE modify (Real CONST p) IS
   (* during this procedure, we re-interpret
      errN as number of just deleted connections
   *)
   Real VAR oldConN, newConN;
   ME.in[].transport (1.0);
   REDUCTION ME.in[].in:sum INTO oldConN;  (* count connections *)
   ME.in[].eliminate (p);
   REDUCTION ME.in[].in:sum INTO newConN;
   ME.consN := Int(newConN);
   ME.errN := Int(oldConN) - ME.consN;
   IF ME.consN = 0    (* no more input connections present -> *)
   THEN REPLICATE ME INTO 0;  END; (* self-delete *)
 END PROCEDURE;
 }
```

This macro is invoked in definition 3.

This procedure does three things: (1) it eliminates all of its inputs connections with weights below threshold `p` by calling the connection type's `eliminate` procedure, (2) it counts how many input connections are present at the node before and after the connection elimination, and (3) it deletes itself, if no input connection are left after the elimination.

Finally, to call the connection elimination we need an appropriate network procedure:

18 *example network eliminate connections procedure*[18] ≡

```
{
PROCEDURE modify (Real CONST p) IS
  ME.out[].modify (p);
  ME.hid[].modify (p);
  ME.countConnections ();
  ME.out[].resetErrors();
END PROCEDURE;
}
```

This macro is invoked in definition 6.

This procedure calls the connection elimination for the output and hidden layer (since there are no input connections to the input layer, a call for the input layer is not useful) and then counts how many weights were present before and after the elimination by calling the procedure `countConnections` shown below. Finally, it resets the "misused" value of `errN` in the nodes of the output layer so that it can be used for actual error bit counting in the next epoch again.

19 *example network count connections procedure*[19] ≡

```
{
PROCEDURE countConnections () IS
  Int VAR outConsN, outConsDeleted;
  REDUCTION ME.out[].consN:sumInt INTO outConsN;
  REDUCTION ME.out[].errN:sumInt  INTO outConsDeleted;
  REDUCTION ME.hid[].consN:sumInt INTO ME.consN;
  REDUCTION ME.hid[].errN:sumInt  INTO ME.errorsN;
  ME.consN += outConsN;
  ME.errorsN += outConsDeleted;
END PROCEDURE;
}
```

This macro is invoked in definition 6.

6.3.10 The reduction function

The definition of the reduction function used in the backpropagation algorithm to compute inputs of nodes and to compute global error values is almost self-explanatory. The reduction is needed for real and for integer values:

20 *example reduction function definition*[20] ≡

```
{
Real REDUCTION sum IS
  RETURN (ME + YOU);
END REDUCTION;

Int REDUCTION sumInt IS
  RETURN (ME + YOU);
END REDUCTION;
}
```

This macro is invoked in definition 30.

A reduction function has two explicit parameters called `ME` and `YOU`. These parameters have the same type as the result of the reduction function. The values of the parameters cannot be changed. Apart of that, a reduction function is just a function like any other, which means that it can declare local variables, call other global functions, and so on.

6.3.11 External program parts

Our CuPit program is not completely stand-alone: Some arithmetic functions and the procedures that supply the examples have been put into external program parts.

For the arithmetic functions this was done deliberately; they could also be defined in CuPit itself. Moving them to an external module, however, offers capabilities for fine-tuned, machine-dependent optimized implementation. We could for example implement the sigmoid function using a lookup table. Such an implementation can not be expressed in CuPit if the lookup table shall exist once per processor of a parallel machine, since CuPit completely hides the structure of the machine.

example external arithmetic functions[21] ≡ 21
```
{
 Real FUNCTION sigmoid (Real CONST x)       IS EXTERNAL;
 Real FUNCTION sigmoidPrime (Real CONST x)  IS EXTERNAL;
 Real FUNCTION absReal (Real CONST x)       IS EXTERNAL;
 Real FUNCTION signReal (Real CONST x)      IS EXTERNAL;
 Real FUNCTION minReal (Real CONST x, y)    IS EXTERNAL;
 Real FUNCTION maxReal (Real CONST x, y)    IS EXTERNAL;
 }
```
This macro is invoked in definition 30.

These functions mean (in order): The activation function, its derivative, the absolute value function, the signum function, the minimum-of-two-values function, the maximum-of-two-values function.

Furthermore, we need a few procedures for getting the examples into the CuPit program and a few procedures for output. Both are declared globally, outside of any type definition.

Here are the procedures for example handling.:

example external example-getting procedures[22] ≡ 22
```
{
 PROCEDURE openDatafile (String CONST filename;
                         Int VAR iInputsN, rInputsN, iOutputsN, rOutputsN,
                         examplesN) IS EXTERNAL;
 PROCEDURE initIOareasxRxR (Real IO rIn; Real IO rOut;
                            Int CONST maxreplicates) IS EXTERNAL;
 PROCEDURE getExamplesxRxR (Real IO rIn; Real IO rOut;
                            Int CONST howmany; Int VAR firstI) IS EXTERNAL;
 }
```
This macro is invoked in definition 30.

`openDatafile` opens the example data file with the given name, reads all the examples from the file into memory, and returns how many examples there are in the file and how many real and integer input and output coefficients each example has. `initIOareas` initializes the I/O buffer objects `rIn` and `rOut` as soon as we know how many network replicates will be used at once. Finally, `getExamples` reads the next `howmany` examples into the I/O objects `rIn` (input coefficients) and `rOut` (output coefficients). The name suffix `xRxR` indicates that the variants for real-coefficients-only (and no integer coefficients) shall be used; there are also `IRIR` etc. variants of the procedures available.

The output procedures used to make the learning process visible are:

example output procedures[23] ≡ 23
```
{
 PROCEDURE protocolError (Int CONST epochNumber; Real CONST totalError;
                          Int CONST nrOfErrors) IS EXTERNAL;
 PROCEDURE pInt (Int CONST me) IS EXTERNAL;
 PROCEDURE pReal (Real CONST me) IS EXTERNAL;
 PROCEDURE pString (String CONST me) IS EXTERNAL;
```

```
}
```
This macro is invoked in definition 30.

`protocolError` writes a line with epoch number, total error value, and number of error bits to the screen and to a file (for graphical visualization). `pInt`, `pReal`, and `pString` output one object of the respective types to the screen.

6.3.12 Global data definitions

Some global constants and variables have to be defined for our backpropagation program. Most of the constants have already been used above.

24 *example global constant definitions*[24] ≡
```
{
Real CONST initialEta     :=   0.05, (* initial learning step *)
           etaPlus        :=   1.1,  (* learning step increase step *)
           etaMinus       :=   0.5,  (* learning step decrease step *)
           maxEta         :=   50.0, (* maximum learning step *)
           minEta         :=   1.0e-6, (* minimum learning step *)
           decayterm      :=   0.0,  (* weight decay *)
           errBitThreshold:=   0.3;  (* max. difference for correct outputs *)
Int  VAR   inputs,
           hidden         :=   4,
           outputs;
}
```
This macro is invoked in definition 30.

25 *example global variable definitions*[25] ≡
```
{
Real IO    x1, x2;  (* I/O-areas, allocated and managed by external program *)
Mlp  VAR   net;     (* THE NETWORK *)
}
```
This macro is invoked in definition 30.

The variable `net` is the central variable of our program. It represents the multi layer perceptron which we want to train. `x1` and `x2` are the buffers (called "I/O areas") through which the examples are communicated between the CuPit program on one side and the external procedures that supply the examples on the other side.

6.3.13 The central agent

This section contains the main routine of the CuPit program. This procedure and all invocations of other procedures and functions that are not network object operations comprise what is called the *central agent* of the CuPit program. Certain operations can only be called by the central agent (most noticeably the replication of networks).

26 *example central agent*[26] ≡
```
{
PROCEDURE program () IS
  Int  VAR i := 0, nrOfExamples, examplesDone := 0, epochNr := 0;
  Int  VAR dummy1, dummy2;
  Real VAR error, oldError, stoperror := 0.1;
  openDatafile ("Data", dummy1, inputs, dummy2, outputs, nrOfExamples);
  stoperror := Real(outputs**2) * 0.5*(stoperror**2);
  net[].createNet (inputs, hidden, outputs);
```

```
REPLICATE net INTO nrOfExamples;  (* better: INTO 1...nrOfExamples *)
initIOareasxRxR (x1, x2, MAXINDEX (net) + 1);
REPEAT
  epochNr += 1;
  example process each example once[27]
  MERGE net;  (* collect and redistribute results *)
  net[].adapt();
  net[].computeTotalError ();
  error := net[0].totError;
  protocolError (epochNr, error, net[0].errorsN);
  IF epochNr % 10 = 0 THEN
    IF epochNr <= 20  THEN oldError := error/2.0;
    ELSIF error < oldError
    THEN REPLICATE net INTO 1;
         net[].modify (0.25);
         pString (">>>>>>>>>>> weights remaining: ");
         pInt (net[0].consN);
         pString ("  deleted: "); pInt (net[0].errorsN);
         pString (" <<<<<\n");
         oldError := error;
         REPLICATE net INTO nrOfExamples;
    END IF;
  END IF;
UNTIL epochNr > 4 AND error <= stoperror  END REPEAT;
END PROCEDURE;
}
```

This macro is invoked in definition 30.

This procedure should be more or less self-explanatory. First, we read the examples into memory, compute the stop criterion, and create the initial network configuration. `REPLICATE net INTO nrOfExamples` requests that exactly as many copies (replicates) of the network shall be made as there are training examples. This works only for small training sets. A better choice would be `REPLICATE net INTO 1...nrOfExamples` which lets the compiler chose any number of replicates it considers sensible for optimal efficiency (but at most one per example). `initIOareasxRxR` allocates space for the I/O areas `x1` and `x2` based on the number of network replicates that the CuPit compiler has chosen to generate (`MAXINDEX(net)+1`).

The following loop embodies the actual training process. `MERGE net` merges data from the network replicates into the first network replicate and then immediately redistributes this data into the other replicates. The number of replicates is thus not changed, but instead all replicates are made identical (at least the *relevant* part of the data in respect to the `MERGE` operations declared in the types is made identical).

The next four statements after the `MERGE` call the weight adaption process and compute and show the global error measures for the network. The following `IF` controls the call of the connection elimination scheme: Connection elimination is performed with a threshold of 0.25 and is called first when the epoch number is divisible by 10 and the error is less than half that of epoch 20. Subsequent calls of the weight elimination occur when the error is less than it was just before the previous connection elimination step. In order to perform the connection elimination, CuPit requires that the number of replicates be one, because otherwise the topology of the replicates might diverge and the replicate correspondence as defined by the `MERGE` operation would be lost. Thus, we call `REPLICATE net INTO 1` before the connection elimination, and `REPLICATE net INTO nrOfExamples` to create replicates again after it.

The most interesting part of the central agent, the part where the backpropagation process is carried out for all the examples is described now:

example process each example once[27] ≡ 27

```
{REPEAT
```

```
  getExamplesxRxR (x1, x2, MAXINDEX (net) + 1, i);
  net.in[].inData   <-- x1;
  net.out[].outData <-- x2;
  net[].example();
  examplesDone += MAXINDEX (net) + 1;
UNTIL examplesDone >= nrOfExamples END REPEAT;
examplesDone %= nrOfExamples;
}
```

This macro is invoked in definition 26.

The `getExamplexRxR` procedure loads one example per network replicate into the I/O buffers: `x1` holds the input values and `x2` holds the output values. These values are then transferred into the input fields of the input layers (one value from `x1` per node per replicate) and the output fields of the output layers (one value from `x2` per node per replicate). Once the examples are loaded into the network replicates, a single call to `net[].example` suffices to process the example in each network replicate. This is how example-level parallelism is made implicit in CuPit.

Finally, here is the `computeTotalError` function which is still missing from the definition of the network type:

28 *example compute total error procedure*[28] ≡

```
{
PROCEDURE computeTotalError () IS
  REDUCTION ME.out[].cumErr:sum INTO ME.totError;
  REDUCTION ME.out[].errN:sumInt INTO ME.errorsN;
  ME.out[].resetErrors();
END PROCEDURE;
}
```

This macro is invoked in definition 6.

...and the procedure `resetErrors` which is missing from the definition of the node type:

29 *example resetErrors procedure*[29] ≡

```
{
PROCEDURE resetErrors () IS
  ME.cumErr := 0.0;
  ME.errN   := 0;
END PROCEDURE;
}
```

This macro is invoked in definition 3.

6.3.14 The whole program

This is how the program parts shown in the last few sections together make up the complete program. You can find the program listed as a whole in appendix E.

30 **backprop.nn**[30] ≡

