Man–Computer Interaction Research
MACINTER-II
F. Klix, N.A. Streitz, Y. Waern, and H. Wandke (Editors)
© Elsevier Science Publishers B.V. (North-Holland), 1989

KNOWLEDGE SPECIFICATION AND INSTRUCTION FOR A VISUAL COMPUTER LANGUAGE

Claus MOEBUS and Olaf SCHROEDER

Project ABSYNT[+]
FB 10, Informatik
Unit on Tutoring and Learning Systems
University of Oldenburg
Oldenburg, FRG

One difficult problem in the development of intelligent computer aided instruction (ICAI) is the proper design of instructions and helps. The problem arises because knowledge diagnosis largely depends on what kind of information is given to the student.
This paper adresses the question of developing instructional and help material concerning the operational knowledge for a visual, functional programming language, ABSYNT. The goal of our project is the construction of a problem solving monitor (PSM) for ABSYNT. First, we will explain our motivation for choosing and developing this task environment. Then, we will describe the programming environment of ABSYNT. Next, we will illustrate some difficulties that arose when we used a first, only verbally specified, nonvisual description of the operational knowledge as instructional material. In particular, it was not clear whether this description was complete and errorfree, and it provided no framework for semantic-bug analyses. Finally, the process is described by which we generate rule-based specifications of the operational knowledge and visual instructions and helps. This iterative specification cycle led to two alternative sets of iconic rules which describe the operational knowledge of ABSYNT to the student.

INTRODUCTION

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 (1); Attisha and Yazdani (2); Brown and Burton (3); Bundy (4); Burton (5); VanLehn and Brown (6); Young and O'Shea (7)), in quadratic equations (S'Shea (8), (9)), in simple algebra problems (Sleeman (10, (11), (12), (13), (14), in geometry (Anderson (15); Anderson, Boyle, Farrell and Reiser (16); Anderson, Greeno, Kline and Neves (17)) and in computer programming (Anderson (18, (19); Anderson, Farrell and Sauers (20), (21); Anderson and Reiser (22); Anderson and Skwarecki (23); Johnson (24); Johnson and Soloway (25), (26); Soloway (27); Wertz (28), (29), (30)). 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 and Sussman (31); Ghezzi and Jazayeri (32); Henderson (33), (34)). Furthermore, there is some evidence that there is a strong correspondency between programmer's goals and use of functions (Pennington (35); Soloway (27); Johnson and Soloway (25)). 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 (27)). 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 and Green (36); Green, Sime and Fitter (37); Payne, Sime and Green (38); Larkin and Simon (39)). 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 (Larkin and Simon (39)). "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 (40); Chase (41)). 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. On proposal is the explicit "debugging" approach (Burton (5); VanLehn (42)): 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 and Greeno (43); Simon and Lea (44); Anderson (45); VanLehn (46), (47)). 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 (48) 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.

When should the tutor administer feedback? Our tutorial strategy is guided by "repair theory" (Brown and VanLehn (49))and follows the "minimalist design philosophy" (Carroll (50),(51)).

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.

2. THE PROGRAMMING ENVIRONMENT OF ABSYNT

The programming environment of ABSYNT was developed in our project, basing on the "calculation sheet machine" (Bauer and Goos (52)). The complete programming environment is implemented in INTERLISP and the object-orientated language LOOPS (Janke and Kohnert (53); Kohnert and Janke (54)) to have a system with direct manipulation capabilities which are absolutely necessary prerequisites for our system (Fähnrich and Ziegler (55); Hutchins, Hollan and Norman (56), Shneiderman (57), (58)). Following Shu's (59) dimensional analysis, ABSYNT is a language with high visual extent, low scope and medium level. ABSYNT consists of three modes: a programming mode, a trace mode, and a prediction mode (Kohnert and Janke (54)).

2.1. The Programming Mode

The programming mode is shown in Figure 1. 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 are language supplied primitive operators or user defined functions.

