A Neural compiler

Frédéric GRUAU*, Jean-Yves RATAJSZCZAK†, Gilles WIBER†

Centre d'Etude Nucléaire de Grenoble
Département de Recherche Fondamentale
Matiére Condensée,
17 rue des Martyrs, 38041 Grenoble

Ecole Normale Supérieure de Lyon
Laboratoire de l'Informatique
du Parallélisme, 46 Allée d'Italie
69364 Lyon Cedex 07, France

Abstract

This paper describes a neural compiler. The input of the compiler is a PASCAL Program. The compiler produces a neural network that computes what is specified by the PASCAL program. The compiler generates an intermediate code called cellular code.

1 Introduction

We all know that neural nets are at the origin of all the computer science, that is, the very existence of digital computers. The automata theory founded by Kleene in 1956 [8] (the date of his fundamental publication, with the revealing title "Representation of events in nerve nets") is directly issued from the neuron model proposed by Mc Culloch and Pitts [1] in 1943. In 1945, Von Neuman was also using a system of formal neurons that was a direct offspring of Mc Culloch and Pitts's model, to describe the logic of the very first computers. Despite these facts, Rosenblatt's perceptron [11] developed in the years 1950, has not been transformed into a working tool. This is due to theoretical difficulties shown in the work from Minsky and Papert [10], but also because at this time, it was not possible to implement simulations on machine. The power of machines was ridiculous compared to now. Moreover, a link was missing in the theory: the fact that a neural network can simulate a Turing machine. In other word, it is the proof that any computable function can be computed by a neural network. Odd enough, this equivalence proof was brought only recently by Siegelmann and Sontag, in 1991 [13]. In [14], Siegelmann and Sontag did a real time simulation for real time equivalence. Going from Turing machines to modern programming

*gruau@lip.ens-lyon.fr

†or Ecole Nationale Supérieure d'Informatique et de Mathématiques Appliquées de Grenoble, Campus de Saint Martin d'Héres, 38000 Grenoble, France

languages with variables, procedures and data structures, took about 15 years. Here, we consider that the first realisation of the Turing machine dates to 1945, and that the first languages like FORTRAN, PASCAL, or LISP (PASCAL can be considered as a form of ALGOL) appeared in 1960. It took ten more years to learn how to compile these languages efficiently. Siegelmann [12] has developed a theoretical language for neural networks called AEL, trying to exploit the parallelism and the analog aspect of the neurons. But her language was not yet implemented with a compiler.

In [3]and [5], Gruau describes the principles of neural compilation. In this paper, we describe a neural compiler that has been actually programmed. The input of the compiler is a PASCAL Program. The compiler produces a neural network that computes what is specified by the PASCAL program. This work is complementary to the work of Siegelmann on AEL. We proposed a method to actually build the neural network from a program, but did not concentrate on which language should be best adapted to neural network compilation (although we did add a few instructions to standard PASCAL that allow to exploit parallelism and hierarchical design). On the other hand Siegelmann studied the language, but not the compilation.

Since the compiled neural network is a dataflow graph, neural compilation shares some similarities with compilation for dataflow machines like the one presented by Veen, 1991 [16]. However, our method of compilation is different. It is entirely based on graph rewriting. As we will see, graph grammars is a simple and efficient tool that not only produces the dataflow graph but also provides each cell with (x, y) coordinates. These coordinates can be used to map the neural net on a multiprocessor system, or to draw the neural net automatically.

The steps of the neural compilation are described in details. The basis of the compiler is to use a list of cellular operators, which are listed in the appendix, and are defined using their microcode. These operators called *program symbols*, add or suppress cells, establish or cut a connection. In other words, they implement truly physical operations on the neural net that is being built. They allow to encode a neural net by specifying how to build it step by step. The encoding is called *cellular encoding*. *Macro program symbols* represented in a geometrical way, translate the PASCAL declarations of the program (that build data structure like array) and the PASCAL instructions, into operations on neural network that can be implemented using a sequence of program symbols.

By applying these program symbols, we can see the neural net that is being built step by step, on the computer screen.

Thanks to the neural compiler, the simulation of Siegelmann and Sontag is transformed into the effective building of a neural net that behaves like a given PASCAL program. Modern computers have completely changed our life. The impact of a neural compiler could be as important, if we carry on the comparison. It could make it possible to manage neural computers of billions of neurons, as conveniently as we are now using computer with billions bytes of memory. Personal computer would be replaced by personal neuro computer that can offer extra learning capabilities.

The compiled neural network structure is fixed and do not change in time according to the particular inputs. The neural network has therefore a limited memory. For this reason, one cannot compile instructions for dynamic memory allocation. If there are recursive procedures one must specify before compilation an upper bound on the number of recursive encapsulations. However in the case of divide and conquer algorithm the compiler itself can compute the exact number of recursive encapsulations that is needed, and the user need not specify bounds.

The compiler produces a cellular code from the PASCAL program, and develops a neural network from the cellular code, which is then fixed. The run of the PASCAL program is simulated by relaxing the compiled network. A future direction it to develop the neural network at run time. The subnetwork simulating a procedure would be developed only when that procedure is invoked. That can be done with relatively few changes with the technique presented in this paper. The generation of the cellular encoding can also be done during run time. Both changes will transform the neural compiler into a neural interpreter. The neural interpreter does not need to be provided with a bound on the number of recursive encapsulations, and it can translate instructions for dynamic memory allocation. The neural interpreter would spare the compilation time but would be slower to run than the compiled neural network.

The paper begins with a presentation of the kind of neural networks that are compiled. The compiler generates an intermediate code which are sequence of program symbols called cellular code. A background on cellular encoding is given. We then describe in detail the neural compiler. We explain the stage of the compilation, and the structure of a compiled neural network. We describe

how to compile separately a PASCAL instruction, and then, how to compile the whole PASCAL program. Two small examples of compilation are reported. They illustrate the explanations. We then give a precise geometrical description of how to translate each construct of the PASCAL language into a macro program symbol. We start by a kernel of the PASCAL language, and extend progressively to control structure, function and procedure, array data structure, and constructs of an extended PASCAL that allow to exploit the power of neural networks (parallelism, learning). We show the interest of the neural compiler with height examples of compilation. In the conclusion we propose future applications of the neural compiler. We will see that neural compilation can be used for the design of huge neural networks, automatic parallel compilation, and the compilation of hybrid systems.

2 The neural networks that are used

A neural network is an oriented graph of interconnected cells. Each cell computes an activity using the activities of the cells connected to its input, and sends its activity to the cells connected to its output. A neuron is determined by the way it computes its activity. An ordinary neuron makes a sum of its inputs, weighted by the weights of the connections, subtracts a threshold, and finally, applies a function called *sigmoïd*. The neural network makes a computation by iterating each of its neurons. The order of iteration determines the dynamic of the neural net. The *dynamic* can be parallel, each neuron updates its activity at the same time, or sequential: neurons update their activity one after the other. We are now going to describe the particular sigmoïd and the particular dynamic we will use for neural compilation.

2.1 Sigmoïd

We use four different kinds of sigmoïd listed in figure 1.

[Figure 1 about here]

Moreover, we use non classical neurons that make the product of their inputs, and that divide

the first input by the second input. This is used to multiply and to divide two numbers. Siegelmann and Sontag allowed for only one kind of sigmoïd, piecewise linear between 0 and 1. We preferred to use a small set of sigmoïds for efficiency. For example, it is possible to simulate arithmetic operations on binary coded number with Siegelmann and Sontag's sigmoïd, but it costs a lot of neurons.

2.2 The Dynamic

The neural network is a data-flow graph. A neuron computing an arithmetic operation with two operands must wait for the two operands to come before starting to compute its activity. We use a particular dynamic (denoted $\mathcal{D}_0$). The neurons decide to start their computation using three rules:

- A neuron computes its activity as soon as it has received all the activities from its input neighbors.
- The input units are initialized with the input vector.
- A neuron sends its activity to the output neighbors, when it is initialized, of if it has just computed its activity.

There is no special threshold units to encode the threshold. At a given moment, all the neurons that can compute their activity, do it. The dynamic $\mathcal{D}_0$ is half way between the sequential dynamic and the parallel dynamic. There is no need for a global entity to control the flow of activities. Two other dynamics are used by the compiler.

- The dynamic $\mathcal{D}_1$ concerns neurons having exactly two inputs. As was the case for $\mathcal{D}_0$, the neuron waits until it has received an activity from both its neighbors. If the first input is strictly positive, the neuron send the activity of the second neighbor. Otherwise, the neuron do not propagate any activity. The dynamic $\mathcal{D}_1$ allows to block a flow of activities.
- Neuron with dynamic $\mathcal{D}_2$ compute their activity whenever an activity is received from one of its input neighbors. Its activity is the activity of that particular neighbor. In order to work correctly, such a neuron must not received two activities at the same time. Dynamic $\mathcal{D}_2$ can merge on the same neuron, activities coming from different parts.

In [3] Gruau (1992) shows that it is possible to simulate the dynamic $\mathcal{D}_0$ $\mathcal{D}_1$ $\mathcal{D}_2$ with the traditional parallel dynamic. The number of units is multiplied by 6, and the time needed to relax by 4. Dynamic $\mathcal{D}_1$ and $\mathcal{D}_1$ correspond to the "branch" and the "merge" node used in data flow models.

3 Overview of Cellular Encoding

Cellular encoding is a method for encoding families of similarly structured neural networks. Cellular encoding was previously proposed by Gruau as a way to encode a neural net on chromosomes that can be manipulated by the genetic algorithm [4] [7] [6].

3.1 The basic Cellular Encoding

In this subsection we present the basic cellular encoding. The cellular code is represented as a *grammar tree* with ordered branches whose nodes are labeled with name of *program symbols*. The reader must not make the confusion between grammar tree and tree grammar. Grammar tree means a grammar encoded as a tree, whereas tree grammar means a grammar that rewrites trees[1]

A cell is a node of an oriented *network graph* with ordered connections. Each cell carries a duplicate copy of the cellular code (i.e., the grammar tree) and has an internal reading head that reads from the grammar tree. Typically, each cell reads from the grammar tree at a different position. The labels of the grammar tree represent instructions for cell development that act on the cell or on connections of the cell. During a step of the development process, a cell executes the instruction referenced by the symbol it reads, and moves its reading head down in the tree. One can draw an analogy between a cell and a Turing machine. The cell reads from a tree instead of a tape and the cell is capable of duplicating itself; but both execute instructions by moving the reading head in a manner dictated by the symbol that is read. We will refer to the grammar tree as a *program* and each label as a *program-symbol*.

A cell also manages a set of internal registers, some of which are used during development, while others determine the weights and thresholds of the final neural net. The link register is used to refer to one of possibly several fan-in connections (i.e., links) into a cell.

[1] In [6], we have demonstrated that cellular encoding represents a parallel graph grammar, this is why we use the term grammar tree.

Consider the problem of finding the neural net for the exclusive OR (XOR) function. Neurons can have thresholds of 0 or 1. Connections can be weighted -1 or $+1$. In this section, we use a variant of sigmoïd (b) of figure 1. If the weighted sum of its input is strictly greater than its threshold, the neuron outputs 1, else it outputs 0. The inputs of the neural net are 0 or 1.

The development of a neural net starts with a single cell called the *ancestor cell* connected to an *input pointer cell* and an *output pointer cell*. Consider the starting network on the right half of figure 2 and the cellular encoding depicted on the left half of figure 2. At the starting step 0 the reading head of the ancestor cell is positioned on the root of the tree as shown by the arrow connecting the two. Its registers are initialized with default values. For example, its threshold is set to 0. As this cell repeatedly divides, it gives birth to all the other cells that will eventually become a neuron and make up the neural network. A cell is said to become a neuron when it looses its reading-head. The input and output pointer cells to which the ancestor is linked (indicated by boxes in the figure) do not execute any program-symbol. Rather, at the end of the development process, the upper pointer cell is connected to the set of input units, while the lower pointer cell is connected to the set of output units. These input and output units are created during the development, they are not added independently at the end. After development is complete, the pointer cells can be deleted. For example, in figure 6, the final decoded neural net has two input units labeled "a" and "c", and one output unit labeled "d". We will now describe the kernel of the program-symbol.

- A division-program symbol creates two cells from one. In a *sequential division* (denoted by `SEQ`) the first child cell inherits the input links, the second child cell inherits the output links of the parent cell. The first child connects to the second with weight 1. The link is oriented from the first child to the second child. This is illustrated in steps 1 and 3. Since there are two child cells, a division program-symbol must label nodes of arity two. The first child moves its reading head to the left subtree and the second child moves its reading head to the right subtree. Finally, when a cell divides, the values of the internal registers of the parent cell are copied in the child cells.

- The em Parallel division (denoted by `PAR`) is a second kind of division program symbol. Both child cells inherit the input and output links from the parent cell (in step 2 and step 6). The sequential and parallel division are canonical because they treat all the input and output links uniformly, regardless of their number. Subsection 3.3 introduces more complex divisions.

- The *ending-program symbol* denoted `END` causes a cell to loose its reading head and become a finished neuron. `END` labels the leaves of the grammar tree (i.e., nodes of arity 0).

- A *value-program symbol* modifies the value of an internal register of the cell. The program-symbol `INCBIAS` increments (and `DECBIAS` decrements) the threshold of a cell. `INCLR` increments (and `DECLR` decrements) the value of the link register, which points to a specific fan-in link or connection. Changing the value of the link register causes it to point to a different fan-in connection. The link register has a default initial value of 1, thus pointing to the leftmost fan-in link. Operations on other connections can be accomplished by first resetting the value of the link register. The program-symbol denoted `VAL+` sets the weight of the input link pointed by the link register to 1, while `VAL-` sets the weight to -1 (see step 7). The program-symbols `VAL+` or `VAL-` do not explicitly indicate to which fan-in connection the corresponding instructions are applied. When `VAL+` or `VAL-` is executed it is applied to the link pointed to by the link register.

- An unary program-symbol `CUT` cuts the link pointed by the link register. This operator modifies the topology by removing a link.

Operators `INCLR`, `DECLR`, `CUT` are not illustrated, they are not required for the development of a neural net for the XOR problem. The sequence in which cells execute program-symbols is determined as follows: once a cell has executed its program-symbol, it enters a First In First Out (FIFO) queue. The next cell to execute is the head of the FIFO queue. If the cell divides, the child which reads the left subtree enters the FIFO queue first. This order of execution tries to model what would happen if cells were active in parallel. It ensures that a cell cannot be active twice while another cell has not been active at all. In some cases, the final configuration of the network depends on the order in which cells execute their corresponding instructions. For example, in the

development of the XOR, performing step 7 before step 6 would produce a neural net with an output unit having two negative weights instead of one, as desired. The waiting program-symbol denoted `WAIT` makes the cell wait for its next rewriting step. `WAIT` is necessary for those cases where the development process must be controlled by generating appropriate delays.

[Figure 2 about here]

[Figure 3 about here]

[Figure 4 about here]

[Figure 5 about here]

[Figure 6 about here]

3.2 How to encode recursive grammars

Up to this point in our description the grammar tree does not use recursion (Note that recurrence in the grammar does not imply that there is recurrence in the resulting neural network.). Non recursive grammar can develop only a single neural network. But one would like to develop a family of neural networks, which share the same structure, for computing a family of similarly

structured problem. For this purpose, we introduce a *recurrent program-symbol* denoted `REC` which allows a fixed number of loops L. Each cell contains a register called life. The cell which reads `REC` executes the following algorithm:

[Figure 7 about here]

[Figure 8 about here]

```
life := life - 1
If (life > 0) reading-head := root of the grammar tree
Else  reading-head := subtree of the current node
```

where *life* is a register of the cell initialized with L in the ancestor cell. Thus a grammar develops a family of neural networks parametrized by L. The use of a recurrent-program symbol is illustrated figure 7 and 8. The cellular code in this figure is almost the same as the cellular code of a XOR network. The only difference is that a program symbol `END` has been replaced by a program symbol `REC`. The resulting cellular code is now able to develop a neural net for the parity function with an arbitrary large number of inputs, by assembling copies of a XOR subnetwork. In figure 8 the network for parity of 3,5 and 7 inputs is shown. This implementation of the recurrence allows a precise control of the growth process. The development is not stopped when the network size reaches a predetermined limit, but when the code has been read exactly L times through. The number L parametrizes the size of the neural network.

3.3 Microcoding

The program symbols introduced in the preceding section are not sufficient to do neural compilation. We had to introduce many other division program-symbols, as well as program-symbols that modify

cell registers, or that make local topological transformations, or that influence the order of execution. Each new program-symbol may have an integer argument that specifies a particular link or a particular new register value, this favors a compact representation. The program symbols are microcoded. The program-symbol's microcode are listed in appendix 3. A microcode is composed of two parts. The first part specifies the category of the program symbol. It uses three capital letters. DIV indicates a division, TOP a modification of the topology, CEL, SIT, LNK a modification of respectively a cell register, a site register or a link register. EXE indicates a program-symbol used to manage the order of execution, BRA tests a condition on the registers of the cells or on the topology, if it is true, the reading head is placed on the left sub-tree, else it is placed on the right sub-tree. HOM refers to a division in more than 2 child cells, AFF enhances the display. The second part of the microcode is an operand. For CEL, SIT, LNK the operand is the name of the register which value will be modified using the argument. If there is an arithmetic sign, the operator applies the arithmetic operation to the actual content of the register and the argument and places the result in the register. Else it simply sets the register with the argument. For BRA the operand is the name of the condition. For DIV, the operand is a list of *segments*. Each segment is composed of an alphabetic letter and some arithmetic signs. The arithmetic signs specify a sublist of links, and the letter specifies whether to move or to duplicate this sublist. When the cell divides, the first child cell inherits all the input links, the second child cell inherits all the output links. The segments are then decoded, they specify how to duplicate or move links between the two child cells. Figure 9 gives an example of a DIV microcode analysis. If the segment's letter is a capital, the links are duplicated or moved from the output site of the second child cell to the output site of the first child cell. If the letter is a small letter, the links are duplicated or moved from the input site of the first child cell to the input site of the second child cell.

- The letter 'm' or 'M' moves the sublist,
- the letter 'd' or 'D' duplicates the sublist.
- the letter 'r' or 'R' is the same as 'd' or 'D' except that the added links are not ordered in the same way, by the neighboring cell. Figure 10 explains the difference between "d" and "r".

A particular segment with the letter 's' means: connect the output site of the first child cell to the input site of the second child. This segment needs no arithmetic sign. Arithmetic signs specify the sublist of links to be moved or duplicated. '<', '>', '=', specify the links lower than, equal, or greater than the argument, '*' specifies all the links. '#' and '$' refer to the first and the last link, ':' refers to links from the site of the neighboring cell, '−' is used when it is necessary to count the links in decreasing order, rather than in increasing order, '−' placed before the three capital letter exchanges the role of the input site and the output site, '_' is used to replace the argument by the value of the link register. '|' is used to replace the argument by the number of input links divided by two.

[Figure 9 about here]

[Figure 10 about here]

A similar microcoding is used for the operands of TOP, that locally modifies the topology, by cutting or reordering links. In this case, the single child cell inherits the output links of the mother cell, and the analysis of the segments indicates movements of links from the input site of the mother cell to the input site of the child cell.

In appendix 3 there are also a number of program symbols which are used only for labeling cells. During the development of the network graph, the software that we have programmed can indicate the program symbols read by the cell. Neuron keep their reading head on the cellular code, on the last program symbol read. Therefore we use dummy program symbols to label leaves of the code. This enables us to see on the computer screen what function a given neuron performs.

3.4 Advanced program symbols

In this section, we describe some other program symbols used for the compilation of PASCAL programs. These program symbols are not described by their microcode, we explain them separately.

The program symbol `BLOC` blocks the development of a cell. A cell that reads `BLOC` waits that all its input neighbors cells become finished neurons. `BLOC` avoids that a particular piece of cellular code is developed to early. The compiler generates many grammar trees of cellular code: one tree for the main program and one tree for each function. These grammar trees have a number. The argument of the `JUMP` program symbol specifies the number of a grammar tree. A reading cell that executes the program symbol `JMP x` places its reading head on the root of the grammar tree which number is x. `JMP x` is equivalent to the recurrent program symbol `REC` if x is the number of the tree that is currently read by the cell.