```
{
(* Example CuPit program for rprop Back-Propagation + weight elimination
   Lutz Prechelt, 94/01/16
   (see description in "Parallel Distributed Processing Vol 1",  pp.322-329)
*)
```

example global constant definitions[24]
example external arithmetic functions[21]
example reduction function definition[20]
example connection type definition[1]
example node type definition[3]

example layer type definitions[5]
example network type definition[6]
example global variable definitions[25]
example external example-getting procedures[22]
example output procedures[23]
example central agent[26]
}
This macro is attached to an output file.

6.4 Usable parallelism

The above formulation of the backpropagation algorithm allows for easy exploitation of all kinds of parallelism that are inherent in the problem (see section 6.1.4) by a compiler.

Some of the parallelism is explicit, e.g. the parallelism in the node and connection operations, and some of the parallelism is implicit, e.g. the broadcast, reduction, and example parallelism. But all parallelism whether explicit or implicit can be extracted straightforwardly from the CuPit program and can be implemented on real parallel machines easily; no complicated program analysis is necessary. Another important point is that the parallelism is formulated in a problem-oriented fashion; it is always object-based parallelism achieved by applying an operation to many objects at once.

All parallelism is defined with asynchronous semantics in CuPit, i.e., all parts of a parallel operation may be executed in absolutely any order by an actual implementation of a CuPit program. Such asynchronous semantics are adequate for a local computation paradigm such as neural networks and allow for implementation with maximum efficiency.

6.5 What is not shown here

There are of course many features of CuPit that are not used in the above tutorial example program. Most of them are not very difficult or important, for instance enumeration types, additional operators, or additional kinds of loops.

A few other features, though, shall be mentioned here:

1. Nodes are numbered within a group (from 0 to `MAXINDEX(group)`). These node indices are available in node procedures as `INDEX`.
2. Node operations can be called on *slices* of node groups: Instead of saying `ME.hid[].adapt()` you might say `ME.hid[4...7].adapt()` in order to call `adapt` only for the nodes 4 to 7 of the `hid` group. There is an interval data type to support this slicing.
3. You can also call a network operation for only some of the existing network replicates by using the same slicing syntax.
4. There are winner-takes-all operations, working with user-defined winner-takes-all operators (similar to reduction functions), that call a connection or node operation only for one connection of a node or one node of a node group based on a maximum criterion.

More important than these operations, though, are the other variants of topology-changing operations not used in the example above:

1. `EXTEND` can be used not only for initial node group creation but also for dynamic extension of a node group by one or several additional nodes.
2. `EXTEND` can also be used with negative numbers in order to delete the node with the highest indices in the group.
3. `DISCONNECT` is the antipode to `CONNECT` and can be used to delete connections under central control.

Together with the self-deletion capabilities of nodes and connections, these operations simply and flexibly allow to create and destroy nodes or connections while the learning is in progress (see section 10.8). Such a capability is important for the description of constructive learning algorithms that learn not only the

values of parameters (weights), but also evolve a suitable network topology during the learning process. The weight elimination scheme shown above is only a very simple example of such an algorithm. It is difficult to support such algorithms efficiently on parallel machines, but CuPit was specifically designed to support dynamic network topologies and can thus be implemented with good performance.

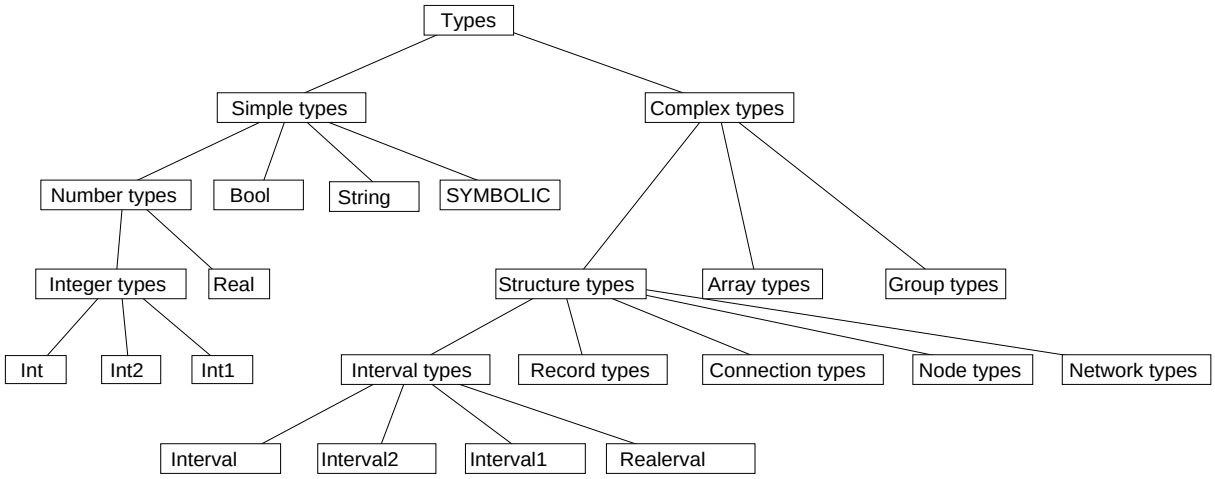


Figure 3: **CuPit type taxonomy**

Part II: Language Reference

7 Type definitions

7.1 Overview

There are several categories of types in CuPit:

1. Elementary types, such as `Int`, `Real`, `Bool`, `String`, and enumerations (`SYMBOLIC` types).
2. Intervals of `Int` or `Real`.
3. Record types.
4. Connection types.
5. Node types.
6. Network types.
7. Arrays of objects of simple types, intervals, record types, node types, or array types.
8. Groups of objects of node types (i.e. collections with or without fixed order)

The elementary types are also called *simple types*. The elementary types except `Bool`, `String`, and enumerations are called *number types*. The interval, record, connection, node, and network types are also called *structure types*. The structure types, interval types, array types, and group types are called *complex types*. You can see the whole taxonomy of the CuPit type system in figure 3.

This is the general syntax of type definitions:

Type Definition[31] ≡ 31

```
{
 TypeDef:
    'TYPE' NewTypeId 'IS' TypeDefBody 'END' OptTYPE.

 NewTypeId:
    UppercaseIdent.

 TypeDefBody:
    SymbolicTypeDef /
    RecordTypeDef /
    NodeTypeDef /
    ConnectionTypeDef /
    ArrayTypeDef /
```

```
   GroupTypeDef /
   NetworkTypeDef.

OptTYPE:
   /* nothing */ /
   'TYPE'.
```

Symbolic Type Definition[32]
Record Type Definition[33]
Node Type Definition[36]
Connection Type Definition[38]
Array Type Definition[39]
Group Type Definition[40]
Network Type Definition[41]
}

This macro is invoked in definition 85.

A type definition may appear only on the outermost level of a CuPit program (i.e. not within procedures). The `TypeId` mentioned in the `TypeDef` is introduced as a new type name and bound to the definition given in the `TypeDefBody`. The new `TypeId` is defined and visible in the rest of the program after the point where it appears first in its own definition, i.e., types must be defined before they can be used in the definition of another type or in the definition of an object. Type names must not be redefined.

All types that occur in a CuPit program have an explicit name and two types are identical only if they have the same name. *Design rationale:* This makes the semantics of the language much simpler.

The individual kinds of definitions will be explained in the next few subsections.

7.2 Simple types

Among the basic types of CuPit are truth values `Bool`, integral numbers `Int` and floating point numbers `Real`. The exact representation and operation semantics of these types is machine dependent. There are three variants of `Int`, namely `Int1`, `Int2`, and `Int`. These are one-byte, two-byte, and four-byte signed integers, respectively.

Other simple types are the `String` type, which represents pointers to arrays of bytes terminated by a byte with value 0 (like in C), and the so-called `SYMBOLIC` types, which are defined by giving a list of names that represent the set of values of the type. Thus a `SYMBOLIC` type is similar to an enumeration type in MODULA-2 or in C, except that in CuPit symbolic values are not ordered and cannot be converted into or created from integer values. The only operations that are defined on symbolic types are assignment and test for equality.

Objects of simple types may occur as members in any other type, as global variables and as local variables in all kinds of procedures and functions.

32 *Symbolic Type Definition*[32] ≡

```
{
SymbolicTypeDef:
   'SYMBOLIC' NewEnumIdList OptSEMICOLON.

OptSEMICOLON:
   /* nothing */ /
   ';'.

NewEnumIdList:
   NewEnumId /
   NewEnumIdList ',' NewEnumId.
```

```
NewEnumId:
   LowercaseIdent.
}
```
This macro is invoked in definition 31.

7.3 Interval types

Types can be defined that can hold two integer or real values and mean the compact integer or real interval between the two. Objects of interval types may occur as elements of any complex type, as global variables and as local variables in all kinds of procedures and functions.

Objects of type `Interval`, `Interval1`, and `Interval2`, use objects of type `Int`, `Int1`, `Int2` respectively, to represent their current maximum and minimum; `Realerval` objects use `Real` values. The strange name `Realerval` is just a play on words.

Design rationale: The reason for introducing `Interval` types explicitly in the language is that some special operations shall be defined for them.

7.4 Record types

Records are compounds of several data *elements* (also called *components* or *fields*) and are similar to RECORDs in Modula-2 or structs in C. Records types consist of internal data elements and a number of operations, which can be performed on them, the so-called *record procedures* (and record functions).

Objects of record types may occur as elements in any other complex type, as global variables, and as local variables in all kinds of procedures and functions.

Record Type Definition[33] ≡ 33
```
{
RecordTypeDef:
   'RECORD' RecordElemDefList.

RecordElemDefList:
   RecordElemDef ';' /
   RecordElemDefList RecordElemDef ';'.

RecordElemDef:
   RecordDataElemDef /
   MergeProcDef /
   ObjProcedureDef /
   ObjFunctionDef.
}
```
This macro is defined in definitions 33 and 34.
This macro is invoked in definition 31.

For the meaning and restrictions of procedure and function definitions in records, see section 9. The data element definition will be explained now:

Record Type Definition[34] ≡ 34
```
{
RecordDataElemDef:
   TypeId InitElemIdList.

InitElemIdList:
   InitElemId /
   InitElemIdList ',' InitElemId.
```

```
    InitElemId:
       NewElemId /
       NewElemId ':=' Expression.

    NewElemId:
       LowercaseIdent.

    Type Identifier[35]
    }
```

This macro is defined in definitions 33 and 34.
This macro is invoked in definition 31.

35 *Type Identifier*[35] ≡

```
    {
    TypeId:
       UppercaseIdent.
    }
```

This macro is invoked in definition 34.

Each name in the initialized-identifier list introduces an element of the record in the sense of a record field in Modula-2 or a component of a struct in C. The name of the element is local to the record, i.e., the same name may be used again as the name of a procedure or data object or as the name of an element in a different structure type.

Elements of records may be of simple type, interval type, record type, or array of those. Initializers for individual elements in a record type may be given. The meaning of an initializer `x` at an element `c` is that for each object A of the record type the element `c` of this object is initialized to the value of expression `x` upon creation of that object A. The initializer may consist of any expression of objects visible at that point in the program. The type of the expression must be compatible to the type of the element.

Other initializers for elements of the record type may exist in the object declarations using the record type or in the declarations of types that contain elements of the record type. These initializers apply later and thus overwrite the effect of the initializers here.

Example:

```
TYPE Atype IS RECORD  Int a, b = 7;  END
TYPE Btype IS RECORD  A x = A (2, 5);
                      Int c = 0;     END
```

Here, element `b` will be initialized to 7 in an `Atype` object, while element `x.b` will be initialized to 5 in an `Btype` object. Element `a` will be initialized to an undefined value in an `Atype` object, because no initializer is given. Programs that rely on certain values in such undefined objects are erroneous.

7.5 Node types

The nodes are the active elements of neural computation (some people call them *units* or even *neurons*). In CuPit, Nodes consist of input and output interface elements, internal data elements, and a number of operations, the so-called *node procedures*, that operate on the internal data elements and the connections attached to the interface elements.

Objects of node types may only occur as members in objects of group types and array types. They are not allowed as global variables or as local variables or parameters in any kind of procedure or function.

36 *Node Type Definition*[36] ≡

```
    {
```

```
NodeTypeDef:
   'NODE' NodeElemDefList.

NodeElemDefList:
   NodeElemDef ';' /
   NodeElemDefList NodeElemDef ';'.

NodeElemDef:
   NodeInterfaceElemDef /
   NodeDataElemDef /
   MergeProcDef /
   ObjProcedureDef /
   ObjFunctionDef.

NodeDataElemDef:
   TypeId InitElemIdList.
}
```

This macro is defined in definitions 36 and 37.
This macro is invoked in definition 31.

For the meaning and restrictions of procedure and function definitions and merge procedure definitions in nodes, see section 9. The data element definitions are analogous to those in record types and obey the same rules. The node interface element definitions will be explained now:

Node Type Definition[37] ≡ 37

```
{
NodeInterfaceElemDef:
   InterfaceMode TypeId InterfaceIdList.

