A RECONFIGURABLE SYSTOLIC NETWORK FOR COMPUTER VISION

Robert M. Haralick

Machine Vision International
325 E. Eisenhower Parkway
Ann Arbor, Michigan, 48104

I. Introduction

There are a variety of parallel architectures which can be used for image processing applications. Among them are processor arrays such as Illiac III, CLIP4, CLIP7, DAP, MPP, and GAPP, pipelines such as Cytocomputer, Genesis, and VAP, multiprocessor systems such as PASM, POLYP, ZMOB, GOP, PICAP, TOSPICS, DIP, FLIP, PM4, and pyramid systems. Reviews of these architectures can be found in the papers listed in the reference section.

There appears to be no use of networks or reconfigurable networks for image processing and little discussion of architectures which are simultaneously suitable for image processing and the rest of computer vision. To cut short the review of these architecture types we make some political parallels which have been informally made by some noted researchers. The array can be likened to a Fascist dictator leading a march. The pipeline can be likened to a capitalist assembly line. The pyramid can be likened to the cell block hierarchy of Communist dictatorships. The multiprocessor systems can be likened to parliamentary commitees at work. The network can be likened to political anarchy. And the reconfigurable network just cannot get its act together.

The purpose of this paper is to provide a perspective by which the act of the reconfigurable network can be put together. Our viewpoint will be different from the usual discussions of computer architecture which tend to concentrate around hardware design issues in a hardware language. We believe that all viewpoints which map image data onto processors like the arrays and pyramids or viewpoints which map the specialized tasks onto the different processors like the multiprocessor systems, must of necessity create specialized and inflexible systems. We believe that the watchword of computer vision is flexibility. There must be the low level vision image in and image out operations. There must be the mid level vision image in and data structure out operations. There must be the high level vision data structure in and data structure out operations. And in any computer vision system whose purpose is to be economically useful in the factories of the society in which it functions, there must be the capability for doing numerical calculations, data formatting operations, communication reporting operations, and communication operations of real time control of external devices such as material handlers and robots. This suggests that the approach needs to be an integrated one. To do the design we must step back and understand that the low level neighborhood operators discussed in today's archival literature can be much more complex than the Roberts and Sobel variety. We must understand that the manipulation and processing of the high level vision data structures may be as complex as the symbolic processing required by artificial intelligence computers.

Within our universe, where can we take a stand so that our viewpoint can unravel the inherent complexity of this computer vision question? The required flexibility suggests that the architecture should be able to naturally execute algorithms of a general nature. The quantity of data processed in a computer vision system suggests that the architecture must be in some sense optimized to spend a substantial amount of its processing time doing uniform pixel pushing operations. If it can have high efficiency in performing a regular pattern of operations on a large data set, it can afford to have a lower efficiency in performing less regular operations on small data sets. Or said in another way, the architecture must spend its time performing a variety of activities. If it can configure itself so that it has high efficiency for the most computationally intensive activities, it can afford the overhead required to reconfigure itself to perform the less computationally intensive

CH2229-3/85/0000/0507$01.00 © 1985 IEEE

activities. High efficiency for computationally intensive activities suggests a systolic network. Flexibility suggests reconfigurability. The combination of the two suggests a data flow architecture, a reconfigurable systolic network.

To understand what must go into a data flow architecture we must have a language in which to discuss its configuration possibilities. Hardware programming languages and graph description languages are at a too low level. The interesting thing about the data flow in a systolic network is that a high level specification of the configuration of the network is a specification of the program the network is executing. This is different from Von Neumann architectures in which a specification of the architecture tells nothing about what program is executing. Now low level specification of a network, or more formally, a graph having labeled arcs and nodes, has nothing about it which is sequential or procedural. Likewise, a high level specification need not be sequential or procedural. A high level specification of a network is just a specification of the relations which hold in the network. So specification of the configuration of a systolic network amounts to specifying relations and since the specification is the program which the network executes, the language used to program a systolic network is a language of relations. The language must be naturally non-procedural. From a high level perspective, the semantics of the language essentially describe the essence of the architecture. Therefore, in the remainder of the paper we describe the language INSIGHT, a language in the LUCID family of data flow language (Wadge and Ashcroft, 1985) which we have developed for the purpose of data flow architecture specification, and for a language in which our computer vision algorithms can be written to execute on such a data flow architecture, an architecture like a reconfigurable systolic network.