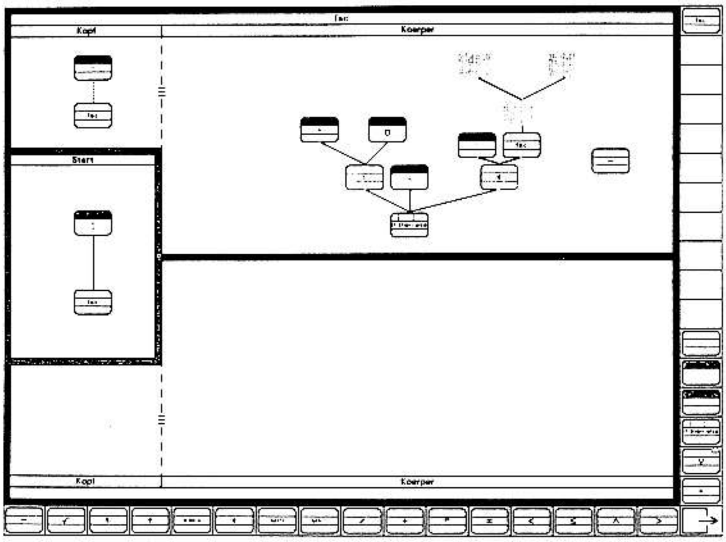


FIGURE 1
The Programming Mode of ABSYNT.

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. When a problem has to be solved a computation has to be initialized 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.

The design of the programming mode is motivated by the operational knowledge for ABSYNT (Möbus and Thole (60)). That is, the features and distinctions necessary for the operational knowledge (i.e., frame name and frame number, division of a node into an input stripe, a name stripe and an output stripe) are visualized in the programming mode as well as in the other modes of the programming environment. We gathered converging evidence for the usefulness of this design by analyzing syntactic and semantic bugs in a feasability study based on the calculation sheet machine (Colonius, Frank, Janke, Kohnert, Möbus, Schröder and Thole (61)).

2.2. Trace Mode and Prediction Mode

If the user has programmed a start tree for his program, he can run the program and get a trace for it. The design of the trace is a result of our iterative specification cycle of developing abstract rules and process icons (to be explained in part 4 of this paper). In case of recursive programs, the actually computed frame is in the upper half of the screen. The lower half shows the frame one level deeper in the stack, so that the recursive call stays visible.

As an experimental tool of the ABSYNT environment, there is also a prediction mode. Here the user can predict the actions of the interpreter, that is, compute ABSYNT-programs by himself, so he can acquire the operational knowledge for ABSYNT. In part 4 we explain the instruction and help material for acquiring this knowledge.

3. PRELIMINARY INSTRUCTIONAL MATERIAL AND SEMANTIC BUGS

Our starting point for developing a functional, visual programming language was the "calculation sheet machine" (Bauer and Goos (52); Möbus (62)). In a first step, we reconstructed it in order to obtain a paper-and-pencil-version for doing explorations. Part of this reconstruction was a verbal specification of the syntax and the operational knowledge, illustrated by simple programs and trees. The essence of the verbal specification of the operational knowledge is shown in Figure 2.

Computation of Calculation Sheet Programs:

A Calculation Sheet Program consists of a Long Form and a Short Form, which may replace the Long Form. Before the calculation starts, write the start value(s) in left-to-right-order into the parameter nodes of the Short Form. Now look at the Long Form. Write into every parameter node the value which is in the parameter node with the same name in the Short Form.
You can start the computation when every node without input connections in the Long Form has a value.

Computation rules:

1. Start with the bottom-most node of the Long Form.

2. Does this node have no input connection? If so, its content is its value.

3. Does this node have at least one input connection? If so, then test whether it is a branching node ("if-then-else").

   a) If it is a branching node: The node connected to its leftmost input connection must get a value according to computation rules 2 to 4.

      - If this value is "True", then the node connected to the middle input connection of the branching node ("then-node") must get a value according to the computation rules 2 to 4.

      - If this value is "False", then the node connected to the rightmost input connection of the branching node ("else-node") must get a value according to the computation rules 2 to 4.

   b) If it is not a branching node: The node is some different operator node.
      Every node connected to the input connection(s) of this node must get a value according to the computation rules 2 to 4.
      The operator is then computed according to computation rule 4, and the obtained value is written into the operator node.

4. a) If the operator is in the following list of primitive operators, it will be computed in one of the following ways:

      + takes at least two numbers. The numbers are added.
        takes at least two numbers. The numbers are multiplied.

      ...

   b) If the operator is not in this list ("unknown operator"), it is the name of a Calculation Sheet Program.