InterfaceMode:
   'IN' /
   'OUT'.

InterfaceIdList:
   NewInterfaceId /
   InterfaceIdList ',' NewInterfaceId.

NewInterfaceId:
   LowercaseIdent.
}
```

This macro is defined in definitions 36 and 37.
This macro is invoked in definition 31.

Interface elements are no data elements but instead have the property that connections can be attached to them. The type name given in an interface element definition must be the name of a connection type; only connections of this type can be attached to the interface element. For interface mode `IN`, the connections are incoming connections: the output of these connections is connected to the interface element. For interface mode `OUT` the connections are outgoing connections: the input of these connections is connected to the interface element. The name of an interface element of a particular node object stands for all the connections that are attached to that interface element at once. The visibility of the name of an interface element obeys the same rules as the visibility of the name of a data element.

It is allowed to have several interface elements with the same interface mode in a single node type. Initializers for interface elements cannot be given in a node type declaration.

7.6 Connection types

Connections are the communication paths along which data flows from one node to another in a network. A connection object may contain arbitrary data and may perform arbitrary operations on it.

Objects of connection types cannot be declared explicitly; they may occur only implicitly connected to an output interface element of one node and to an input interface element of another (maybe the same) node. They are not allowed as members in any other type, nor as global variables nor as local variables in any kind of procedure or function. They can, however, be passed as parameters to external functions.

Connections are directed, i.e., they are not connections *between A and B*, but either *from A to B* or *from B to A*. Nevertheless, data can be transported along a connection in both directions. *Design rationale:* Connections must be directed because otherwise it is very difficult to provide an efficient implementation: Without direction it is not possible to store the actual connection data always at, say, the input end of the connection; thus we could not achieve data locality between connections and (at least one of the two) attached nodes.

38 *Connection Type Definition*[38] ≡

```
{
 ConnectionTypeDef:
    'CONNECTION' ConElemDefList.

 ConElemDefList:
    ConElemDef ';' /
    ConElemDefList ConElemDef ';'.

 ConElemDef:
    ConDataElemDef /
    MergeProcDef /
    ObjProcedureDef /
    ObjFunctionDef.

 ConDataElemDef:
    TypeId InitElemIdList.

}
```

This macro is invoked in definition 31.

For the meaning and restrictions of procedure and function definitions and merge procedure definitions in connections, see section 9. The data element definitions are analogous to those in record and node types and obey the same rules.

7.7 Array types

Arrays are linear arrangements of several data elements of the same type (called the *base type* of the array). The number of data elements in the array is called the *size* of the array. The elements can be accessed individually by means of an *index* as known from Modula-2. The lowest index to an array is always 0, the highest index is the size of the array minus one. An attempt to access an array using a negative index or an index that is too large is a run-time error.

Objects of array types may be used wherever objects of their element type may be used.

39 *Array Type Definition*[39] ≡

```
{
 ArrayTypeDef:
    'ARRAY' '[' ArraySize ']' 'OF' TypeId.
```

```
ArraySize:
   Expression.
}
```
This macro is invoked in definition 31.

The array size expression must be of integer type and must contain only constant values, so that it can be evaluated at compile time. The value of the expression determines the size of the array.

7.8 Group types

Groups are linear arrangements of several data elements of the same type (called the *base type* of the group). The number of data elements in the group is called the *size* of the group. The elements can be accessed individually by means of an *index* just like for an array. The lowest index to a group is always 0, the highest is the size of the group minus one. An attempt to access a group using a negative index or an index that is too large results in a run-time error.

Objects of group types may occur only as elements of network types. The base type of a group type must be a node type.

This far, groups and arrays are mostly the same. The main difference between groups and arrays is that groups are dynamic in size: There are operations to add new elements to a group at the end of the current index range, or to delete elements from the end of the current index range (see section 10.8). These operations cause the size of the group to change, but keep the indices of those elements of the group constant that already existed before the operation and still exist after it. In contrast to these operations there are others, which also change the size of the group, but do *not* necessarily leave the indices of constantly existing elements unchanged: Elements of a group can self-delete, even if they are not the last ones of the group and elements of a group can self-replicate (i.e. make one or several additional copies of themselves), even if they are not the last element of the group (see section 10.8). Such operations cause the indices of all the elements of the group to be recomputed. For arrays, the identity of an element with index `i` remains constant for the whole lifetime of the element. This is not true for groups: A constant index `i` is not guaranteed to refer to the same object in a group after a self-delete or self-replicate operation has been performed on the group (see section 10.8). The initial size of a group is 0.

Group Type Definition[40] ≡ 40
```
{
 GroupTypeDef:
    'GROUP' 'OF' TypeId.
}
```
This macro is invoked in definition 31.

7.9 Network types

Networks are the central data structures of neural algorithms. A network contains one or more groups or arrays of nodes, which are interconnected by connections. Other data may also be present in a network. Objects of network types may occur only as global variables.

Network Type Definition[41] ≡ 41
```
{
 NetworkTypeDef:
    'NETWORK' NetElemDefList.

 NetElemDefList:
    NetElemDef ';' /
    NetElemDefList NetElemDef ';'.
```

```
NetElemDef:
   NetDataElemDef /
   MergeProcDef /
   ObjProcedureDef /
   ObjFunctionDef.

NetDataElemDef:
   TypeId InitElemIdList.
}
```

This macro is defined in definitions 41.
This macro is invoked in definition 31.

The data element definitions for a network type are similar to those of a node type, except that arrays and groups of nodes are allowed as elements additionally. Arrays and groups cannot be initialized explicitly. Nodes can be used as elements of a network only in groups or arrays — individual nodes are not allowed.

8 Data object definitions

42 *Data Object Definition*[42] ≡

```
{
DataObjectDef:
   TypeId AccessType InitDataIdList.

AccessType:
   'CONST' /
   'VAR' /
   'IO'.

InitDataIdList:
   InitDataId /
   InitDataIdList ',' InitDataId.

InitDataId:
   NewDataId /
   NewDataId ':=' Expression.

NewDataId:
   LowercaseIdent.
}
```

This macro is invoked in definition 85.

Objects can be defined as either constants or variables or I/O areas. The only difference between constants and variables is that constants *must* be initialized and cannot be assigned to at any other point in the source code. It is possible, though, that a constant is not really allocated in memory as a data object at run-time when its properties are completely known at compile-time.

The I/O area data object category is CuPit's way to handle input and output. The exact layout and handling of I/O area objects are machine-dependent and must be specified separately for each compiler. *Design rationale:* Since the semantics of actual parallel I/O are tricky, CuPit defines only buffer operations and leaves the actual transfer of these buffers to machine-dependent external procedures.

An I/O area is a data object that is used to move data into a CuPit program from and out of a CuPit program to an external program part. Defining an I/O area basically means to declare a name for a variable whose storage must be allocated by an external program part and whose memory layout is defined by each CuPit compiler in a target machine dependent way. I/O areas can be used as arguments to external functions and special CuPit operators exist to move data from an I/O area into a group or

array of nodes or vice versa (see section 10.3). I/O areas are allowed to occur everywhere. However, in network or node or connection procedures they are usually useless.

9 Subroutine definitions

9.1 Overview

There are several types of subroutines in CuPit:

1. Procedures.
2. Functions.
3. Object procedures and object functions, which are much like normal procedures and functions.
4. Reduction functions, to combine many values into one.
5. Winner-takes-all functions, to reduce a parallel context into a sequential one.
6. Object merge procedures, to unite multiple replicates of a data object into one.

We will explain each of these types in order.

Subroutine Definition[43] ≡ 43
```
{
 Procedure Definition[44]
 Function Definition[45]
 Reduction Function Definition[46]
 Winner-takes-all Function Definition[47]
 Object Merge Procedure Definition[48]
 Statements[49]
 }
```
This macro is invoked in definition 85.

9.2 Procedures and functions

Procedure Definition[44] ≡ 44
```
{
 ProcedureDef:
    'PROCEDURE' NewProcedureId SubroutineDescription OptPROCEDURE.

 NewProcedureId:
   LowercaseIdent.

 SubroutineDescription:
    ParamList 'IS' SubroutineBody 'END' /
    ParamList 'IS' 'EXTERNAL'.

 ParamList:
    '(' ')' /
    '(' Parameters ')'.

 Parameters:
    ParamsDef /
    Parameters ';' ParamsDef.

 ParamsDef:
    TypeId AccessType ParamIdList.
```

```
ParamIdList:
   NewParamId /
   ParamIdList ',' NewParamId.

NewParamId:
   LowercaseIdent.

SubroutineBody:
   Statements.

OptPROCEDURE:
   /* nothing */ /
   'PROCEDURE'.

ObjProcedureDef:
   'PROCEDURE' NewObjProcedureId SubroutineDescription OptPROCEDURE.

NewObjProcedureId:
   LowercaseIdent.
}
```

This macro is invoked in definition 43.

The semantics of a procedure definition is similar to that of a procedure definition in Modula-2:
`PROCEDURE p (CONST T1 a, b; VAR T2 c) IS stmts END`
defines a procedure with the name `p` with three parameters. The parameters `a` and `b` have type `T1` and are available in the body of the procedure just like constants of same name and type, i.e. they may be read but not assigned to. Parameter `c` has type `T2` and is available in the body of the procedure just like a variable of same name and type. The body of the procedure consists of `stmts`.

If the procedure definition is part of the definition of a record type, node type, connection type, or network type, the procedure is called an *object procedure*. In this case, the object for which the procedure has been called is visible as `ME` in the procedure body. All elements of that object are visible and can be accessed using the selection syntax (e.g. `ME.a` to access a element `a`). `VAR` parameters are allowed for an object procedure only when the procedure is only called from other object subroutines of the same type. Otherwise, object procedure definitions are just like normal procedure definitions.

If the procedure body is replaced by the `EXTERNAL` keyword, the procedure is only declared, but not defined and must be implemented externally. *Design rationale:* The purpose of an `EXTERNAL` procedure definition is to make procedures and their parameter lists visible, so that a CuPit program can call them.

Parameters of node or connection types are allowed for external procedures only.

45 *Function Definition*[45] ≡

```
{
FunctionDef:
   TypeId 'FUNCTION' NewFunctionId SubroutineDescription OptFUNCTION.

NewFunctionId:
   LowercaseIdent.

OptFUNCTION:
   /* nothing */ /
   'FUNCTION'.

ObjFunctionDef:
   TypeId 'FUNCTION' NewObjFunctionId SubroutineDescription OptFUNCTION.
```

```
NewObjFunctionId:
   LowercaseIdent.
}
```
This macro is invoked in definition 43.

The semantics of a function definition is analogous to that of a procedure definition. The difference is that for a function a return type has to be declared. The value is returned in the function body using the `RETURN` statement with an expression. Connection, node, and network types are not allowed as return types of functions. An object function definition looks exactly like a normal function definition. The only difference is that for an object function definition the `ME` object that denotes the object the function was called for is visible in the body; neither `ME` nor its elements can be changed. No function may have a `VAR` parameter.

9.3 Reduction functions

Reduction Function Definition[46] ≡ 46
```
{
ReductionFunctionDef:
   TypeId 'REDUCTION' NewReductionFunctionId 'IS'
      ReductionFunctionBody 'END' OptREDUCTION.

NewReductionFunctionId:
   LowercaseIdent.

ReductionFunctionBody:
   Statements.

OptREDUCTION:
   /* nothing */ /
   'REDUCTION'.
}
```
This macro is invoked in definition 43.

The definition of a reduction function introduces a binary operator (which must be commutative and associative). This operator is used to reduce multitudes to single values in an implicit way.
For a declaration `T REDUCTION op IS body END`, the objects `ME` and `YOU` are implicitly declared as `T CONST` and are visible in `body`. `T` is the type of the values that can be reduced by this reduction function.

Reduction functions can be declared only globally (i.e. outside of type definitions and procedure definitions) and are used in three different contexts: First, in a node subroutine to reduce the values delivered to a node by the set of connections attached to a single connection interface; second, in a network subroutine to reduce the values of a particular data element of all nodes of a single node group or node array, and third, in a global subroutine to reduce the values of a particular data element of all replicates of a single network.

Design rationale: A reduction function declaration can be used to construct an efficient reduction procedure that runs in logarithmic time on a parallel machine and uses knowledge about the specific data distribution in order to avoid communication operations.

Example: Given the definition
`Real REDUCTION sum IS RETURN (a+b) END`
then in a node procedure of a node type having a connection interface `in` of a connection type having a `Real` data element `val`, the statement
`REDUCTION ME.in[].val:sum INTO inSum;`
means to apply the `sum` reduction to the `val` fields of all connections attached to the `in` interface. Assuming that there are exactly three connections whose `val` values are `x, y, z`, respectively. The value of `inSum` after the statement will be either `(x+y)+z` or `x+(y+z)` or `(x+z)+y` or any commutation of one of these.

9.4 Winner-takes-all functions

47 *Winner-takes-all Function Definition*[47] ≡

```
{
 WtaFunctionDef:
    TypeId 'WTA' NewWtaFunctionId 'IS' WtaFunctionBody 'END' OptWTA.