II. INSIGHT Language Specification

II.1 Motivation

All computation can be thought of in terms of generating simple sequences and transforming and/or combining them. Thus the sequence is the dominant data object of INSIGHT, the language which defines the configuration of the reconfigurable systolic network of the MVI integrated technology machine and the language whose expressions are evaluated by the systolic network. INSIGHT is in the family of LUCID languages.

Mathematically, a sequence is a mapping which associates to each natural integer a value. If we let

$$S = < s_0, s_1, ..., s_n, ..., >$$

denote a sequence, then the value of the sequence associated with the natural integer 0 is s_0, the value of the sequence associated with the natural integer 1 is s_1, and the value of the sequence associated with the natural integer n is s_n. In the systolic network, a sequence is the discretized time history of any signal line where signal line can mean an associated group of 8, 16, or 32 wires. The continuous time history of a signal line is discretized by the clock ticks. A successive value is generated for the sequence at each clock tick. This value may be the previous value if the computation for the next term has not completed. Also associated with each signal line is its state indicating whether the sequence is in active generation or has terminated. Active sequences are those in which the concatenation process which constructs the sequence is still going on. A sequence is inactive after it terminates. In effect, the continuous time history of a signal line is infinite. The sequence's state, active or terminated, enables a finite sequence to be extracted from the potentially infinite time history or sequence.

The sequence construct of INSIGHT is a precise symbolic representation of any discretized time function manipulated by the systolic network. An INSIGHT identifier denoting a sequence is not like a variable in a common imperative or procedural programming language such as FORTRAN or C. In the procedural languages the value of a variable can be thought of as the contents of a particular memory location. The same variable appearing in defferent places in the same program may have different values. This is undoubtedly the case if it appears on the left side of an assignment statement more than one time. In INSIGHT, all occurrences of the same sequence name designate the same sequence. Or said another way, the value of each occurrence of the same sequence is the same, term for term. In INSIGHT, there is no assignment statement. There is an equals statement which is used to define a sequence. In INSIGHT, the same sequence name can only be defined once, that is, it can appear to the left side of an equal sign only once. The sequence of INSIGHT can be likened to a variable in motion. The sequence is the variable's motion history and not a snapshot of the variable at any particular moment.

II.2 Succession

Since all computation can be thought of in terms of generating simple sequences and transforming and/or combining them and since the sequence is the basic entity of INSIGHT, it should come as no surprise that the first definitions we give of INSIGHT syntax is that for generating a sequence. To guarantee that our definitions are clear we will work up to them from a mathematical example and associated hardware implementation.

Let $I = < i_1, ..., i_N, ... >$ represent an input sequence and $S = < s_0, ..., s_N, ... >$ represent an output sequence. The term s_n of the output sequence for our example is defined as the sum of all the terms of I up to and including the term i_n. That is,

$$s_n = \sum_{k=0}^{n} i_k$$

$$= s_{n-1} + i_n$$

In hardware, all sequences are generated by a sequential machine. The output of a sequential machine is a function of its current state and current input. The next state of a sequential machine is a function of its current state and current input.

The sequential machine hardware which accomplishes the transformation required in our example consists of two basic elements: a combinational logic adder, and a sequential logic delay element whose output is the input delayed by one time unit. The delay one element is perhaps the simplest of all sequential machines. Its output is always its current state. Its next state is always the current input. The two elements are connected as shown in Figure 1 to compute s_n.

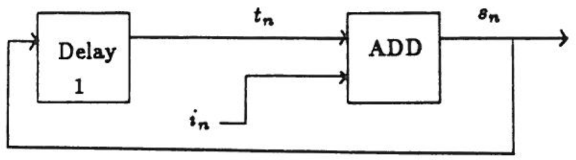

Figure 1 illustrates a single circuit to compute the running sum of the input.