[Figure 11 about here]

Links can be distributed into groups of links called sub-sites. Figure 11 (a) and (b) describes how a site divides into two sub-sites. Two conditions are required for a site to split: First the site flag-register called `divisible` must be set to 1. Second, a neighboring cell must divide. The distribution of links into group of links is interesting only with respect to the `SPLIT` program symbol. The execution of the `SPLIT` program symbol is done in two stage described in figure 11 (c), (d) and (e): First, some links are duplicated, so that the number of links on each output and input sub-site is the same (figure 11 (e)). We call n the number of links on each sub-site. In the second stage, the cell splits into n child cells. Each child cell has as much input (resp. output) links as there are input (resp. output) sub-sites in the mother cell.

4 Principles of the neural compiler

4.1 The stages of the compilation

[Figure 12 about here]

We will now present the neural compiler. The neural compiler has been programmed. The software is called JaNNeT (Just an Automatic Neural Network Translator). JaNNeT encompasses three stages that are shown in figure 12 (a) (b) (c). The input of the compiler is a program written in an enhanced PASCAL. The output is a neural net that computes what is specified by the PASCAL program. The PASCAL is said to be "enhanced" because it proposes supplementary instructions compared to standard PASCAL.

The first stage is the parsing of the program and the building of the parse tree. It is a standard technic that is used for the compilation of any language. Nevertheless, this parse tree has a somewhat unusual form. In appendix 3, a grammar defines the kind of parse tree that we use. The third stage uses cellular encoding. It is simply the decoding of the cellular code that has been generated at the second step. This decoding uses the development of a network graph, seen in section 2.

The second stage is the heart of the compiler. This stage is a rewriting of trees. The rewriting of a tree (in a simple case) consists in replacing one node of the tree by a sub tree. The added subtree must specify where to glue the sub trees of the node that is being rewritten. During the second stage each node of the parse tree is replaced by a sub tree labeled with program-symbols of the cellular encoding. When the program symbols of that sub tree will be executed by a cell c, they will make a local graph transformation, by replacing cell c into many other cells, connected to the neighbors of cell c. Each node of the parse tree can therefore be associated to a local transformation of a network graph of cells. This transformation is called a *macro program symbol*. A macro program symbol makes a transformation bigger that the one done by a program symbol of the cellular encoding scheme. A program symbol creates no more that one cell (except for the `SPLIT`), or it modifies no

more than one register. It uses no more than a single integer parameter. A macro program symbol can create many cells, modify many registers, and often needs more than one parameter. The macro program symbols are implemented using subtrees of program symbols listed in the appendix 3. The presentation of the compilation method will be done with few reference to the cellular encoding. We will consider the compilation at the level of macro program symbols.

In its present release, JaNNeT does not produce instructions that can be executed by a particular neural computer. It produces a neural network, which is a format that suits a neural computer with many processors dedicated to neuron simulation. Presently, the neural network generated by JaNNeT are simulated on a sequential machine. The last stage shown in figure 12 (d) consists in mapping the neural network on a physical architecture of a neural computer. This step must take into account the memory size of one neuron, the communications between processors, and the granularity of the computation made by one neuron.

4.2 Structure of the compiled neural network

The compiled neural network behaves like a data flow graph. Figure 13 describes a simple example of compiled neural network. Each variable V of the PASCAL program contains a value. The variable is initialized at the beginning of the program. Then it changes following the assignments that are made. For each variable V, there corresponds as many neurons as there are changes in V's value. All these neurons represent the values taken by V during a run of the program. At a given step of the run, V has a given value v. The value v is contained in one of the neuron that represents V. This neuron is connected to the input neighbors that contain values of other variables, which have been used to compute v. The same neuron is connected with output neighbor that contain values of other variables. The output neighbors use the value of V to compute the new value of the variable that they represent.

The environment in compilation means the set of variables that can be accessed from a given point of the program. Before the execution of each instruction, the environment is represented by a row r of neurons. This row contains as many neurons as there are variables. Each of these neurons contains the value of the variable that it represents, at this particular moment.

An instruction modifies the environment, by modifying the value of some given variables. An instruction is compiled into a neural layer that computes the modified values, from the old values stored in the row r of neurons. The new values are stored in a row r_1 of neurons. Hence the whole PASCAL program is compiled into a sequence of rows r, r_1, r_2, ... alternated with neural layers that compute the new values. There are as many such neural layers as there are PASCAL instructions.

[Figure 13 about here]

4.3 Compilation of a PASCAL instruction

The idea of the compilation method is to translate each word of the PASCAL language into a modification of a network graph of neurons. At a coarse granularity, the compilation of a program is decomposed in the compilation of each instruction. Each instruction is translated into a neural layer that is laid down, and a row of neurons that contains the environment modified by the instruction. So the compilation is a construction process. The compiler builds the row of neurons that contains the initial environment, by translating the part of the parse tree where the variables are declared. Then the compiler lays down consecutively neural layers, by compiling the program instruction by instruction.

This idea of progressive building of the compiled neural network can be applied with a granularity smaller than a PASCAL instruction. A PASCAL instruction can be decomposed into words of the PASCAL language, these words are organized according to a tree data structure called parse tree (see figure 14). we can associate each word of the PASCAL language with a local modification of a network graph of cells, so that the combined effect of these small modifications transforms a single cell into a neural layer that computes what is specified by the PASCAL instruction. This is modeled using a system similar to the cellular encoding. The neural network will be developed during many steps. Certain neurons will have a copy of the parse tree, and a reading head that reads a particular node of the parse tree. They will be called reading cell rather than neuron,

because their role is not to do the computation corresponding to a neuron, but to divide, so as to create the neural layer associated to the PASCAL instruction. Each cell reads the parse tree at a different position. The labels of the parse tree represent instructions for cell development that on the cell. As we already pointed out in subsection 4.1, these instructions called *macro program symbol* can be decomposed into a sequence of program symbols of the cellular encoding scheme. During a step of development, a cell executes the macro program symbol read by its reading head, and moves its reading head towards the leaves of the parse tree. A cell also manages a set of internal registers. Some of them are used during the development, while others determine the weights and the thresholds of the final neurons.

[Figure 14 about here]

Consider the problem of the development of a neural net for compiling the PASCAL instruction `"a:=a+b"`. The development begins with a single cell called the ancestor cell, connected to neurons that contain the initial values of the environment. Consider the network described on the right half of figure 15 on the right. At the beginning, the reading head of the ancestor cell is placed on the root of the parse tree of the PASCAL instruction, as indicated by the arrow connecting the two. Its registers are initialized with default values. This cell will divide many times, by executing the macro-program symbols associated to the parse tree. It will give birth to the neural network associated to the parse tree. At the end, all the cells loose their reading head, and become finished neurons. When there are only finished neurons, we have obtained the translated neural network layer. In all the following examples it is important to keep in mind that in the figures, the input and output connections are ordered. The connection number is coded by the position of the connection on the circle that represents the cell. For example, in figure 15, the link from cell “b” to cell “1” has number 2.

[Figure 15 about here]

[Figure 16 about here]

[Figure 17 about here]

4.4 Compilation of the PASCAL program

[Figure 18 about here]

We have shown how the parse tree of a PASCAL instruction can be interpreted as a tree labeled with macro program symbols. When these program symbols are executed by cells, they develop a neural layer that translates what is specified by the PASCAL instruction. This method can be generalized to the whole PASCAL program. The total program can be represented by its parse tree. Figure 18 on the left represents the parse tree of the PASCAL program "`program p; var a : integer; begin read(a); write (a); end.`". The first nodes of the parse tree are used to declare variables. They will create the first layer of neurons that contain the initial values of the variables. The following nodes of the parse tree correspond to the instructions. They will create many layers of neurons that make the computations associated to these instructions. Consider the starting neural network, on the right half of figure 18. The input pointer cell and the output pointer cell to which the ancestor cell is linked, do not execute any macro program symbols. At the end of the development, the input pointer cell points to the input units of the network, and the output pointer cell points to the output units. The development of the network is reported in appendix 1.

5 The macro program symbols

We have presented the method of neural compilation and the compilation of a small PASCAL program. The method is based on the association of each label of the parse tree with a macro

program symbol. We call each macro program symbol with the name of the associated label. A macro program symbol m replaces the cell c that executes it, by a graph d of reading cells and neurons. This graph is connected to the the neighbors of c. The term "neighbor" refers to two cells connected either with a direct connection or, indirectly, through special neurons called pointer neurons. The graph d must specify where the reading cells are going to read in the parse tree. In general, each reading cell will read one of the child node of the node labeled by m.

In this section, we details all the macro program symbols, by drawing its associated graph d, and explaining it. We will put forward an invariant pattern: when a cell is being rewritten by a macro program symbol, its neighbor cells are always of the same type, in the same number. If the environment contains n variables, the cell will have $n + 2$ input links, and two output links. The first input (resp. output) link points to the input (resp. output) pointer cell. The second input link is connected to a cell called "start" cell, the last n input links point to neurons that contain the values of the variables. The second output link points to a cell labeled "next". The input and output pointer cell points to the input and output units of the neural net. The start cell starts the neural network by sending a flow of null values. The "next" cell connects the graph generated by the macro program symbol, with the rest of the network graph. By extension, the cell that is rewritten is called the *ancestor cell*.

The parse tree that we use are described by a simple grammar in appendix 3. This grammar details the list of macro program symbols and how they are combined. We will use the non terminal of this grammar to explain the macro program symbols. We classify two kinds of macro program symbols, the macro program symbols of the type "expression", they correspond to labels generated using the non terminal <expr>, and produce neurons that are used to compute values. And the macro program symbols of the type "modification of environment" that modify the environment. In the invariant patter, if the macro program symbol is of the type "expression", the "next" cell is always a neuron, else it is a reading cell.

The first macro program symbol executed during the compilation of a program is the macro program symbol `PROGRAM` (figure 23 on the left). This macro program symbol creates the invariant pattern. By recurrence, this invariant is kept afterwards because the reading cells generated by

each macro program symbol have their neighbors that verify the invariant pattern.

We sometime use cellular encoding in our description. In the implementation, the macro program symbols are decomposed in program symbols. For each macro program symbol, these decompositions are reported in appendix 3. The program symbols of cellular encoding implement small modifications of graph. In the contrary, the macro program symbols create many cells, connected in a complex manner. The decomposition of macro program symbols into program symbols is a quick and simple way of implementing macro program symbols. During the compilation, it may happen that some cells, after the execution of a macro program symbol, block their development until all their input neighbors are neurons. When they unblock, they usually execute a piece of cellular code before going back to read the PASCAL parse tree.

5.1 Kernel of the PASCAL

We consider in this subsection, programs that are written with a reduced instruction set: the kernel of the PASCAL, that allows to write only very simple programs. We consider for the moment only scalar variables, we will see arrays later. The declaration of variables are translated into a string of `DECL` macro program symbols in the PASCAL parse tree. The effect of each `DECL` is to add an input link to the ancestor cell. This link is connected to a cell that is itself connected to the "start" cell with a weight 0 (figure 23 on the right and 24 on the left). This last connection ensures that the neurons corresponding to the variable will be activated, and will start the network. We suppose that each variable is initialized with the value 0.

The sequential execution is implemented using a string of reading cells, associated to a list of instructions. The cell of the string at position i reads the sub parse tree of instruction number i of the list. On right half of figure 24, the input neighbors of cell "1" are neurons that contain the values of the environment. Cell "$i+1$" has a single input neighbor which is the cell"i". Cell "2" must delay its development until cell "1" has finished to develop the subnetwork associated to instruction 1. Cell "2" blocks its development until all the input neighbor cells have became neurons.

The assignment (figure 15 on the right) is the most simple example of instruction. Each variable

corresponds to a particular input link of the ancestor cell. The storage of a value in a variable is translated by connecting the neuron that contains the value to the ancestor cell. The connection must have the right number, in order to store the value in the right variable. The management of the connection number is done by the macro program symbol `IDF-AFF`, figure 16 on the left. Similarly, in order to read the content of a variable, it is enough to know its link number (figure 17). The translation of `IDF-LEC` is done by connecting the neuron that contains the value of the variable to the neuron "next" that needs this value.

The input units of the neural net correspond to the values read during the execution of the PASCAL program. If in the PASCAL program the instruction `read a` appears, there will be one more input unit created. By setting the initial activity of this input unit to v, the variable a will be initialized with v. The output units correspond to the values written during the execution of the PASCAL program. If the instruction `write a` appears, there will be one more output unit created. Each time the instruction `write a` is executed, the value of this output unit will be the value of a. A neuron is an input unit if it is connected to the input pointer cell. It is an output unit if it is connected to the output pointer cell. The macro program symbol `READ` adds a connection from the input pointer cell to the neuron that represents the variable, and the macro program symbol `WRITE` adds a connection between the neuron that represents the variable and the output pointer cell (figure 25 on the left and 26 on the right).

An arithmetic expression is translated in a tree of neurons, each neuron implements a particular arithmetic operation using a particular sigmoïd. Unary operators are implemented in the same way as binary operators (figure 16). Two cells are created instead of three. The first cell will develop the sub network corresponding to the sub expression to which the unary operator is applied. The second one will place its reading head on the cellular code that describes how to develop a subnetwork for computing the unary operator. When a constant is needed in a computation, this constant is stored in the bias of a neuron n whose sigmoïd is the identity (figure 29). This neuron is linked to the "start" cell that will control its activity, that is to say, at what time, and how many times, the neuron n sends the constant to other neurons that need it.

5.2 Control Structures

Until now, the macro program symbols were rewriting one cell into less than 3 cells. We are going to present a new kind of macro program symbols. The `IF` rewrites a cell into a graph of cells whose size is proportional to the size of the environment. This represents a step in the complexity of the rewriting. This new class of macro program symbols can be implemented using the `SPLIT` program symbol and sub-sites (see section 3.4). The PASCAL line `if a then b else c` can be translated by the boolean formula: `(a AND b) OR ( (NOT a) AND c)` or by the neuronal formula: `(a EAND b) AOR ( (NOT a) EAND c)`. The neurons `EAND` (extended AND) and `AOR` (arithmetic OR) have respectively the dynamic $\mathcal{D}_1$ and $\mathcal{D}_2$ described in section 2.2. These neuron do not perform a real computation, they are used to control the flow of activities. Their behavior is thus completely describe by the dynamic. tt AOR and `EAND` correspond to the `MERGE` and `BRANCH` nodes in the dataflow formalism. The `TRUE` value is coded on 1, and the `FALSE` is coded on -1. A logical `NOT` can be implemented with a single neuron that has a -1 weight. On figure 30, we can see a layer of 6 neurons `EAND`, divided into two sub layers of three neurons. These sub layers direct the flow of data in the environment towards either the subnetwork that computes the body of the "then" or the subnetwork that computes the body of the "else". A layer of 3 `AOR` neurons retrieves either the output environment of the "then" subnetwork, or the output environment from the "else" subnetwork, and transmits it to the next instruction.

The instruction `while <condition> do <body>` corresponds to a recurrent neural network in which an infinite loop of computations can take place using a finite memory. The flow of activity enters the recurrent neural network through a layer of `AOR` neurons. Then, it goes through a network that compute the condition of the while. If the condition is false, the flow of activities goes out of the recurrent neural networks. Otherwise, it is sent to a subnetwork that computes the body of the loop. After that, it goes through a layer of synchronization neurons and is sent back to the input layer of `AOR`. The synchronization is done to avoid that a given neuron in the body of the loop updates its activity two times, whereas in the mean time, another neuron has not updated it at all. In the case of two encapsulated loops, the neurons corresponding to the inner loop come back to their initial state when the computations of the inner loop are finished. They are ready

to start another loop if the outer loop commands it. The neuron "start" is considered as if it was storing the value of a variable. There is a copy of it in the layer of `AOR` neurons. It is connected to all the neurons that contain constants used in the body of the loop, and activate these neurons at each loop iteration.

The compilation of the `REPEAT` is very similar to the `WHILE`. The computation of the condition is done at the end, instead of at the beginning.

5.3 Procedures and functions

We now explain how to compile procedures and functions. We will refer to the following PASCAL Program:

```
program proc_func

Var glob: integer                (* global variable*)

Procedure proc2(paf2: integer)  (*formal parameter *)
var loc2: integer;               (*variable local to  procedure proc2*)
   begin
   ...
   end;

Procedure proc1(paf1: integer)  (*formal parameter *)
var loc1: integer;               (*variable local to  procedure proc1*)
   begin
   proc2(pef2);                 (*parameter  *)
   end;                         (*passed  by value to proc2*)

BEGIN                           (*body of the main program*)
   proc1(pef1)                  (*parameter *)
END.                            (*passed  by value to proc1*)
```

We use a slightly modified invariant pattern, where the environment contains three variables. The first variable is global, the second is a parameter passed to a procedure, the third is one is a local variable of a procedure. In all the figures, the first output link of cell "1" points to the output pointer cell and the second link points to the "next" cell. If, sometimes, the inverse is represented, it is only for a purpose of clarity in the representation. It allows to have a planar graph.

Consider the moment when `proc1` calls `proc2`. The environment on the neuronal side encompasses the input pointer cell, the "start" cell, the global PASCAL variable `glob` and the local

variables pef1 and loc1. We want to translate the calling of procedure proc2. First we suppress the local variables pef1 and loc1. This is done by the macro program symbol CALL-P described figure 34. Then we develop the parameters passed by value from proc1 to proc2 (macro program symbol COMMA figure 36). Finally, the ancestor cell will develop the body of proc2. The body of proc2 begins with the macro program symbol PROCEDURE that inserts a cell. When the translation of procedure proc2 is finished, this cell will pop the local variables of proc2 (cellular code POP figure 37 on the right). After that, a cell (inserted by a CALL-P) executes a cellular code RESTORE that allows to recover the local variables of proc1. Putting aside the local variables of proc1 ensures that each variable can be assigned a fixed connection number. The macro program symbols CALL-P, CALL-F and POP use two parameters: GLOBAL is the number of global variables in the environment and LOCAL is the number of local variables. The macro program symbol CALL-F is simpler than CALL-P. It does not need to create a cell that will restore the environment of the calling procedure, because a function returns a value instead of a complete environment. The macro program symbol FUNCTION has no effect. It is not necessary to create a cell that will pop the environment of the function. The macro-program symbol RETURN also has a null effect.

5.4 The arrays

The arrays are a very important data structure in a programming language like PASCAL. We had to use a special kind of neurons in order to be able to handle arrays. These neurons are called pointer neurons. They are used only to record the array data structure in a neural tree. The pointer neurons are nodes of the neural tree. The data are the leaves of the neural tree. The use of pointer neurons ensures that the ancestor cell possesses exactly one input link per variable. It allows to handle the variables separately, when one want to read or to modify them. Using pointer neurons, one can represent array with arbitrary dimension. The invariant must be extended. It now contains pointer neurons. Figure 40 uses an example of extended invariant, that represents a 1D array with two elements. A tree of depth d can represent an array with d dimensions. The use of many layers of pointer neurons for storing an array with many dimensions is necessary to handle each column separately, in each dimension.

When an array variable is declared, the structure of neural tree is created for the first time. For this purpose, the type of a variable is coded in the parse tree as a string of macro program symbols `TYPE-ARRAY` (see figure 41). Each of these macro program symbol is associated to one dimension of the array, and creates one layer of pointer neurons, in the neural tree. Each macro program symbol `TYPE-ARRAY` has a single parameter which is the number of columns along its associated dimension. The string of `TYPE-ARRAY` is ended by a macro program symbol `TYPE-SIMPLE` which was already described in figure 24 on the left.

Many macro program symbols must handle neural trees. For example, the `IF, WHILE, X-AOR, X-SYNC` are macro program symbol that must re-build the structure of pointer neurons. Since the depth of the tree can be arbitrary big, the tree of neuron must be processed recursively. Cellular encoding allows recursive development as described in section 3.2. A 2-ary program symbol `BPN x` is used to stop the recursion. It tests whether the neighbor x is a pointer neuron, or not. Depending on the result of the test, the left or the right subtree will be executed.