- Make a copy of this program.
- Write the value(s) of all node(s) connected to the input connection(s) of the unknown operator in left-to-right-order into the parameter node(s) of the Short Form of the copy.
- Compute the Long Form of the copy according to the rules given above.
- Write the obtained value into the bottom-most node ("name node") of the Short Form of the copy.
- Write the value into the unknown operator node of the previous Calculation Sheet Program.

FIGURE 2
Part of the initial verbal specification of the operational knowledge.

With this first version of the language, we performed a feasability study. Its aims were

- getting hints for the design of the language and the interface
- collecting syntactic and semantic bugs
- studying the memory representations of example programs (cf. Hoc (63); Adelson (64); Brooks (65); Letovsky (66); Rist (67)) in order to find reasons for bugs and conditions under which they occur.

In the sessions, the subject computed calculation sheet programs with paper and pencil. Moreover, they reproduced them and compared different programs. The verbal specification of the syntactic and operational knowledge was provided as help.

The subjects had no programming knowledge, but prior to the sessions they were introduced to the calculation sheet machine and to the verbal specification. The programs can be partially ordered in accordance with the programming concepts which they exemplify (see Figure 3).

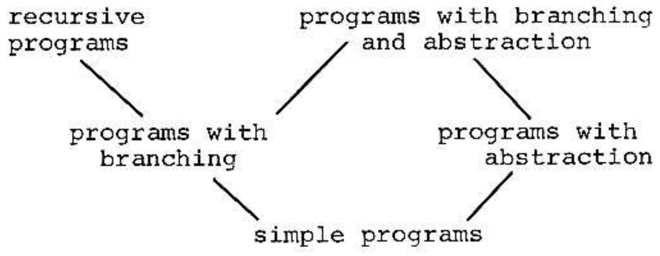

recursive programs
programs with branching and abstraction
programs with branching
programs with abstraction
simple programs

FIGURE 3
Partial ordering of "calculation sheet" program.

A detailed description of the feasability study is provided in Colonius, Frank, Janke, Kohnert, Möbus, Schröder and Thole (68). Here we will focus on the semantic bugs. Altogether, the subjects computed 75 programs. 18% of the computations contained bugs. It follows a short description:

- Buggy computation of primitive operators (except branching): In some case, the arithmetic operator "-" was first used correctly, but then additionally interpreted as the sign for the obtained result (i.e., 7 - 3 = -4). This bug supported a decision concerning the design of operator nodes in ABSYNT.

- Buggy computation of the branching operator (if-then-else): In most cases of buggy computation, the result of the predicate was taken as the result of the branching operator, although these subjects computed the then-branch resp. else-branch correctly.

- Buggy computation of abstraction: When an abstract operator appeared more than once in a program, it was computed correctly for the first time. The obtained result was then taken as the result of other occurrences of the abstract operator, too, in spite of different arguments.

- Buggy computation of recursion: In some cases the recursive call was treated as a primitive operator (i.e., addition). In some other cases, the subjects interrupted the computation when reaching the recursive call. Then they started computing the other branch of the branching operator. In still other cases, the recursive calls were computed correctly, but the result of the deepest-level incarnation was taken as the result of the whole program. Postponed computations were ignored.

However, this collection of semantic bugs gave rise to the following problems:

- It is unclear whether the bugs arose because of ambiguities in the instructional material (the verbal description of the operational knowledge). Therefore, we cannot be certain if this description can be viewed as the semantic "expert" knowledge, which in our opinion is a prerequisite for a user of our language to plan and debug efficiently.

- The verbal description of the operational knowledge is a poor base for a more detailed and systematic description of the observed bugs in terms of missing or wrong pieces of knowledge.

- It seems unnatural to construct a verbal specification of the operational knowledge for a visual programming language. The design of a visual language has to be based on the concept of generalized icons (Chang (69)), which can be divided into object icons and process icons. Object icons define the representation of static language constructs, whereas process icons specify the representation of data flow and control flow (see also Möbus and Thole (60)).

Therefore, we decided to use a runnable specification (Davis (70)) of the language, which was implemented as rule sets in the course of our project, as a foundation for constructing process icons. These process icons may then be used as instructional and help material for the operational knowledge.

Moreover, with a first version of this runable specification (rule set A, see below) we realyzed the observed semantic bugs described above (Colonius, Frank, Janke, Kohnert, Möbus, Schröder and Thole (68)). This made clear that the rules can provide a systematic account of most of the bugs. In this view they could be described as

- missing rules (i.e., buggy computation of a primitive operator).

