paper submitted to
"Visualization in Human-Computer Interaction"
7th Interdisciplinary Workshop in Informatics and Psychology
Schärding, Austria, May 24-27, 1988

Representing Knowledge with Iconic Rules in the Domain of Functional Programming

Claus Möbus

University of Oldenburg
FB 10, Informatik
Unit on Tutoring and Learning Systems
D-2900 Oldenburg
W-Germany

This research was sponsored by the Deutsche Forschungsgemeinschaft (DFG) in the SPP Psychology of Knowledge under Contract No. MO 293/3-2

5. The Design of our Reactive Graphical Programming Environment ABSYNT

5.1 Psychological Arguments in Favor of a Functional Visual Programming Language

The main research goal of ABSYNT is the construction of a problem solving monitor (PSM). Some PSM-relevant research has been reported about solving problems in simple arithmetic tasks (ATTISHA, 1984; ATTISHA & YAZDANI, 1983; BROWN & BURTON, 1978; BUNDY, 1983; BURTON, 1982; VanLEHN & BROWN, 1980; YOUNG & O'SHEA, 1981), in quadratic equations (O'SHEA, 1979, 1982), in simple algebra problems (SLEEMAN, 1982, 1983, 1984, 1985, 1986), in geometry (ANDERSON, 1983c; ANDERSON, BOYLE, FARRELL & REISER, 1987; ANDERSON, GREENO, KLINE & NEVES, 1981) and in computer programming (ANDERSON, 1983b, 1987; ANDERSON, FARRELL & SAUERS, 1982, 1984; ANDERSON & REISER, 1985; ANDERSON & SKWARECKI, 1986; JOHNSON, 1986; JOHNSON & SOLOWAY, 1985, 1987; SOLOWAY, 1986; WERTZ, 1982, 1985, 1987).

We chose the domain of computer programming because problem solving is the main activity of each programmer. Furthermore, errors can be diagnosed easily. We had to make some more design decisions. Because the PSM should mainly supervise the planning processes of the programmer, we decided to use a simple programming language, the syntax and semantics of which can be learned in a few hours. We decided to take a purely functional language. From the view of cognitive science functional languages have some beneficial characteristics. So less working memory load on the side of the programmer is obtainable by their properties, referential transparency and modularity (ABELSON, SUSSMAN & SUSSMAN, 1985; GHEZZI & JAZAYERI, 1987[2]; HENDERSON, 1980, 1986). Furthermore, there is some evidence that there is a strong correspondency between programmer's goals and use of functions (PENNINGTON, 1987; SOLOWAY, 1986; JOHNSON & SOLOWAY, 1985, 1987). So we avoid the difficult problem of interleaving plans in the code which show up in imperative programming languages because it makes the diagnosis of programmer's plans rather difficult (SOLOWAY, 1986). If we take for granted that a goal can be represented by a function, we can gain a great flexibility in the PSM concerning the programming style of the student. We can offer him facilities to program in a bottom-up, top-down or middle-out style. The strategy of building up a goal hierarchy can correspond to the development of the functional program.

There are some similar psychological reasons for the use of a visual programming language, too. There is some evidence that less working memory load is obtainable through the use of diagrams if they support encoding of information or if they can be used as an external memory (FITTER & GREEN, 1981; GREEN, SIME & FITTER, 1981; PAYNE, SIME & GREEN, 1984; SIMON & LARKIN, 1987). Especially if we demand the total visibility of control and data flow the diagrams can serve as external memories.

The diagrammatic structuring of information should also reduce the amount of verbal information which is known to produce a higher cognitive processing load than "good" diagrams (SIMON & LARKIN, 1987). "Good" diagrams produce automatic control of attention with the help of location objects. These are in our case object icons, which are made of two sorts: straight connection lines and convex objects. Iconic objects of these types are known to control perceptual grouping and simultaneous visual information processing (POMERANZ, 1985; CHASE, 1986).

5.2 Computer Science Based Design Guidelines for Visual Languages

Work on the design of visual languages has just started twenty years ago (e.g. LAKIN, 1980), but some results have been obtained which are not controversial among scientists. Visual programming languages can be described by a profile in a three-dimensional system according to 1) visual extent, 2) scope and 3) language level (SHU, 1986). A language has a high visual extent if graphics are not mere illustrations but play a central role in programming. They must be "executable graphics" (LAKIN, 1986). The scope of a language is a measure of the generality of their applicability. The language level gives a hint how abstract and hardware independent the constructs of the language are.

The design of a visual language has to be based on the concept of generalized icons (CHANG, 1987), which are dual representations of abstract and visual parts. The type of generalized icons can be divided into object icons and process icons. Object icons define the represenation of static language constructs, whereas process icons specify the representation of dataflow and controlflow. We want to quote CHANG (1987, p.9f) on this subject:

An **iconic system** is a structured set of related icons. A complex icon can be composed from other icons in the iconic system, and therefore express a more complex visual concept. An **iconic sentence** ... is a spatial arrangement of icons from an iconic system. A **visual language** is a set of visual sentences constructed with given syntax and semantics. **Syntactic analysis of visual language** (spatial parsing) is the analysis of the spatial arrangement of icons (i.e. visual sentences) to determine the underlying syntactic structure. Finally, **semantic analysis of visual language** (spatial interpretation) is the interpretation of a visual sentence to determine its underlying meaning.

From the view point of system implementation, to design an iconic system and a visual language, two major software tools are required: a) an iconic editor to edit a generalized icon; and b) an icon interpreter to perform syntactic analysis and semantic analysis of the visual system."

Similar ideas stem from GLINERT & GONCZAROWSKI (1987) and GLINERT, GONCZAROWSKI & SMITH (1987). CHANG (1986) proposed a specification cycle in the language design (Figure 1). We used this cycle for our design. The abstract parts of the language were specified in PROLOG according to some ideas of (PEREIRA, 1986) as a runnable specification (DAVIS, 1982). The corresponding visual parts were specified obeying results of our own empirical research or generalizing findings and standards from cognitive psychology and cognitive science.

Figure 1: The Specification Cycle of a Visual Language (CHANG, 1986)

The complete programming environment is implemented in INTERLISP and the object-orientated language LOOPS (JANKE & KOHNERT, 1988; KOHNERT & JAHNKE, 1988) to have a system with direct manipulation capabilities which is an absolutely necessary prerequisite for our system (FÄHNRICH & ZIEGLER, 1985a,b; HUTCHINS, HOLLAN & NORMAN, 1986; SHNEIDERMAN, 1983, 1987). Following SHU's dimensional analysis, ABSYNT is a language with high visual extent, low scope and medium level.

5.3 Cognitive Science and Psychology Based Guidelines for the Design of Graphical Objects and Diagrams

Though through decades research in psychology, physiology and computer science has been devoted to human, animal and machine perception, results which could guide decisions in building tutorial applications have remained on a rather informal level of "gestalt laws" or phenomenological principles (BERTIN, 1981, 1983; CAMPBELL & ROSS, 1987; CLEVELAND, 1985; FITTER & GREEN, 1979; LUTZE, 1987; TUFTE, 1980; WOOD & WOOD, 1987).