The convention used in Figure 1 is that we have snapped a picture of the state of affairs at time n. Hence every line has an associated variable name with a subscript n. Now by the computational meaning of the elements we can write the following equations:

$$t_n = s_{n-1}$$

$$s_n = t_n + i_n$$

Upon eliminating the value t_n, we obtain

$$s_n = s_{n-1} + i_n$$

The required relationship defining the nth term of the sequence S. The diagram of Figure 1 can be illustrated directly using the above equation as shown in Figure 2. The snapshot of Figure 2 is also taken at time n.

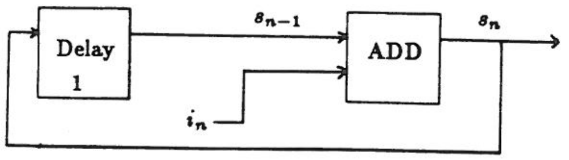

Figure 2 illustrates a simple circuit computing the running sum of the input.

Any line having an associated symbol whose subscript is not n, such as the one labeled s_{n-1}, does not mean that the time associated with the line is not n. It means that at time n the value of the line is the value that the line designed s_n had at time $n - 1$.

Now the recursive equation

$$s_n = s_{n-1} + i_n$$

is equivalent to the equation

$$s_n = \sum_{k=0}^{n} i_k$$

only when the initial condition for s_0 is specified as

$$s_0 = i_0$$

The hardware correspondence to this fact is clear upon examining the delay element if Figure 1. To make s_0 be equal to i_0 through the adder, the output t_0 of the delay memory at time 0 must have the value 0. Delay memories have initial values that must be set. A memory with delay 1 has one value to be set. In this case, the initial value of the delay memory must be set to be 0.

The INSIGHT statement which accomplishes the sequence generation of $S = < s_0, ..., s_n, ..., >$ needs to contain the initial information that $s_0 = i_0$ and that the sequential generation equation is $s_n = s_{n-1} + i_n$. Because the sequence is the basic entity of INSIGHT, it would be more appropriate to define the sequence generation using a syntax which deals directly with the entity sequence and not with the nth term of the sequence. INSIGHT does precisly this.

We begin the INSIGHT expression defining the sequence S from the sequence I by first describing the relationship between the sequence $T =< t_0, ...t_n, ..., >$ and $S =< s_0, ..., s_n, ..., >$ of Figure 1. The mathematical relationship the syntax has to capture is $t_n = s_{n-1}$, $t_0 = 0$. The INSIGHT statement which does this is

$$T = 0\ FOLLOWED\ BY\ S$$

Here T designates the sequence $< t_0, ..., t_n, ..., >$, S designates the sequence $< s_0, ..., s_n, ..., >$, and 0 designates the sequence $< 0, ..., 0, ..., >$. The meaning of the statement is that T is a sequence whose first term t_0 is the first term of the sequence 0 and whose remainder terms are the terms of the sequence S. In other words, insert a 0 in front of the first term of the S sequence. The result is the T sequence.

The adder of Figure 1 adds the sequences T and I, term by term to produce the sequence S. INSIGHT expresses this by

$$S = T + I$$

Thus the function of the circuit of Figure 1 is completely described by

$$T = 0\ FBY\ S$$

$$S = T + I$$

where we use FBY as an abbrevation for $FOLLOWED$ BY. The function of the circuit of Figure 1 is also completely described by

$$S = T + I$$

$$T = 0\ FBY\ S$$

The order of the statements appearing in an INSIGHT clause is not material. INSIGHT is a non-procedural language. Its equal sign is the equal sign of algebra. The semantics of an equals statement is the semantics of the algebraic relationship of equality.