The reading of an array, like `a:=t[i1, i2]` is translated by an unusual parse tree. The parse tree used for `t[i1, i2]` is indicated figure 43. The reading of an array is interpreted as the result of a function `read_index` defined by a cellular code that has $d+1$ inputs for an array of d dimensions. The inputs are the d indices plus the name of the array. The function `read_index` returns the value read in the array. The writing of an array like for example `t[i1, i2]:=v` is translated in a parse tree indicated figure 44. The writing of an array of d dimensions is interpreted as the result of a function `write_index` defined by a cellular code, with $d+2$ inputs and one output that is an array. The inputs are the d indices, the value to assign, the name of the array. The function`write_index` returns the modified array after the writing. For either reading or writing in arrays, the number of used neurons is proportional to the number of elements in the array. However if the indices used to read or to write the array are known at compilation time (for example the instruction `a[0]=1`) the neural net is automatically simplified during the development, and the final number of used neurons is constant.

5.5 The enhanced PASCAL

We now describe the compilation of new instruction that have been added to the standard PASCAL. These instructions exploit the power of neural networks. The instruction `CALLGEN` allows to include a neural network directly defined by a cellular code. Alternatively, `CALLGEN` can be considered as a call to a function defined by a cellular code. It is similar to a machine call in classic programming languages. At the beginning of the PASCAL program, there must be an instruction `#include <file-name>` that indicates the name of a file where the cellular codes of the functions to be called are stored. The syntax of the `CALLGEN` is the following: `<idf0> := CALLGEN("code-name", <idf1>, .., <idfk>)`. the result of the call will be a value assigned to the variable `idf0`. The code name indicates the particular cellular code of the called function. `idf1, .. idfk` are the parameter passed to the function. The macro program symbol `CALLGEN` has the same effect as `CALL-F` except that the body of the called function is not specified by a parse tree, but by a cellular code that has been defined before the compilation takes place. The `CALLGEN` includes a neural layer that computes the called function. The neurons that contain the values of `idf1, ..., idf0` are connected to the input of the included neural network. The output of the included neural network is connected to the neurons representing the variable `idf0`. The philosophy of the `CALLGEN` is to define a number of functions that can be used in various context, or to include some neural networks that have been trained. The neural networks included by a `CALLGEN` never use global variables. Hence the global variables are suppressed before the calling. Examples of predefined functions in the appendix 3 are:

- function `left` returns the left half of an array
- function `right` returns the right half of an array
- function `concat` merges two arrays.
- function `int_to_array` adds a dimension to an array
- function `array_to_int` suppresses a dimension to an array
- function `random` initialize an array with random values in $\{-1, 0, 1\}$

The compiler can do automatic parallelization of a program written with a divide and conquer strategy. In this strategy, a problem of size n is decomposed into two subproblems of size $n/2$ and then into 4 subproblems of size $n/4$ and so on until problems of size 1. The array functions `left` and `right` are used to divide the data of a problem into two equal parts. The decomposition process must be stopped when the size of the problem is 1. We must test during the development of the neural net, when the number of elements of the array is 1. In one case, the part of the parse tree corresponding to the decomposition must be developed. In the other case, the part of the parse tree corresponding to the problem of size one must be developed. We use a static `IF` to do this test. The syntax is the same as the normal `IF`. The name `IF THEN, ELSE` are replaced by `#IF #THEN, #ELSE`. However, the condition of the `#IF` is always of the type `t=1`, where t is an array. This expression tests whether the array t has one or more than one element. If it is the case, the ancestor cell goes to read the left sub-tree of the `#ELSE` parse tree label. Otherwise it goes to read the right sub tree.

6 Examples of compilation

In this section, we propose a few examples of compilation that illustrate the principles of the compiler and its interest. In order to run a compiled network, on must initialize the input unit with the value read during the execution of the PASCAL program. The neural net makes its computation with a data driven dynamic which is halfway between parallel and sequential. When the network is stable, the computation is finished. The output units contain the values that are written by the PASCAL program.

Examples of compiled neural networks are drawn in figure 19 and 22. The drawing have been produced automatically by the compiler. Each cell possesses a rectangular window. The ancestor cell possesses a window that covers all the drawing area. Whenever a cell divides, each child cell inherits half of the window. The division is made horizontally if the two cells are connected, and vertically otherwise. A cell is drawn at the middle of its windows. An additional mechanism ensures that each cell possesses a window of approximately the same area.

6.1 Compilation of two standard PASCAL programs

[Figure 19 about here]

Figure 19 (a) shows a network compiled with the following program:

```
program p;
type tab = array [0..7] of integer;
var t : tab; i, j, max, aux : integer;
begin
  read (t);
  while i < 7 do
  begin
    j := i + 1;max := i;
    while j <= 7 do
    begin
      if t[max] < t[j] then max := j  fi;
      j := j + 1;
    end;
    aux := t[i];t[i] := t[max];
    t[max] := aux; i := i + 1;
  end;
  write (t);
end.
```

There are 9 input units, the "start" cell plus the 8 values to be sorted. There are 8 output units that correspond to the instruction `write(t)`. Recurrent connections are drawn using three segments. The two encapsulated loops that are mapped on two sets of recurrent connections. Figure 20 shows the step of development for the bubble sorting neural network. During one step, all the cells that are not blocked are iterated sequentially. At step 63, the 8 cells colored in grays are deleted. At step 95, the first recurrent neural network for the outer loop is born. At step 59, the second recurrent neural network for the inner loop is born. At step 223, all the recurrent links of each of the two recurrent neural network are drawn as a single line to spare room. From step 223 to 356, instruction for reading and writing in arrays are translated.

[Figure 20 about here]

Figure 19 (b) shows a neural net that has been compiled with the following program:

```
program Fibonacci;
var resul: integer; a : integer;
function fib(n: integer):integer;
begin
  if (n<=1) then resul := 1
  else resul := fib(n-1)+fib(n-2) fi;
  return (resul);
end;
begin
  read (a); write (fib (a));
end.
```

This example illustrates the fact that if the program to be compiled uses recursive calls, the depth of recursion must be bounded, before the compilation begins. The initial life of the ancestor cell is 6. Hence the depth of recursion is been bounded by 6. This net can compute `fib(i)` for`i<7`. The number 6 is the value used to initialize the life register of the ancestor cell. Figure 21 shows how the neural net evolves when the life is increased from 1 to 4.

[Figure 21 about here]

6.2 How to use the `CALLGEN` instruction.

We have added to the PASCAL language two instructions: `CALLGEN` and `#include` that allow to include in the final network, smaller networks whose weights have been found using a learning algorithm. The code of these nets are stored in a file which name is indicated by the instruction `#include`. We now show an example of PASCAL program that uses the `CALLGEN` instruction. The compiled net is shown in figure 19 (c).

```
#include "animal.a"

program animal;
type tab=ARRAY[0..19] of integer; tab2=ARRAY[0..9] of integer;
var pixel: tab; feature: tab2; position, is-bad, move: integer;

function opposite(position: integer):integer;
begin  return(1-position); end;
```

```
begin
  read(pixel); feature:=CALLGEN("retina",pixel);
  position:=CALLGEN("position_object",feature);
  is-bad:=CALLGEN("predator",feature);
  if(is-bad=1) then move:=opposite(position)
  else move:=0 fi;
  write(CALLGEN("motor",move));
end.
```

In the file `animal.a` are stored the cellular codes of different layered neural networks. The network `retina` inputs 20 pixels, and outputs a list of 10 relevant features. The network `position-object` determines from these features, the position of an object, if an object lies in the visual field. The neural net `predator` determines whether the object is a predator animal. The neural net `motor` commands 4 motor neurons for moving the legs. The whole compiled neural net simulates the behavior of an animal in front of a predator. The input units are the pixels, and the output units are the neurons of the leg muscles. This neural net can be stored in a library, in order to be included in a bigger neural network. It illustrates how easy it is to interface the PASCAL language with predefined neural nets. The compiled neural net encompasses many neural layers, plus one part that is purely symbolic. This is the part that corresponds to the `if` instruction. So the compiler not only links together various neural networks, but it allows to make symbolic computations from the outputs of the neural networks, and it can manage the input/output.

6.3 How to use the #IF instruction

The instructions `#IF #THEN #ELSE` allow to test at compile time whether an array has more than one elements. This instruction allows automatic parallelization of a program written with a "divide and conquer" strategy (subsection 5.5). We need other predefined functions to enrich the array data structure: functions `left` and `right` extract the left part and the right part of an array, `int-to-array` and `array-to-int` add or suppress a dimension. The following PASCAL program uses the instruction `#IF`. The compiled net is shown figure 19 (d).

```
#include "array.a"
program merge sort;
const infini=10000; type tab = ARRAY [0..7] of integer; var w : tab;
```

```
function merge(t: tab; n: integer; u: tab; m: integer): tab;
var v: tab; i, j, a, b: integer;
begin
  v:=CALLGEN("concat", t, u); a:=t[0]; b:=u[0];
  while(i+j<n+m) do
  begin
    if(a<b) then
    begin  v[i+j]:=a;i:=i+1; if (i<n) then a:=t[i] else a:=infini  fi; end
    else
     begin v[i+j]:=b;j:=j+1; if (j<m) then b:=u[j] else  b:=infini fi; end
    fi;
  end;
  return(v);
end;

function sort(t: tab; n: integer) : tab;
begin  #IF t=1
       #THEN return(t)
       #ELSE return(merge(sort(CALLGEN("left",t),n DIV 2),  n DIV 2,
                sort(CALLGEN("right",t),n-(n DIV 2)), n-(n DIV 2)))
       #FI;
end;

begin read(w); write(sort(w,8)); end.
```

Function `merge` is written in standard PASCAL (except one use of `CALLGEN`). Function `sort` uses the `#IF` instruction. The file `array.a` contains the library of the cellular codes of the functions manipulating arrays. The compiled neural net has 9 input units: the start cells plus 8 integers to sort. It has 8 output units for the 8 sorted integers. The structure of the net can be understood. The first layer of 4 recurrent neural nets outputs 4 sorted lists of two integers each. The second layer outputs two sorted lists of four integers. Although it is recursive, the function `sort` can be compiled on a finite graph, because it is applied to a finite array of integers. As opposed to the Fibonacci function, there is no need to specify a bound on the number of recursive encapsulations. Since we have used the divide and conquer strategy, the network sorts n integers in a time which is $O(n)$ (here we assume that the neurons can do their computations in parallel). We have also compiled many other programs using the divide and conquer strategy. The convolution, the matrix product, the maximum of an integer list, are programs that are compiled on a neural net able to do the job in a time $O(\ln(n))$. Figure 22 shows the compiled neural nets. We wrote a PASCAL program that simulates a feed-forward layered neural network, with matrix vector multiplication

and random weights in $\{0, -1, 1\}$. Figure 22 (c) shows the compiled neural net, it is a layered neural net!

[Figure 22 about here]

7 Conclusion and applications

In this paper, we describe a neural compiler that has been actually programmed. The input of the compiler is a PASCAL Program. The compiler produces a neural network that computes what is specified by the PASCAL program. The principle of compilation is not difficult. It consists in the rewriting of the parse tree in a cellular code, and the development of the cellular code, which can be formalize as the derivation of a graph grammar. Experimental results show that the compiler works. The neural compiler is called JaNNeT (Just a Neural Network Translator). Neural compilation is based on a new paradigm: Automatic building of neural networks using an algorithmic description of the problem to be solved. This paradigm is at the exact opposite of the existing trend which present neural networks as machine that learns by themselves. As a conclusion, we want to show that JaNNeT can reproduce and even improve three kinds of compilation.

7.1 Neural network design

The interest of a language for describing neural networks like SESAME([9]) is to facilitate the design of large and complex neural networks. These languages propose to first define some "building block" neural networks. Then, the building blocks are used to build more complex neural networks. The latter can again be composed, and so on... This hierarchical design is very practical. We have added an instruction `CALLGEN` to the standard PASCAL language that allows to achieve the same effect with JaNNeT. When it compiles a neural net, JaNNeT produces its cellular code. This cellular code can be stored in a file, and called using CALLGEN, so as to be included in a bigger neural network. This technic allows a modular compilation, from the compilation point of view. It permit a hierarchical design, from the neural network design point of view. But JaNNeT goes further than

a language of description like SESAME. JaNNeT gives the possibility to combine neural building blocks without mentioning the physical connections between the networks. The specification is made on a logical, soft level. A human being understand much better a soft description where it is enough to describe the logical steps of a computation, rather than an intricate set of connections. JaNNeT allows the design of huge neural networks. In the near future, machines with millions of neurons will be available, and the advantages of a logical description will become obvious. Moreover, JaNNeT produces a graphical representation of the neural net that exploit the regularities of the graph. To our knowledge, this is the very first software able to do that.

7.2 Tool for the design of hybrid systems

JaNNeT is a process that can compile a PASCAL program towards a neural network. If we would like to compile a base of rules used in expert systems (artificial intelligence), towards a neural network, we just need to write the base of rules in PASCAL, as well as the inference motor. Hence JaNNeT includes the compiler of base of rules, used in the so called hybrid systems, which goal is to combine artificial intelligence and neural networks. However, JaNNeT can be used for the design of hybrid system in a more efficient way. The compiler can put in the same connectionist model, algorithmic knowledge, (that knowledge is compiled) and "fuzzy knowledge", (that one is learned in a fixed size neural networks). The compiler can build hybrid systems that encompass two layers. The first layer is learned and it is sub symbolic. It grounds symbols on fuzzy and distributed data. The second layer is compiled, it makes a computation on these symbols.

7.3 Automatic parallelization

The numerical examples show that JaNNeT can automatically parallelize algorithms written with a "divide and conquer" strategy. Many algorithm can be programmed in that way. However, in its actual version, JaNNeT does not produce instructions for a particular parallel machine. It produces a neural network. The missing step consists in mapping this network on a multi processor system. This step must take into account the size of the memory of each processor, the communications between processors, and the granularity of the neuronal computation. The neural networks produced by JaNNeT can be mapped on a 3D grid. The three dimensions represent

the size of the environment, the functional parallelism, and the time. It is possible to project the network along the third dimension so as to map the neural net on a 2D array of processors.

We must also parallelize the compiler itself. If the compiler takes one hour to compile a program that afterwards runs in one second on a parallel machine, it prevents the interactive use of JaNNeT. We have a model of abstract parallel machine that may allow both parallel compilation and parallel run. This machine is a 2D grid of processors were a particular processor can dynamically decide to become a connection. A line of such processors can pass information from arbitrary distant point in constant time.

Since the compiled neural net is a data flow graph, a comparison with existing compiler for data flow machine is needed. The main feature of our parallel compiler will be to use a precise architecture (a 2D grid), and to ensure that two tasks which need to communicate data will be executed by two neighboring processors. The review done by Veen, 1986, in [15], and the research monograph [2] (1991) do not report such direction of research in the dataflow community. Last, the advantage of JaNNeT compared to the data parallel approach described by Zima et al. [17] (1993) or the dataflow approach by Veen et al. [16] is that the user does not need to add in his program indications about where (on which processor) to store the data.

Acknowledgment

This work was supported by the Centre d'études nucléaire de Grenoble, the European community within the working group ASMICS, and the NFS grant IRI-9312748. Michel Cosnard gave us the idea to compile a language like PASCAL. Without his advice, instead of PASCAL, we would have compiled a poor language without variables. We are thankful to Pierre Peretto and Michel Cosnard for their many encouragements. The introduction is inspired of a report on Gruau's PhD thesis from Maurice Nivat. We thank Pascal Koiran and Hava Siegelmann for their helpful comments about this paper.

References

[1] Mc Cullochs and W.S. Pitts. A logical calculus of the ideas immanent in nervous activity.

Bull. Math. Biophy., 5:115–133, 1943.

[2] J. Gaudiot and L Bic. *Advanced topics in dataflow computing*. Prentice Hall, 1991.

[3] F. Gruau. Cellular encoding of genetic neural network. Research report 92.21, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, 1992.

[4] F. Gruau. Genetic synthesis of boolean neural networks with a cell rewriting developmental process. In *Combination of Genetic Algorithms and Neural Networks*, 1992.

[5] F. Gruau. Process of translation and conception of neural networks based on a logical description of the target problem. Patent EN 93 158 92, December 30, 1993., 1993.

[6] F. Gruau. *Neural Network Synthesis using Cellular Encoding and the Genetic Algorithm*. PhD Thesis, Ecole Normale Supérieure de Lyon, 1994. anonymous ftp: lip.ens-lyon.fr (140.77.1.11) directory pub/Rapports/PhD file PhD94-01-E.ps.Z (english) PhD94-01-F.ps.Z (french).

[7] F. Gruau and D. Whitley. Adding learning to the the cellular developmental process: a comparative study. *Evolutionary Computation V1N3*, 1993.

[8] S.C. Kleene. Representation of events in nerve nets. In Shannon and Mc Carthy, editors, *Automata Studies*, pages 3–40. Princeton University press, 1956.

[9] A. Linden and C. Tietz. Combining multiple neural network paradigm and application using sesame. In *International Joint Conference on Neural Networks*. IEEE computer society press, 1992.

[10] M. Minsky and S. Pappert. *Perceptrons: an introduction to computational geometry*. MIT press Cambridge, England, 1969.

[11] F. Rosenblatt. A probabilistic model for information storage and organization in the brain. *Psych. Rev.*, 62:386–407, 1958.

[12] H Siegelmann. Neural programming language. In *conference of the American Association for Artificial Intelligence*, 1994.

[13] H. Siegelmann and E. Sontag. Turing computability with neural networks. *Applied Mathematics letters*, 4(6):77 – 80, 1991.

[14] H Siegelmann and E. Sontag. On the computational power of neural networks. In *ACM workshop on computational learning, Pittsburg*, pages 440 – 449, 1992.

[15] A. Veen. Data flow machine architecture. *ACM computing surveys*, 18 (04):365–396, 1986.

[16] A. H. Veen and R. Born. Compiling c for the ddtn data-flow computer. In J. L. Gaudiot and L. Bic, editors, *Advances Topics in Data-Flow Computing*. Prentice Hall, 1991.

[17] H. Zima, P. Brezany, B.Chapman, and J. Hulman. Automatic parallelization for distributed memory system. In *European Informatics Congress System Architecture*, 1993.

Appendix 1: Small example of compilation

This appendix describes the compilation of a very simple PASCAL program: "`program p; var a: integer; begin write(a); read(a); end.`". Figure 18 describes the initial setting, at step 0.

[Figure 23 about here]

[Figure 24 about here]

[Figure 25 about here]

[Figure 26 about here]

[Figure 27 about here]

Appendix 2: The other macro program symbols

In this appendix we describe the macro program symbols that have not yet been illustrated.

[Figure 28 about here]

[Figure 29 about here]

[Figure 30 about here]

[Figure 31 about here]

[Figure 32 about here]

[Figure 33 about here]

[Figure 34 about here]

[Figure 35 about here]

[Figure 36 about here]

[Figure 37 about here]

[Figure 38 about here]

[Figure 39 about here]

[Figure 40 about here]

[Figure 41 about here]

[Figure 42 about here]

[Figure 43 about here]

[Figure 44 about here]

[Figure 45 about here]

[Figure 46 about here]

8 Appendix 3 : Technical implementation

Registers of the cell

[Table 1 about here]

Moreover, links have 3 registers, one for the weight `weight`, one for the state `state`, and one for the mark of beginning of a sub site `begin-sub-site`. Sites have a single register called `divisible`.

Syntax of the micro coding of cellular operators

The following grammar generates the microcode of operators

```
<microcode> ::= <top> | <div> |  <reg> | <exe> | <aff>
      <top> ::= -TOP<segments> | TOP<segments>
      <div> ::= <div1> | <div2>
     <div1> ::= -DIV<segments> | DIV<segments>
 <segments> ::=<segment><segments> | <segment>
  <segment> ::=<operator><operand> |<operator>^<operand> | s
 <operator> ::=  m | M | r | R | d | D
  <operand> ::= * | > | >= | < | <= | = | $ | #
     <div2> ::= HOM<number>
      <reg> ::= CELL<character> | SIT<character> | LNK<character>
      <exe> ::= EXE<character> | BRA<character>
      <aff> ::= AFF<character>
```

The non terminal <number> is rewritten into a number, The non terminal <character> is rewritten into a character. The characer is: For the non terminal CEL, SIT, LNK, the abbreviated name of a cell register, a site register or a link register; for EXE, it reminds what kind of movements the reading head does. For BRA, it specifies the kind of condition that is tested, for AFF is indicated the kind of the neuron that is displayed.