In our work we relied on empirical studies partly done by others (e.g. WEBER & KOSSLYN, 1986) and partly conducted by ourselves. The former gave us hints concerning the synchronisation of the properties of graphics and the mental imagery system. The latter dealt with the memory representation of the tree programs (SCHRÖDER, COLONIUS & FRANK, 1987) and errors which resulted from misinterpretations of the syntax and the semantics of the original language as appeared in BAUER & GOOS (1982). The last version of the language which is used for ABSYNT was strongly influenced by an empirical study of POMERANTZ (1985) and a theoretically orientated article by SIMON & LARKIN (1987). POMERANTZ made some careful studies about selective and divided attention information processing. One consequence for our design was that time-indexed information had to be spatial indexed by locations, too. Information with the same time index should have the same spatial index. This means that this information should appear in the same location. is In our design a location is identical with a visual object. These insights were supported by the formal analysis of SIMON & LARKIN (1987). They showed under what circumstances a diagrammatic representation of information consumes less computational resources as an informational equivalent sentential representation.

5.4 The Iterative Specification Cycle for the Derivation of Iconic Objects and Iconic Rules Concerning the Operational Semantics of ABSYNT

Though our main research goals lie in the exploration and debugging of planning processes we have to deal with the computational knowledge of the programmer, too. It is our opinion that a user of our language should have sufficient knowledge about the interpreter so that he/she is able to predict the set of the successor states from knowledge of the current state. To get and maintain this expertise we have to implement an instructional component and a help system. The specification of the operational knowledge was made in an iterative specification cycle (MÖBUS, 1987a,b,c) (Figure 2).

Figure 2: The Iterative Specification Cycle for Computational or Operational Semantic Knowledge

The first step consisted of the knowledge acquisition phase. The next step led to a ruleset A of 9 main Horn clauses (plus some operator-specific rules). The set contained the minimal abstract knowledge about the interpretation of ABSYNT programs. The abstract structure of a program was formalized by a set of PROLOG facts similar to an approach of GENESERETH (1987, ch. 2.5).

Then we tried an iconic representation of the facts and Horn clauses. Soon we realized that a visual representation according to the recommendations of SIMON & LARKIN (1987) was only possible, if we "enriched" the iconic structure. This means that we had to add iconic elements which were not present in the abstract structure. This lead to an iconic structure which remained unchanged and was used as the interface of the programming environment (FIGURE 3).

Problems occured if we wanted to keep the number of iconic objects fixed during a computation of a recursive program. The postulate of total visibility led to a visual trace with an information overload (FIGURE 6). Time indexed information was not location indexed. So selective attention according to POMERANTZ (1985) was not possible: computional errors were inevitable.

This forced us to relax our requirement to use only a minimal number of object icons. We came up with a relaxed rule set B with 14 main rules (plus operator-specific rules).

The behavior of these rules lead to a new visual trace. Time indexed information was now location indexed so that undesired perceptual grouping could not occur. But computational goals and intermediate results were kept visible only as long as the were absolutely necessary for the ongoing computation.

Empirical considerations showed that the programmer had to reconstruct former computations mentally, because their result disappeared from the screen. So we had to relax the minimum assumption a second time and introduce even more visual redundancy. This was e.g. in accordance with the third principle of FITTER & GREEN (1979).

But there were some other reasons which influenced the decision to modify the ruleset a third time. First, rules were still recursive. This leads in computations to pending rules. The derivation of instructions from recursive rules forces a higher working memory load because of the mental maintanance of a goal stack with return points. Second, if we had derived iconic rules from the abstract rule set B we would have gotten two disjunctive rules. But there is a fair amount of experimental evidence that for humans conjunctive rules are easier to process than disjunctive rules (BOURNE, 1974; HAYGOOD & BOURNE, 1965; MEDIN, WATTENMAKER & MICHALSKI, 1987). So we decided to avoid disjunctive iconic rules.

A third ruleset C was developed with 29 (plus operator specific) rules. Now there was even more redundant iconic information on the screen. This computational behavior was "frozen" in our INTERLIPS/LOOPS implementation.

5.5 The Programming Environment of ABSYNT

The programming environment (KOHNERT & JANKE, 1988) as the result of our specification cycle is shown in figure 3. The screen is split into several regions. On the right and below we have a menu bar for nodes. A typical node is divided into three stripes: an input stripe (top), a name stripe (middle) and an output stripe (bottom). These nodes can be made to constants or variables (with black input stripe) or arelanguage supplied primitive operators or user defined functions.

Figure 3: The programming Environment of ABSYNT

The programmer sees in the upper half of the screen the main worksheet and in the lower half another one. Each worksheet is called frame. The frame is split into a left part: "head" (in german: "Kopf") and into a right part "body" (in german: "Körper"). The head contains the local environment with parameter-value bindings and the function name. The body contains the body of the function.

Programming is done by making up trees from nodes and links. The programmer enters the menu bar with the mouse, chooses one node and drags the node to the desired position in the frame. Beneath the frame is a covered grid which orders the arrangements of the nodes so that everything looks tidy. Connections between the nodes are drawn with the mouse. The connection lines are the "pipelines" for the control and data flow. If a node is missed the programmer is reminded with a phantom node that there is something missing. The editor warns with flashes if unsyntactic programs are going to be constructed: crossing of connections, hiding of nodes etc. The function name is entered by the programmer with the help of pop-up-menus in the root node of the head and the parameters in the leaves of the head.

If the function is syntactically correct, the name of the function appears in the frame title and in one of the nodes in the menu bar so that it can be used as a higher operator. The frame number of the original program is 0. When a problem has to be solved a computation has to be initialised by the call of a function. This call is programmed into the "Start"-Tree. Initial numbers are entered by pop-up-menus in constant nodes in the start tree. This tree has a frame without a name, so that the iconic bars are consistent.

5.6 The integration of rules in the architecture of an PSM or ITS

A very crucial point concerning the "intelligence" of an PSM lies in the quality of the design for the feedback system. In literature two approaches have been proposed. One proposal is the explicit "debugging" approach (BURTON, 1982; VanLEHN, 1981): tracing an error with the help of a diagnostic procedure and an extensive bug collection back to underlying malrules or misconceptions. The other idea rests solely on the specified expert knowledge and a model of human learning (EGAN & GREENO, 1974; SIMON & LEA, 1974; ANDERSON, 1983; VanLEHN, 1987a,b). According to these rule-based theories of human skill acquisition a learner has to be aware of at least two types of information: the current goal within the problem and the conditions under which rules apply. McKENDREE (1987) could show in three experiments, that "goal" information is even more important than "condition" information in promoting learning of skill. This type of feedback design is more simple to implement than the "debugging" strategy. But there are still no experimental comparisons between the two methods.

Either way, we have to specify goals and rules an expert would use when predicting the computational behavior of the ABSYNT interpreter. So in the last part of our paper we show how we try to achieve the design of iconic rules and visual helps.

When should the tutor administer feedback? Our tutorial strategy is guided by "repair theory" (BROWN & VanLEHN, 1980) and follows the "minimalist design philosophy" (CARROLL, 1984a,b).

This means, that if the learner is given less (less to read, less overhead, less to get tangled in), the learner will achieve more. Explorative learning should be supported as long as there is preknowledge on the learner side. Only if an error occurs feedback becomes necessary and information should be given for error recovery.

According to repair theory an impasse occurs, when the student notices that his solution path shows no progress or is blocked. In that situation the person tries to make local patches in his problem solving strategy with general weak heuristics to "repair" the problem situation. In our tutorial strategy we plan to give feedback and helps only, when this repair leads to a second error.

5.7 The Genealogy of Rule Sets Concerning the Operational Semantics of ABSYNT

5.7.1 The First Rule Set A: a Minimal Interpreter in PROLOG for ABSYNT Programs

The specification cycle (Figure 2) led to the first ruleset A. The program is described abstractly by a set of nodes and a set of connections which are represented by PROLOG facts. The nodes possess the attributes frame-name, tree-type, instance-number, name and value. These attributes determine the location, the within structure and the value of the node.