- overgeneralized rules: Components of a rule are missing (i.e., no distinction between different calls of the same function is made. This would lead to ignoring postponed computations in recursive calls, as described above).

- overly restricted domains of rules: The appropriate rule is not applied in certain situations (i.e., the general rule for dealing with function calls is not applied in case of recursive calls. This would lead to an impasse followed by tinkering (Brown and VanLehn (49)). So, treating the recursive call as a primitive operator or switching to the other branch of the if-then-else-operator (see above) could be viewed as such attempts to repair the situation).

On the other hand, the acquisition of the operational knowledge could be viewed as acquisition, refinement and generalization (cf. Norman (71); Goldstein (72) of the rules (Colonius, Frank, Janke, Kohnert, Möbus, Schröder and Thole (61)).

4. CONSTRUCTION OF IMPROVED INSTRUCTIONAL MATERIAL: PROCESS ICONS

The specification of the operational knowledge was made in an iterative specification cycle (Möbus (72), (74); Möbus and Thole (60) (Figure 4)).

The first step consisted of the knowledge acquisition phase. The next step led to a rule set 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 was formalized by a set of PROLOG facts similar to an approach of Genesereth and Nilsson (75)).

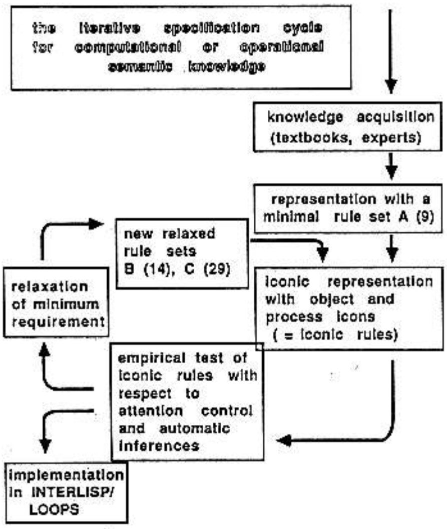

the iterative specification cycle for computational or operational semantic knowledge
knowledge acquisition (textbooks, experts)
representation with a minimal rule set A (9)
new relaxed rule sets B (14), C (29)
relaxation of minimum requirement
iconic representation with object and process icons (= iconic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

FIGURE 4
The iterative specification cycle for operational semantic knowledge.

4.1. Rule Set A and Process Icons

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 output-field 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 5) 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 into 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 5
The Abstract Input Rule.

As a further example we include the "output"-rule for a higher operator (Figure 6). 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(Paremeter))),
         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))))),!.

/+ 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 6
The Abstract Output Rule for a Higher Operator

In the next step of the specification cycle, we tried an iconic representation of the facts and Horn clauses of rule set A. Thereby, we kept in mind design principles which are motivated by Pomerantz (40) and Larkin and Simon (39). 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. In our design a location is identical with a visual object. These insights were supported by the formal analysis of Larkin and Simon (39). They showed under what circumstances a diagrammatic representation of information consumes less computational resources as an informational equivalent sentential representation. Figure 7 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 only 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. Time indexed information was not location indexed. Pomerantz (40) 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. Here, the postulate of total visibility would lead to a visual trace with an information overload. Computational errors would be inevitable.

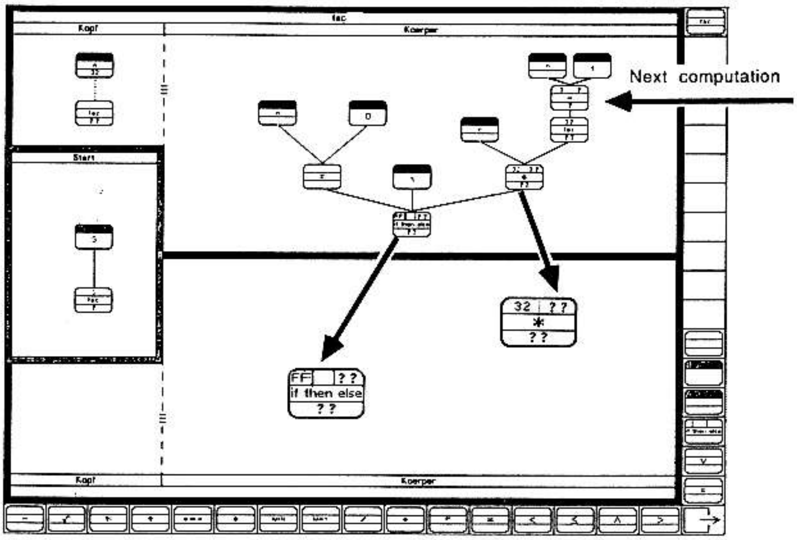


FIGURE 7
Trace within a Hypothetical Environment According to Rule Set A.