List of program symbols and their microcode

The letter x indicates that the program symbol uses an integer argument. i is the number of input links, o is the number of output links, r is the value of the link register.

```
*****************Local Topological Transformation**************************
  CUTL x   TOPm-<m->    Cuts the input links i-x
  CUTRI x  TOPm<        Cuts the right input links starting from link x
  CUT x    TOPm<m>      Cuts input link  x
  CLIP     TOPm_<m_>    Cuts input link  r
  CUTO x   -TOPm<m>     Cuts output link  x
  MRG x    TOPm<d^*m>   Merges the input link of input neighbour x
  MG x     TOPm_<d_^*m_>Merges the input link of input neighbor r
  CPFO x  -TOPm<d^#m>=  Copies  first output of output cell x
  CHOPI x  TOPm=        Cuts all the input links except link x
  CHOPO x  -TOPm=       Cuts all the output links except link x
  CHHL     TOPm|>       Cuts the first input links from 1 until i/2
  CHHR     TOPm|<=      Cuts the last input links from i/2+1 until i.
```

```
  PUTF x   TOPm=m<m>    Puts link x in first position
  PUTL x   TOPm<m>m=    Puts link x in last position
  SWITCH x TOPm<m$m>$m= Makes a link permutation between  last link and link x
  KILL     TOP          Deletes the neurone
  CYC      TOPs         Creates a recurrent link
********Cell division into two childs which will execute a separate   code******
  PAR      DIVd*R*      Parallel  division
  SEQ      DIVs         Sequential division
  XSPL x   DIVm<=s      Gradual division
  SEP      DIVd*M|>     Separation division
  ADL x    DIVm<sm>     Adds a cell on link x
  AD  x    DIVm_<sm_>   Adds a cell on link r
  IPAR x   DIVm<sd=m>   Duplicates link x, Add a cell on link x
  SPLFI x  DIVd<=s      Duplicates first x inputs
  SPLFIL x DIVd-<=s     Duplicates first i-x inputs
  SPLLI x  DIVsd->=     Duplicates last input starting at link x
  SPLLIL x DIVsd>=      Duplicates last input starting at link i-x
******Cell division into more than two childs which will execute the same code***
  CLONE x  OLC          Clones into x childs, x is the argument
  SPLIT    HOM0         Splits into y childs, y is computed from the topology
  TAB      HOM1         Same has split, but child i has its bias set to i
  SPLITN   HOM2         Same as split, but the sub-sites are merged
************************Modification of  a register*******************************
  SBIAS x  CELb         Sets the bias to the argument
  SBIAS0   CELb 0       Sets the bias to 0
  SBIAS1   CELb 1       Sets the bias to 1
  INCBIAS  CEL+b 1      Increments the bias
  SSIGMO x CELs         Sets the type of sigmoid
  SDYN x   CELd         Sets the dymamic
  EVER x   CELe         Sets the level of simplifiability
  LR x     CELl         Sets the link register
  INCLR    CEL+l 1      Increments the link register
  DECLR    CEL+l -1     Decrements the link register
  SITE     SITD         Sets the divisability of the sub sites to the argument
  SITI     SITd         Sets the divisability of the input sub-sites to the argument
  SITO    -SITd        Sets the divisability of the output sub-sites to the argument
  DELSITE  SITO         Merges all the input sites and all the output sites
  DELSITI  SITo         Merges all the input sites
  DELSITO  -SITo        Merges all the output sites
  MULTSIT  SITm         Creates one input site for each input link
  VAL x    LNKv         Sets the value of the input weight r to x
  VALO     -LNKv        Sets the value of the output weight r to x
  VAL-     LNKv -1      Sets the value of the input weight r to -1
  RAND x   LNK?         Sets weight x to a random value in {-1, 0, 1}
************************Managing of the execution order*************************
  WAIT x   EXEw         Waits x steps
  WAIT     EXEf         Waits one step if no arguments are supplied
  JMP  x   EXEj         Includes subnetwork x
  REC      EXEr         Moves the reading head back to the root of the currently read subtree
  END      EXEe         Becomes a finished neuron
  BLOC     EXEq         Waits for the neighbor to loose their reading head
  BLIFE    BRAf         Tests if the value of the life register is one
  BUN x    BRAu         Tests if the neighbor has x input links
  BPN      BRAp         Tests if the neighbour is a pointer neuron
  BLR x    BRAl         Tests if the value of the link register equals the argument
```

```
**********Enhancing display of the graph**********
  PN       AFFpn        Draws a pointer neuron
  VAR      AFFvar       Draws a neuron that contains the value of a variable
**********Operators for labeling**********
  RINS     EXEf         It indicates insertion of the right sub-tree
  LINS     EXEf         It indicates insertion of the left sub-tree
  PN_      CELe         It is a pointer neuron
  VAR_     EXEe         It contains the initial value of a variable
  EAND_    EXEe         It is a dynamic EAND
  AOR_     EXEe         It is a dynamic AOR
  INF_     EXEe         It compares the 2 inputs
  SUP_     EXEe         It compares the 2 inputs
  INFEQ_   EXEe         It compares the 2 inputs
  SUPEQ_   EXEe         It compares the 2 inputs
  PLUS_    EXEe         It adds the 2 inputs
  MOINS_   EXEe         It substracts the first input from the second inputs
  MULT_    EXEe         It multiplies the 2 inputs
  QUO_     EXEe         It divides the first input by the second inputs
  NOT_     EXEe         It does a logical NOT
  OR_      EXEe         It does a logical OR
  AND_     EXEe         It does a logical AND
  NEQ_     EXEe         It tests if the two inputs are different
  EQ_      EXEe         It tests if the two inputs are equal
  EQB_     EXEe         It tests if the input is equal to bias
  START_   EXEe         It is a Start neuron
```

Syntax of the parse trees that are used.

The following grammar generates parenthesized expressions that can be interpreted into well formed parse trees. A parse tree can be the parse tree of the main program, the parse tree of a procedure, or the parse tree of a function. The compiler generates a list of trees, one tree for the main program, and one tree for each function or procedure. The parenthesized representation used here is a bit unusual. We do not put parenthesis when there is a single subtree. When there are two sub trees, we consider that the right sub tree is a trunc, and put only the left subtree between parenthesis. This special representation allows to write a simple grammar. Capital letters and parenthesis are terminals of the grammar. Non terminals are written using small letters, between brackets.

```
<parse_tree> ::= <program> | <procedure> | <function>
   <program> ::= PROGRAM <decl_list> <inst_list>
 <procedure> ::= PROCEDURE <decl_list> <inst_list>
  <function> ::= FUNCTION <decl_list> <inst_f>
 <decl_list> ::= DECL attrb (<type>)<decl_list> | DECL attrb <type>
```

```
      <type> ::= TYPE_ARRAY attrb <type> | TYPE_SIMPLE
 <inst_list> ::= SEMI_COLON (inst) <inst_list> | <inst>
    <inst_f> ::= SEMI_COLON (<inst>) <inst_f> |  RETURN <expr>
      <inst> ::= <write> | <read> | <assign> | <if> | <#if> |
                 <while> | <repeat> |<call_p> | <callgen>
    <assign> ::= ASSIGN ( <idf_aff> ) <expr>
      <read> ::= READ <idf_aff>
   <idf_aff> ::= IDF_AFF attrb
     <write> ::= WRITE <expr>
        <if> ::= IF(<expr>) <then>
      <then> ::= THEN (<inst_list>)<inst_list>
       <#if> ::= #IF(<idf_lect>) <#then>
     <#then> ::= #THEN (<inst_list>)<inst_list>
     <while> ::= WHILE (<expr>) <inst_list>
    <repeat> ::= REPEAT (<expr>) <inst_list>
    <call_p> ::= CALL_P attrb <param_list>
<param_list> ::= COMA (<expr>)<param_list> | NO_PARAM
   <callgen> ::= CALLGEN attrb0 <param_list>
      <expr> ::= <read_array> | <write_array> | <idf_lect> |
                 <const> | <bin_op> | <un_op> | <call_f>
<read_array> ::= CALLGEN attrb1  <param_list>
<write_array> ::= CALLGEN attrb2 <param_list>
  <idf_lect> ::= IDF_LEC attrb
     <const> ::= INT_CST attrb
    <bin_op> ::= BINOP attrb (<expr>)<expr>
    <bin_op> ::= UNOP attrb <expr>
    <call_f> ::= CALL_F attrb <param_list>
```

This grammar generates parse trees with nodes having sometimes some attributes. Table 2 indicates what each attribute represents.

[Table 2 about here]

Predefinitions

The predefinitions are elementary cellular codes that are used many times. They are defined separately in order to reduce the size of the cellular code generated by the neural compiler.

Arithmetic operators.

Elementary codes implements arithmetic operators, using neurons, as well as the AOR and EAND dynamic. The sigmo"ids are in increasing order: Identity(1), Pi-units(3), stair step-L(5), stair step-R(6), equality to zero (7) Div-unit (8)

OR SBIAS 2 (SSIGMO 5 *stair step-L* (OR-))

AND SSIGMO 5 *stair step -L* (AND-)

NEG LR 1 (VAL -1 (NOT-))

INFEQ LR 1 (VAL -1 (SSIGMO 6 *stair step-R* (INFEQ-)))

SUPEQ LR 2 (VAL -1 (SSIGMO 6 *stair step-R* (SUPEQ-)))

INF LR 1 (VAL -1 (SSIGMO 5 *stair step-L* (INF-)))

SUP LR 2 (VAL -1 (SSIGMO 5 *stair step–L* (SUP-)))

EQ LR 1 (VAL -1 (SSIGMO 7 *equality to zero* (EQ-)))

NEQ SEQ (JMP EQ) (JMP NEG)

PLUS SSIGMO 1 *identity* (PLUS-)

MOINS LR 2 (VAL -1 (SSIGMO 1 *identity* (MOINS-)))

MULT SSIGMO 3 *PI Unit* (MULT-)

QUO SSIGMO 8 *DIV Unit* (QUO-)

EOR SDYN 2 (AOR-)

EAND SDYN 3 (EAND-)

Reading and writing in an array

We list the cellular codes of the two functions for reading and writing in a multi dimensional array. The function WRITE-INDEX needs three elementary pieces of code that recursively call themselves. The function READ-INDEX needs two.

WRITE-INDEX SITE 1 (CUTLE 3 (CUTO 1 (MULTSIT (SEQ (ADL 1 (SITE 0 (JMP CREAT-EQB)) (MRG -2 (SPLIT (JMP WRITE-INDEX-1)))) (SITE 0 (BLOC (PN- 2)))))))

CREAT-EQB TAB (VAL -1 (SSIGMO 7 (EQB-)))

WRITE-INDEX-1 BUN 3 (JMP WRITE-INDEX-2) (SEQ (SPLLIL 3 (SITE 0 (CUTRI 3 (ADL 2 (JMP CREAT-EQB) (SPLIT (JMP AND))))) (MRG -2 (SPLIT (JMP WRITE-INDEX-1)))) (SITE 0 (BLOC (PN- 2))))

WRITE-INDEX-2 BPN 2 (SITE 0 (JMP WRITE-INDEX-3)) (MRG 3 (MRG 2 (SEQ (SPLIT (JMP WRITE-INDEX-2)) (SITE 0 (BLOC (PN- 2)))))))

WRITE-INDEX-3 SEQ (PAR (CUT 3 (LR 1 (VAL -1 (JMP EAND)))) (CUT 2 (JMP EAND))) (JMP AOR)

READ-INDEX SITE 1 (CUTLE 3 (CUTO 1 (MULTSIT (SEQ (ADL 1 (SITE 0 (JMP CREAT-EQB)) (MRG -1 (SPLIT (JMP READ-INDEX-1)))) (BLOC (JMP READ-INDEX-2))))))

READ-INDEX-1 BUN 2 (JMP X-EAND) (SPLLIL 3 (SITE 0 (CUTRI 3 (ADL 2 (JMP CREAT-EQB) (SPLIT (JMP AND))))) (MRG -1 (SPLIT (JMP READ-INDEX-1))))

READ-INDEX-2 SPLIT (BPN 1 (JMP AOR) (SEQ (LR -1 (JMP X-MRG)) (SITE 0 (BLOC (PN- 2)))))

Recursive macro program symbol for the manipulation of the environment

These are elementary codes that are used in many places in the cellular code generated by the compiler. Theses code apply recursively the same operation to all the environment and the neural trees.

X-MRG BLR 0 (MG (ADDLR -1 (JMP X-MRG))) (JMP READ-INDEX-2)

X-EAND SPLIT (BPN 2 (JMP EAND) (MRG 2 (SEQ (JMP X-EAND) (SITE 0 (BLOC (PN- 2))))))

X-AOR SEQ (MRG 2 (MRG 1 (SPLIT (BPN 1 (JMP AOR) (JMP X-AOR))))) (SITE 0 (BLOC (PN- 2)))

X-DYN SPLIT (BPN 1 (EVER 1 (SDYN 2)) (SEQ (MRG 1 (JMP X-DYN)) (SITE 0 (BLOC (PN- 2)))))

X-SYNC IPAR 1 (VAL 0 (EVER 0)) (SITE 0 (BLOC (DELSITO (SPLITN (EVER 1)))))

REPEAND SPLIT (BPN 2 (JMP EAND) (MRG 2 (JMP REPEAND)))

WHILEND SPLIT (BPN 1 (END) (MRG 1 (JMP WHILEND)))

READ-1 BPN 3 (CUT 3 (LR 2 (VAL 0))) (SEQ (MRG 3 (SPLIT (JMP READ-1))) (SITE 0 (BLOC (PN- 2))))

WRITE-1 BPN 1 (END) (MRG 1 (SPLIT (JMP WRITE-1)))

Rules for rewriting nodes of the parse tree

There is one rules for each label of the PASCAL parse tree. The left member of each rule is a label of the PASCAL parse tree, the right member is a cellular code.

IF SITE 1 (SPLLIL 1 (CPFO 1 (SITE 0 (DELSIT (RINS)))) (SEQ (SITE 0 (SPLFI 2 (CUT 2 (PAR (WAIT 2 (SITE 1 (JMP X-EAND))) (LR 1 *NOT SIMPLIFIE* (VAL -1 (SITE 1 (JMP X-EAND))))))) (CUT 1 (CPFO 1 (LINS))))) (BLOC (CUTO 1 (JMP X-AOR)))))

THEN PAR (CUT 3 (BLOC (DELSIT (RINS)))) (CUT 2 (BLOC (DELSIT (LINS))))

REPEAT SITE 1 (CYC (SITE 0 (XSPL 1 (SITE 1 (JMP X-DYN)) (SPLFI 1 (CPFO 1 (BLOC (DELSIT (RINS))))) (BLOC (MRG 2 (SITE 1 (SPLLIL 2 (CPFO 1 (SITE 0 (DELSIT (LINS))))) (PAR (CUTO 1 (CUTO 1 (LR 1 (VAL -1 (SEQ (WAIT 2 (JMP REPEAND)) (JMP X-SYNC)))))) (CHOPO 2 (SEQ (JMP X-EAND) (SITE 0 (BLOC (END)))))))))))))))

WHILE SITE 1 (CYC (SITE 0 (XSPL 1 (SITE 1 (JMP X-DYN)) (BLOC (SITE 1 (SPLLIL 1 (CPFO 1 (SITE 0 (DELSIT (LINS)))) (PAR (CUTO 2 (SPLFI 2 (CUT 2 (JMP X-EAND)) (SITE 0 (CUT 1 (BLOC (SEQ (CPFO 1 (RINS)) (BLOC (SITE 1 (CUTO 1 (SEQ (MRG 1 (WAIT (JMP WHILEND))) (JMP X-SYNC)))))))))))))) (CHOPO 2 (LR 1 (VAL -1 (SEQ (CUT 2 (JMP X-EAND)) (SITE 0 (BLOC (END))))))))))))))

PROGRAM IPAR 1 (START-) (SEQ (CPFO 1 (SWITCH 1 (LINS))) (BLOC (MRG 1 (CUTO 1))))

NO-INST CUTO 1 (CUT 1)

DECL IPAR 2 (RINS) (PUTL 2 (BLOC (LINS)))

TYPE-ARRAY SEQ (CLONE ATTR (LINS)) (BLOC (EVER 2 (PN)))

TYPE-SIMPLE VAR (LR 1 (VAL 0 (EVER 2)))

SEMICOL SPLFI 1 (CPFO 1 (LINS)) (BLOC (MRG 2 (RINS)))

ASSIGN SPLFIL 1 (CPFO 1 (RINS)) (BLOC (LINS))

IDF-AFF SWITCH attr (CUTL 1 (CUT 1 (CUTO 1)))

IDF-LEC CUTO 1 (CHOP attr (BPN 1 (END) (MRG 1 (PN- 2))))

COMMA PAR (LINS) (RINS)

UNOP SEQ (CPFO 1 (LINS)) (CUTO 1 (JMP attr))

BINOP SEQ (CPFO 1 (PAR (LINS) (RINS))) (CUTO 1 (JMP ATTR))

INT-CST CHOP 2 (CUTO 1 (SBIAS ATTR (LR 1 (VAL 0 (EVER 2)))))

CALL-F SPLFI GLOBAL (CPFO 1 (RINS)) (BLOC (JMP ATTR (CUTO 1 (CHOP 2 (LR 1 (VAL 0)))))))

NO-PARAM KILL ()

FUNCTION ou RETURN LINS ()

CALL-P SPLLI LOCAL (CPFO 1 (SPLFI GLOBAL (CPFO 1 (RINS)) (BLOC (JMP ATTR (END))))) (BLOC (MRG 1 (CUTO 1)))

PROCEDURE SEQ (CPFO 1 (LINS)) (BLOC (MRG 1 (CUTRI GLOBAL (CUTO 1)))))

READ SPLFIL 1 (SPLFI 2 (CHOP ATTR) (MRG 3 (SITE 1 (MULTSIT (JMP READ-1))))) (BLOC (RINS))

WRITE PAR (CHOPO 1 (SEQ (CPFO 1 (RINS)) (BLOC (JMP WRITE-1])))) (CUT 1 (CUTO 1))

#IF SEQ (CPFO 1 (PAR (CUTO 1) (RINS))) (BLOC (MRG 2 (LINS)))

#THEN BUN ATTR (CHOP 1 (MRG 1 (RINS))) (CHOP 1 (MRG 1 (LINS)))

CALLGEN SPLFI 2 (CPFO 1 (RINS)) (BLOC (JMP ATTR (CUTO 1 (CHOP 2 (LR 1 (VAL 0)))))))

Library of functions for handling arrays.

These functions enrich the array data structure, they are used to implement the automatic parallelization of divide and conquer algorithms.

left CUTLE 3 (CUTO 1 (MRG 1 (CHHR 1 (PN- 2))))
right CUTLE 3 (CUTO 1 (MRG 1 (CHHL 1 (PN- 2))))
concat CUTLE 3 (CUTO 1 (MRG 2 (MRG 1 (PN- 2))))
int-to-array CUTLE 3 (CUTO 1 (PN- 2))
array-to-int CUTLE 3 (CUTO 1 (MRG 1 (BPN 1 (END) (MRG 1 (PN- 2)))))
randomize CUTLE 3 (CUTO 1 (JMP rand))
rand BPN 1 (LR 1 (VAL 0 (RAND))) (SEQ (MRG 1 (SPLIT (JMP rand))) (PN- 2))

Library of layered neural networks

The following cellular code correspond to neural networks that are included in the PASCAL program animal.p

retina CUTLE 3 (CUTO 1 (SEQ (MRG 1 (SPLIT)) (SEQ (CLO 10 (EVER 0)) (BLOC (PN- 2)))))
position-object CUTLE 3 (CUTO 1 (SEQ (SEQ (MRG 1 (CLO 2 (EVER 0))) (CLO 2 (EVER 0))) (EVER 0)))
predator CUTLE 3 (CUTO 1 (SEQ (SEQ (MRG 1 (CLO 3 (EVER 0))) (CLO 3 (EVER 0))) (EVER 0)))
motor CUTLE 3 (CUTO 1 (SEQ (EVER 0) (SEQ (CLO 2 (EVER 0)) (SEQ (CLO 4 (EVER 0)) (PN- 2)))))

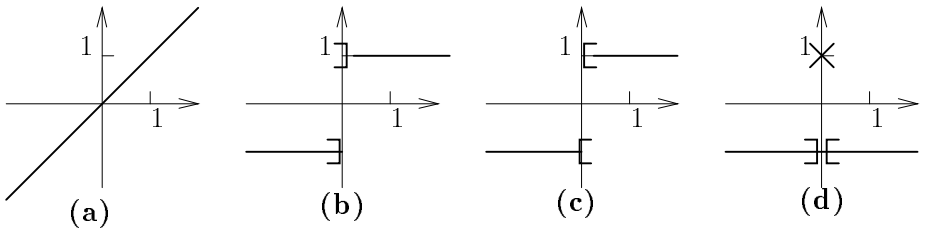


Figure 1: Different kind of sigmoïds used. (a) adds two numbers. (b) and (c) compare two numbers, (d) tests the equality of two numbers.