The connections possess the attributes frame, tree, out-instance, in-instance and input-number. They link the outputfield of a node with the inputfield of another node.

Semantic knowledge is moulded into two types of rules. One consists only of one "input" rule and the other of several "output" rules. The "input" rule (Figure 4) contains the knowledge about the migration of computation goals and data between the nodes. The "output" rules contain the knowledge about computations within one node. Because the nodes have different meanings, we need different "output" rules. There is one for each primitive operator, one for the parameters in the tree "head", one for constant nodes, one for parameter nodes in the tree "body", one for the root in the tree "head" and one for the computation of higher (self defined) operators. In the last rule parameters are bound in a parallel fashion to their arguments (call by value) and the new leaves of the tree "head" are put onto the stack. Furthermore we have rules which contain the knowledge to generate roots and leafs or to check nodes with respect to their root or leaf status.

```
input(frame(Frame),tree(Tree),instance(Instance),inputno(Inputno),value(Value))
:-  connection(frame(Frame),tree(Tree),out_inst(Out_inst),in_inst(Instance),in_inst_no(Inputno)),
    output(frame(Frame),tree(Tree),instance(Out_inst),name(Name),value(Value)).

/* IF     there is the goal to compute the value of the input with number Inputno in node Instance in the tree
          Tree in the frame Frame,
   THEN   there is a subgoal to look for a connection, which leads to this input from a yet unknown node
          Out-inst, which is the source of this connection
   AND    there is another subgoal to compute the value of the node Out-Inst
          (this value is then the value of the goal in the IF part of this rule).      */
```

FIGURE 4: The Abstract Input Rule

As a further example we include the "output"-rule for a higher operator (FIGURE 5). This rule describes the call-by-value mechanism.

```
output(frame(Frame),tree(Tree),instance(Instance),name(Name),value(Value))
:-  node_name(frame(Frame),tree(Tree),instance(Instance),name(Name)),
    findall(Argument,input(frame(Frame),tree(Tree),instance(Instance),
        inputno(Inputno),value(Argument)),List_of_arguments),
    set_of(Parameter,(leaf(frame(Name),tree(head),instance(Inst_leaf)),
        node_name(frame(Name),tree(head),instance(Inst_leaf),name(Parameter))),
        List_of_parameters),
    forall(parm_arg_pair(Parm,Arg,List_of_parameters,List_of_arguments),
        (node_name(frame(Name),tree(head),instance(Inst_parm),name(Parm)),
        asserta(node(frame(Name),tree(head),instance(Inst_parm),name(Parm),
        value(Arg))))),
    root(frame(Name),tree(head),instance(Inst_root_head)), ! ,
    output(frame(Name),tree(head),instance(Inst_root_head),name(Name),value(Value)),
    forall(parm_arg_pair(Parm,Arg,List_of_parameters,List_of_arguments),
        (node_name(frame(Name),tree(head),instance(Inst_parm),name(Parm)),
        retract(node(frame(Name),tree(head),instance(Inst_parm),name(Parm),
        value(Arg))))), ! .
```

```
/* IF there is the goal to compute the output value of a higher operator node,
   THEN the following subgoals have to be solved:
        - determine the node name
        - compute all input values of the node
        - determine all parameters of the frame whose name is identical to the node name
        - put the parameter-argument bindings into the new local environment
        - find the head root of the frame
        - compute the output value of the head root
            (this value is then the value of the goal in the IF part of this rule)
        - destroy the local environment
```

FIGURE 5: The Abstract Output Rule for a Higher Operator

As we wrote in 5.4 it is not possible to make a visual represention of facts and rules from set A. We "enriched" the iconic structure by adding some iconic elements. FIGURE 6 demonstrates how the computation of the well-known factorial would look like, if we keep the number of object icons to a minimum: there is onely one frame for recursive computations and intermediate results and computation goals (represented by "?") disappear when no longer needed for the computation.

We see that value and goal stacks are collapsed into the various fields of a node. For the application of an operator we have to select all numbers with the same time index. POMERANTZ (1985) showed that this kind of selective attention is extremely difficult and not trainable. If the function gets more complicated like a tree recursive function, a diagrammatic information of this kind would be completely misleading.

FIGURE 6: Trace within a Hypothetical Environment According to Rule Set A

5.7.2 The second rule set B

We had to modify rule set A because of the following reasons, which result from constraints in the human information processor:

1) any undesired perceptual grouping of information in operator nodes and 2) because we wanted to represent the abstract rules visually iconic rules with disjunctive conditions have to be avoided. Iconic rules with disjunctive conditions require selective attention, which causes matching errors and longer processing time (BOURNE, 1974; HAYGOOD & BOURNE, 1965; MEDIN, WATTENMAKER & MICHALSKI, 1987). This required various modifications of the abstract rules. The "output" rule for a higher operator had to be modified. When a higher operator is called, a fresh copy of the original frame is created. The copies of the frames are ordered by frame number and are put on a frame stack. The arguments are copied in parallel into the parameter leaves of the head. Nodes and connections get the new attribute frame number, too. This allows to location-index time-indexed information. The "output" rule for higher operators is split into three rules corresponding to the call location (start tree, body tree within frame with different name: "abstraction", body tree within frame with same name as called operator: "recursion").

Because we used recursive rules, the control and data flow occured through the parameters. An iconic representation would require that intermediate results should be visible only when they belong to a pending operation. So intermediate results "die" before the corresponding frame "dies". This is not optimal from a cognitive science point of view, because a programmer who wants to recapitulate the computation history has to reconstruct mentally the already obtained results. This leads to higher working memory load for the programmer.

5.7.3 The third set C: objects and rules

The third rule set was motivated by the postulate, that the extent of the intermediate result should not end before the life of a frame ends. This seemed to require only a few changes to the visual interface. But the abstract rules had to be rewritten completely. There is no "input" rule any longer. We have 18 "output" rules instead which all lost their parameters. Like production rules they manipulate the nodes directly via the databasis. Computation goals ("?") and input and output values are written into the nodes. For this purpose a new attribute input-stripe is added to the nodes.

We have included examples for abstract parts of object icons in FIGURE 7 and examples for abstract rules in FIGUREs 8 and 9. The PROLOG facts in FIGURE 7 describe two nodes and two connections in the incomplete program of FIGURE 3. Both nodes are in the root position of the head and the body of the program, respectively.

```
node(frame_name(fac),frame_no(0),tree_type(head),instance_no(2),
    input-stripe([empty]),name_stripe(fac),output_stripe(empty)).
node(frame_name(fac), frame_no(0), tree_type(body), instance_no(11),
    input-stripe([empty,empty,empty]),name-stripe(if), output-stripe(empty)).
connection(frame_name(fac),frame_no(0), tree_type(head), out_instance_no(1),
    in_instance_no(2), input_no(1)).
connection(frame_name(fac),frame_no(0), tree_type(body),out_instance(10),
    in_instance_no(11),input_no(3)).
```