So we realized that a visual representation of the facts and Horn clauses of rule set A according to the recommendations of Pomerantz and Larkin and Simon 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.

A second reason for an enrichment and, thereby, a modification of rule set A, was that rule set A led to iconic representations with disjunctive rules. Iconic rules with disjunctive conditions require selective attention, which causes matching errors and longer processing time (Bourne (76); Haygood and Bourne (77); Medin, Wattenmaker and Michalski (78)).

So we had to modify rule set A because of the following reasons, which result from constraints in the human information processor: We wanted to avoid 1. any undesired perceptual grouping of information in operator nodes, 2. iconic rules with disjunctive conditions, and 3. visual hinding of dynamic successor frames already put on a stack.

4.2. Rule Set B and Process Icons

As shown above, an attempt to visually represent rule set A forced us to relax our requirement to use only a minimal number of object icons (see the iterative specification cycle, Figure 4). This required various modifications of the abstract rules. We came up with a relaxed rule set B with 14 main rules (plus operator-specific rules).

In rule set B, the "output" rule for a higher operator is modified. When a higher operator is called, a fresh copy of the original frame is created. In order to avoid a only partly visible "spaghetti"-stack in the sense that from one frame several new successor frames could be opened by calling "higher" operators, we allowed only one call per frame at the moment. This results in a depth-first search in the call tree. The copies of the frames are ordered by frame number and 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 two rules corresponding to the call location (start tree, body tree). Figure 8 shows the abstract output rule for higher operators in the start tree.

```
output(frame_name(Frame_name),frame_no(Frame_no),tree_
      type(Tree_type),
      instance_no(Instance_no)name_stripe(Name_stripe),
      output_stripe(Output_stripe)):-
         node_name(frame_name(Frame_name),frame_no
         (Frame_no),tree_type(Tree_type),
            instance_no(Instance_no),name_stripe(Name_
            stripe)),
         higher_op(name(Name_stripe)),Tree_type=start,
         not(inverted_name_stripe(frame_name(Frame_name),
         frame_no(Frame_no),
            tree_type(Tree_type),instance_no(Any_instance_
            no))),
         findall(Argument,input(frame_name(Frame_name),
         frame_no(Frame_no),
            tree_type(Tree_type),instance_no(Instance_no),
            inputno(Inputno),output_stripe(Argument)),
          List_of_arguments),
      assert(inverted_name_stripe(frame_name(Frame_name),
      frame_no(Frame_no),
          tree_type(Tree_type),instance_no(Instance_noo))),
      copy_frame_on_top(frame_name(Name-stripe),top_frame_
      no(Top_frame_no)),
      findall(Parameter,(leaf(frame_name(Name_stripe),
      frame_no(Top_frame_no),
          tree_type(head),instance_no(Inst_leaf)),
          node_name(frame_name(Name_stripe),
          frame_no(Top_frame_no),tree_type(head),
          instance_no(Inst_leaf),
          name_stripe(Parameter))), List_of_parameters),
      forall(parm_arg_pair(Parm,Arg,List_of_parameters,
      List_of_arguments),
          (node_name(frame_name(Name_stripe),frame_no
          (Top_frame_no),
          tree_type(head),instance_no(Inst_parm),
          name_stripe(Parm)),
          modify(frame_name(Name_stripe),frame_no
          (Top_frame_no),
          tree_type(head),instance_no(Inst_parm),
          output_stripe(Arg)))),
      root(frame_name(Name_stripe),frame_no(Top_frame_
      no),tree_type(head),
          instance_no(Inst_root_head)),!,
      output(frame_name(Name_stripe),frame_no(Top_frame_
      no),tree_type(head),
          instance_no(Inst_root_head),name_stripe
          (Name_stripe),
          output_stripe(Output_stripe)),
      retract(inverted_name_stripe(frame_name(Frame_
      name),frame_no(Frame_no),
          tree_type(Tree_type),instance_no(Instance_no))),
      delete_frame_from_top,!.


/+ IF   there is the goal to compute the output value of a
        node AND
        (1) the node name is a higher operator in the
            start tree,
        (2) there is no inverted name stripe in the tree
            which contains the node,

   THAN create the subgoal to compute all input values of
        the node,

   AND  after this subgoal is fulfilled,
        (1) invert the name stripe of the node,
        (2) create the frame with the operators name and
            place it on top of the frame stack,
        (3) bind the parameters,
        (4) determine it's head root,
        (5) create the subgoal to compute the output
            value of the head root
            (this value is then the value of the goal
            in the IF part of this rule),

AND         after this subgoal is fulfilled,

            (6) undo the inversion of the name stripe
                of the node,

            (7) delete the upper visible frame.      +/
```