 NewWtaFunctionId:
    LowercaseIdent.

 WtaFunctionBody:
    Statements.

 OptWTA:
    /* nothing */ /
    'WTA'.
}
```

This macro is invoked in definition 43.

The definition of a winner-takes-all function introduces a binary operator. This operator is a comparison operator and is used to induce an ordering on the type for which the operator is defined.

For a declaration `T WTA op IS body END`, the objects `ME` and `YOU` are implicitly declared as `T CONST` and are visible in `body`. The `body` must return a `Bool` result; `true` means that `ME` is above `YOU` in the ordering defined by the operator and `false` means that it is not.

Winner-takes-all functions can be declared only globally (i.e. outside of type definitions) and are used in three different contexts: First, to select one connection per node from the sets of connections attached to a certain node interface of each node in a group of nodes; second, to select one node from a group of nodes; and third, to select a network from a set of replicated networks. See section 10.6.

Design rationale: The winner of a winner-takes-all call is always unique. Thus it is not easily possible to emulate a winner-takes-all function by a reduction and subsequent rebroadcast of the result, because the winning *value* need not be unique. A winner-takes-all function declaration can be used to construct an efficient reduction procedure that runs in logarithmic time on a parallel machine and uses knowledge about the specific data distribution in order to avoid communication operations.

9.5 Merge procedures

48 *Object Merge Procedure Definition*[48] ≡

```
{
 MergeProcDef:
    'MERGE' 'IS' MergeProcedureBody 'END' OptMERGE.

 MergeProcedureBody:
    Statements.

 OptMERGE:
    /* nothing */ /
    'MERGE'.
}
```

This macro is invoked in definition 43.

Merge procedures are similar to reduction functions; they also perform a reduction. Their purpose is to reunite replicated exemplars of networks or individual network elements. For a description of network replication, see section 10.8.

While replication and subsequent merging can only be *executed* for a whole network, merging is *defined* in network types, node types, and connection types separately. This way, the knowledge about how merging works for particular object types remains local to the definitions of these types.

When network merging is called, each merge procedure of the network elements is implicitly called as an object procedure, i.e., the object for which it has been called is available as `ME`. This object is also where the result of the merging has to be placed by the merge procedure. The object to be merged into `ME` is available as `YOU` with `CONST` access, i.e., writing to elements of `YOU` is not allowed. The task of the merge procedure body is to construct in `ME` the reunion of `ME` and `YOU`. A merge procedure of a network type should merge all relevant non-node elements of the network. The node elements are merged one-by-one by the respective merge procedures of the node types. A merge procedure of a node type should merge all relevant non-interface elements of the node. The connections attached to the node are merged one-by-one by the respective merge procedures of the connection types.

If no merge procedure is defined for a particular type, no merging occurs and the reunited exemplar of each object of this type is identical to a random one of the replicated exemplars of the object. *Design rationale:* Often, most of the data structures do not really need to be merged in neural algorithms.

However, merging is still performed on the enclosed parts of the data structure. That is, if no merge procedure for a network is defined, merging can still occur for the nodes and connections of this network, if merging procedures for them are defined. If no merge procedure for a node is defined, merging can still occur for its connections. *Open question: Do we need the capability to declare multiple merge procedures for the same type ?*

10 Statements

10.1 Overview

The statements available in CuPit can be divided into the following groups:

1. Statements that are common in sequential procedural languages, such as assignment, control flow, procedure call.
2. Statements that imply parallelism, such as group procedure call and reductions.
3. Statements that modify the number of data objects, i.e., create new objects or delete existing ones.

Each list of statements can have some data object definitions at its beginning. The objects declared this way are visible only locally in the list of statements. They are created at run-time just before the list is executed and vanish as soon as the execution of the list is over. This introduces a kind of block structure for local data objects into CuPit that is similar to that of C.

Statements[49] ≡ 49

```
{
Statements:
   DataObjectDefList StatementList.

DataObjectDefList:
   /* nothing */ /
   DataObjectDefList DataObjectDef ';'.

StatementList:
   /* nothing */ /
   StatementList Statement ';'.
}
```

This macro is invoked in definition 43.

Statement[50] ≡ 50

```
{
 Statement:
    Assignment /
    InputAssignment /
    OutputAssignment /
    ProcedureCall /
    ObjectProcedureCall /
    MultiObjectProcedureCall /
    ReductionStmt /
    WtaStmt /
    ReturnStmt /
    IfStmt /
    LoopStmt /
    BreakStmt /
    DataAllocationStmt /
    MergeStmt.
```

Assignment[51]
I/O Assignment[52]
Procedure Call[53]
Reduction Statement[56]
Wta Statement[57]
Return Statement[58]
If Statement[59]
Loop Statement[60]
Break Statement[61]
Data Allocation Statement[62]
Merge Statement[63]
}

This macro is invoked in definition 85.

All these kinds of statements will now be explained individually.

10.2 Assignment

51 *Assignment*[51] ≡

```
{
 Assignment:
    Object AssignOperator Expression.

 AssignOperator:
    ':=' / '+=' / '-=' / '*=' / '/=' / '%='.
}
```

This macro is invoked in definition 50.

The assignment `a := b` stores a new value (as given by the expression `b`) into a data object (`a`, in this case). The types of `a` and `b` must be compatible (see section 11.2) and `b` is converted into the type of `a` if necessary. The computation of the memory location to store to (the address of `a`) may involve the evaluation of expressions, too. In this case, the the left hand side and the right hand side are evaluated in undefined order (e.g. in parallel). The assignments `a += b, a -= b, a *= b, a /= b, a %= b` have the same meaning as `a = a+b, a = a-b, a = a*b, a = a/b, a = a%b`, except that any expressions involved in computing the address of `a` are evaluated only once.

The assignment to a node object N is defined only, when this node N does not yet have any connections attached to it.

Design rationale: This is because assignment to node objects is intended to be used for initialization only. During the rest of the program run, nodes should only be changed by themselves by means of node procedures.

10.3 I/O assignment

I/O Assignment[52] ≡ 52

```
{
 InputAssignment:
    Object '<--' Object.

 OutputAssignment:
    Object '-->' Object.
 }
```

This macro is invoked in definition 50.

The purpose of these special assignments is to provide a way of communication between a network and the "outer world": Since the mapping of nodes onto processors is completely left to (and known only by) the compiler, external procedures can not directly read data from nodes or write data into nodes. On the other hand, the memory mapping of I/O areas is statically defined by any compiler (see sections 8 and 17), so that external procedures can easily access them for reading and writing.

The object on the left hand side must be a data element of a group of nodes (i.e., a parallel variable), the object on the right hand side must be a global `X IO`, where `X` is the type of the data element field of the nodes mentioned on the left hand side.

A single input or output assignment statement provides one value for each node of the group in each of the replicates of the respective network. For the input assignment each such value is copied from the I/O area into the data element of the appropriate node according to the I/O area data layout defined for the particular compiler. For the output assignment the value is copied from the data elements of the nodes into the I/O area.

Input and output assignments are allowed in the central agent only. *Design rationale:* The central agent is conceptually the only part of the program where knowledge about network replication is present. Since input and output assignments work on all replicates at once and the program who fills or reads an I/O area must know that; the central agent is the only program part where input and output assignments make sense.

10.4 Procedure call

Procedure Call[53] ≡ 53

```
{
 ProcedureCall:
    ProcedureId '(' ArgumentList ')'.

 ProcedureId:
    LowercaseIdent.

 ArgumentList:
    /* nothing*/ /
    ExprList.
 }
```

This macro is defined in definitions 53, 54, and 55.
This macro is invoked in definition 50.

The semantics of a procedure call are similar to that known in languages such as Modula-2 or C: First, all formal parameters of the procedure are bound to the arguments of the procedure call. Then control

is transferred to the body of the procedure. This body is executed until its end or a **RETURN** statement is encountered. Then control is transferred back to the point immediately after the point at which the procedure was called. Procedure calls can be nested and so will be the extent of any local variables or parameters procedures created during a call.

The binding of arguments to parameters involves evaluating the arguments. This occurs in an undefined order (e.g. in parallel). Parameter binding may have either call-by-value or call-by-reference semantics for constant parameters and either copy-in-copy-out or call-by-reference semantics for variable parameters. Which of these is used for any single procedure call is left to the compiler. Any program that relies on a certain selection within these possibilities is erroneous.

Design rationale: The appropriateness of one or the other parameter passing mechanism depends on the particular data type to be passed and the actual parallel machine on which the program shall run. Thus, the compiler should have the freedom to choose the most efficient mechanism in each situation.

54 *Procedure Call*[54] ≡

```
{
 ObjectProcedureCall:
    Object '.' ObjectProcedureId '(' ArgumentList ')'.

 ObjectProcedureId:
    LowercaseIdent.
}
```

This macro is defined in definitions 53, 54, and 55.
This macro is invoked in definition 50.

An object procedure call, for which the object is an array or a group of nodes or an input or output interface of a node (referring to a set of connections) is called a *group procedure call*. A group procedure call means that the called object procedure is executed for all objects of the group in an asynchronously parallel fashion (i.e. in any sequential or overlapping order). This language construct introduces a level of object-centered parallelism. Such object-centered parallelism is similar to data parallelism but is more expressive than pure data parallelism, because more than a single assignment or expression can be evaluated in a single parallel statement and additional parallelism can be introduced in the body of an object-centered parallel operation.

Object procedure calls for network procedures are allowed in the central agent and in network procedures and functions. Object procedure calls for individual nodes or for arrays or groups of nodes are allowed in network procedures and network functions. Object procedure calls for individual nodes are allowed in node procedures and functions. Object procedure calls for input or output interfaces of nodes (thus calling a connection type object procedure) is allowed in node procedures and node functions. Object procedure calls for individual connections are allowed in connection procedures and functions.

55 *Procedure Call*[55] ≡

```
{
 MultiObjectProcedureCall:
    ObjectProcedureCall 'AND' ObjectProcedureCall /
    ObjectProcedureCall 'AND' MultiObjectProcedureCall.
}
```

This macro is defined in definitions 53, 54, and 55.
This macro is invoked in definition 50.

A multiple object procedure call means the execution of the individual object procedure calls in an asynchronously parallel fashion (i.e. in any sequential or overlapping order). This language construct introduces a level of process parallelism. This is the only kind of process parallelism supported in CuPit.

10.5 Reduction statement

56 *Reduction Statement*[56] ≡

```
{
 ReductionStmt:
    'REDUCTION' Object ':' ReductionFunctionId 'INTO' Object.

 ReductionFunctionId:
    LowercaseIdent.
}
```

This macro is invoked in definition 50.

Reduction statements are allowed in the central agent, in network procedures and functions and in node procedures and functions. For the meaning of the statement `REDUCTION obj.d:op INTO x` there are three cases:

`obj` can be a network variable. Then the call must be in the central agent and `d` is a data element of the network variable (i.e. not a node or a node group). In this case, the `op` reduction of the `d` elements in all replicates of the network is determined and stored into `x`.

Or `obj` is a group of nodes. Then the call must be in a network procedure and `d` is a data element of the base type of the node group (i.e. the node type). In this case, the `op` reduction of the `d` elements of all nodes of the node group is determined and stored into `x`.

Or `obj` is a connection interface element of a node. In this case, the call must be in a node procedure and `d` must be a data element of the connections attached to the interface. In this case, the `op` reduction of the `d` elements for each node of the node group are determined and stored into `x`.

If the set of objects to perform the reduction on is empty, `x` is not changed. The types of the object to reduce, the object to reduce into, and the reduction function must be the same.

10.6 Winner-takes-all statement

Wta Statement[57] ≡ 57

```
{
 WtaStmt:
    'WTA' Object ':' Elementname '.' WtaFunctionId ':'
    ObjectProcedureId '(' ArgumentList ')'.