FIGURE 7: An example for Abstract Nodes and Connections

```
output :-
    first_solution(focus(frame_name(Frame_name),frame_no(Frame_no),tree_type(Tree_type),
        instance_no(Instance_no))),
    node(frame_name(Frame_name),frame_no(Frame_no),tree_type(Tree_type),
        instance_no(Instance_no),input_stripe(Input_stripe),name_stripe(Name_stripe),
        output_stripe(Output_stripe)),
    Tree_type = start,
    Output_stripe = ? ,forall(on(Element,Input_stripe),value(Element)),
    higher_operator(name(Name_stripe)),
    New_frame_no = 1,
    not(exist_frame(frame_name(Name_stripe),frame_no(New_frame_no))),
    copy_frame(frame_name(Name_stripe),frame_no(New_frame_no)),
    root(frame_name(Name_stripe),frame_no(New_frame_no),tree_type(head),
        instance_no(Instance_no_root_head)),
    modify(frame_name(Name_stripe),frame_no(New_frame_no),tree_type(head),
        instance_no(Instance_no_root_head),output_stripe( ? )),
    modify(frame_name(Name_stripe),frame_no(New_frame_no),tree_type(head),
        instance_no(Instance_no_root_head),input_stripe(Input_stripe)),
    bind_parameter(input_stripe(Input_stripe),frame_name(Name_stripe),frame_no(New_frame_no)),
    asserta(focus(frame_name(Name_stripe),frame_no(New_frame_no),tree_type(head),
        instance_no(instance_no_root_head))),
output.


/*  IF  the focus is on the node
        and that node has the following features:
        (1) The node is located in the start tree.
        (2) The output_stripe of the node contains a "?".
        (3) The input_stripe of the node contains all input values.
        (4) The node name is a higher operator.
        and there is no related frame.
    THEN create the related frame.
        Determine it`s head root, put a "?" into it`s output_stripe.
        Transfer the input_stripe of the node to the head root.
        Bind the parameters and put
        the new focus "head root" onto the stack.  */
```

FIGURE 8: Abstract Rule 8 (First part of Call-by-Value, call in start tree)

```
output :-
    first_solution(focus(frame_name(Frame_name),frame_no(Frame_no),tree_type(Tree_type),
        instance_no(Instance_no))),
    node(frame_name(Frame_name),frame_no(Frame_no),tree_type(Tree_type),
        instance_no(Instance_no),input_stripe(Input_stripe),name_stripe(Name_stripe),
        output_stripe(Output_stripe)),
    Tree_type = start,
    Output_stripe = ? ,forall(on(Element,Input_stripe),value(Element)),
    higher_operator(name(Name_stripe)),
    New_frame_no = 1,
    check_value_of_related_frame(related_frame_name(Name_stripe),
        related_frame_no(New_frame_no),
        related_frame_value(Output_stripe_root_head)),
    modify(frame(Frame_name),frame_no(Frame_no),tree_type(Tree_type),
        instance_no(Instance_no),output_stripe(Output_stripe_root_head)),
    delete_frame(frame_name(Name_stripe),frame_no(New_frame_no)),
    retract(focus(frame_name(Frame_name),frame_no(Frame_no),tree_type(Tree_type),
        instance_no(Instance_no))),
output.


/*  IF  the focus is on the node
        and that node has the following features:
        (1) The node is located in the start tree.
        (2) The output_stripe of the node contains a "?".
        (3) The input_stripe of the node contains all input values.
        (4) The node name is a higher operator.
        and the head root of the frame related to the node contains a value.
    THEN  transfer this value into the output_stripe of the node.
        Delete the frame.
        Pop the focus.  */
```

FIGURE 9: Abstract rule 9 (Second part of Call-by-Value, call in start tree)

5.9 Iconic Rules for the Instructional Component and a Help System

On the basis of rule set B and C we developed iconic rules to describe the operational behavior of the interpreter. Because of space restrictions we can only show the rules from set C (FIGUREs 10, 11), which are representation of the production-like PROLOG rules of FIGUREs 8 and 9. At the present moment these rules are not implemented in an instructional or help component. But they are used successfully in experiments where novices are requested to predict the computation steps of the interpreter. Each rule consists of a description of the triggering situation and a description of the situation after the rules has been applied. We tried to make the rules self-explanatory as much as possible. So we need only a short introduction to explain the syntax of the iconic rules.

FIGURE 10: Iconic Rule on the Basis of Abstract Rule 8 in FIGURE 8
(First part of Call-by-Value, call in start tree)

FIGURE 11: Iconic Rule on the Basis of Abstract Rule 9 in FIGURE 9
(Second part of Call-by-Value, call in start tree)

We found that the sentential information is used only in situations when an impasse in the computational process occurs. Novices were able to predict the interpreter in less than five hours learning time.

The next step is to implement the rules for instructional purposes so that the interpreter becomes selfexplaining. This situation can arise when the student is uncertain about the calculation process of the machine.

6. Summary

We reviewed the literature on CAI and ICAI systems and discussed some design principles from a cognitive science view. We discussed student models and plan recognition as important research goals but also stressed the importance of a rather neglected topic: design of the "student input". We need very careful and knowledge-crafted instructions, interfaces and helps so that our plan diagnosis and student modelling components get the opportunity to work satisfactorily. As an example we introduced iconic rules which transmit information in a diagrammatic form. Only if no phase of the design process is omitted we will achieve true "intelligent" CAI.

8. Appendices

8.1 Appendix A: Instructions for reading the book "Analysis of Behavior" (HOLLAND & SKINNER, 1961,p.viif., p.1f.) with an excerpt of part I:

To the Student

With this book the student should be able to instruct himself in that substantial part of psychology which deals with the analysis of behavior - in particular the explicit prediction and control of behavior of people. The practical importance of such a science scarcely needs to be pointed out, but understanding and effective use of the science require fairly detailed knowledge. This program is designed to present the basic terms and principles of the science. It is also designed to reveal the inadequacy of popular explanations of behavior and to prepare the student for rapidly expanding extensions into such diverse fields as social behavior and psychopharmacology, space flight and child care, education and psychotherapy. This book is itself one application of the science.

How to Use the Book

The material was designed for use in a teaching machine. The teaching machine presents each
item automatically. The student writes his response on a strip of paper revealed through a window in the machine. He the operates the machine to make his written response inaccessible, though visible, and to uncover the correct response for comparison.

Where machines are not available, a programmed textbook such as this may be used. The correct response to each item appears on the following page, along with the next item in the sequence. Read each item, write your response on a separate sheet of paper, and then turn the page to see whether your answer is correct. If it is incorrect, mark an "x" beside it. Then read and answer the next question, and turn the page again to check your answer.

Writing out the answer is essential. It is also essential to write it *before* looking at the correct answer. When the student, though well-intentioned, glances ahead without first putting down an answer of his own, he commits himself to only a vague and poorly formulated guess. This is not effective and in the long run makes the total task more difficult.

It is important to do each item in its proper turn. The sequence has been carefully designed, and occasional apparent repetitions or redundancies are there for good reason. Do not skip. If you have undue difficulty with a set, repeat it before going on to the text. A good rule is to repeat any set in which you answer more than 10 per cent of the items incorrectly. Avoid careless answers. If you begin to make mistakes because you are tired or not looking at the material carefully, take a break. If you are not able to work on the material for a period of several days, it may be advisable to review the last set completed.

The review sets will help you to find your weaknesses. When you miss an item in a review set, jot down the set number given in the answer space and review that set after you have completed the review set.

Please insert pages 1 + 2 from HOLLAND & SKINNER here

8.2 Appendix B: Excerpt from FRIEDMAN's and FELLEISEN's "The Little LISPer" (1987)

Please insert pages 3 + 4 from FRIEDMAN & FELLEISEN here

8.3 Appendix C: A dialog with ANDERSON's LISP tutor

(ANDERSON & REISER, 1985; ANDERSON & SKWARECKI, 1986; ANDERSON, 1987)

Appendix C

8.4 Appendix D: A verbal description of the operational semantics of the recursive "calculation sheet" machine, an example of a diagrammatic form program and the corresponding trace

8.4.1 A verbal instruction

The verbal instruction stems from the pages 110-113 of BAUER & GOOS (1982).