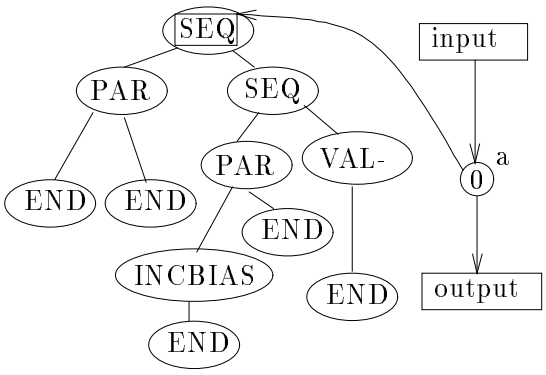


Figure 2: Step 0: Development of the neural net for the exclusive-OR (XOR) function in the right half of this figure. The development starts with a single ancestor cell labeled "a" and shown as a circle. The 0 inside the circle indicates the threshold of the ancestor cell is 0. The ancestor cell is connected to the neural net's input pointer cell (box labeled "input") and the neural net's output pointer cell (box labeled "output"). The cellular encoding of the neural net for XOR is shown on the left half of this figure. The arrow between the ancestor cell and the symbol `SEQ` of the graph grammar represents the position of the reading head of the ancestor cell.

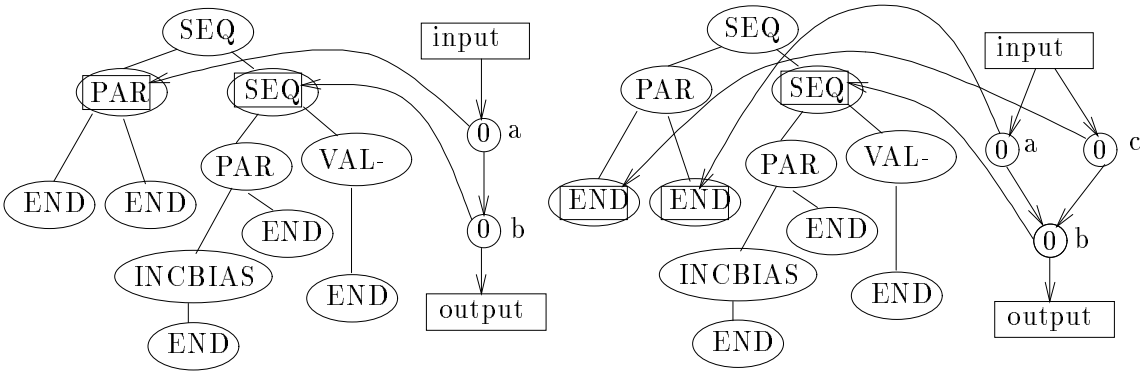


Figure 3: On the left: the execution at step 1 of the sequential division `SEQ` pointed to by the cell "a" of preceding figure causes the ancestor cell "a" to divide into two cells "a" and "b". Cell "a" feeds into cell "b" with weight +1. The reading head of cell "a" now points to the left sub-tree of the Cellular Encoding on the left (the box with `PAR`) and the reading head of new cell "b" points to the right subtree (the box at the second level down with `SEQ`). On the right: The execution of the parallel division `PAR` at step 2 causes the creation of cell "c". Both cell "a" and "c" inherit the input formerly feeding into "a". Both cells "a" and "c" output to cell "b" (The place where "a" formerly sent its output). The reading head of cell "a" now points to the left sub-tree (`END`) and the reading head of the new cell "c" points to the right sub-tree (which also has an `END`).

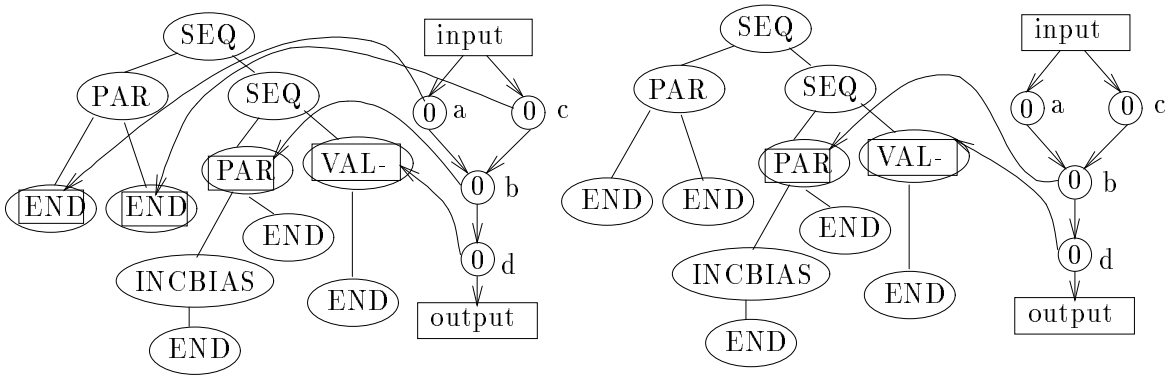


Figure 4: On the left: the execution at step 3 of the sequential division `SEQ` pointed to by the cell "b" of preceding figure causes the cell "b" to divide into two cells "b" and "d". Cell "b" feeds into cell "d" with weight +1. The reading head of cell "b" now points to the left sub-tree (the box at the third level with `PAR`) and the reading head of new cell "d" points to the right subtree (the box at the third level down with `VAL-`). On the right: The execution of the two end-program symbols. The `END`'s causes the cells "a" and "c" to loose their reading head and become finished neurons. Since there are two `END`'s, it takes two time steps, one time step for each `END`

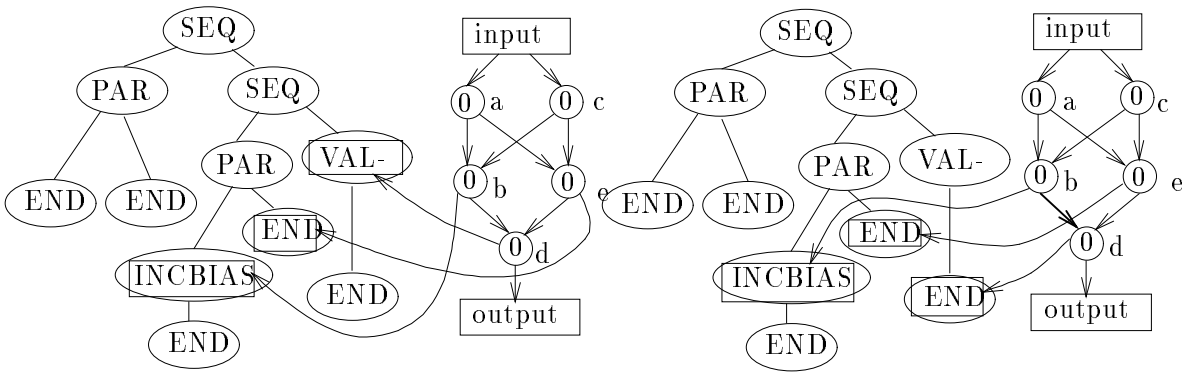


Figure 5: On the left: the execution of the parallel division "`PAR`" at step 6 causes the creation of cell "e". Both cell "b" and "e" inherit the input from cell "a" and "c" formerly feeding into "b". Both cells "b" and "e" send their output to cell "d" (The place where "b" formerly sent its output.) The reading head of cell "b" now points to the left sub-tree (`INCBIAS`) and the reading head of the new cell "e" now points to the right sub-tree (which has an "`END`"). On the right: The cell "d" executes the value-program symbol `VAL-`. The link register is one (the default value) it points the left-most link. The action of `VAL-` is to set the weight of the first link to -1. The heavy line is used to indicate a -1 weight.

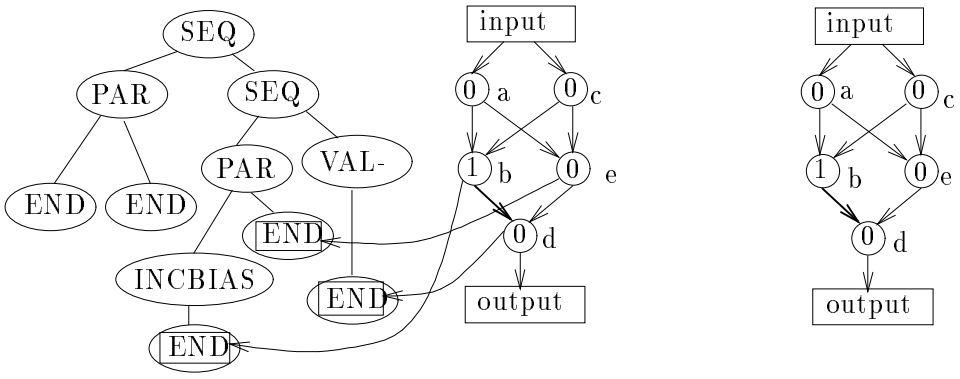


Figure 6: On the left: Neuron "b" executes the value-program symbol `INCBIAS`. The action of `INCBIAS` is to increase the threshold of cell "b" by 1. After execution, the threshold of cell "b" is 1, and the reading head of cell "b" points to an `END` at the fifth level of the tree. On the right: The last tree steps consist in executing three `END`-program symbols. The three `END`'s cause the cells "b", "e" and "d" to lose their reading head and become finished neurons. Since there are now only finished neurons, the development is finished. The final neural net has two input units "a" and "c", and one output unit "d". The neuron "c" is the second input unit, because the link from the input pointer cell to "c" is the second output link of the input pointer cell

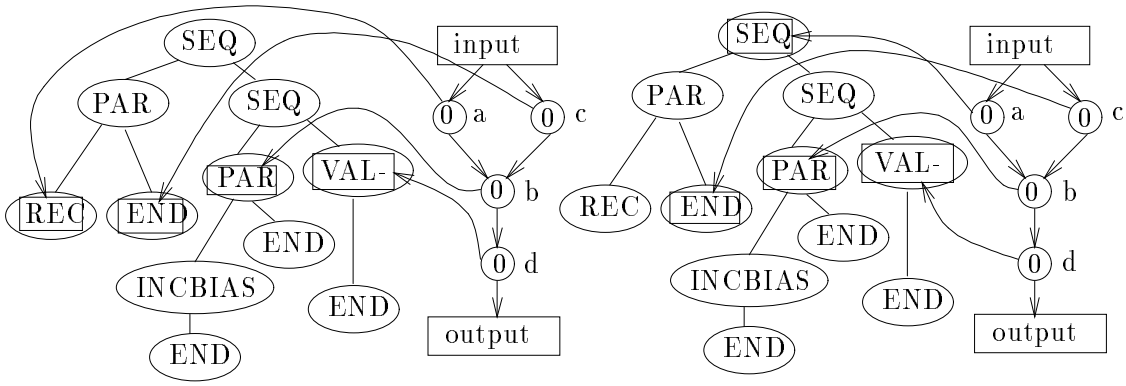


Figure 7: A recurrent developmental program for a neural network that computes the even parity for three binary inputs. The life of the ancestor cell is 2. The first three steps are the same as those shown in the preceding figures. During the fourth step, cell "a" executes a recurrent-program symbol. Its reading head is backtracked the label (`SEQ`) on the root of the grammar-tree. The situation of cell "a" is similar to the beginning of the development at step 0. The life of "a" has been decremented and is now equal to 1. Cell "a" will give birth to a second XOR network, whose outputs will be sent to the child cells generated by cell "b".

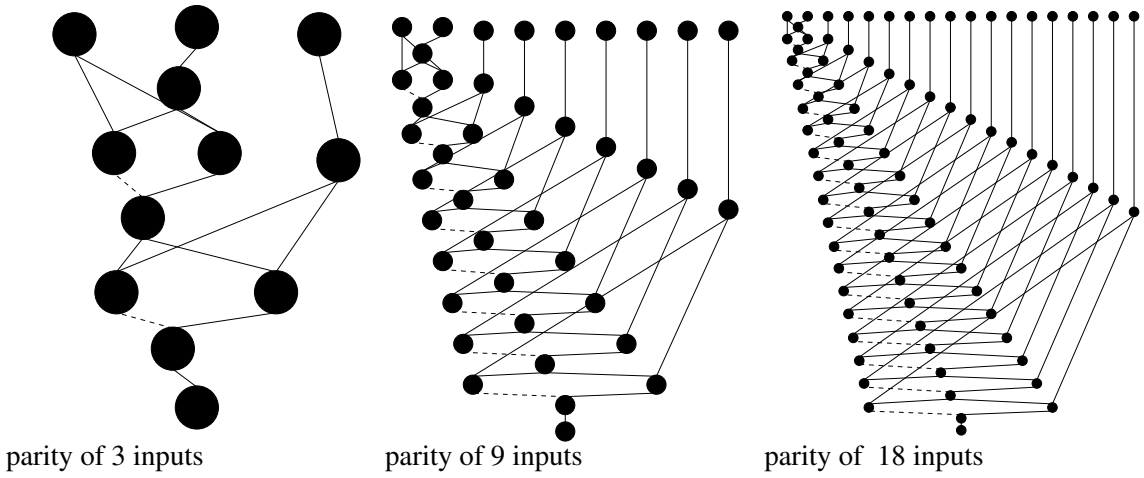


Figure 8: The final decoded neural network that computes the even parity of 3, 9 and 18 binary inputs. The initial life is respectively 2,8 and 17. A weight -1 is represented by a dashed line. With an initial life of L we develop a neural network for the $L+1$ input parity problem, with L copies of the XOR network.

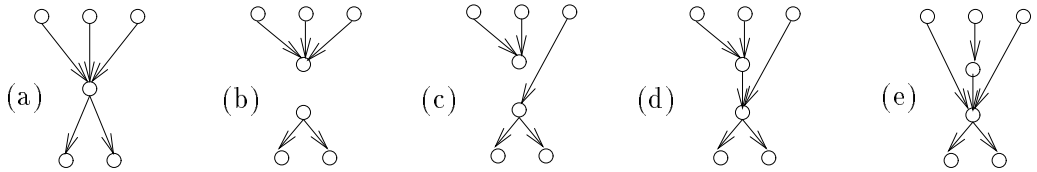


Figure 9: Microcoding of the program-symbol ADL 2. (Add on Link), 2 is an argument. This program symbol adds a cell on the link number 2. The microcode is DIVm<sm> 2 (a) before the division ADL, (b) division in two child cells. (c) the segment "m>" is analyzed,. (d) the segment "s" is analyzed (e) the segment "m<" is analyzed.

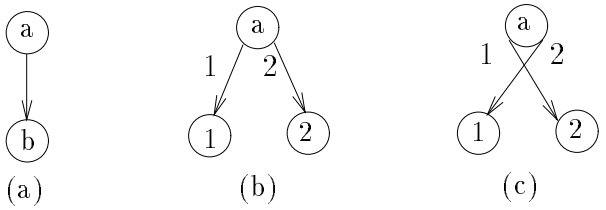


Figure 10: The difference between segement's microcode with "d" and with "r". (a) before the cell division; the cell which divides is labeled "b", it has an input neighbor labeled "a". (b) the input link is duplicated using a "d" microcoding. The first child is the first output neighbor of cell "a" (c) the input link is duplicated using a "r" microcoding. The first child is the second output neighbor of cell "a".

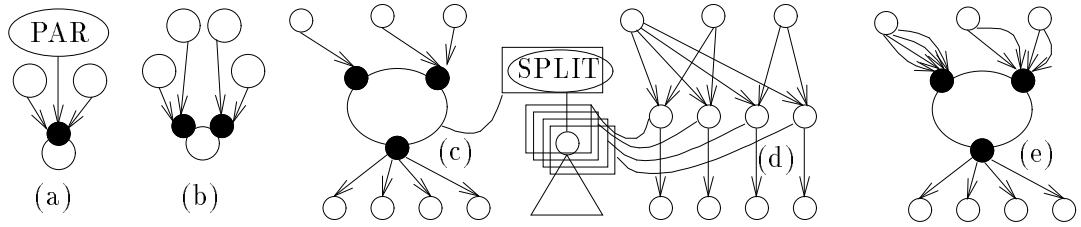


Figure 11: Use of sub-sites. On the left: Splitting of a site into 2 sub-sites. A sub-site is indicated by a small black disk. (a) Before the parallel division of the cell labeled `PAR` (b) After the parallel division, a sub-site has been created by the neighbor cell. On the left: the `SPLIT` program symbol. (c) Before the execution of program symbol `SPLIT`. (d) Result of the splitting division. The four child cells read the same node. (e) The intermediate stage.

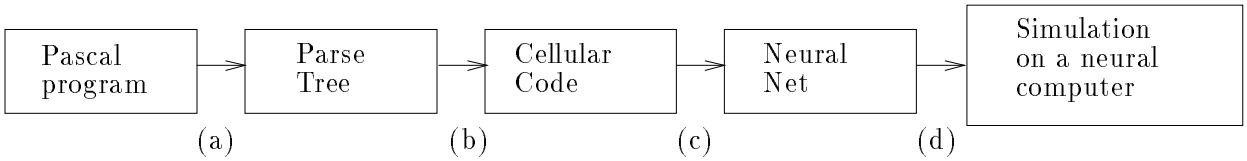


Figure 12: Stages of the neural compilation. (a) Parsing of the PASCAL Program. (b) rewriting of the parse tree into a cellular code, using a tree grammar. (c) decoding of the cellular code into a neural network. (d) scheduling and mapping of the neural network on a physical machine. This stage is not yet done.

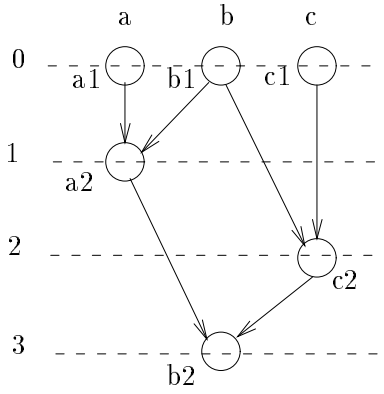


Figure 13: Result of the compilation of the PASCAL program: *var a,b,c: integer; begin a=a+b; c=c+b; b=a+c; end.* Each dotted line corresponds to the compilation of one instruction. The neurons make a simple addition of their inputs (the weights are 1, the sigmoïd is the identity). The line 0 represents the initial environment. It encompasses three values "a", "b" and "c" stored in three neurons a_1, b_1 and c_1. At line 1, the value of "a" has changed, it is now contained in a new neuron a_2, that computes the sum of "a" and "b". In order to make this sum, neuron a_2 is connected to neuron a_1 and b_1. The value of "c" changes at the next line, finally, the value of "b" also changes. One can see that the two first instructions: `a:= a+b; c:=c+b` can be executed in parallel by neurons a_2 and c_2. On this example, each variable is represented by two neurons during the run of the program.