Now in algebra, any expression equal to a quantity may be substituted for that quantity in all places where the quantity appears and there will be no resulting change in the meaning of the algebraic system. Since the equals of INSIGHT is the equals of algebra the same substitutability holds. The sequence T appears in the statement

$$S = T + I$$

The sequence T is defined by the statement

$$T = 0\ FBY\ S$$

Substituting the defining expression the T into the first statement we obtain the valid INSIGHT statement defining the sequence S

$$S = (0\ FBY\ S) + I$$

The definition now appears recursive. It corresponds to the circuit diagram of Figure 2. We can rewrite the above statement. The first term of $0\ FBY\ S$ is 0. The first term of I we denote by $FIRST(I)$. Hence, $S = FIRST(I)\ FBY\ (S + I)$.

For our second example, we consider the generation of the sequence of natural integers $I =< 0, 1, ..., n, ..., >$. The recursive equation is $i(n) = i(n-1) + 1$ with initial condition $i(0) = 0$. The corresponding INSIGHT statement is

$$I = 0\ FBY\ I + 1$$

which means that I is the sequence whose first term can be obtained as the first term of the sequence 0. The remaining terms of the sequence I are obtained by adding the sequence I to the sequence 1, a sequence all of whose terms are ones.

For our third example, we consider the generation of the sequence S defined by the recursive equation

$$s_n = s_{n-1} + i_n * s_{n-2},\ s_0 = s_1 = 0$$

To see how to do this, we first rewrite the recursive equation as

$$\begin{aligned}
&s_n = q_n + r_n \\
&\quad where \\
&\qquad q_n = s_{n-1} \\
&\qquad r_n = i_n * t_n \\
&\qquad\quad where \\
&\qquad\qquad t_n = s_{n-2}
\end{aligned}$$

The parallel INSIGHT statements to this mathematical expression are

```
S = Q + R
    WHERE
        Q = 0 FBY S
        R = I * T
            WHERE
                T = 0 FBY S DBY 2
```

The statement

$$T = 0\ FBY\ S\ DBY\ 2$$

means that T is the sequence whose first two terms are the first two terms of the sequence and whose remaining terms are those of the sequence S. The statement

$$Q = 0\ FBY\ S$$

is really a short form for the statement

$$Q = 0\ FBY\ S\ DBY\ 1$$

Any followed by phrase (FBY) not including a delayed by (DBY) defaults to a delayed by 1.

The WHERE clauses permit nested definitions. In the statement

$$S = Q + R$$

both Q and R have not already been defined. The first *WHERE* clause defines Q and R. However, R is defined in terms of T. So the second *WHERE* clause gives the required definition for R.

If A and B are sequences, INSIGHT permits the concatenation of A with B, A first and B second, by the statement

$$S = A\ CONCATENATED\ WITH\ B$$

II.3 Finite Sequences

In actual practice, the sequences required in computation are not infinite. They have a definite starting time and a definite ending time. The sequences are finite. Also, the generation of any term of a sequence may depend on terms previous to the current term. To do this, the syntax of INSIGHT has to be able to express two new concepts. The first one is a way of specifying how a sequence can terminate and the second one is a way of coordinating or time shifting the values of one sequence with respect to another. To generate only the positive integers up to and including N in INSIGHT we use the *UNTIL* condition phrase in the *FBY* statements.

$$I = 1\ FBY\ I\ +\ 1\ UNTIL\ I = N$$

To generate the factorial sequence $F = < f_1, f_1, ..., f_N >$ from the integer sequence

$$I\ = < i_1\ , ...,\ i_N > = < 1,\ 2, ...,\ N >,$$

we want to express the recursive equation $f_n = i_n f_{n-1}$, initial condition $f_0 = 1$. Since $i_n = n$, we could have more naturally written $f_n = n f_{n-1}$ as the recursive equation, but doing so would confuse the distinction between n which designates which time moment we fix upon and i_n which is the value of the sequence I at time moment n. The expression $i_n f_{n-1}$ involves the coordination of two sequences time shifted with respect to one another. At any time moment n, the term we want from the sequence F is the term previous to the current term f_n. The natural INSIGHT syntax expressing $f_n = i_n f_{n-1}$, initial condition $f_0 = 1$, is

$$F = 1\ FBY\ F\ *\ NEXT\ I$$

The *NEXT I* phrase indicates that the term of the I sequence which mulitplies the terms of the F sequence is that term which is one ahead.