Appendix D.1

8.4.2 A "calculation sheet" program

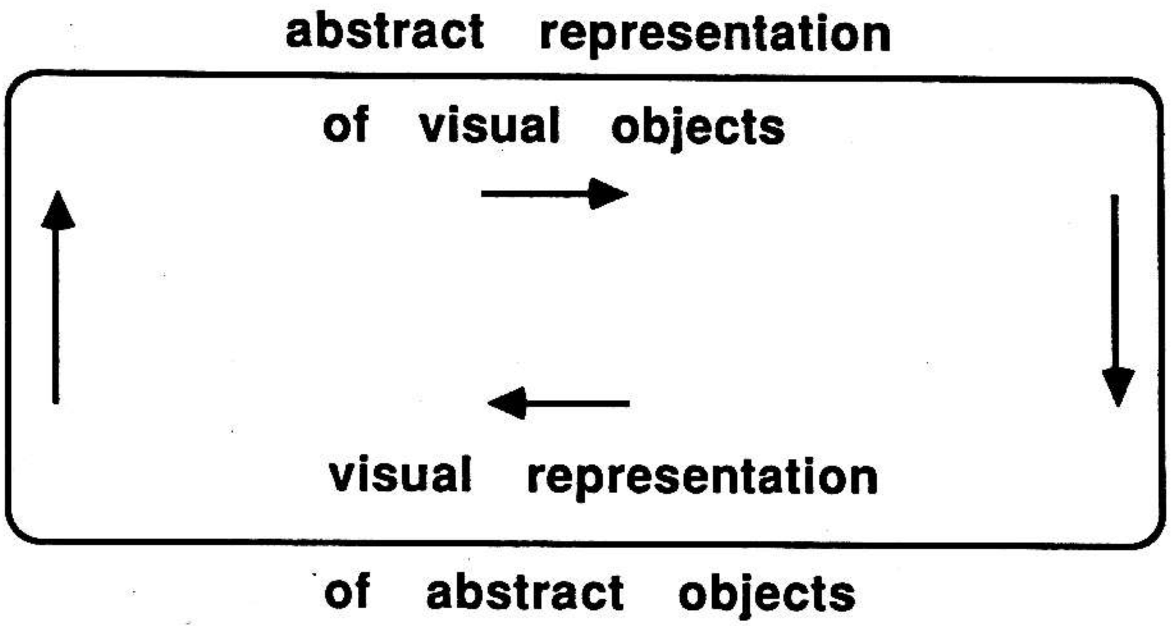
This program is shown on page 104 of BAUER & GOOS (1982).