 WtaFunctionId:
    LowercaseIdent.
}
```

This macro is invoked in definition 50.

Winner-takes-all statements are allowed in the central agent, in network procedures and functions and in node procedures and functions. For the meaning of the statement `WTA obj:d.op:p(params)` there are three cases:

`obj` can be a network variable. Then the call must be in the central agent and `d` is a data element of the network variable (i.e. not a node or a node group). In this case, the winner of the `d` elements in all replicates of the network with respect to the WTA function `op` is determined and the function `p` is called only for the winning network replicate.

Or `obj` is a group (or array) of nodes. Then the call must be in a network procedure and `d` is a data element of the base type of the node group (i.e. the node type). In this case, the winner of the `d` elements of all nodes of the node group with respect to the WTA function `op` is determined and the function `p` is called only for the winning node in each network replicate.

Or `obj` is a connection interface element of a node. In this case, the call must be in a node procedure and `d` must be a data element of the connections attached to the interface. In this case, the winners of the `d` elements for each node of the node group with respect to the WTA function `op` are determined and the function `p` is called only for the winning connection of each node in each network replicate.

The types of `d` and `op` must be the same. If the set `obj` of objects to pick the winner from is empty, the procedure `p` is not called at all.

10.7 Control flow statements

58 *Return Statement*[58] ≡

```
{
ReturnStmt:
   'RETURN' /
   'RETURN' Expression.
}
```

This macro is invoked in definition 50.

The `RETURN` statement is allowed in all kinds of functions and procedures. Its semantics is the immediate termination of the execution of the current procedure or function. In functions (and only in functions) an expression must be given, which must have a type that is compatible to the declared return type of the function. This expression is evaluated and (perhaps after an implicit type conversion to the return type) returned as the result of the function. Since a `RETURN` statement is the only way to return a value in a function, each function must have at least a `RETURN` statement at its end.

In group function or procedure invocations, the `RETURN` statement of course terminates only the calls that execute it, the others continue normal execution.

59 *If Statement*[59] ≡

```
{
IfStmt:
   'IF' Expression 'THEN' Statements ElsePart 'END' OptIF.

ElsePart:
   /* nothing */ /
   'ELSE' Statements /
   'ELSIF' Expression 'THEN' Statements ElsePart.

OptIF:
   /* nothing */ /
   'IF'.
}
```

This macro is invoked in definition 50.

The semantics of the `IF` statement is the same as in Modula-2.

60 *Loop Statement*[60] ≡

```
{
LoopStmt:
   OptWhilePart 'REPEAT' Statements OptUntilPart 'END' OptREPEAT /
   'FOR' Object ':=' Expression ForLoopStep Expression
      'REPEAT' Statements OptUntilPart 'END' OptREPEAT.

OptWhilePart:
   /* nothing */ /
   'WHILE' Expression.

OptUntilPart:
   /* nothing */ /
   'UNTIL' Expression.
```

```
OptREPEAT:
   /* nothing */ /
   'REPEAT'.

ForLoopStep:
   'UPTO' / 'TO' / 'DOWNTO'.
}
```

This macro is invoked in definition 50.

Loops are available in two forms: the normal loop and the `FOR` loop.

The normal loop can have two boolean conditions, both are optional. This combines the WHILE, UNTIL, and LOOP loop types of Modula-2 and has the intuitive semantics. The `WHILE` test defaults to true and the `UNTIL` test defaults to false. The `WHILE` test is evaluated immediately before each iteration of the loop body, the `UNTIL` test is evaluated immediately after each iteration of the loop body. Whenever a `WHILE` test yields false or an `UNTIL` test yields true, the loop terminates.

Design rationale: You won't need a combined while/until loop very often. But once you need it, it is really nice to have it.

The semantics of the `FOR` loop are be defined by the following transformation pattern: A loop of the form `FOR i := f TO t REPEAT s; UNTIL c END` has the meaning

```
i  := f;                (* initialization *)
t2 := t;                (* limit computation *)
WHILE i <= t2 REPEAT    (* FOR termination test *)
  s;                    (* body *)
  IF c THEN BREAK END;  (* UNTIL termination test *)
  i += 1;               (* count step *)
END
```

where `i` is an existing variable of integral type, `f` and `t` are arbitrary expressions of integral type, `s` is a list of statements, and `c` is a boolean expression. `t2` is an implicitly declared anonymous variable of the same type as `t` that is used for this loop only. The keyword `TO` may be replaced by `UPTO` without change in meaning. It may also be replaced by `DOWNTO`. In this case the "for termination test" is `i >= t2` and the "count step" is `i += -1`. In all three forms, the `UNTIL` test defaults to false, just as for the normal loop.

Break Statement[61] ≡ 61

```
{
 BreakStmt:
   'BREAK'.
 }
```

This macro is invoked in definition 50.

The `BREAK` statement is allowed in loops only. Its semantics is the immediate termination of the innermost textually surrounding loop, just like the `break` statement in C.

10.8 Data allocation statements

Data Allocation Statement[62] ≡ 62

```
{
 DataAllocationStmt:
   'REPLICATE' Object 'INTO' Expression /
   'EXTEND' Object 'BY' Expression /
   'CONNECT' Object 'TO' Object /
   'DISCONNECT' Object 'FROM' Object.
```

```
}
```

This macro is invoked in definition 50.

These statements allocate or deallocate nodes or connections or create or reunite network replicates.

10.8.1 Connection creation and deletion

The `REPLICATE` statement can in its first form be used in a connection procedure. In `REPLICATE ME INTO n`, the expression must be non-negative integral and gives the number of identical exemplars of this connection that shall exist after the replication statement has been executed. Zero means "delete myself", one means "do nothing". In connection procedures, only `REPLICATE ME INTO 0` and `REPLICATE ME INTO 1` are allowed (*Design rationale:* Only one connection can exist between any two node interfaces at any given time).

The rest of the procedure in which `REPLICATE` was called is not executed, i.e., the `REPLICATE` statement implies a `RETURN`. It is a run time error to call `REPLICATE` for a connection with an operand that is negative or larger than one or to call it while the whole network is replicated.

The `CONNECT` and `DISCONNECT` statements can only be used in network procedures to create or delete connections between two groups of nodes, which have to be given in the order origin–destination. The statement `CONNECT a[2...4].out WITH b[].in1` has the following semantics: `a` and `b` must be node arrays or node groups of the network for which the statement was issued. `out` must be an output interface of the nodes in `a`, `in1` must be an input interface of the nodes in `b`; the types of `in1` and `out` must be identical. The statement creates a connection from each of the nodes 2, 3, and 4 of `a` to each node of `b`. Generally speaking, the objects given in a `CONNECT` or `DISCONNECT` statement must be parallel variable selections (see section 12.3 on page 60) of connection type, where the first one is an output interface and the second an input interface. All newly created connections are initialized using the default initializers given in the respective connection type declaration. Connections that already exist are not created again and are not initialized again.

The `DISCONNECT` statements works in the same way, except that it deletes connections instead of creating them. If `CONNECT` is used to create connections that already exist, an additional exemplar of these connections may or may not be created; such use is non-portable and should be avoided. If `DISCONNECT` is used to delete connections of which multiple exemplars exist, all exemplars will be deleted. It is no error if some or all of the connections that a `DISCONNECT` statement conceptually would delete do not exist. It is a run time error to call `CONNECT` or `DISCONNECT` while the network is replicated. `CONNECT` may produce a run time error if there is not enough memory available on the machine.

10.8.2 Node creation and deletion

The `REPLICATE` statement can in its first form be used in a node procedure. In `REPLICATE ME INTO n`, the expression must be non-negative integral and gives the number of identical exemplars of this node that shall exist after the replication statement has been executed. Zero means "delete myself", one means "do nothing" and larger values mean "create `n`-1 additional exemplars". All incoming and outgoing connections of the node are cloned for each new exemplar when `REPLICATE` is called with a value of 2 or higher. The new nodes are inserted in the index range at the point of the old node (i.e. the replicates of a node will be in a contiguous subrange of the new index range). The new indices are computed in a way that maintains the order of the indices of the nodes (although not the indices itself). Example: In a node group with four nodes 1, 2, 3, 4, after a replicate statement where the nodes request 3, 1, 0, 1 replicates, respectively, the new indices 1, 2, 3, 4, 5 will be given to the nodes stemming from the nodes with old indices 1, 1, 1, 2, 4, respectively. The statement can produce a run time error if it creates so many new nodes that the machine runs out of memory, if it is called for nodes that are not part of a `GROUP` but part of a node `ARRAY` instead, and if it is called while the network is replicated.

`REPLICATE` implies `RETURN`, i.e., the procedure that calls it terminates after the replication has been performed. *Open question: This is a bit ugly. But what is the semantics otherwise? And how would you implement it?*

The `EXTEND` statement can only be used in network procedures for nodes that belong to a node `GROUP`. `EXTEND g BY n` means that the group of nodes `g` shall be extended by `n` new nodes (or reduced by `-n` nodes if `n` is negative). The nodes are added or removed at the upper end of the group's current index range. The new nodes, if any, are initialized using the default initializers as given in the type declaration of the node type, if any. The new nodes do not have any connections initially. It is a run time error if the size that the group `g` would have after the `EXTEND` is negative, if `EXTEND` is called while the network is replicated, or if there is not enough memory on the machine.

10.8.3 Network replication

The network replication statement is allowed in the central agent only. The object must be a network variable. The expression must have integral or integer interval type.

Design rationale: At the beginning of the existence of a network variable, the corresponding object exists as a single exemplar (as one would usually expect for any variable of any type). Since many Neural Algorithms allow input example parallelism, i.e., the simultaneous independent processing of several input examples, CuPit allows network objects to be *replicated*. This is what the network replication statement is for.

`REPLICATE nw INTO 3`, for example, tells CuPit to create 3 exemplars of the network object designated by the variable `nw`. The exemplars are identical copies of the original object. The input assignment and output assignment statements, though, allow to feed different data into and read different data from each of the exemplars. `REPLICATE nw INTO 3...20`, tells CuPit to create any number of exemplars of the network object it would like to, provided it is in the range 3 to 20. The compiler chooses the number of replicates that it thinks will make the program run fastest.

Design rationale: The compiler may have a lot more knowledge about available memory and the cost of replicating, reuniting (merging), and operating on several replicates in parallel than the programmer has. It should thus be given some freedom to optimize the parameter "number of network replicates". A compiler may for example choose to prefer network replication with numbers of replicates that are powers of two, because this is the most efficient on the particular target machine.

While a network is replicated, all network procedure calls are automatically executed by all exemplars of the network. For network functions the behavior is different, depending on where they are being called from: If a network function is called from the central agent or from another network function that has been called from the central agent, the function is executed and the results are returned for the first exemplar of the network only. If it is called from a network procedure or from another network function that has been called from a network procedure, execution occurs on all exemplars of the network and a value is returned for all exemplars as well.

`REPLICATE nw INTO 1` reunites the replicated exemplars to a single object again, using the `MERGE` procedures as defined in the network type and the relevant node and connection types. The two states 'replicated' and 'non-replicated' have an important difference: While a network is replicated, no `CONNECT`, `DISCONNECT`, `REPLICATE`, or `EXTEND` commands must be issued for its parts. This restriction is necessary because it is not clear how replicates with differing topology could be merged. The advantage of the restriction is that it may allow the compiler to work with a more efficient data distribution in replicated state. Even if a program uses only one replicate all the time, it can switch between "topology changes allowed but data distribution maybe less efficient" and "topology changes forbidden but data distribution is most efficient" by using `REPLICATE nw INTO 1...1` for the latter.

The number of exemplars minus one that currently exist can be inquired for any network variable using the `MAXINDEX` operation. It is a run-time error, to request a number of replicates that is not strictly positive or to request network replication while the network is already replicated.

10.9 Merge statement

Merge Statement[63] ≡ 63