In hardware, the active/terminated state for the sequence I of the factorial example remains logically true until immediately after the clock tick associated with the term i_N which is the last term of the sequence. At this point it changes to logically false. Since the sequence F is dominated by the active/terminated line for the sequence I, the active/terminated line for F is dominated by the active/terminated line for the sequence I. Hence the termination of the sequence I forces the termination of the sequence F. The sequence F changes to an inactive state when the sequence I changes to an inactive state.

We have already seen that one way to terminate a sequence is by the use of the *UNTIL* condition phrase which is part of a sequence definition statement. An alternative way to terminate a sequence is by the use of the *AS LONG AS* condition phrase. Its key word abbreviation is *ALA*. The semantics of the *AS LONG AS* phrase is dual to the semantics of the *UNTIL* phrase. For the *AS LONG AS* phrase, a sequence generates (stays in the active state) until the first clock tick during which the condition of the *AS LONG AS* phrase becomes false. Then the sequence terminates and its active state permanently changes to terminated. In comparison, with the $UNTIL$ phrase, a sequence generates until the first clock tick during which the condition of the until phrase becomes true.

II.4 Sequence Generation Control

A sequence may be active (not terminated) but its generation mechanism may not have the next value of the sequence ready or available by clock tick time. Hence, if the state of a sequence is active, there are two possibilities: either the next term of the sequence will be generated by the next clock tick or the next term of the sequence will not be generated by the next clock tick. Thus, active sequences can be in a state active and output ready or active and output not ready. This can happen for a variety of reasons. For example, the next term may not be ready because new inputs to the generating mechanism have not arrived. Or, the next term may not yet have been accepted by all the places to which it goes. Finally, it may not be ready because of INSIGHT conditional control.

II.5 Condition Control

There are three classes of INSIGHT conditional control. They are selection control, generation control, and modification control. We begin by describing selection control. There are two selection control statements: $WHENEVER$ and $AS\ SOON\ AS$. The $WHENEVER$ statement can force the state of a sequence to be active and output not ready on a user specified condition. For example suppose B denotes a data sequence and C denotes a Boolean control sequence, and the sequence A is defined by

$$A = B\ WHENEVER\ C$$

Then at each clock tick time, the term in A has the same value as the term in B when the term in C is true. Under these conditions, the state of A will be active and output ready. But should the term in C be false, then the sequence A will go into the state active and output not ready. All generation mechanisms for which A is an input will then not have any input to accept for at the clock tick at which they would ordinarily have an output value which would have been computed based on an input from A, when A is in this state of active and output not ready, their output sequences will be forced into the state active and output not ready. From a computational point of view, the $WHENEVER$ statement causes A to be a subsequence of B selected by the sequence C.

The Boolean sequence C can also be expressed as a condition. For example, if D and E are sequences, the INSIGHT permits

$$A = B\ WHENEVER\ (D \leq E).$$

In this case, the condition sequence is true for all clock ticks during which the term in D is less than or equal to the term in E. Operators permitted in the condition clause include $=$, $\neq$, $<$, $>$, $\leq$, $\geq$.

The $AS\ SOON\ AS$ selection control of INSIGHT is also a subsequence detection mechanism. It has the form

$$A = B\ AS\ SOON\ AS\ C$$

With the $AS\ SOON\ AS$ mechanism, the initial terms of B are not selected. The first term to be selected from B is the term for which the $AS\ SOON\ AS$ condition is true. Then it and all subsequent terms are selected. Hence the sequence A remains in the active and output not ready state up until the clock tick that the first term of C becomes true. At that clock tick, the sequence A changes to the active and output not ready, then even though the condition C may be true, the sequence A will go into an active and output not ready state too. If either sequences B or C are terminated, then the sequence A becomes terminated.