FIGURE 8
Abstract Rule 5 of Rule Set B (Call-by-Value, call in start tree).

The behavior of these rules led to a new visual trace. Time-indexed information was now location-indexed so that undesired perceptual grouping could not occur any longer.

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 computational goals and intermediate results are kept visible only as long as they are absolutely necessary for the ongoing computation. 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 former computations mentally. This leads to higher working memory load for the programmer.

So we had to relax the minimum assumption a second time (see Figure 4) and introduce even more visual redundancy. This was i.e. in accordance with the third principle of Fitter and Green (36).

But there were some other reasons which influenced the decision to modify the rule set a second time. First, as mentioned, rules were still recursive. If process icons derived from recursive rules are used as instructional and help material, they force higher working memory load because of the mental maintainance of a goal stack with return points. Second, derivation of iconic rules from rule set B still leads to two disjunctive rules.

4.3. Rule Set C and Process Icons

The third rule set with 29 (plus operator-specific) rules 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. Computations 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 9 and examples for abstract rules in Figures 10 and 11. The PROLOG facts in Figure 9 describe two nodes and two connections in the incomplete program of Figure 1. Both nodes are in the root position of the head and the body of the program, respectively. The rules in Figures 10 and 11 are comparable to parts of the abstract rule 5 of rule-set B shown in Figure 8. The computational behavior of rule set C was "frozen" in our INTERLISP/LOOPS-Implementation (Kohnert and Janke (54)). This completes the specification cycle (Figure 4).

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

FIGURE 9
An example for Abstract Nodes and Connections.

```
output:-
      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)),
      higher_operator(name(Name_stripe)),
      Tree_type = start,
      not(inverted_name_stripe(frame_name(Frame_name),
      frame_no(Frame_no),tree_type(Tree_type),
        instance_no(Any_instance_no))),
      Output_stripe = ?,
      forall(on(Element,Input_stripe),value(Element)),
      copy_frame_on_top(frame_name(Name_stripe),top_frame_no
      (Top_frame_no)),
      assert(inverted_name_stripe(frame_name(Frame_name),
      frame_no(Frame_no),tree_type(Tree_type),
        instance_no(Instance_no))),
      root(frame_name(Name_stripe),frame_no(Top_frame_no),
      tree_type(head),
        instance_no(Instance_no_root_head)),
      modify(frame_name(Name_stripe),frame_no(Top_frame_no),
      tree_type(head),
        instance_no(Instance_no_root_head),output_stripe(?)),
      modify(frame_name(Name_stripe),frame_no(Top_frame_no),
      tree_type(head),
        instance_no(Instance_no_root_head),input_stripe
        (Input_stripe)),
      bind_parameter_of_top_frame(input_stripe(Input_stripe)),
output.

/+ IF    there is a node which has the following features:
         (1) The node name is a higher operator.
         (2) The node is located in the start tree.
         (3) The name stripe of the node is the only
             inverted one in the tree which contains the node.
         (4) The output_stripe of the node contains a "?".
         (5) The input_stripe of the node contains all
             input values.

   THEN  create the frame with the operators name and place
         it on top of the frame stack.
         Invert the name stripe of the node.
         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.     +/
```