```
{
 MergeStmt:
    'MERGE' Object.
 }
```

This macro is invoked in definition 50.

The statement `MERGE nw` applies the respective `MERGE` procedures to all parts of all replicates of the network `nw`, thus collecting the data from all the replicates in the first replicate, and then redistributes this data from the first replicate to all other replicates again. After a `MERGE`, the values of all corresponding data elements that are merged by the merge procedures of the respective data types are identical in the different network replicates. It is undefined whether the data elements not modified by the individual `MERGE` procedures retain their previous values in all replicates or are all changed to the values of the corresponding data elements of the first replicate. The `MERGE` statement can only be called from the central agent.

Design rationale: It is often useful to reunite the data in network replicates without actually destroying the replicated network, because the next thing the program does is to create replicates again, anyway. This is the case when the purpose of reuniting the replicates is not a change in network topology but only the collection of data from the replicates.

11 Expressions

11.1 Overview

Most of the expression syntax and semantics of CuPit is well-known from common procedural languages: Mentioning an object uses it as a value, a function can be called with arguments and returns a value, operators are used to combine values generating new values, all values have a type, there are restrictions on type compatibility for the application of operators, and values of some types can explicitly be converted into values of other types. There are, though, a few special expressions, which are concerned with handling dynamic data structures and accessing object elements. The concrete operators that are available can be seen in table 1.

11.2 Type compatibility and type conversion

For most binary operations (including assignment and parameter passing), the two operands must be *compatible*. In the current version of CuPit, two types A, B are compatible only if they are the same; the exception to this rule is automatic promotion from smaller to larger integer types and integer interval types according to the following rules: Two integer types A and B are compatible if and only if either

1. they are the same or
2. A is smaller than B and A is `not` the type of the left-hand object in an assignment or the formal parameter in an argument passing, or
3. B is smaller than A and B is `not` the type of the left-hand object in an assignment or the formal parameter in an argument passing.

In the latter two cases, the smaller operand is converted into the type of the larger one. Formal parameters can not be converted, nor can objects that are passed as arguments to a `VAR` or `IO` formal parameter. Integer denoters have smallest integer type that can represent their value. Analogous rules apply to integer intervals.

For explicit type conversion, see page 57. The set of explicit type conversions that are available can be described as follows. There are type constructors that generate an object of a certain type T from objects of the component types of X: For each record type there is a conversion from a complete set of record elements to the record type, e.g. an object of `TYPE Rec IS RECORD REAL a; INT b; BOOL c; INT d; END` can be constructed by `Rec(3.0,7,false,0)`. The order of the arguments for the conversion is the order in which the elements of the record were defined. Type constructors for array or group types do not exist.

Prio	Appearance	Purpose
1	`?:`	ternary if-then-else expression
2	`OR`	Boolean or
2	`XOR`	Boolean exclusive or
3	`AND`	Boolean and
4	`= <> < >`	Equality, inequality, less than, greater than
4	`<= >=`	Less than or equal, greater than or equal
4	`IN`	Interval hit
5	`BITOR`	Bitwise or
5	`BITXOR`	Bitwise exclusive or
5	`...`	Interval construction
6	`BITAND`	Bitwise and
7	`LSHIFT`	Leftshift
7	`RSHIFT`	Rightshift
8	`+ -`	Addition
9	`* / %`	Multiplication, Division, Modulo
10	`**`	Exponentiation
11	`NOT`	Unary boolean not
11	`BITNOT`	Unary bitwise not
11	`-`	Unary arithmetic negation
11	`MIN`	Access minimum of interval
11	`MAX`	Access maximum of interval
11	`RANDOM`	Random number generation
11	`-`	Unary arithmetic negation
11	`Type`	Explicit type conversion or construction
12	`[]`	Array/group subscription, parallel variable creation
13	`.`	Record element selection
14	`()`	Grouping

Table 1: Operators in CuPit

11.3 Operators

Expression[64] ≡ 64

```
{
 Expression:
    E1.

 ExprList:
    Expression /
    ExprList ',' Expression.
}
```

This macro is defined in definitions 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, and 77.
This macro is invoked in definition 85.

All operators can be used in a constant expression. The only requirement is that the values of all operands must be available at compile time. The compiler performs as much constant folding as possible with real, integer, and boolean values in order to produce constant expressions where necessary. The compiler may, but need not, fold constants in other contexts, too. Note that this may change the semantics of a program, if the compilation machine's arithmetic is not exactly equivalent to that of the target machine.

Expression[65] ≡ 65

```
{
 E1:
    E2 '?' E2 ':' E2 /
```

```
      E2.
   }
```

This macro is defined in definitions 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, and 77.
This macro is invoked in definition 85.

This is the useful if-then-else operator known from C. Note that it is non-associative in CuPit: In order to nest it, parentheses must be used. The first expression must have boolean type, the second and third must have compatible types.

`A ? B : C` has the following semantics: First, `A` is evaluated and must yield a `Bool`. Then, if `A` is true, `B` is evaluated and returned, otherwise `C` is evaluated and returned. The types of `B` and `C` must be compatible; implicit type conversion is performed on them as if they were an operand pair.

66 *Expression*[66] ≡

```
   {
    E2:
       E2 OrOp E3 /
       E3.

    OrOp:
       'OR' / 'XOR'.

    E3:
       E3 AndOp E4 /
       E4.

    AndOp:
       'AND'.
   }
```

This macro is defined in definitions 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, and 77.
This macro is invoked in definition 85.

These are the usual logical operators: both of their operands must have type `Bool`, the result has type `Bool`, too. `a OR b` is true iff either `a` or `b` or both are true. `a XOR b` is true iff either `a` or `b` but not both are true. `a AND b` is true iff both, `a` and `b`, are true. The operands are evaluated in undefined order.
Open question: Do we need this freedom ? Or would it be better to define left-to-right shortcut evaluation ?

67 *Expression*[67] ≡

```
   {
    E4:
       E5 CompareOp E5.

    CompareOp:
       '='  /  '<>'  /  '<'  /  '>'  /  '<='  /  '>='.
   }
```

This macro is defined in definitions 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, and 77.
This macro is invoked in definition 85.

These are the usual comparison operators: both of their operands must be numbers or enumerations; their types must be compatible. The result has type `Bool`. The result for the comparison of `SYMBOLIC` values is well-defined only for the `'='` and `'<>'` test. The other tests yield a result without any special meaning for these types, but this result is constant within the same run of the program. The operands are evaluated in undefined order.

68 *Expression*[68] ≡

```
   {
    E4:
       E5 InOp E5 /
       E5.
```

```
InOp:
    'IN'.
}
```

This macro is defined in definitions 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, and 77.
This macro is invoked in definition 85.

This is the interval test operator. The left operand must have a number type, the right operand must have the corresponding interval type. The `IN` operator returns true if the value of the left operand lies in the interval and false otherwise.

Expression[69] ≡ 69

```
{
E5:
    E5 BitorOp E6.

BitorOp:
    'BITOR' / 'BITXOR'.
}
```

This macro is defined in definitions 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, and 77.
This macro is invoked in definition 85.

These are logical operators that operate bitwise. Both of their operands must be integral numbers; their types must be compatible. The operation `a BITOR b` means that for every bit position in the internal representation of `a` and `b` (after the type conversion required by the compatibility has been performed) a logical OR operation is performed, just as the `OR` operator does. A zero bit corresponds to false and a one bit corresponds to true. `BITXOR` is defined analogously. Since the internal representation of integral numbers is not defined in CuPit, the result of these operators is generally machine-dependent. The operands are evaluated in undefined order.

Expression[70] ≡ 70

```
{
E5:
    E6 IntervalOp E6 /
    E6.

IntervalOp:
    '...'.
}
```

This macro is defined in definitions 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, and 77.
This macro is invoked in definition 85.

This is the interval construction operator: both operands must be numbers and must have compatible types. The result of `a...b` is a `Realerval` if `a` and `b` have type `Real` and an `Interval` if `a` and `b` have types compatible to `Int`. Objects of type `Interval1` and `Interval2` can only be generated by explicit type conversion.

The interval is empty, if `a` $>$ `b`, otherwise it contains all numbers x of type `Int` or `Real`, respectively, for which `a`$\leq x \leq$`b`. `a` and `b` are evaluated in undefined order.

Expression[71] ≡ 71

```
{
E6:
    E6 BitandOp E7 /
    E7.

BitandOp:
    'BITAND'.
}
```

This macro is defined in definitions 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, and 77.
This macro is invoked in definition 85.

This is the bitwise logical AND operator. Works analogous to `BITOR`, but performs a bitwise `AND` operation.

72 *Expression*[72] ≡

```
{
 E7:
    E7 ShiftOp E8 /
    E8.

 ShiftOp:
    'LSHIFT' / 'RSHIFT'.
}
```

This macro is defined in definitions 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, and 77.
This macro is invoked in definition 85.

These are shift operators working on the bit representation of the left operand. Both operands must be of integral type. The result for negative values of `a` or `b` is machine-dependent; otherwise `a LSHIFT b` (where `a` has type `A`) is equivalent to `A(a*2**b)` and `a RSHIFT b` (where `a` has type `A`) is equivalent to `A(a/2**b)`. The operands are evaluated in undefined order.

73 *Expression*[73] ≡

```
{
 E8:
    E8 AddOp E9 /
    E9.

 AddOp:
    '+' / '-'.
}
```

This macro is defined in definitions 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, and 77.
This macro is invoked in definition 85.

Addition and subtraction of numbers. Both operands must be of compatible type. The exact semantics of these operations is machine-dependent (but will be the same on almost all machines). The operands are evaluated in undefined order.

74 *Expression*[74] ≡

```
{
 E9:
    E9 MulOp E10 /
    E10.

 MulOp:
    '*' / '/' / '%'.
}
```

This macro is defined in definitions 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, and 77.
This macro is invoked in definition 85.

Multiplication, division, and modulo operation on numbers. For multiplication and division, both operands must have compatible type. The exact semantics of these operations is machine-dependent (but will be the same on almost all machines, except perhaps for division and modulo by negative integers). For modulo, the right operand must have integral type. `a % b` where `a` is integral is defined as `a-b*(a/b)` for positive `a` and `b` and is machine-dependent if either is negative. `a % b` where `b` has type `Real` is defined as `a-b*R(Int((a/b)))`. The operands are evaluated exactly once, in undefined order.

75 *Expression*[75] ≡

```
{
```

```
E10:
   E11 ExponOp E10 /
   E11.

ExponOp:
   '**'.
}
```

This macro is defined in definitions 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, and 77.
This macro is invoked in definition 85.

The exponentiation `a**b` is defined

1. for integral `a` and non-negative integral `b` with result type `Int`.
2. for real `a` and integral `b` with result type `Real`.
3. for non-negative real `a` and arbitrary real `b` with result type `Real`.

In all cases, the meaning is a^b, where 0^0 equals 1. The behavior upon overflow is machine-dependent. The compiler may provide run-time checking. The operands are evaluated in undefined order.

Expression[76] ≡ 76

```
{
E11:
   UnaryOp E12 /
   TypeId '(' ExprList ')' /
   'MAXINDEX' '(' Object ')' /
   E12.

UnaryOp:
   'NOT' / 'BITNOT' / '-' / 'MIN' / 'MAX' / 'RANDOM'.
}
```

This macro is defined in definitions 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, and 77.
This macro is invoked in definition 85.

These are unary operators. All unary operators have the same precedence.

`NOT a` is defined iff `a` has type `Bool`; it returns false, if `a` is true and true if `a` is false.

`BITNOT a` is defined iff `a` has an integer type. The internal representation of `a` is returned complemented bitwise. The result has the same type as `a`.

`-a` is defined iff `a` has a number type. The result is the same as `(0-a)`.

`MIN(a)` and `MAX(a)` are defined iff `a` has an interval type. The result is the minimum or maximum, respectively, of the interval.

`RANDOM a` is defined for real or integer intervals `a`. It returns a pseudorandom number in the given interval, with even distribution. This operator can usually not be evaluated as a constant expression, i.e., each time `RANDOM a` is executed at run time, it may return a different value, even if `a` is not changing. The exception to this rule occurs when it is possible to guarantee that the expression will be evaluated only once; this is always the case for the initialization of global variables.

A typename `X` can be applied to a parenthesized expression like a unary operator in order to specify a type conversion into type `X`. All the usual conversions between the types `Real, Int, Int1, Int2` are available; their exact semantics is machine-dependent.

For structure types, it is possible to list all data elements of an object of that type separated by commas and enclosed in parentheses in order to use the typename as a constructor for values of that type. The elements must appear in the order in which they are defined in the type definition.

`MAXINDEX(a)` returns the highest currently available index to the object `a` as an `Int`. `a` must be a connection interface, group, array, or network. For connection interfaces, the number of connections at the interface minus one is returned. For networks the meaning is the number of currently existing replicates minus one.

Expression[77] ≡ 77
```
{
E12:
   '(' Expression ')' /
   Object /
   Denoter /
   FunctionCall /
   ObjectFunctionCall.

Data Object Access[80]
Denoter[78]
Function Call[79]
}
```
This macro is defined in definitions 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, and 77.
This macro is invoked in definition 85.

78 *Denoter*[78] ≡
```
{
Denoter:
   IntegerDenoter /
   RealDenoter /
   StringDenoter.
}
```
This macro is invoked in definition 77.

11.4 Function call

79 *Function Call*[79] ≡
```
{
FunctionCall:
   FunctionId '(' ArgumentList ')'.

FunctionId:
   LowercaseIdent.

ObjectFunctionCall:
   Object '.' ObjectFunctionId '(' ArgumentList ')'.

ObjectFunctionId:
   LowercaseIdent.
}
```
This macro is invoked in definition 77.

Function calls look exactly like procedure calls. The difference is that functions return a value. This value can usually be used just like any other value of the same type. Function calls that involve parallelism, however, are allowed only if they do not change the amount of parallelism: Object function calls to network functions from within network functions or network procedures and object function calls to node functions from within node functions or node procedures and object function calls to connection functions from within connection functions or connection procedures work just like other normal function calls; object function calls to node functions from network functions or network procedures and object function calls to connection functions from node functions or node procedures are not allowed. Object function calls to network functions from the central agent are an exception: They return the result from the first network replicate. *Design rationale:* We could allow function calls into a higher level of parallelism instead of the data element in a reduction statement. I didn't do it in order to keep the semantics and implementation of the reduction statement simple.

12 Referring to data objects

Data objects are referred to either by identifiers, by record element selection, by subscription, by connection addressing, or by special keywords.

12.1 ME, YOU, INDEX, and explicit variables

Data Object Access[80] ≡ 80

```
{
 Object:
    Objectname /
    'ME' /
    'YOU' /
    'INDEX'.