There are three INSIGHT generation control statements. They are: $STALLED\ BY$, $SUSPENDED\ BY$, and $CONTINUED\ BY$. The generation control statements work differently than the selection control statements. The selection control accomplished what it had to do by controlling the state of the sequence. It enables user selection of one of the states active and output ready or active and output not ready. This kind of control is from the output side looking forward. In contrast, the generation control is from the input side looking back.

Suppose that the sequence B is an input to the generating mechanism for the sequence A. When B is an output ready state, the generating mechanism for A can accept the term from B. It does so by communicating to the generating mechanism of B "input accepted". Before the generating mechanism for B can put B in a state of active and output ready, it must make sure that its current output has been accepted by all the generating mechanisms to which it is connected. Until acceptance has been received from all these places, B must maintain an active and output not ready state. INSIGHT generation control permits a conditional control on the "input accepted" communication.

Consider the statement

$$A = B\ STALLED\ BY\ C$$

If at a clock tick, C has the value false, then sequence B is not stalled and everything happens normally. The term in sequence A assumes the value that the term in sequence B has if sequence B is in a state of active and output ready. If at this clock tick, sequence B is in a state of active and output not ready, then sequence A takes the state active and output not ready. However, if at a clock tick, C has the value true, then sequence B is stalled. The generating mechanism for A does not communicate the message "input accepted" to B so that the generating mechanism for B stays in a state of waiting until its current output is accepted in all the places to which it goes. At this clock tick, sequence A takes the state active and output not ready.

Just as an INSIGHT sequence has a state, so does the generation mechanism of the sequence have a state. The possible states of the generation mechanism are accepting data input and not accepting data inputs. The $STALLED\ BY$ statement is one which sets A's generation mechanism to the not accepting data input state when the condition C is true and sets A's generation mechanism state to the accepting input state then condition C is false.

The two other INSIGHT generation control statements are $SUSPENDED\ BY$ and $CONTINUED\ BY$. The statement

$$A = B\ SUSPENDED\ BY\ C$$

puts A's generation mechanism in the state of not accpeting data inputs from B upon the first clock tick that C takes the value true. The sequence A then takes the state active and output not ready and the sequence B is stalled until it is continued. We call the condition of being constantly stalled "suspended". To change A's generation mechanism to the state of accepting data inputs INSIGHT uses the statement

$$A = B\ CONTINUED\ BY\ C$$

Typically $SUSPENDED\ BY$ and $CONTINUED\ BY$ are used together as in

$$A = (B\ SUSPENDED\ BY\ C)\ CONTINUED\ BY\ D$$

Here the next clock tick during which C takes the value true causes A's generation mechanism to constantly take the not accepting data input state. This continues until the next clock tick during which D takes the value true causing A's generation mechanism to constantly take the not accepting data input state. This continues until the next clock tick during which D takes the value true. At this clock tick, A's generation mechanism takes the accepting data input state.

There are four modification control statements. They are $EXTENDED\ BY$, $SUSTAINED\ BY$, $PROLONGED\ BY$, and $UPON$. First we consider

$$A = B\ SUSTAINED\ BY\ C$$

This control statement does not control a sequence's state or the sequence's generation mechanism's state. If during a clock tick C takes the value false, the term for A takes the value of the term from B. If during a clock tick C takes the value true, the term from A takes the value of the previous term from A.

Next we consider

$$A = B\ EXTENDED\ BY\ C$$

This control statement has an action similar to $SUSTAINED\ BY$. In $SUSTAINED\ BY$, for each clock tick for which C takes the value true, A takes the previous value of A. In $EXTENDED\ BY$, from the first clock tick on that C takes the value true, A takes the previous value of A. For all clock ticks before the first clock tick that C takes the value true, the term in A takes its value from the term in B.

Finally, we consider

$$A = B\ PROLONGED\ BY\ C$$

This control statement has an action which is a combination of the $SUSTAINED\ BY$ and $STALLED\ BY$ statements. If during a clock tick C takes the value true, then the term in A takes the value of the previous term in A and the generation mechanism of A is put into the not accepting input data state. If during a clock tick, C takes the value false, then the generation mechanism of A is put into the accepting input data state and A takes the value of the term in B.