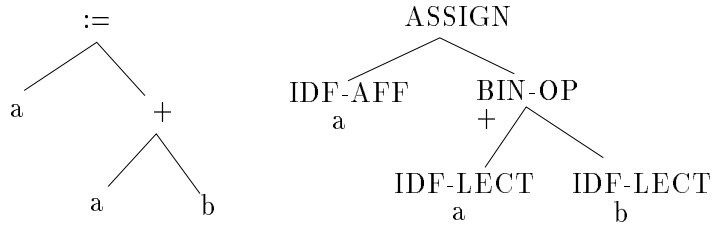


Figure 14: Parse tree of the instruction "*a:=a+b*". On the link side: the parse tree is labeled with words of the PASCAL program. We will use another format described on the right. The labels are made of two parts: the first part specifies the kind of the node, the second part is an attribute. For example,`IDF-AFF` indicates that the node contains the name of a variable that is modified, `IDF-LEC`: the name of a variable which is read, `BIN-OP` the name of a binary operator like the addition, `ASSIGN` corresponds to an assignment. For nodes `IDF-AFF` and `IDF-LEC`, the attribute contains the name of the variable to modify or to read. For `BIN-OP`, the attribute contains the name of the particular binary operator.

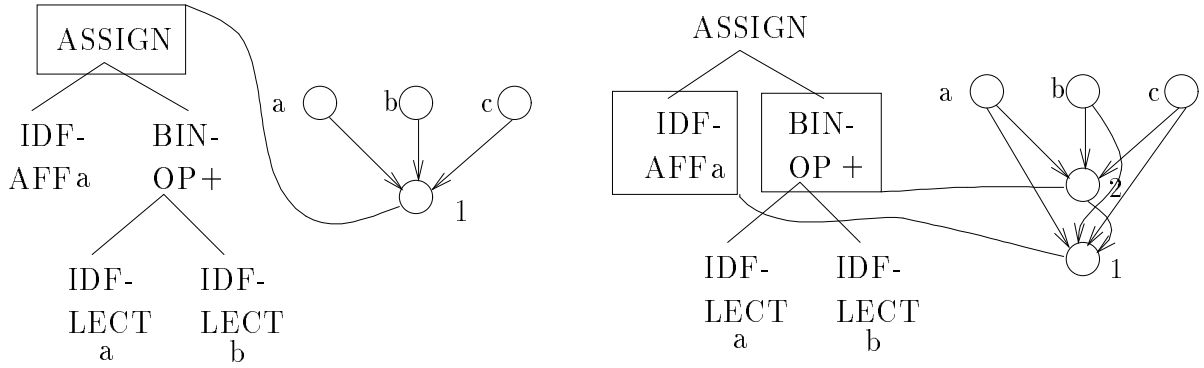


Figure 15: On the left: Development of the neural network translating the PASCAL instruction "`a:=a+b`". The development begins with an ancestor cell labeled "1" connected to three neurons labeled "a", "b" and "c" which store the state of the environment before the instruction is executed. On the left side, we represent the parse tree of the PASCAL instruction. The arrow between the ancestor cell and the symbol `ASSIGN` of the parse tree represents the reading head of the ancestor cell. On the right: execution at step 1 of the macro program symbol `ASSIGN`, read by the ancestor cell "1". Cell "1" divides into two child cells "1" and "2". Both child cells inherit the input links of the mother cell. Child "2" is connected to child "1". The reading head of cell "1" now points to the node `IDF-AFF` and the reading head of cell "2" points to the node `BIN-OP`

.

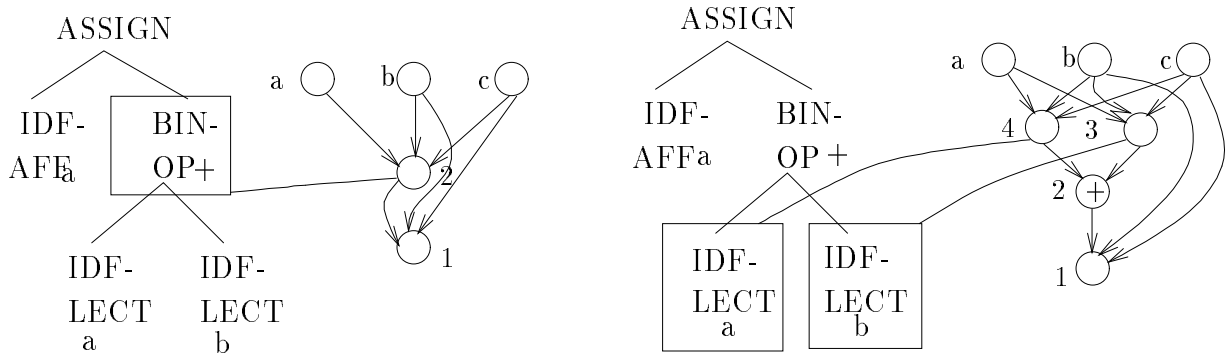


Figure 16: On the left: execution at step 2 of the macro program symbol `IDF-AFF`, read by the cell "1". Cell "1" deletes its first link, and moves its fourth link back in the first position. On the right: execution at step 3 of the macro program symbol `BINOP` read by cell "2". Cell "2" divides into three cells "2", "3" and "4". Cells "3" and "4" inherit the input links of their mother cell, and are connected on the output to cell "2". Whereas cell "2" inherits the output links of the mother cell, looses its reading head and becomes a neuron. The sign + inside the circle indicates that it makes the addition of its inputs.

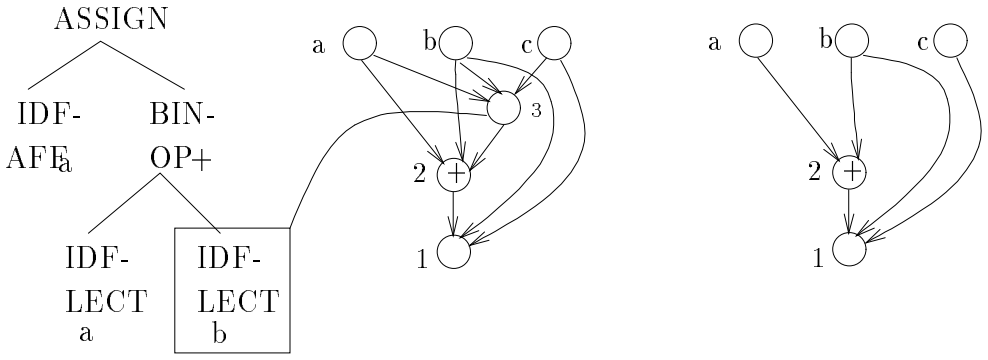


Figure 17: (a) Step 4: On the left: execution at step 4 of the macro program symbol `IDF-LEC`, read by cell "4". The cell "4" disappears, after having connected its first input neighbor to its output neighbor. On the right: execution at step 5 of the macro program symbol `IDF-LEC`, read by cell "3". The cell "3" disappears, after having connected its second input neighbor to its output neighbor. There are no more reading cells. The compilation of the PASCAL instruction "`a:=a+b`" is finished. The ancestor cell "1" is connected to the three neurons that contain the values of the modified environment. It can start to develop the next instruction.

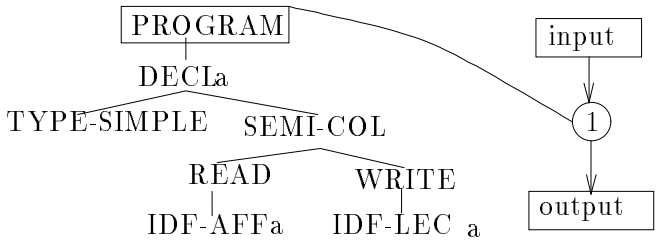
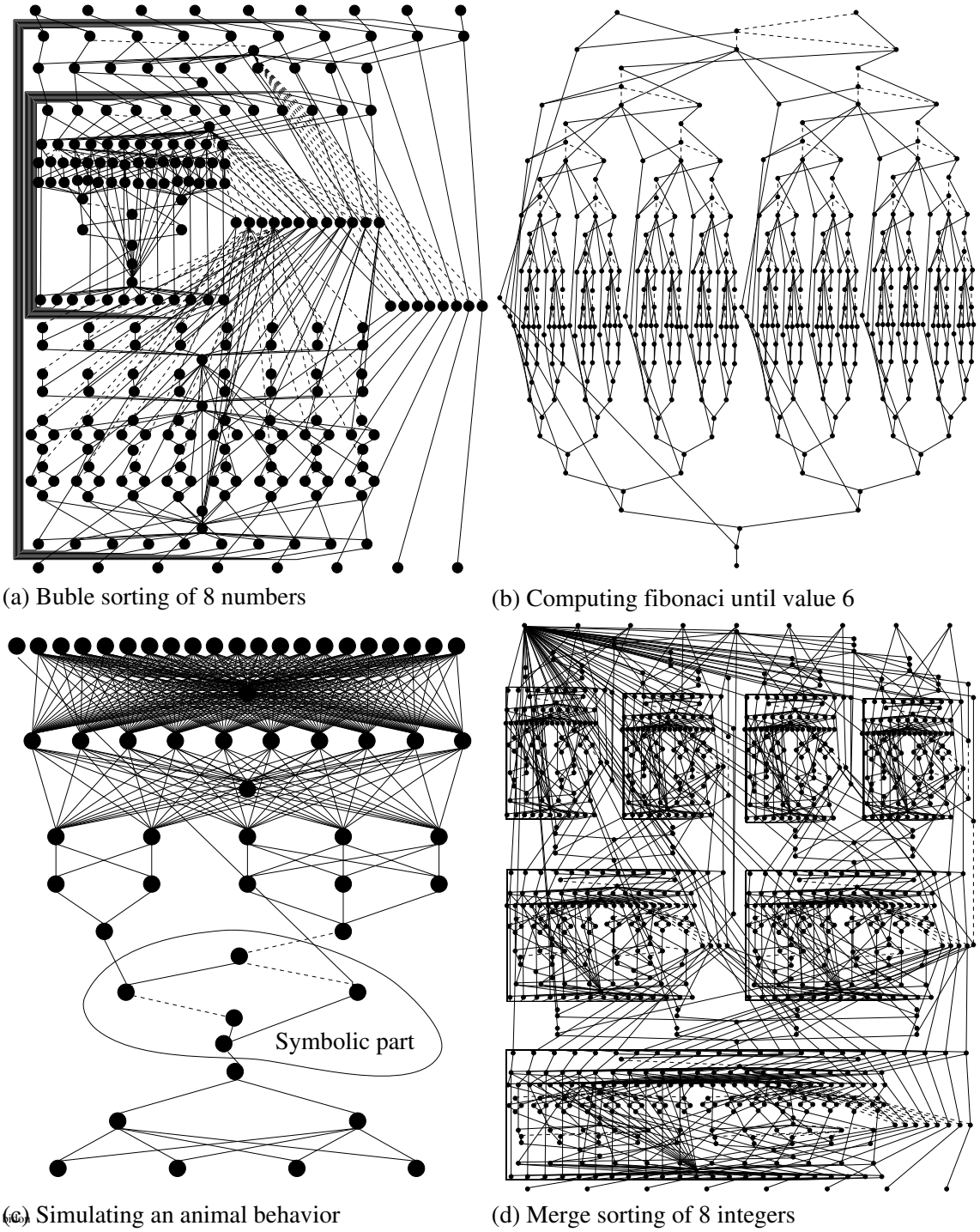


Figure 18: On the left: The parse tree of the PASCAL Program: "`program p; var a : integer; begin read(a); write (a); end.`" This parse tree does not have the usual form. It contains nodes for the declaration of variables, and the labels are not organized in a standard way. This particular form helps the implementation of the neural compiler. The appendix 3 contains a BNF grammar that defines these parse trees. On the right: The development begins with an ancestor cell labeled "1" and represented as a circle. The ancestor cell is connected to an input pointer cell (box labeled "input") and an output pointer cell (box labeled "output"). The arrow between the ancestor cell and the parse tree represents the reading head of the ancestor cell. The development is reported in appendix 1.



(a) Buble sorting of 8 numbers

(b) Computing fibonaci until value 6

(c) Simulating an animal behavior

(d) Merge sorting of 8 integers

Figure 19: Examples of compiled neural networks

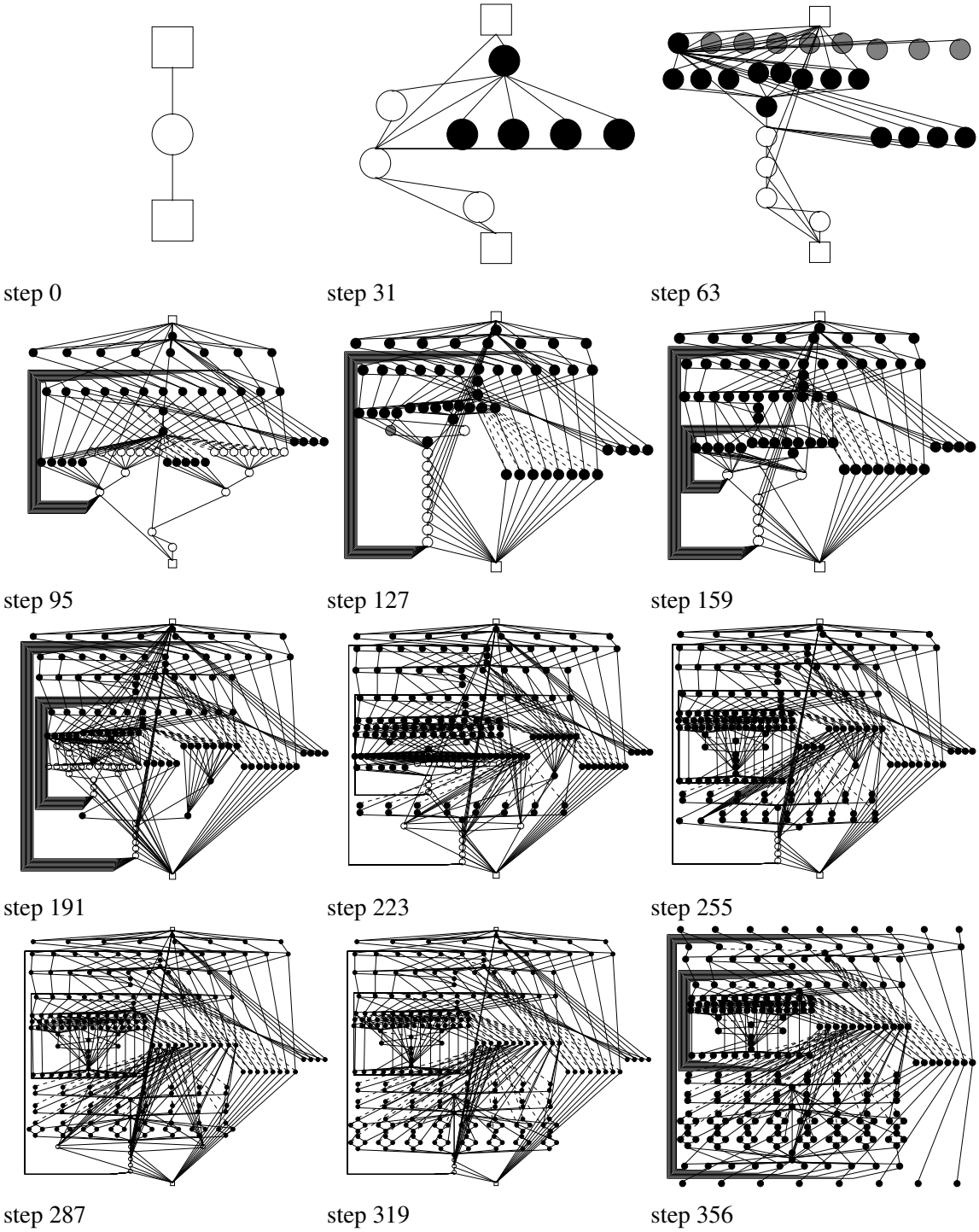


Figure 20: Steps of the development of the neural network for the bubble sorting of 8 integers.

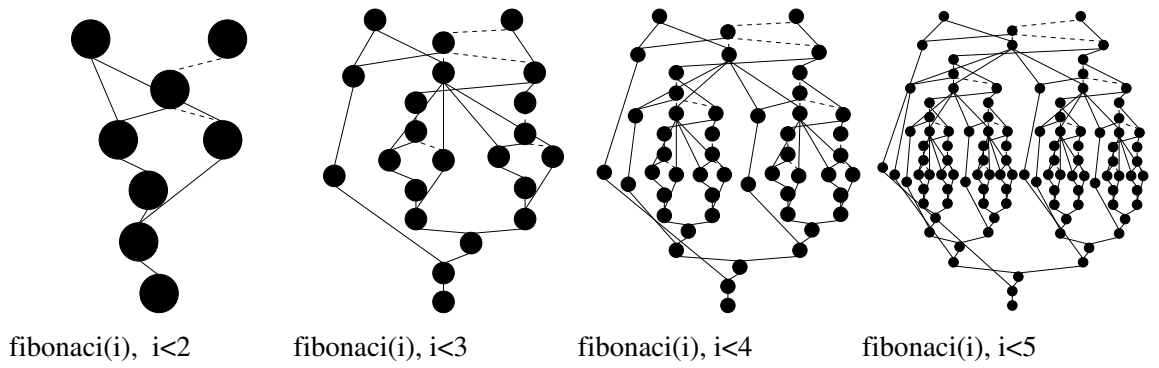
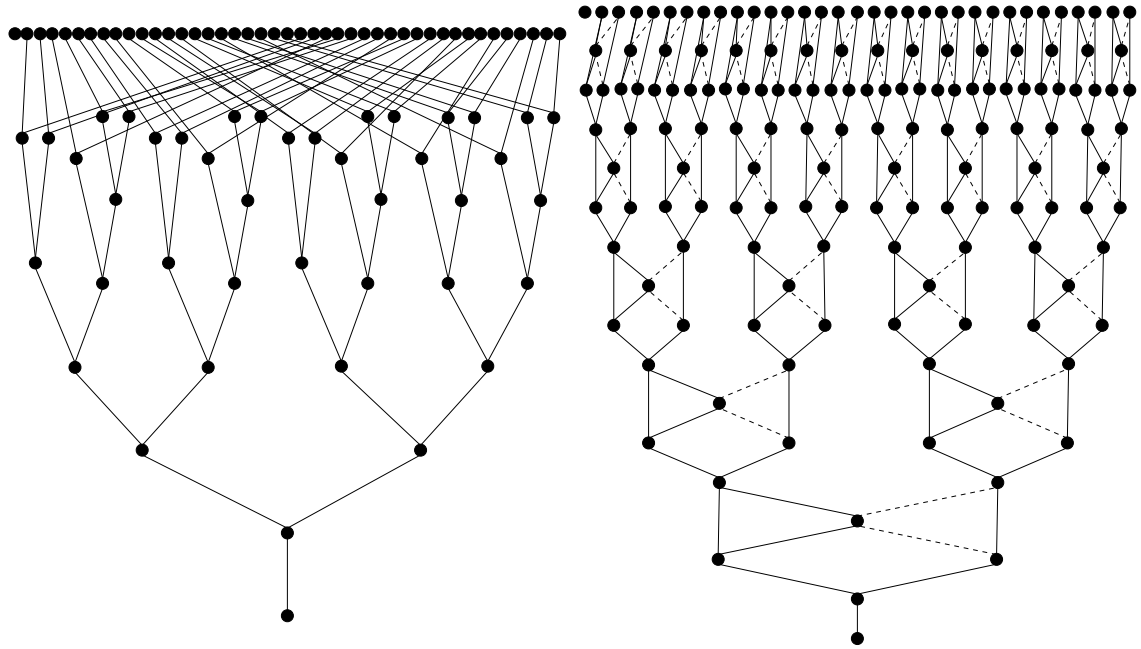
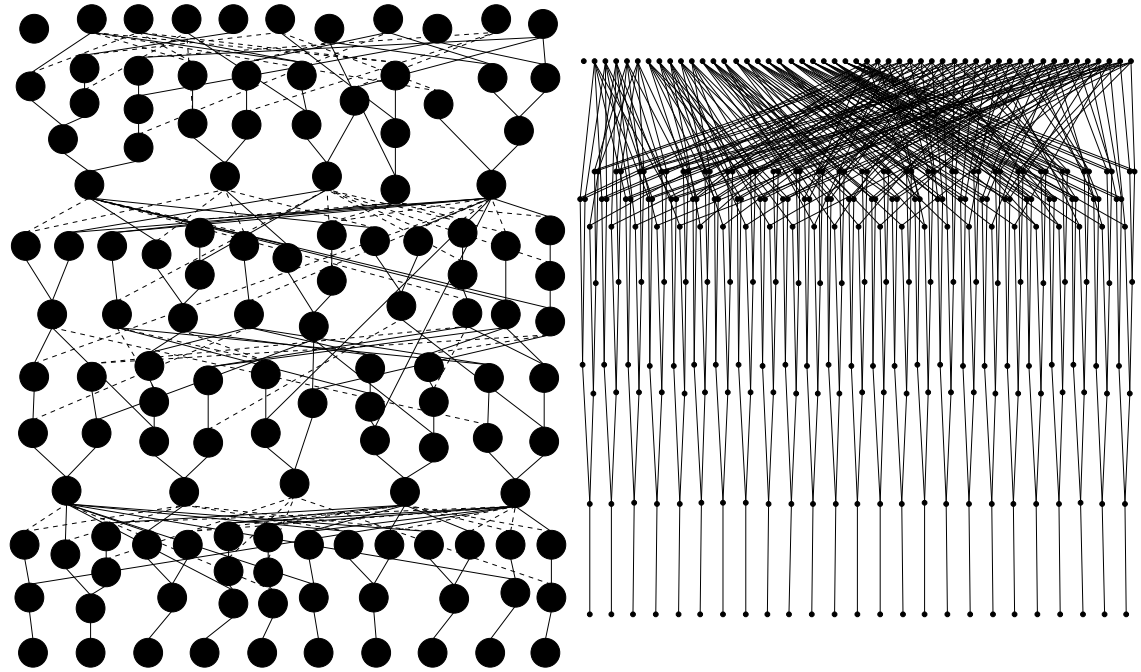


Figure 21: The neural network for the Fibonacci function developed with an initial life respectively L=1, 2, 3 and 4. The corresponding networks are self-similar. The networks can compute Fibonaci(i), i¡L+2.