 Objectname:
    LowercaseIdent.
}
```

This macro is defined in definitions 80, 81, 82, and 83.
This macro is invoked in definition 77.

An identifier used as an object refers to an object by name. We call this an *explicit variable* (using the term "variable" also for `CONST` and `IO` objects, meaning an object occupying some data storage — as opposed to, say, a denoter).

The special object `ME` can only be used in object subroutines, object merge procedures, and winner-takes-all and reduction functions. It denotes the object for which the procedure or function was called. `ME` is called an *implicit variable*. In reduction and winner-takes-all functions, `ME` is `CONST`, otherwise it is `VAR`.

The special object `YOU` can only be used in object merge procedures and winner-takes-all and reduction functions. It denotes the second object for which the procedure or function was called and is also an implicit variable. For merge procedures, the result of the merging has to be constructed in `ME` (i.e. `YOU` has to be merged into `ME` — not the other way round). For reduction functions, the value constructed from `ME` and `YOU` is returned as the function result (i.e. here `ME` and `YOU` have equal rights). `YOU` is always `CONST`.

The special object `INDEX` is an implicit `Int CONST` and can only be used in object procedures and object functions. When read in a network procedure, it returns the replicate index of the network replicate it is read from. When read in a node subroutine, its value is the current index number of the object for which the subroutine has been called in the array or group it belongs to. `INDEX` is undefined in connection subroutines.

Note that for the very same node object, the value of `INDEX` may change from call to call if sister objects are created or deleted with the `REPLICATE` statement.

12.2 Selection

Data Object Access[81] ≡ 81

```
{
 Object:
    Object '.' Elementname.

 Elementname:
    LowercaseIdent.
}
```

This macro is defined in definitions 80, 81, 82, and 83.

This macro is invoked in definition 77.

Selections pick an element of a structure type object (node, connection, network, or record) by name. The selected element can be a data element (of a network, node, connection, or record), a node group or node array (of a network), or an interface element (of a node).

12.3 Subscription and parallel variables

82 *Data Object Access*[82] ≡

```
{
Object:
   Object '[' Expression ']' /
   Object '[' ']'.
}
```

This macro is defined in definitions 80, 81, 82, and 83.
This macro is invoked in definition 77.

Subscriptions pick one or several elements of an array or group by position. To pick a single element, the expression must have integral type. If the value of the expression is n, e.g. `ME.nodes[n]`, the subscription refers to the element of the array or group that has index n. The first object of an array or group has index 0. To pick several elements at once, the expression must have `Interval` type, e.g. `ME.nodes[3...5]`. The object that is referred to by such a subscription is called a *slice* of the group or array and consists of all elements whose index is contained in the interval. There is a special slice, called the *all-slice* containing all objects of an array or group that is denoted by just using selection brackets without any expression at all, e.g. `ME.nodes[]`. Slice subscriptions that contain indices of objects that are not existing in the sliced object are automatically clipped to only the existing objects without producing an error.

As we have seen in the description of the object procedure calls, all parallelism in CuPit is created by operating on a multitude of objects. To describe this, the notion of a *parallel variable* is used: a parallel variable is a multiple object that can induce parallelism; parallel variables are either multiple replicates of a network, multiple nodes of a node array or group, or multiple connections attached to a connection interface of a node.

Slice subscription is used to create parallel variables. In order to do this, connection interfaces and explicit network variables are implicitly treated as groups . Given a network variable called `net`, a node group (or array) called `nodes` and a connection interface called `cons`, we find the following cases: In a global subroutine, `net[]` and `net[1...3]` are parallel variables while `net` and `net[3]` are ordinary variables. In a network subroutine `ME.nodes[]` and `ME.nodes[3...5]` are parallel variables while `ME.nodes` and `ME.nodes[3]` are ordinary variables. In a node subroutine, `ME.con[]` is a parallel variable while `ME.con` is an ordinary variable; actual subscription is not allowed for connection interfaces.

Parallel variables can be used for calls to object procedures (but not object functions, since that would return a multitude of values). Further subscription is not allowed on parallel variables. Further selection from a parallel variable creates a *parallel variable selection*. Such an object can be used only in `REDUCTION` and `WTA` statements and in `CONNECT` and `DISCONNECT` statements. E.g. given a `Real` data element called `r` in `cons`, in `REDUCTION ME.cons[].r:sum INTO x` the term `ME.cons[]` denotes a parallel variable of connection type and `ME.cons[].r` is a parallel variable selection of `Real` type.

12.4 Connection addressing

83 *Data Object Access*[83] ≡

```
{
Object:
   '{' Object '-->' Object '}'.
}
```

This macro is defined in definitions 80, 81, 82, and 83.

This macro is invoked in definition 77.

The connection addressing syntax uses the right arrow to address a connection by giving the node output interface from which it originates and the node input interface at which it ends, e.g. `{net.nd[1].out-->net.hid[1].in}`. Such objects can be used to create and initialize connections at the beginning of the program run. They may appear on the left hand side of assignments only, cannot be further selected, and have the side effect to create the connection described by the object pair. Both node interfaces must belong to nodes in the same network. The construct can only be used in the central agent and only while the number of network replicates is 1.

13 The central agent

All global (i.e. non-object) procedures and functions of a CuPit program either belong to the *central agent* or are called *free*. The *central agent* of a CuPit program consists of the global procedure with the name `program` plus a number of other subroutines according to the rules given below.

Design rationale: The significance of the central agent is that certain operations are allowed only there. The idea behind the central agent is that it is the (sequential) control program from which the (possibly parallel) network operations are called. All parallelism occurs outside the central agent hidden in object procedures.

A function or procedure is free if and only if

1. it does not mention a `NETWORK` variable explicitly and
2. it does not call any global procedure or function that is not free

All subroutines that are not free are part of the central agent.
All subroutines that are part of the central agent are not free.

The global procedure `program` must exist and is always part of the central agent (unless the program does not use a `NETWORK` variable at all); the `program` procedure is implicitly called when a CuPit program is invoked. Object subroutines are never part of the central agent. Note that since the CuPit compiler cannot check external procedures they are always assumed to be free. It is not allowed to call subroutines that belong to the central agent from an object subroutine. Object-subroutines may, however, call free global subroutines.

14 Overall program structure

A CuPit program is simply a sequence of type definitions, data object definitions, and procedure or function definitions. Any object must be defined before it can be used.

Cupit Program[84] ≡ 84

```
{
 Root:
    CupitProgram.

 CupitProgram:
    CupitParts.

 CupitParts:
    /* nothing */ /
    CupitParts CupitPart ';'.

 CupitPart:
    TypeDef /
    DataObjectDef /
```

```
    ProcedureDef /
    FunctionDef /
    ReductionFunctionDef /
    WtaFunctionDef.
  }
```

This macro is invoked in definition 85.

All these definitions are now put into the Eli [GHL+92] grammar specification file `grammar.con`:

85 **grammar.con**[85] ≡

```
{
 Cupit Program[84]
 Type Definition[31]
 Subroutine Definition[43]
 Data Object Definition[42]
 Statement[50]
 Expression[64]
 }
```

This macro is attached to an output file.

15 Basic syntactic elements

All keywords and operators in a CuPit program must appear exactly as shown in the grammar. The syntactic structure of identifiers, denoters (value literals), and comments will be described in this section.

These are the contents of the scanner definition for CuPit:

86 **scanner.gla**[86] ≡

```
{
 Lowercase Identifier[87]
 Uppercase Identifier[88]
 Integer Denoter[89]
 Real Denoter[90]
 String Denoter[91]
 Wrong Keywords[92]
 Comment[94]
 }
```

This macro is attached to an output file.

15.1 Identifier

Identifiers appear in two forms: Starting with an uppercase letter (for type names) or starting with a lowercase letter (for everything else).

87 *Lowercase Identifier*[87] ≡

```
{
 LowercaseIdent:  $[a-z][a-zA-Z0-9]*   [mkidn]
 }
```

This macro is invoked in definition 86.

A `LowercaseIdent` is a sequence of letters and digits that starts with a lowercase letter.

88 *Uppercase Identifier*[88] ≡

```
{
 UppercaseIdent:  $([A-Z][a-zA-Z0-9]*[a-z0-9][a-zA-Z0-9]*)|[A-Z]   [mkidn]
```

```
}
```

This macro is invoked in definition 86.

An `UppercaseIdent` is either a single uppercase letter or a sequence of letters and digits that starts with an uppercase letter and contains at least one lowercase letter or digit. This has the consequence that for instance `T`, `T1` and `TreeIT2` are `UppercaseIdent`s while `TREE` is not.

15.2 Denoter

There are denoters for integer, real, and string values.

Integer Denoter[89] ≡ 89

```
{
 IntegerDenoter:  $([0-9]+|0[xX][0-9a-fA-F]*)   [c_mkint]
}
```

This macro is invoked in definition 86.

Integer denoters are defined exactly as in the C programming language, except that the `L` and `U` suffixes are not supported in CuPit.

Real Denoter[90] ≡ 90

```
{
 RealDenoter:  $([0-9]+\.[0-9]+)([eE][\+\-]?[0-9]+)?   [mkstr]
}
```

This macro is invoked in definition 86.

Real denoters are similar to floating point denoters in the C programming language. However, there must always be a decimal point that is surrounded by digits in a CuPit floating point denoter.

String Denoter[91] ≡ 91

```
{
 StringDenoter:  $\"  (auxCString)   [c_mkstr]
}
```

This macro is invoked in definition 86.

String denoters are defined exactly as string literals in the C programming language.

15.3 Keywords and Comments

Wrong Keywords[92] ≡ 92

```
{
 $[A-Z][A-Z]+   [ComplainKeyword]
}
```

This macro is invoked in definition 86.

Eli extracts the keywords from the parser grammar and automatically constructs the scanner in a way to recognize them. However, if you misspell a keyword (or use a nonexisting one) you get one syntax error per character in your wrong keyword after the point where the wrong keyword looks different from any existing one. This is awful. Therefore, we introduce a scanner rule that catches any token that looks like a keyword (but is not a true keyword — those always take precedence) and produces an error message that says "I have never heard of a keyword like that and do not like it, too". Here is the procedure that produces this message:

scanerr.c[93] ≡ 93

```
{
 #include "err.h"
```

```
void ComplainKeyword (char *start, int lgth, int *extCodePtr, char *intrPtr)
{
  message (ERROR, "Huh ?  What's that ??  An unknown keyword!", 0, &curpos);
}
}
```

This macro is attached to an output file.

94 *Comment*[94] ≡