FIGURE 10
Abstract Rule 8 of Rule Set C (First part of Call-by-Value, call in start tree).

```
 output:-
      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)),
      higher_operator(name(Name_stripe)),
      Tree_type = start,
      inverted_name_stripe(frame_name(Frame_name),frame_no
      (Frame_no),tree_type(Tree_type),
        instance_no(Instance_no)),
      Output_stripe = ?,
      forall(on(Element?input_stripe),value(Element)),
      value_of_upper_visible_frame(Output_stript_root_head),
      no_exist_lower_visible_frame,
      modify(frame(Frame_name),frame_no(Frame_no),
      tree_type(Tree_type),
        instance_no(Instance_no),output_stripe(Output_stripe_
        root_head)),
      delete_frame_from_top,
      retract(inverted_name_stripe(frame_name(Frame_name),
      frame_no(Frame_no),tree_type(Tree_type),
        instance_no(Instance_no))),
output.

/+ IF   there is a node which has the following features:

        (1) The node name is a higher operator.
        (2) The node is located in the start tree.
        (3) The name stripe of the node is inverted.
        (4) The output_stripe of the node contains a "?".
        (5) The input_stripe of the node contains all
            input values.
        (6) The head root of the upper visible frame
            contains a value.
        (7) There is no other visible frame.

   THEN transfer this value into the output_stripe of the
        node.
        Delete the upper visible frame.
        Undo the inversion of the name stripe of the node. +/
```

FIGURE 11
Abstract rule 9 of Rule Set C (Second part of Call-by-Value, call in start tree).

In the visual trace, intermediate results now live as long as their frame. As with rule set B, there is no undesired perceptual grouping. Process icons derived from rule set C would not be applied recursively, and there would be no disjunctions.

4.4. Two iconic rule sets a instruction and help material

On the basis of rule sets B and C we developed iconic rules to describe the operational behavior of the ABSYNT-interpreter so it can be used by a student. We got two different iconic rule sets B and C with 8 resp. 16 iconic rules, based on the abstract rule sets B and C explained above, respectively. The iconic rules are visual representations only of the "input"-rules and "output"-rules of the abstract rule sets. Additional rules of the abstract rule sets (i.e., for testing if a node is a root or a leaf) as well as the operator-specific rules are explained in an appendix which is added to the iconic rule sets when used by a student. The appendix also contains a short introduction to the syntax of the iconic rules. So, complete instructional and help material is provided.

We tried to make the iconic rules as self-explaining as possible. Figure 12 shows the rule from the iconic rule set B which is based on the abstract rule shown in Figure 8. Figures 13 and 14 are partially corresponding to the rule shown in Figure 12. They belong to the iconic rule set C, and are based on the abstract rules shown in Figures 10 and 11.

The thick arrows on the left side of the rule of the iconic rule set B in Figure 12 indicate that this rule may be entered here. The thick arrows to the right side indicate that the rule may be left here. So, if the first situation description is true, the first action can be executed. Now the user may temporarily have to leave the rule in order to produce the computational state which satisfies the second situation description. He will have to do this with the help of other rules. If the second situation description is true, the second action can be performed. In contrast, the rules of the iconic rule set C (Figures 13 and 14) are individual situation-action pairs.

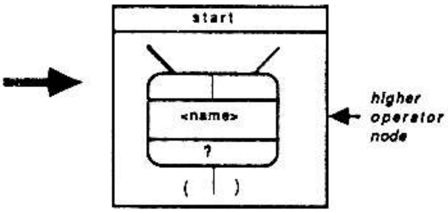
Rule 5: **Computing a higher operator node in start tree**

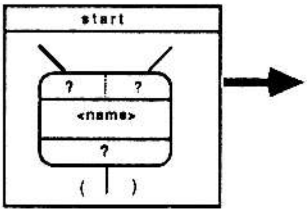
First situation:

A "?" is in the output stripe of a higher operator node in the start tree.
It's input stripe is empty.
There is no inverted name stripe in the start tree.

First action:

Write a "?" into each input field.



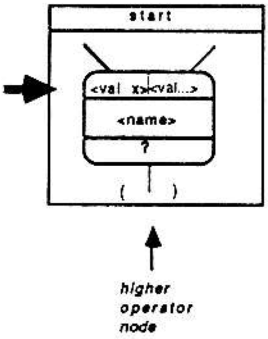


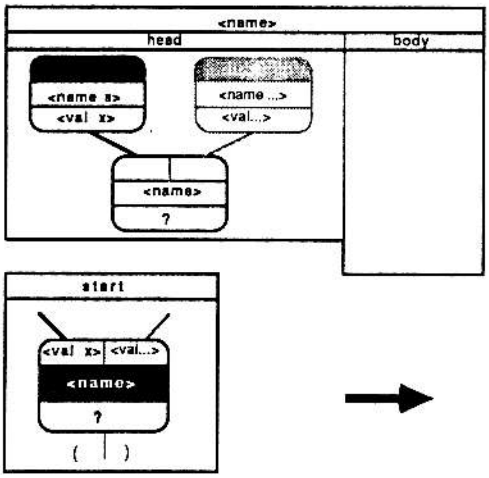
Intermediate situation:

A "?" is in the output stripe of a higher operator node in the start tree. It's input stripe contains only values.
There is no inverted name stripe in the start tree.