(a) Multiplication Vector Vector

(b) Maximum of 32 integers



(c) Multiplication Matrix Vector

(d) Multiplication Matrix Matrix

Figure 22: Other examples of compiled neural networks that can do computation in parallel.

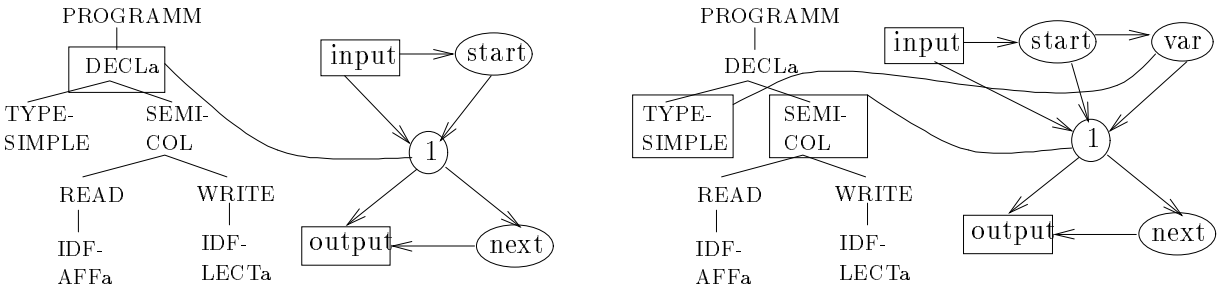


Figure 23: On the left: execution at step 1 of the macro program symbol `PROGRAM`. The ancestor cell "1" gives birth to two other cells. A cell labeled "start" and another one labeled "next". The "start" cell controls the propagation of the activities. The "next" cell has not finished its development. It waits all its neighbors to become finished neurons. The cell "1" now reads the `DECL` node. On the left: execution at step 2 of the macro program symbol `DECL`. The ancestor cell gives birth to another cell labeled "var". This cell represents the variable "a". The input and output connections of the ancestor cell are now complete and represent a topological invariant. The first input (resp. output) link points to the input (resp. output) pointer cell. The second input link points to the "start" cell, the rest of the input links point to "var" neurons. The second output link points to the "next" cell.

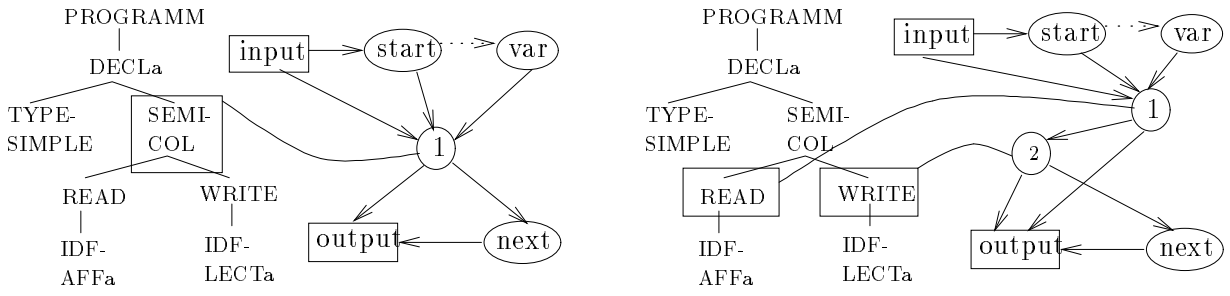


Figure 24: On the left: execution at step 3 of the macro program symbol `TYPE-SIMPLE`. The effect is to nullify the weight of the connection to the neuron that represents variable "a". A weight 0 is represented by a dotted line. On the right: execution at step 4 of the macro program symbol `SEMI-COL`. This macro program symbol composes two instructions. Cell "1" gives birth to another cell labeled 2. Cell "1" goes to read the macro program symbol that corresponds to the instruction `read a`. Cell "2" is blocked on the instruction `write a`

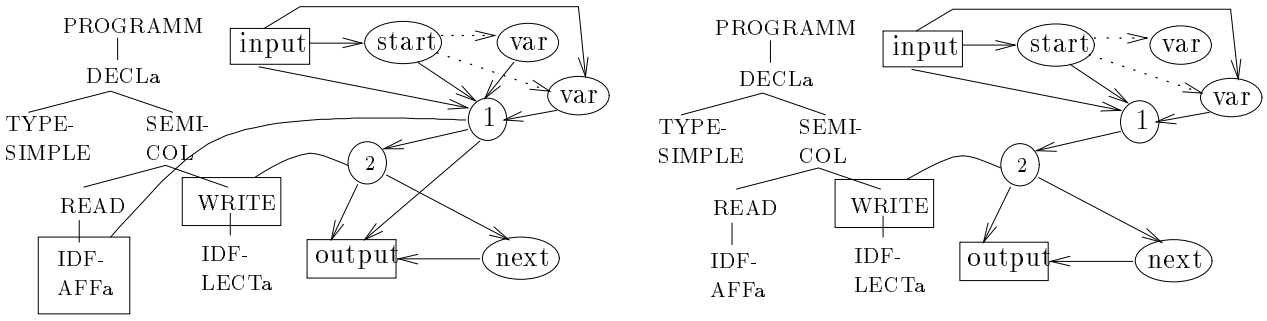


Figure 25: On the left: execution at step 5 of the macro program symbol `READ a`. A neuron labeled `var` is created. It contains the new value of variable "a". This neuron is connected to the input pointer cell. On the right: execution at step 6 of the macro program symbol `IDF-AFF`. The neuron "var" that contains the old value of "a" will be deleted because it has no more output links. The cell "2" is unblocked, and carries on its development.

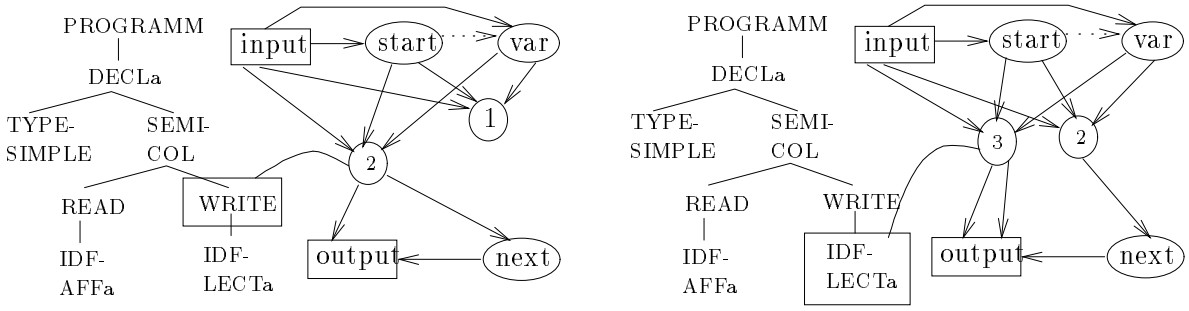


Figure 26: On the left: unblocking at step 7 of the cell "2". Before to start the compilation of the `WRITE`, cell "2" executes a small piece of cellular code. It merges the input links of cell 1. The cell "1" will be deleted. Now, the invariant is restored. Cell "2" can start the compilation of the next instruction. On the right: execution at step 8 of the macro program symbol `WRITE`. The cell "2" gives birth to another cell "3", connected two times to the output pointer cell. The cell "3" possesses all the input connections of cell "2". Cell "2" becomes a neuron.

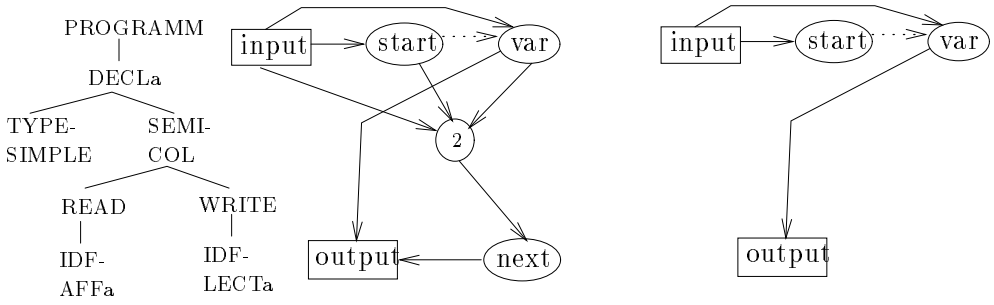


Figure 27: On the left: execution at step 9 of the macro program symbol `IDF-LECT`. Cell "3" selects the input that corresponds to variable "a", and connect that input to the output pointer cell. Then, cell "3" disappears. On the right: unblocking at step 10 of the cell "next" which is deleted. Its deletion leads to the suppression of neuron "2". The final neural net encompass two input neurons: the start cell and the input variable "a"; and one output neuron: the variable "a". This neural net reads "a" and write "a". It translates what is specified by the PASCAL program.

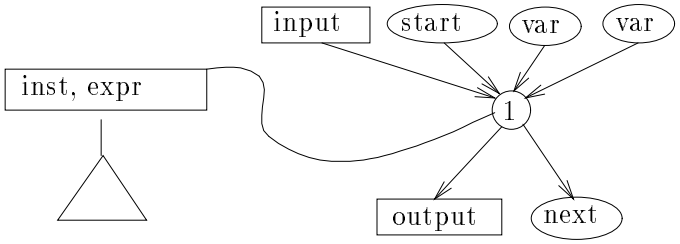


Figure 28: Invariant pattern. We show here the network graph of cells after the declaration of two scalar variables. We can see the two corresponding neurons labeled "var". These neurons "var" with the neuron "start" will constitute the first layer of the final neural network. The graph of this figure is an invariant pattern that will be taken for the initial configuration, for all the next macro program symbols. This avoids repetition and stresses the importance of the invariant pattern in the design of the macro program symbol. Cell "1" will be called the ancestor cell.

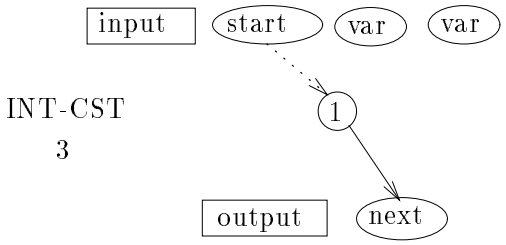


Figure 29: Execution of the macro program symbol `INT-CST` by the ancestor cell of the invariant pattern. This macro program symbol has one parameter which is the value of the constant. The macro program symbol `INT-CST 3` has the following effect: It sets the bias of the ancestor cell to the value 3, and deletes all the links, except the link to the "start" neuron and the link to the "next" neuron. The sigmoïd of the ancestor cell is set to the identity, and the ancestor cell becomes a neuron labeled 1. When the constant is needed, the neuron "start" is used to activate the neuron 1, which delivers the constant coded in its bias.

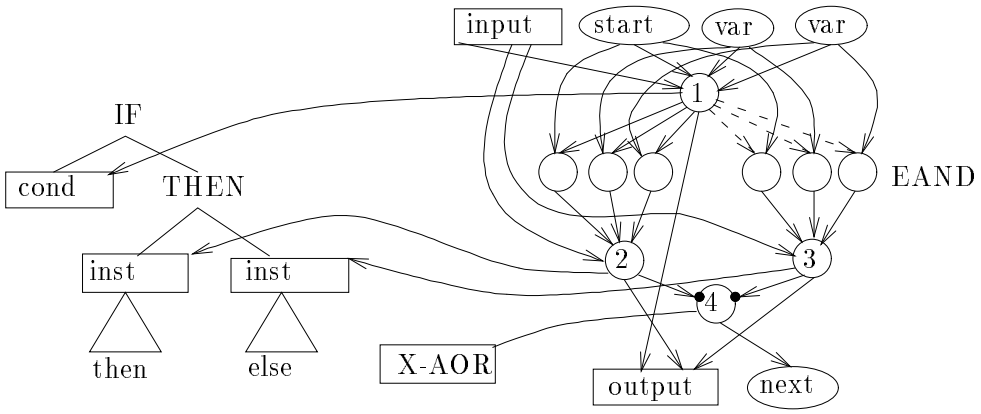


Figure 30: Execution of the macro program symbols `IF` and `THEN` by the ancestor cell of the invariant pattern. For the sake of clarity, we have represented the combined action of `IF` and `THEN`. Four reading cells, and a layer of 6 neurons `EAND` are created. Cell "1" will develop a sub-neural net that computes the value of the condition. `TRUE` is coded on value 1, and `FALSE` is coded on value -1. The logical operation `NOT` is realized using a weight -1 (dashed line). Depending on the value of the condition, the flow of values contained in the environment is sent to the left or to the right. The reading cell "2" and 3 develop respectively the body of the "then" and the body of the "else". Cell "4" possess two input sub-sites which are represented as small black disks. It is blocked on a piece of cellular code called `X-AOR`.

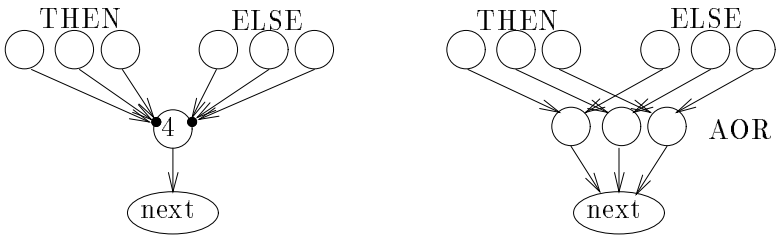


Figure 31: Execution of the cellular code `X-AOR` by cell "4" of the preceding figure. The cell "4" has two input sub-sites that distribute its input links into two lists. The cell "4" produces a layer of `AOR` neurons. This layer retrieves either the output environment from the body of the "then" or the output environment from the body of the "else"; and transmits it to the next instruction.

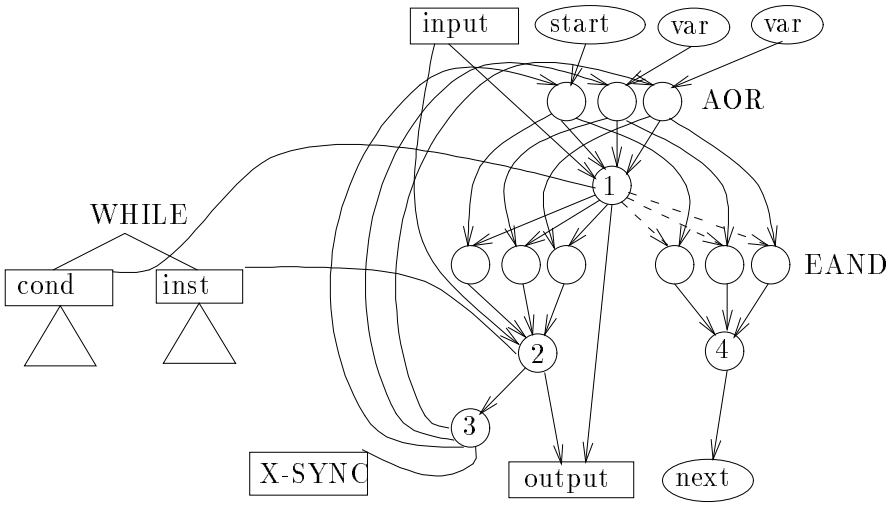


Figure 32: Execution of the macro program symbol `WHILE` by the ancestor cell of the invariant pattern. Many cells are created: three reading cells, 3 `AOR` neurons, 6 `EAND` neurons and one neuron labeled "4", used to forward the environment. The layer of `AOR` neurons merges the flow of values that comes either from the top of the network (first iteration) or from the bottom (other iterations). The cell "1" develops a neural net that computes the condition of the while. As in the case of the `IF`, the `EAND` neurons are used as a switching for the flow of activities. If the condition is true, the flow of values is forwarded in the body of the loop, else it turns to the right and exits the loop. The cell "2" develops the body of the `WHILE`. The cell "3" develops an intermediate layer in the loop, which will synchronize the flow of activities.

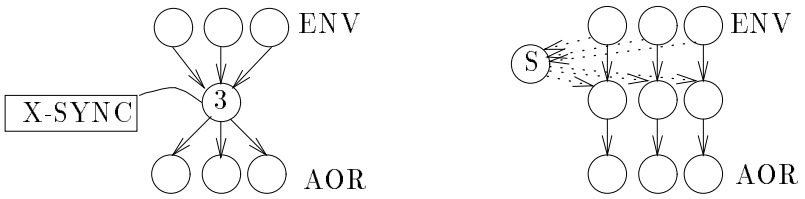


Figure 33: Execution of the cellular code `X-SYNC` by cell "3" of the preceding figure. Cell "3" gives birth to a layer l of three neurons plus one neuron "S" whose dynamic is the normal dynamic. This layer is inserted between the layer of neuron "ENV" that stores the output environment of the body of the `WHILE`, and the layer of `AOR` at the beginning of the loop. The neuron "S" is connected from its input site, to all the neurons in the "ENV" layer. The neuron "S" is active only when all the environment is available. The neuron "S" is linked to the three neurons of layer l from its output site, with weights zero (dotted line). These connections block the neurons while "S" is not active. So the three neurons of layer l are unblocked at the same time, and the flow of values is synchronized.

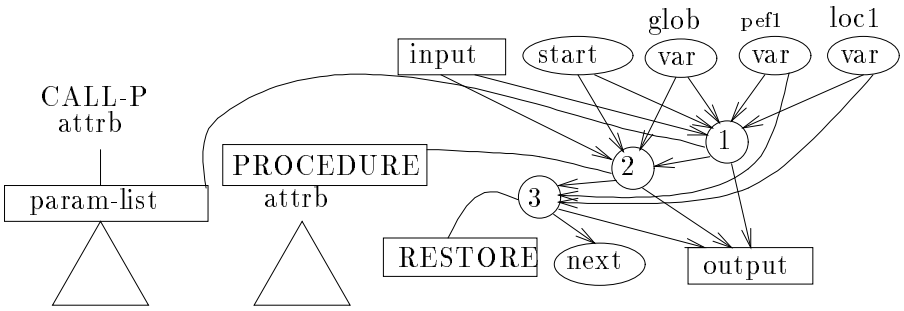


Figure 34: Execution of the macro program symbol `CALL-P` by the ancestor cell of the invariant pattern. The environment has three variables. The `CALL-P` has one parameter (`attrb`) which is the name of the called procedure. Three cells are created. Cell "1" develops different subnetworks: one subnetwork for each parameter to pass to the called procedure. The cell "2" develops the body of the called procedure. Its reading head is placed on the root of the parse tree that defines the called procedure. The cell "2" is not connected to the neurons that contain the local variables of the calling procedure: `pef1` and `loc1`. These variables cannot be accessed from `proc2`. Cell "3" is blocked on a cellular code `RESTORE`. It will restore the environment of the calling procedure, when the translation of the called procedure is finished.