The dual to $PROLONGED\ BY$ is the $UPON$ statment. The statement

$$A = B\ \ UPON\ \ C$$

is also a combination of the $SUSTAINED\ BY$ and $STALLED\ BY$ statements. If during a clock tick C takes the value true, then the generation of mechanism of A is put into the accepting input data state and the term in A

takes the value of the term in B. If during a clock tick, C takes the value false, then the term in A takes the value of the previous term in A and the generation mechanism of A is put into the not accepting input data state. Thus

$$A = B \ UPON \ C$$

has exactly the same effect as

$$A = B \ PROLONGED \ BY \ NOT \ C$$

III. BNF INSIGHT Grammar

This is a modified BNF grammar where the symbol | specifies an alternative and the expression $\{x\}$ means zero or more instances of x. Square brackets are really characters in INSIGHT, not symbols of the grammar.

```
<program> ::= prog <identifier>(<formals list>) =
                  <where expression> endprog |
        prog <identifier>(<formals list>) <where clause>
                  endprog |
        prog <identifier>() = <expression> endprog |
        prog <identifier>() <where clause> endprog
<expression> ::= <constant> |
        <identifier> |
        <memory expression> |
        <seqarray expression> |
        <prefix operator> <expression> |
        <expression> <infix operator> <expression> |
        <if expression> |
        <cond expression> |
        <function call> |
        <where expression> |
<constant> ::= <numeric constant> |
        <boolean constant> |
        <binary constant> |
        <character constant>
<numeric constant> ::= <integer constant> |
        <real constant> |
        <extended real constant> |
        <pointer constant>
<integer constant> ::= <digit> { <digit> } |
        <n-sign> <digit> { <digit> }
<real constant> ::= <integer constant>.{ <digit> } |
        <integer constant>.{ <digit> } E <integer constant>
<extended real constant> ::= <integer constant>.{ <digit> } |
        <integer constant>.{ <digit> } D <integer constant>
<pointer constant> ::= <digit> { <digit> }
<boolean constant> ::= true | false
<binary constant> ::= 0 | 1
<character constant> ::= ' <ASCII character> '
<n-sign> ::= -
<memory expression> ::= <memory identifier> [<subscript list>]
<memory identifier> ::= <identifier> |
        <identifier> [<subscript list>]
<subscript list> ::= <exprcssion> |
        <subscript list>,<expression>
<seqarray expression> ::= <sequence identifier> [<subscript list>]
<sequence identifier> ::= <identifier>
<declaration list> ::= { <global declaration> ; }
<global declaration> ::= <scalar type> <scalar idlist> |
        <composite type> <composite idlist>
<scalar type> ::= <constant type> | <sequence type>
<composite type> ::= <memory type> | <seqarray type>
<constant type> ::= integer constant | real constant |
        boolean constant | pointer constant |
        binary constant | extended real constant |
        character constant
<sequence type> ::= integer sequence | real sequence |
        boolean sequence | pointer sequence |
        binary sequence | extended real sequence |
        character sequence
<memory type> ::= integer memory | real memory |
        boolean memory | pointer memory |
        binary memory | extended real memory |
        character memory
<seqarray type> ::= integer seqarray | real seqarray |
        boolean seqarray | pointer seqarray |
        binary seqarray | extended real seqarray |
        character seqarray
<scalar idlist> ::= <identifier> |
        <scalar idlist> , <identifier>
<composite idlist> ::= <ident> |
        <composite idlist> , <ident>
<ident> ::=<identifier> <dimension declaration>
<dimension declaration> ::= [ <dimlist> ]
<dimlist> ::= <dimspec> |
        <dimlist> , <dimspec>
<dimspec> ::= <numeric constant> : <numeric constant>
<alphanumeric> ::= <digit> |<letter>