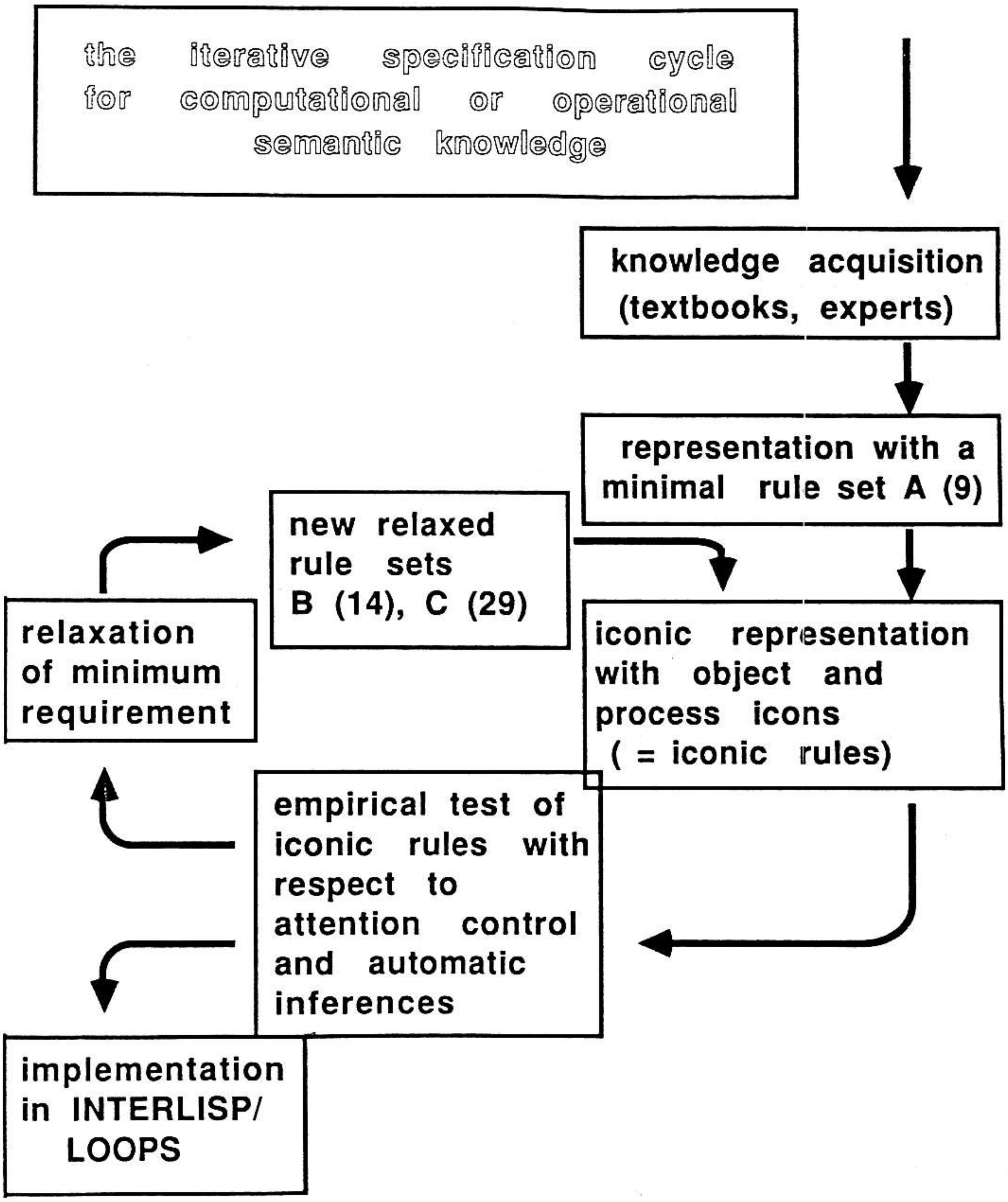
Appendix D.2

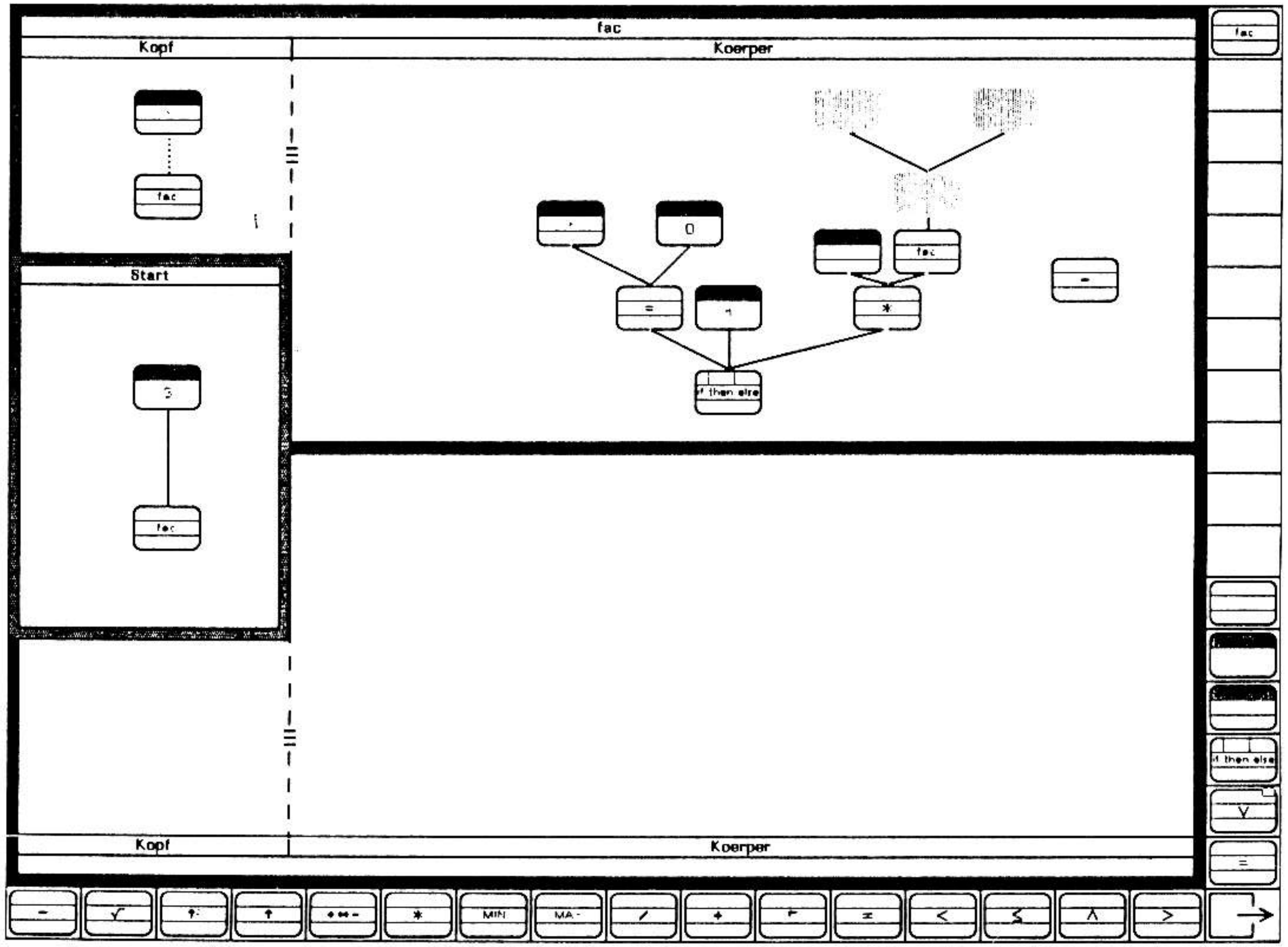
8.4.2 A trace in computing the factorial(3)

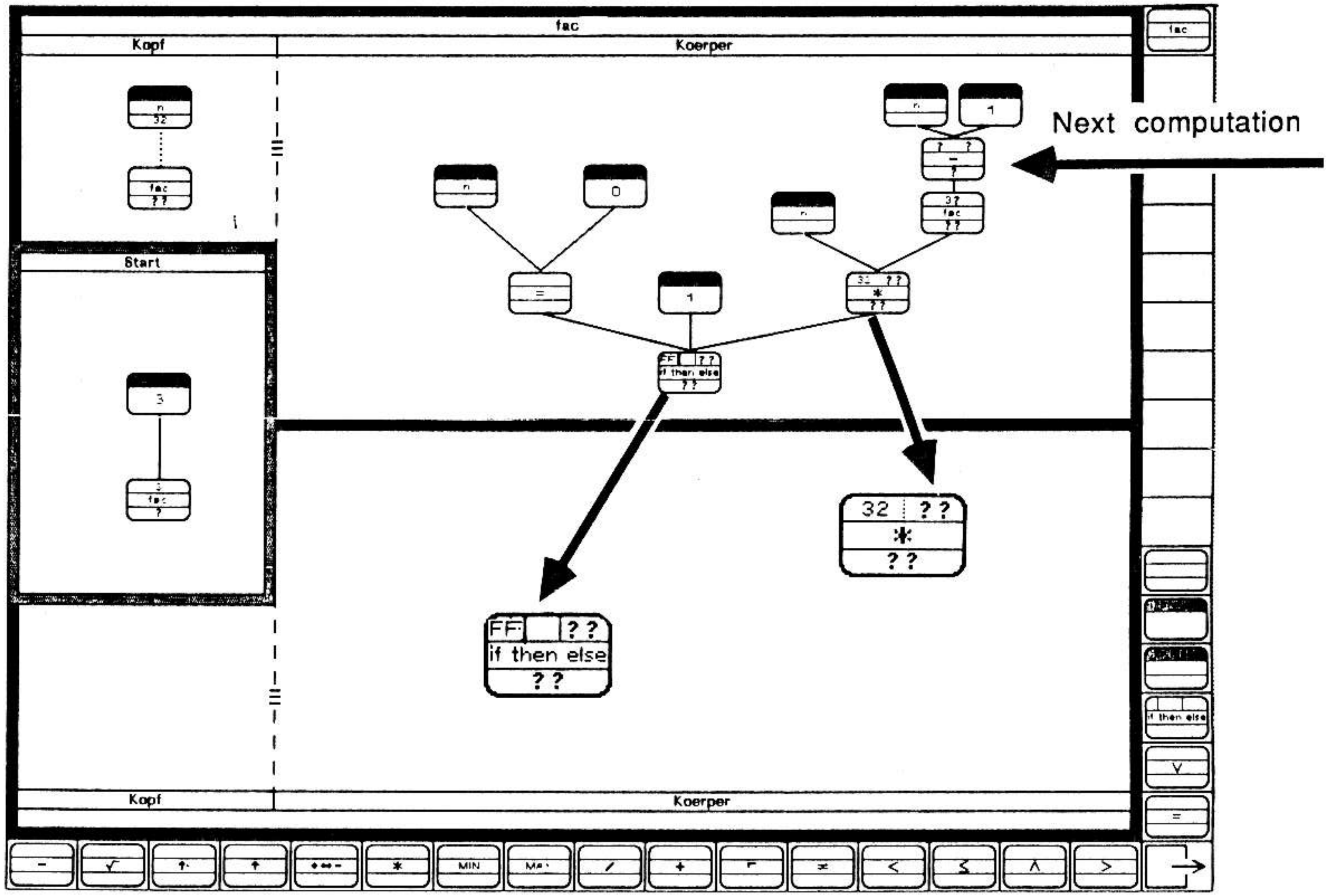
This trace can be found on page 112 of BAUER & GOOS (1982).