```
{
$\(\*  (auxM3Comment)
}
```

This macro is invoked in definition 86.

Comments are defined exactly as in Modula-2 or in Modula-3, i.e. comments begin with `(*`, end with `*)`, and can be nested. `auxM3comment` is a so-called "canned description" in Eli; so to say a miniature re-usable module.

16 Predefined entities

There are a number of types, objects, and operations that are, although not inherently part of the language, predefined and built into the CuPit compiler. These predefined entities are described in this section.

Among the predefined types are the integer types `Int, Int1, Int2`. It is guaranteed that `Int1` can represent at least the range -127...127, that `Int2` can represent at least the range -32767...32767, and that `Int2` can represent at least the range -2147483647...2147483647. Furthermore, it is guaranteed that the range that `Int1` can represent is not smaller than that of `Int2`, and that the range that `Int2` can represent is not smaller than that of `Int`. Integers are represented in a two's complement format.

The predefined type `Real` is a floating point type of at least 32 bit precision. The details of floating point format and arithmetic are machine-dependent.

The predefined types `Interval, Interval1, Interval2`, and `Realerval` are represented as a pair of `Int, Int1, Int2`, or `Real` objects, respectively, as described in section 7.3.

The predefined type `Bool` defines the truth-value type. The predefined constants `true` and `false` are its only values.

The predefined types `String` and `Opaque` represent character strings and general pointers, respectively, in a machine-dependent way.

17 Compiler-dependent properties

Certain aspects of the behavior of CuPit are not completely defined in the language definition itself, but left to the individual compiler. These aspects can be divided into the following groups:

1. Machine arithmetic.
2. Reaction on run-time errors.
3. Mapping of data to memory locations.
4. Handling of parallelism.

The properties of the target machine's arithmetic and the handling of the individual run-time errors must be individually documented for every CuPit compiler. A compiler may provide compilation options to change the behavior of the generated program in some of these respects (e.g. select single or double precision floating point format, turn array subscript checking on or off, etc.)

The mapping of data into memory needs not be documented, with one exception: The mapping of I/O variables must be described because external procedure are needed that access such objects in order to get data into and out of CuPit programs.

How parallelism is handled is completely left to the compiler and needs not be documented.

Appendix

A Terminology and Abbreviations

The term *group* is used to refer to `GROUP`s of nodes, `ARRAY`s of nodes, or slices thereof.

The terms *component*, *field*, and *element* all mean the same. Usually only *element* is used in the CuPit description.

These are the most important abbreviations that occur in the report (especially in the syntactical description of the language). Some of these abbreviations have more than one meaning, but which one is meant can always easily be deduced from context.

Arg	Argument
Cmpd	Compound
Con	Connection, Connect
Decl	Declaration
Def	Definition
El, Elem	Element
Enum	Enumeration
Expr	Expression
Func	Function
Id, Ident	Identifier, Identification
iff	if and only if
Init	Initialization, Initializer
Obj	Object
Op	Operator, Operation
Opt	Optional
Param	Parameter
Proc	Procedure
Stmt	Statement
Val	Value

B Possible future language extensions

The following may be included in future versions of CuPit:

1. Handling single precision and double precision floating point numbers.
2. Fixed-point numbers.
3. Explicit subrange types with range checking or range clipping (saturation).
4. Anonymous types, i.e., using a type declarator instead of a type name.
5. Use of objects before their definition.
6. Overloaded functions and procedures, overloaded operators.
7. Modules (in the sense of MODULA-2)
8. Inner-procedural refinements (in the sense of C-Refine).[4]
9. Derived types with inheritance and run-time dispatch of procedure calls.
10. More support for declarative specification of connections.
11. More support for dynamic node groups (e.g. dynamic groups of dynamic groups)
12. Support for genetic algorithms (populations of networks).

[4] C-Refine is available by anonymous ftp for example from `ftp.ira.uka.de` in `/pub/gnu/crefine.3.0.tar.Z` or from any `comp.sources.reviewed` archive

CuPit	Modula-2	C
?:		?:
OR	OR	||
XOR	# <>	!=
AND	AND	&&
= <> < >	= # <> < >	= != < >
<= >=	<= >=	<= >=
IN	IN	
BITOR		|
BITXOR		^
...	..	
BITAND		&
LSHIFT		<<
RSHIFT		>>
+ -	+ -	+ -
* / %	* / DIV MOD	* / %
**		
NOT	NOT	!
BITNOT		~
-	-	-
Type	Type	(Type)
.	.	.
()	()	()

Table 2: Operator correspondences: CuPit vs. Modula-2 vs. C

There are several reasons why these features have not been included in the current definition of the CuPit language: Some are simply not very important to have, most would make the language semantics and the compiler more complicated and some are not easy to integrate into the CuPit semantics at all. Since the purpose of CuPit is to allow for not-too-complicated yet efficient compilation onto parallel machines, these reasons were considered sufficient to leave the above features out of the language. Practical application of CuPit may show, however, that the usefulness of some of these features outweighs the effort to integrate them.

C Operator correspondences to Modula-2 and C

Many of the operators of CuPit have identical or similar counterparts in Modula-2 or in C. Table 2 lists these correspondences. Note that the application requirements or the semantics for certain special cases may differ between the corresponding operators in the three languages.

D Implementation status

Currently (January 1994), one implementation of CuPit exists. It is a compiler that generates MPL code for the MasPar MP-1/MP-2 massively parallel SIMD machine (16384 processors). MPL is MasPar's data parallel C variant. The compiler is fully functional and is implemented using the Eli compiler construction system [GHL+92]. The compiler source code is written as a FunnelWeb literate programming document. This means the source code is available as a well-structured 300 page document with table of contents, global keyword index, and interspersed documentation text.

The compiler is used to explore optimization strategies that achieve both data locality and load balancing for irregular networks. A second implementation for the KSR-1 will probably be done later.

```
  Real        outData;    (* forward:  sigmoid(in),
                             backward: sum(deltas*weights) or teachinput *)
  IN Weight   in;
  OUT Weight  out;
  Real        cumErr := 0.0;  (* cumulated error (output units only) *)
  Int         errN  := 0;     (* number of error bits (output units only) *)
  Int         consN;          (* number of input connections *)

  PROCEDURE forward (Bool CONST doIn, doOut) IS
    IF doIn
    THEN REDUCTION ME.in[].out:sum INTO ME.inData;    END;
    IF doOut THEN
      ME.outData := sigmoid (ME.inData);
      ME.out[].transport(ME.outData);
    END;
  END PROCEDURE;


  PROCEDURE backward (Bool CONST doIn, doOut) IS
    (* (for output nodes, out is the teaching input)
       for all nodes, set  in := delta of the node
       for all nodes that are not input nodes, back-transport the delta
    *)
    Real VAR sout, diff;
    IF doIn  (* i.e. is not an output node *)
    THEN REDUCTION ME.out[].in:sum INTO ME.outData;
         ME.inData := ME.outData * sigmoidPrime(ME.inData);
    ELSE (* output node: *)
         (* error function:  E (target, actual) = 1/2*(target-actual)**2  *)
         sout := sigmoid (ME.inData);
         diff := ME.outData - sout;
         ME.inData := diff * sigmoidPrime(ME.inData);
         ME.outData := sout;
         ME.cumErr += 0.5 * diff * diff;
         IF absReal (diff) > errBitThreshold THEN ME.errN += 1;  END;
    END;
    IF doOut (* i.e. is not an input node *)
    THEN ME.in[].btransport (ME.inData);  END;
  END PROCEDURE;


  PROCEDURE adapt () IS
    ME.in[].adapt ();
  END PROCEDURE;


  MERGE IS
    ME.cumErr += YOU.cumErr;
    ME.errN   += YOU.errN;
  END MERGE;


  PROCEDURE resetErrors () IS
    ME.cumErr := 0.0;
    ME.errN   := 0;
  END PROCEDURE;


  PROCEDURE modify (Real CONST p) IS
    (* during this procedure, we re-interpret
       errN as number of just deleted connections
    *)
    Real VAR oldConN, newConN;
    ME.in[].transport (1.0);
    REDUCTION ME.in[].in:sum INTO oldConN;  (* count connections *)
    ME.in[].eliminate (p);
    REDUCTION ME.in[].in:sum INTO newConN;
    ME.consN := Int(newConN);
    ME.errN := Int(oldConN) - ME.consN;
    IF ME.consN = 0    (* no more input connections present -> *)
    THEN REPLICATE ME INTO 0;  END; (* self-delete *)
  END PROCEDURE;

END TYPE;


TYPE Layer  IS GROUP OF SigmoidNode END;


TYPE Mlp IS NETWORK
  Layer   in, hid, out;
  Real    totError;
  Int     errorsN;
  Int     consN;

  PROCEDURE createNet (Int CONST inputs, hidden, outputs) IS
    EXTEND ME.in BY inputs;
    EXTEND ME.hid BY hidden;
    EXTEND ME.out BY outputs;
    CONNECT ME.in[].out  TO ME.hid[].in;
    CONNECT ME.hid[].out TO ME.out[].in;
    (* ...and bias for the output nodes: *)
    CONNECT ME.in[0...0].out TO ME.out[].in;
  END;
```

```
  PROCEDURE  example () IS
    ME.in[].forward   (false, true);
    ME.hid[].forward  (true,  true);
    ME.out[].forward  (true,  false);
    ME.out[].backward (false, true);
    ME.hid[].backward (true,  true);
    (* ME.in[].backward  (true,  false); *) (* not needed *)
  END PROCEDURE;


  PROCEDURE adapt () IS
    ME.out[].adapt ();
    ME.hid[].adapt ();
  END PROCEDURE;


  PROCEDURE computeTotalError () IS
    REDUCTION ME.out[].cumErr:sum INTO ME.totError;
    REDUCTION ME.out[].errN:sumInt INTO ME.errorsN;
    ME.out[].resetErrors();
  END PROCEDURE;


  PROCEDURE countConnections () IS
    Int VAR outConsN, outConsDeleted;
    REDUCTION ME.out[].consN:sumInt INTO outConsN;
    REDUCTION ME.out[].errN:sumInt  INTO outConsDeleted;
    REDUCTION ME.hid[].consN:sumInt INTO ME.consN;
    REDUCTION ME.hid[].errN:sumInt  INTO ME.errorsN;
    ME.consN += outConsN;
    ME.errorsN += outConsDeleted;
  END PROCEDURE;


  PROCEDURE modify (Real CONST p) IS
    ME.out[].modify (p);
    ME.hid[].modify (p);
    ME.countConnections ();
    ME.out[].resetErrors();
  END PROCEDURE;

END TYPE;


Real IO    x1, x2;  (* I/O-areas, allocated and managed by external program *)
Mlp  VAR   net;     (* THE NETWORK *)


PROCEDURE openDatafile (String CONST filename;
                        Int VAR iInputsN, rInputsN, iOutputsN, rOutputsN,
                        examplesN) IS EXTERNAL;
PROCEDURE initIOareasxRxR (Real IO rIn; Real IO rOut;
                           Int CONST maxreplicates) IS EXTERNAL;
PROCEDURE getExamplesxRxR (Real IO rIn; Real IO rOut;
                           Int CONST howmany; Int VAR firstI) IS EXTERNAL;


PROCEDURE protocolError (Int CONST epochNumber; Real CONST totalError;
                         Int CONST nrOfErrors) IS EXTERNAL;
PROCEDURE pInt (Int CONST me) IS EXTERNAL;
PROCEDURE pReal (Real CONST me) IS EXTERNAL;
PROCEDURE pString (String CONST me) IS EXTERNAL;


PROCEDURE program () IS
  Int  VAR i := 0, nrOfExamples, examplesDone := 0, epochNr := 0;
  Int  VAR dummy1, dummy2;
  Real VAR error, oldError, stoperror := 0.1;
  openDatafile ("Data", dummy1, inputs, dummy2, outputs, nrOfExamples);
  stoperror := Real(outputs**2) * 0.5*(stoperror**2);
  net[].createNet (inputs, hidden, outputs);
  REPLICATE net INTO nrOfExamples;  (* better: INTO 1...nrOfExamples *)
  initIOareasxRxR (x1, x2, MAXINDEX (net) + 1);
  REPEAT
    epochNr += 1;
    REPEAT
      getExamplesxRxR (x1, x2, MAXINDEX (net) + 1, i);
      net.in[].inData   <-- x1;
      net.out[].outData <-- x2;
      net[].example();
      examplesDone += MAXINDEX (net) + 1;
    UNTIL examplesDone >= nrOfExamples END REPEAT;
    examplesDone %= nrOfExamples;

    MERGE net;  (* collect and redistribute results *)
    net[].adapt();
    net[].computeTotalError ();
    error := net[0].totError;
    protocolError (epochNr, error, net[0].errorsN);
    IF epochNr % 10 = 0 THEN
      IF epochNr <= 20  THEN oldError := error/2.0;
      ELSIF error < oldError
      THEN REPLICATE net INTO 1;
           net[].modify (0.25);
           pString (">>>>>>>>>>> weights remaining: ");
           pInt (net[0].consN);
```

```
            pString ("  deleted: "); pInt (net[0].errorsN);
            pString (" <<<<<\n");
            oldError := error;
            REPLICATE net INTO nrOfExamples;
        END IF;
      END IF;
    UNTIL epochNr > 4 AND error <= stoperror  END REPEAT;
END PROCEDURE;
```

References

[AR88] J.A. Anderson and E. Rosenfeld, editors. *Neurocomputing: Foundations of Research*. MIT Press, Cambridge, MA, 1988.

[GHL+92] Robert W. Gray, Vincent P. Heuring, Steven P. Levi, Anthony M. Sloane, and William M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–131, February 1992.

[Hoa83] C. A. R. Hoare. Hints on programming language design. In Ellis Horowitz, editor, *Programming Languages: A Grand Tour*, pages 31–40. Springer Verlag, 1983.

[KR77] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1977.

[MP43] Warren McCulloch and Walter Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943. Reprinted in [AR88].

[RB93] Martin Riedmiller and Heinrich Braun. A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In *Proceedings of the IEEE International Conference on Neural Networks*, San Francisco, CA, April 1993. IEEE.

[RM86] David Rumelhart and John McClelland, editors. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume Volume 1. MIT Press, Cambridge, MA, 1986.

[Uni81] United States Department of Defense, Springer Lecture Notes in Computer Science 106. *The Programming Language Ada*, 1981.

[Wer74] Paul Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, 1974.

[Wil92] Ross N. Williams. *FunnelWeb User's Manual*, version 1.0 for funnelweb 3.0 edition, May 1992.

[Wir83] Niklaus Wirth. On the design of programming languages. In Ellis Horowitz, editor, *Programming Languages: A Grand Tour*, pages 23–30. Springer Verlag, 1983.

[Wir85] Niklaus Wirth. *Programming in Modula-2*. Springer, Berlin, Heidelberg, New York, Tokyo, 1985.

Index