<digit> ::= 0 |1 |2 |3 |4 |5 |6 |7 |8 |9
<letter> ::= A |B |C |D |E |F |G |H |I |J |K |L |
        M |N |O |P |Q |R |S |T |U |V |W |X |Y |Z |
        a |b |c |d |e |f |g |h |i |j |k |l |
        m |n |o |p |q |r |s |t |u |v |w |x |y |z
<identifier> ::= <letter> { <alphanumeric> }
<prefix operator> ::= <p-numeric operator> |
        <p-insight operator> |
        <p-logic operator> |
        <p-sequence operator>
<p-numeric operator> ::= - |sin |cos |tan |sqrt |
        abs |log10 |log |isnumber
<p-insight operator> ::= first | previous
<p-logic operator> ::= not
<p-sequence operator> ::= isbegun | isfinished | isactive
<infix operator> ::= <i-numeric operator> |
        <i-insight operator> |
        <i-logic operator> |
        <i-character operator>
<i-numeric operator> ::= + |- |** |* |div |mod |/ |
        tan2 |== |^= |<= |< |>= |>
<i-insight operator> ::= fby |whenever |upon |asa |
        attime |ala |dby |until |repeat |cycle |tfby |
        stalby |suspby |contby |extby |sustby |prolby
<i-logic operator> ::= and |or |exor |nand |nor |exnor |
        leq |lge |lgt |lle |llt |lne
<i-character operator> ::= lexeq |lexge |lexgt |
        lexlt |lexle |lexne
<if expression> ::= if <expression> then <expression> endif |
        if <expression> then <expression> else <expression>
        endif
<cond expression> ::= cond <cbody> endcond
<cbody> ::= { <expression> : <expression> ; } <defaultcase>
<defaultcase> ::= else : <expression>;
<function call> ::= <identifier> ( <actuals list> )
<actuals list> ::= <expression> |
        <expression> , <actuals list>
<where expression> ::= <expression> <where clause>
<where clause> ::= where
        { arguments <declaration list> }
        { declare <declaration list> }
        relations <wherebody> endwhere
```

```
<wherebody> ::= <current list>
            <function list>
            <activity list>
            <definitions list>
<current list> ::= { <current declaration> ; }
<current declaration> ::= <identifier> is current <expression>
<function list> ::= { <function definition> ; }
<activity list> ::= { activity <identifier>
            <definitions list> endactivity ; }
<definitions list> ::= { <definition> ; }
<definition> ::= <identifier definition> |
            <memory assignment> |
            <iterated definition>
<identifier definition> ::= <identifier> = <expression>
<memory assignment> ::= <memory expression> = expression
<function definition> ::= <identifier>(<formals list>) =
            <where expression> |
            <identifier>(<formals list>) <where clause>
<formals list> ::= <identifier> |
            <identifier> , <formals list>
<iterated definition> ::= foreach <identifier> := <expression> to
            <expression> by <expression>
            { ( <array definition> | <memory definition> ) ; }
            endfor
<array definition> ::= <seqarray expression> = <expression>
<memory definition> ::= <memory identifier> = <expression>
```

IV. References

S. Yalamanchili, K.V. Palem, L.S. Davis, A.J. Welch, and J.K. Aggarwal, "Image Processing Architectures: A Taxonomy and Survey", Progress in Pattern Recognition 2, L.N. Kanal and A. Rosenfeld (Eds.), Elsevier Science Publishers, B.V. (North Holland), Amsterdam, 1985, pp. 1-37.

B. Kruse, "State-of-the-Art Systems For Pictorial Information Processing", Fundamentals in Computer Vision, O.D. Faugeras (Ed.), Cambridge University Press, Cambridge, 1983, pp. 425-442.

S.R. Sternberg, "Parallel Architectures For Image Processing", Real-Time/Parallel Computing, M. Onoe, K. Preston, and A. Rosenfeld (Eds.), Plenum Press, N.Y., 1981, pp. 347-359.

T.A. Rice and L.H. Jamieson, "Parallel Processing For Computer Vision", Integrated Technology For Parallel Image Processing, S. Levialdi (Ed.), Academic Press, Inc., London, 1985, pp. 57-78.

W.W. Wadge and E.A. Ashcroft, LUCID: The Dataflow Programming Language, Academic Press, London, 1984.