Appendix D.3


abstract representation
of visual objects
visual representation
of abstract objects

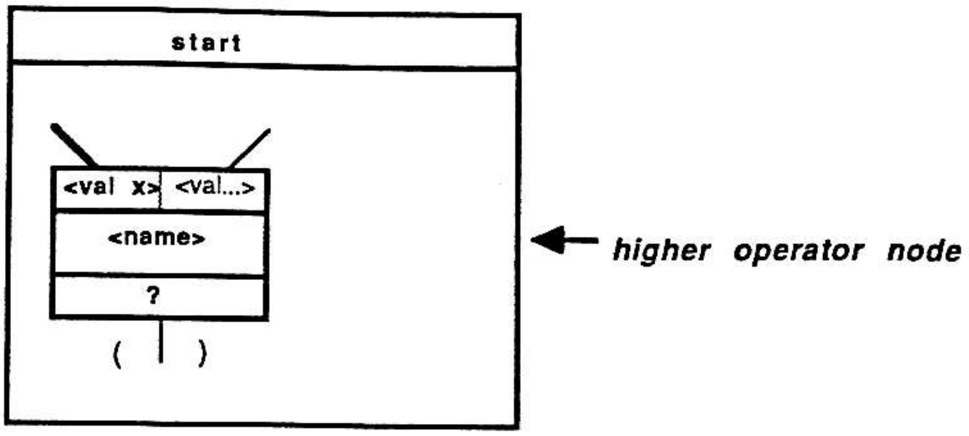

the iterative specification cycle for computational or operational semantic knowledge
knowledge acquisition (textbooks, experts)
representation with a minimal rule set A (9)
iconic representation with object and process icons (= iconic rules)
empirical test of iconic rules with respect to attention control and automatic inferences
relaxation of minimum requirement
new relaxed rule sets B (14), C (29)
implementation in INTERLISP/ LOOPS


fac
Kopf
Koerper
fac
Start
3
fac
0
=
1
*
fac
if then else
-
Kopf
Koerper
fac
if then else
V
=
-
√
*
MIN
MA-
/
+
<
≤
∧
>

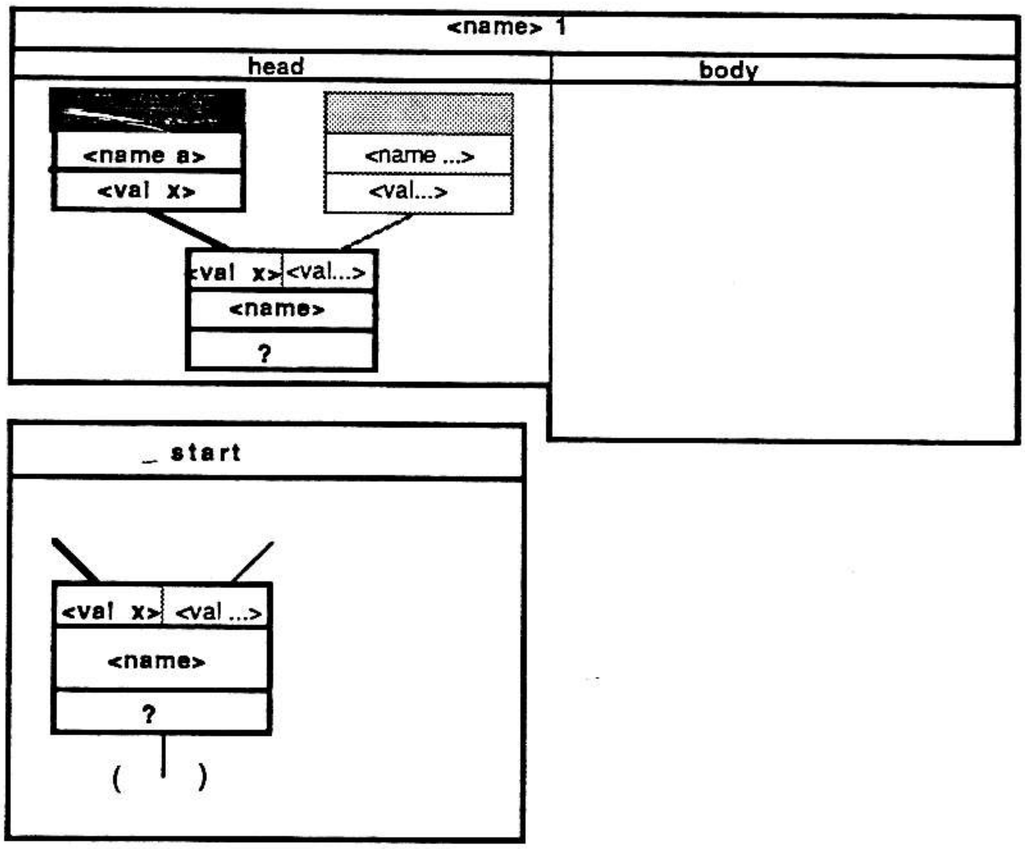

fac
Kopf
Koerper
Start
Next computation
FF ??
if then else
??
32 ??
*
??
MIN
MAX

Rule 8 : **Computing higher operator node in start tree: Making frame, binding parameters, passing goal to head root**

Situation: A "?" is in the output stripe of a higher operator node.
The node is part of the start tree.
The input stripe of the node contains only values.
There is no frame with the operator's name.

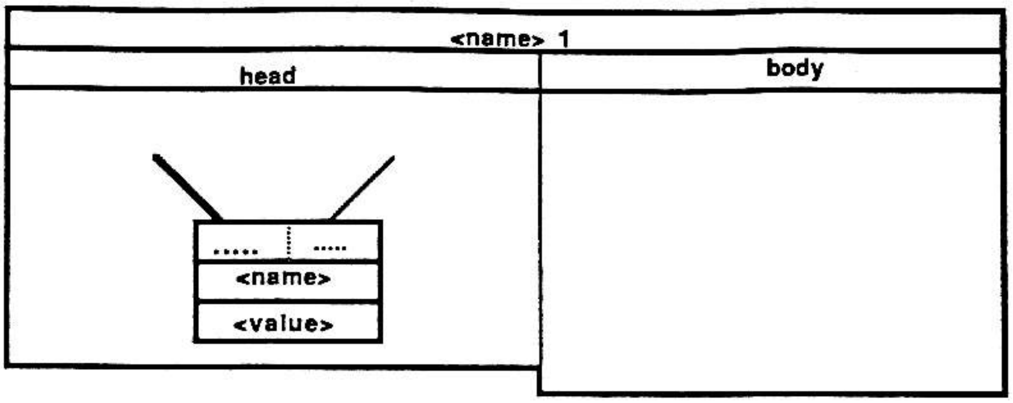


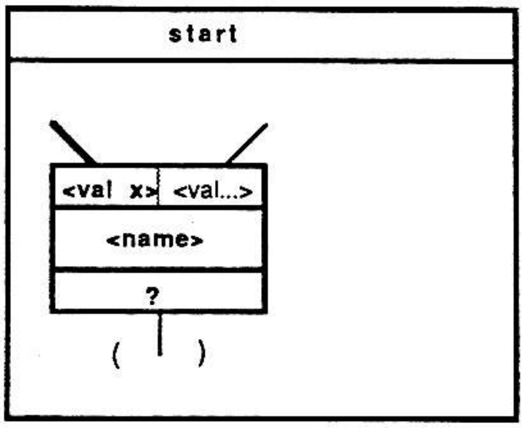
Action: Make a frame with the operator's name and frame-no. 1.
Write a "?" into the output stripe of the head root of this frame.
Write the input values of the higher operator node into the input stripe of this head root, preserving their order.
Write the value of each input field into the output field of the linked head leaf.



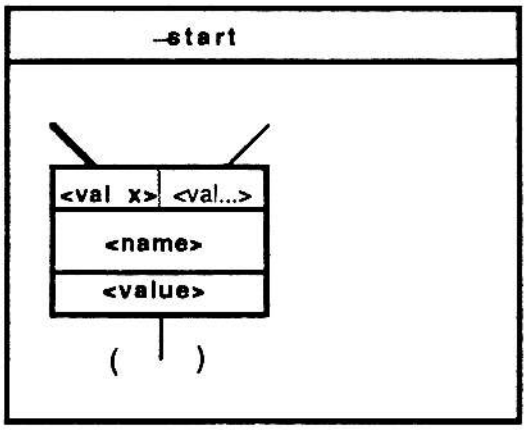
Rule 9 : **Fetching value for higher operator node in start tree**

Situation: A "?" is in the output stripe of a higher operator node.
The node is part of the start tree.
The input stripe of the node contains only values.
There is a frame with the operator's name and frame-no. 1.
The output stripe of the head root of this frame has a value.