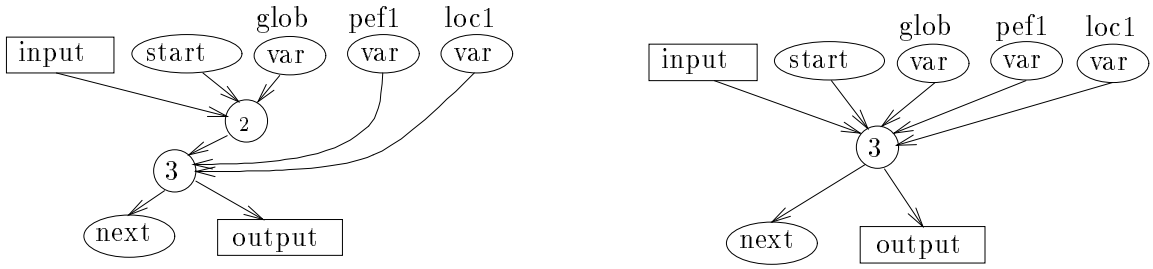


Figure 35: Execution of the cellular code `RESTORE` by cell "3" of the preceding figure. Cell "3" is unblocked when the body of `proc2` is developed. Cell "3" is linked to cell "2" which is linked to the global variables, that may have been modified by the called procedure. The execution of `RESTORE` brings together the modified global variables, and the unchanged local variables of `proc1`.

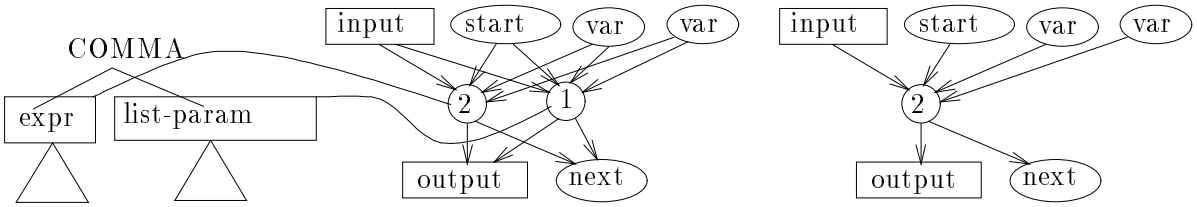


Figure 36: On the left: execution of the macro program symbol `COMMA` by the ancestor cell of the invariant pattern. This macro program symbol creates two cells. Cell "2" develops a subnetwork for computing the value of a parameter, cell "1" continues to read the parameter list. On the right: execution of the macro program symbol `NO-PARAM` by cell "1", assuming that the end of the parameter list is reached. The end of the list is marked by a `NO-PARAM` that simply eliminates cell "1".

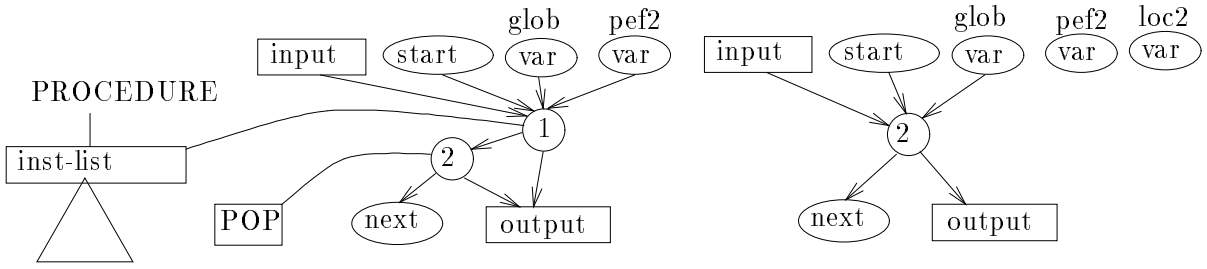


Figure 37: On the left: execution of the macro program symbol `PROCEDURE` by the ancestor cell of the invariant pattern. This macro program symbol adds an intermediate cell "2", blocked on a `POP` cellular code. On the right: when the translation of the procedure is finished, cell "2" is unblocked. and execute the cellular code `POP`. `POP` retrieves only the global variables. The local variables are not connected to neuron "2" after the execution of `POP`. This macro program symbol is equivalent to the pop instruction of a machine language, used for the return of subroutines.

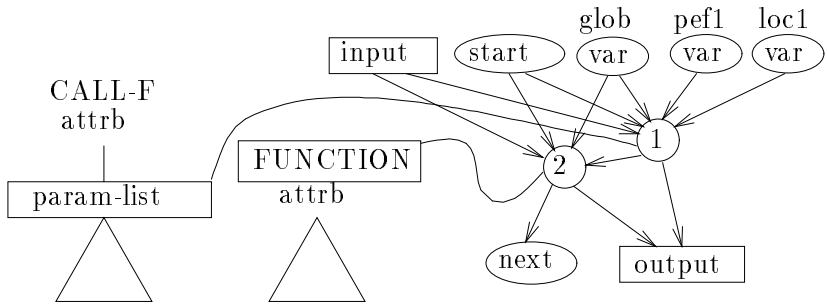


Figure 38: Execution of the macro program symbol `CALL-F` by the ancestor cell of the invariant pattern. This macro program symbol has the same effect as `CALL-P` except that it does not create the cell 3, because it is not necessary to restore the environment.

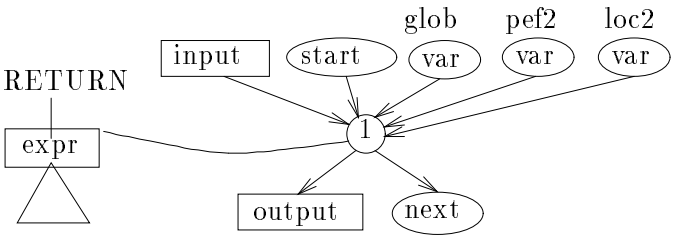


Figure 39: Execution of the macro program symbol `RETURN` by the ancestor cell of the invariant pattern. This macro program symbol has a null effect.

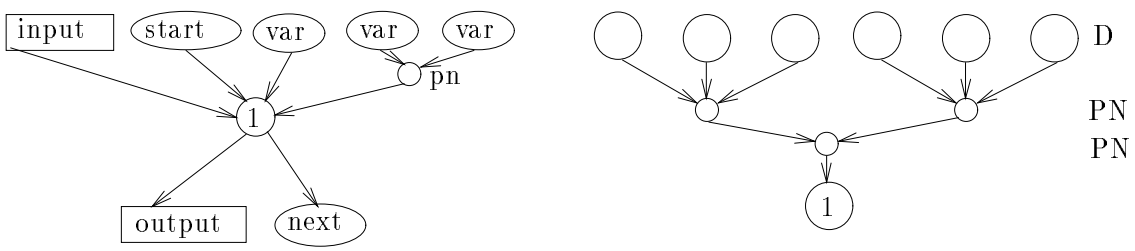


Figure 40: Representation of arrays. On the left: extended invariant pattern, the environment contains a scalar variable and a 1D array with two elements. On the right: representation of a 2D array, using a neural tree of pointer neurons, of depth 2. Layers of pointer neurons are marked by "PN". The layer "D" contains the values of the array.

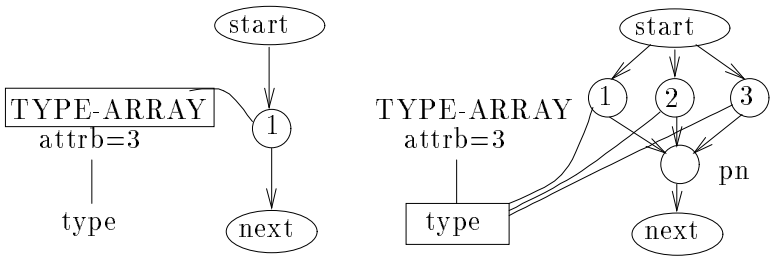


Figure 41: Execution of the macro program symbol `TYPE ARRAY`. This macro program symbol develops the neural tree of pointer neurons that is required to encode an array. We need d macro program symbols `TYPE ARRAY` in order to represent an array of d dimensions. `TYPE-ARRAY` has one parameter which is the number of columns along the associated dimension.

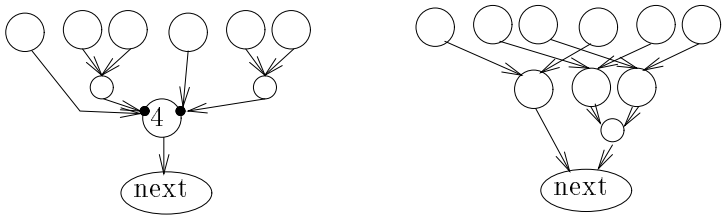


Figure 42: Execution of the cellular code `X-AOR` by cell "4" of the figure that describes the `IF` macro program symbol. The result is the production of a layer of `AOR` neurons, and the duplication of the neural tree that gives a structure to the data.

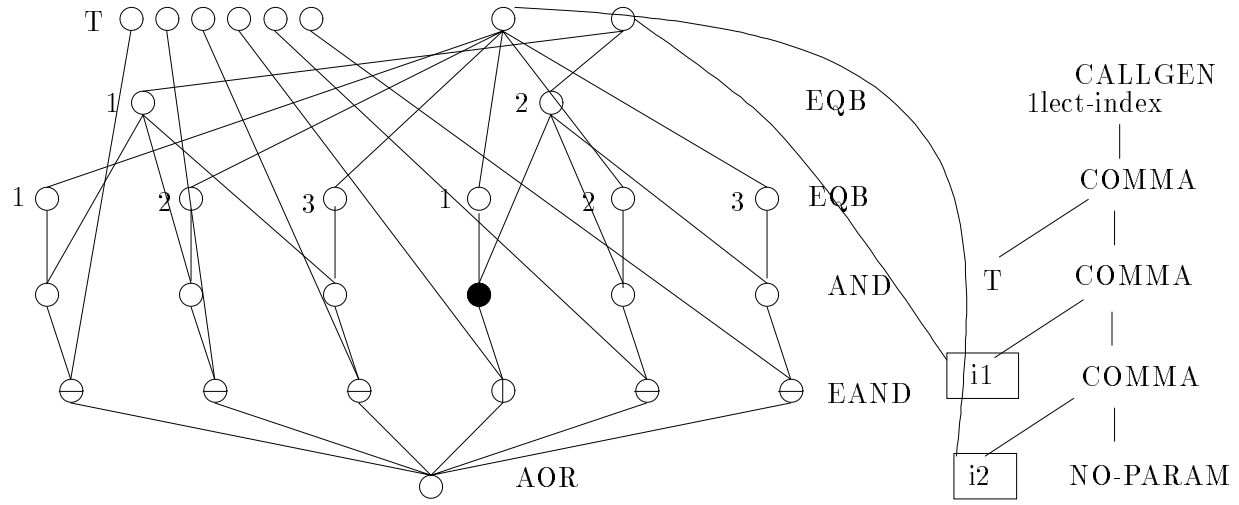


Figure 43: Reading of a 2D array [0..1][0..2] of integers. The index `i1` is 2, the index `i2` is 1. The two first layers of neurons `EQB`s have a sigmoïd that test the equality to zero. They output 1 if the net input is 0, otherwise they output -1. These `EQB` neuron's threshold are indicated. The third layer of neurons computes logical `AND`s. In this layer, a single neuron outputs 1, the one that corresponds to the element that must be read. All the other neurons of this layer output -1. The fourth layer of neurons is a layer of `EAND` neurons. only one of these neuron is activated. It propagates the element of the array that is read. Last, a neuron `AOR` merges all the output lines and retrieves the element that is read. The instruction `CALLGEN` is almost like `CALL-F`, except that all the links to the global variables are deleted. `CALLGEN` is described in the next to the next figure.

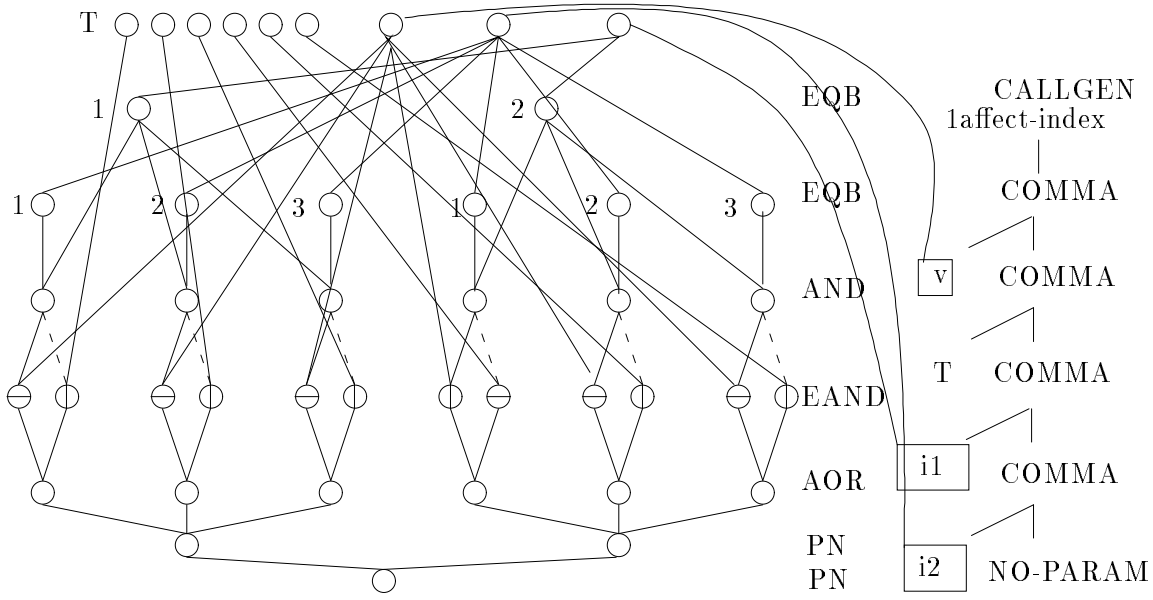


Figure 44: Writing in a 2D array [0..1][0..2] of integers. The index `i1` is 2, the index `i2` is 1. The first three layers are the same as in the preceding figure. The fourth layer contains twice as much `EAND` neurons arranged in pairs. Each pair corresponds to an element of the array. In each pair, one neuron is activated, and the other one is not. The left neuron is never activated except in a single case, when the element corresponds to the place where we want to write. The next layer is a layer of `AOR` neurons. Each `AOR` neuron retrieves the element of the old array, or the value v that is written if the element corresponds to the place where we are writing. Last, a neural tree of pointer neurons preserves the structure of 2D array.

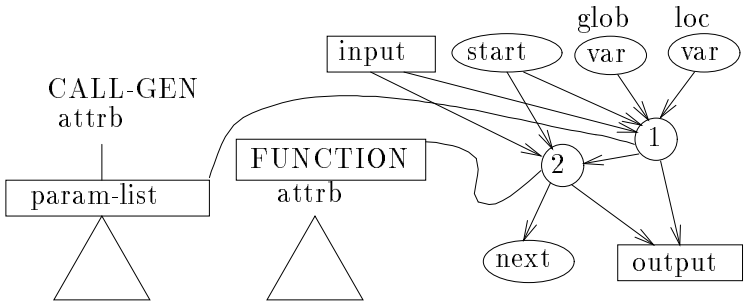


Figure 45: Execution of the macro program symbol `CALLGEN` by the ancestor cell of the invariant pattern. `CALLGEN` has the same effect as `CALL-F` except that the cell "2 " which will execute the body of the function written in cellular code, does not have connections to the global variables of the PASCAL program.

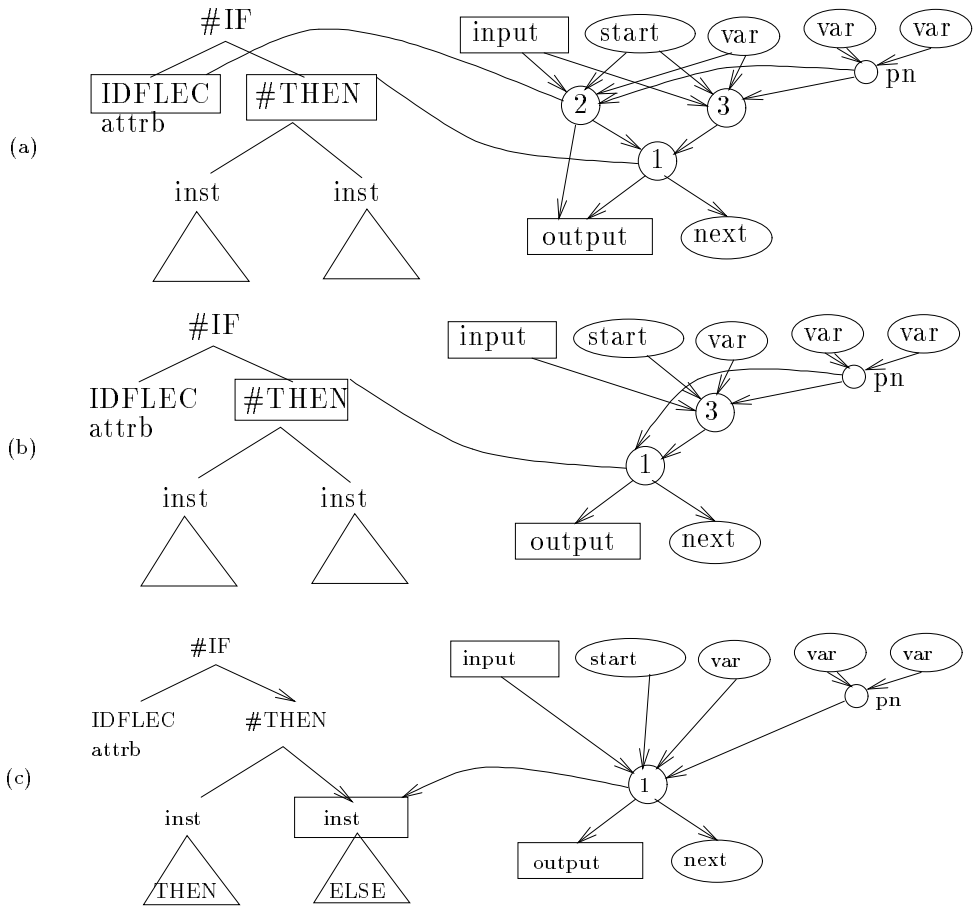


Figure 46: On the top (a): execution of the macro program symbol `#IF` by the ancestor cell of the invariant pattern. Three cells are created. Cell "2" goes to read a macro program symbol `IDF-LEC` that will select the array variable which is concerned by the test. Cell "3" is a neuron that will be used later by cell "1" to restore the environment, once the test is finished. On the middle (b): execution of the macro program symbol `IDF-LEC`, by cell number "2" of the preceding figure. On the bottom (c): execution of the macro operator `#THEN` by cell "1". Cell "1" tests the number of input links of its first neighbor. Here this number is 2 because the array has two elements. If this number is greater than one, cell "1" places its reading head on the right subtree, as it is the case here. Otherwise it goes to read the left subtree. Finally, cell "1" merges the links of the neuron "3".

Name	Role
Reading Head	Reads at a particular position on the cellular code
Life	Counts the number of recursive iterations
Bias	Stores the bias of the future neuron
Sigmo	Stores the sigmoïd
Dyn	Stores the dynamic
link register	Points to a particular link
simplif	Specifies the level of simplifiability
type	Specifies whether the cell is a neuron or a reading cell
x,y,dx,dy	Specifies a window where to draw the cell

Table 1: The registers of a cell

Name	Function of the attribute
DECL	name of the declared variable
TYPE-ARRAY	dimension of the array
IDF-AFF	name of the variable to assign
IDF-LEC	name of the variable to read
INT-CST	value of the integer constant
CALL-P	name of the procedure to call
CALL-F	name of the function to call
CALL-GEN	attrb0 is the name of a pre-encoded solution
CALL-GEN	attrb1 is the name of a procedure for reading an array
CALL-GEN	attrb2 is the name of a procedure for writing an array
BINOP	name of the binary operator
UNOP	name of the unary operatot

Table 2: The attributes of the parse tree