Action: Write this value into the output stripe of the higher operator node and delete the frame specified above.



2.3.3 Die (rekursive) Formularmaschine

Der Gang der Berechnung einer Rechenvorschrift ist bis auf Kollateralität durch ein zugehöriges Formular festgelegt. Kommt im Formular selbst wieder eine Rechenvorschrift als Operation vor[44], so ist ein Formular dieser Rechenvorschrift anzulegen (‚Aufruf') und deren Ergebnis schließlich rückzuübertragen. Dies gilt auch für eine rekursiv definierte Rechenvorschrift - mit der Besonderheit, daß im Lauf der Berechnung entsprechend den rekursiven Aufrufen weitere Exemplare des Formulars eben dieser Rechenvorschrift benötigt werden.

Zu jedem Aufruf werden in ein neues Exemplar des Formulars zunächst linksseitig die jeweiligen Argumentwerte eingetragen (*'call by value'*). Man nennt jedes solche Exemplar eine **Inkarnation** der Rechenvorschrift; um die Übersicht zu behalten, kann man die Inkarnationen und die entsprechenden Aufrufe im Verlauf der Berechnung durchnumerieren.

Für den rekursiven Fall ist es nun besonders bedeutsam, daß die Fallunterscheidung eine arbeitssparende Auswahl trifft: nachdem die Parameterbezeichnungen durch die linksseitig festgestellten Argumentwerte ersetzt sind, werden daher auf dem Urformular und allen folgenden Inkarnationen möglichst zuerst die Bedingungen ausgewertet und sodann die unzulässigen Zweige gekappt. Die Rekursion endet mit Inkarnationen, in denen kein Zweig mehr verbleibt, der einen rekursiven Aufruf enthält. Die ganze Berechnung **terminiert** (für einen bestimmten Parametersatz), wenn sie nur endlich viele Inkarnationen benötigt.

Die Tätigkeit eines Menschen, der auf diese Weise mit Formularen arbeitet, kann in einsichtiger Weise auch mechanisiert werden. Man gelangt so zum Begriff einer rekursiven (Gedanken-)Maschine, der **Formularmaschine**, in der die volle Freiheit der Berechnung noch erhalten ist. Man beachte, daß ein neues Exemplar eines Formulars auch dann angelegt wird, wenn die gleichen Argumente schon einmal aufgetreten sind: die Formularmaschine macht (auf der hier geschilderten Stufe) von einer möglichen Mehrfachverwendung eines Ergebnisses keinen Gebrauch.

Das oben erwähnte Kappen von Zweigen ist insbesondere dann ohne weiteres möglich, wenn in den Bedingungen keine rekursiven Aufrufe vorkommen. Noch übersichtlicher ist der Fall der **linearen Rekursion**, bei der außerdem in den einzelnen Zweigen der Fallunterscheidung höchstens ein rekursiver Aufruf vorkommt; dann wird nämlich in jeder Inkarnation höchstens eine neue Inkarnation angestoßen. Fast alle bisher behandelten Beispiele fallen übrigens in diese Klasse.

Für *fac* von 2.3.2 arbeitet eine Formularmaschine wie in Abb. 59 angegeben. Typisch für die Rekursion ist das ‚Nachklappern' der Berechnung: Erst wenn die Rekursion mit der Inkarnation $fac^{(3)}$ geendet hat, werden die zurückgestellten Berechnungen in $fac^{(2)}$, $fac^{(1)}$ und $fac^{(0)}$ durchführbar und auch durchgeführt[45]; das Urformular $fac^{(0)}$ liefert schließlich das Endergebnis. Das Nachklappern kann in besonders gelagerten Fällen von Aufrufen zu einem bloßen Rückübertragen der Ergebnisse der einzelnen Inkarnationen degenerieren, wie Abb. 60 für das Beispiel *gcd1* (15, 9), vgl. 2.3.2 zeigt. Ein solcher Aufruf heißt **schlicht**. Wenn in linearer Rekursion ausschließlich schlichte Aufrufe vorliegen, spricht man von **repetitiver Rekursion.**

Bei linear rekursiven Rechenvorschriften ist - abgesehen von der sonstigen Kollateralität des Formulars - die Reihenfolge, in der die benötigten Inkarnationen angestoßen werden, eindeutig bestimmt. Dies ist nicht notwendig so im allgemeinen Fall: wenn in einem Zweig mehrere Aufrufe vorkommen, so erlaubt die Kollateralität unter Umständen verschiedene Reihenfolgen und sogar Parallelarbeit.

---

[44] Für primitive, d. h. den zugrundeliegenden Rechenstrukturen entstammende Operationen ist kein Formular erforderlich.

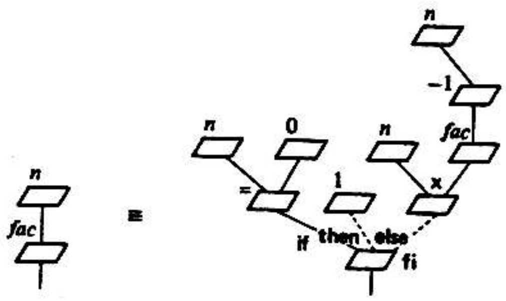


Abb. 54. Formular von *fac*

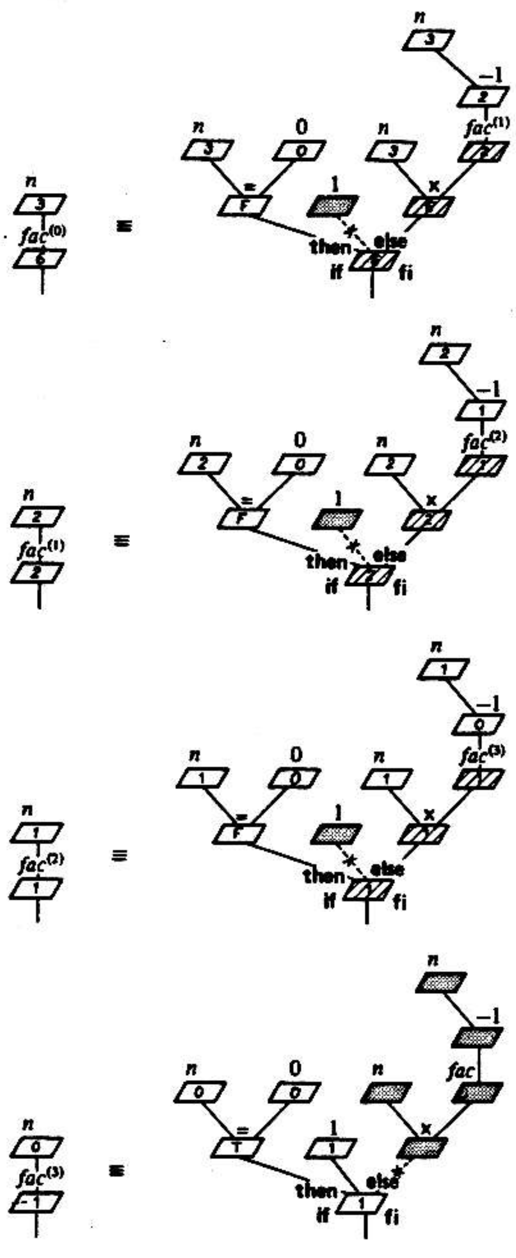


Abb. 59. Arbeitsweise der Formularmaschine am Beispiel *fac*(3)