Intermediate action:

Invert the name stripe of the higher operator node.
Make a frame with the operator's name and place it on top of the frame stack.
Write the value of each input field into the output field of each head leaf of this frame, preserving their order.
Write a "?" into the output stripe of the head root.



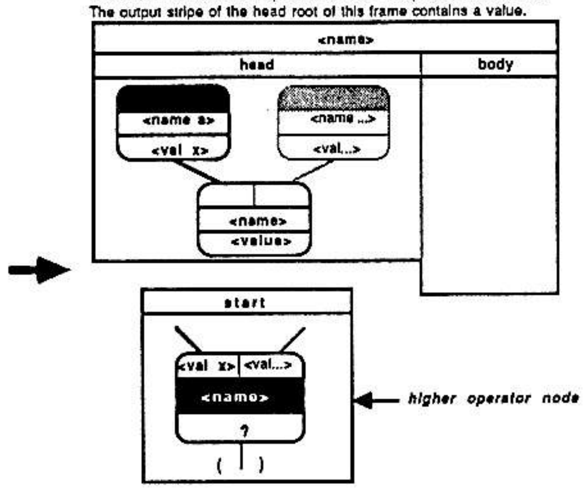


continued on next page

Rule 5 continued: Computing a higher operator node in start tree

Last situation:

A "?" is in the output stripe of a higher operator node in the start tree.
It's input stripe contains only values. It's name stripe is inverted.
There is a frame with the operator's name on top of the frame stack.
The output stripe of the head root of this frame contains a value.



Last action:

Write this value into the output stripe of the higher operator node of the start tree.
Undo the inversion of the name stripe of the node.
Delete the frame from top of the stack.

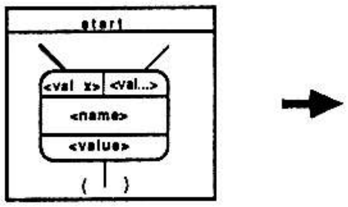
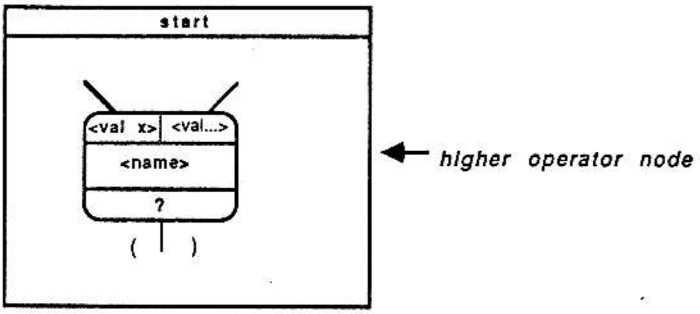


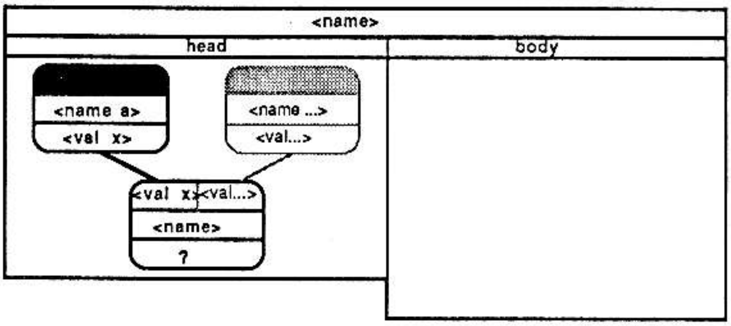
FIGURE 12
Iconic Rule 5 of Iconic Rule Set B on the Basis of Abstract Rule 5 in Figure 8.

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

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



Action: Make a frame with the operator's name and place it on the top of the frame stack.
Invert the name stripe of the higher operator node.
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.



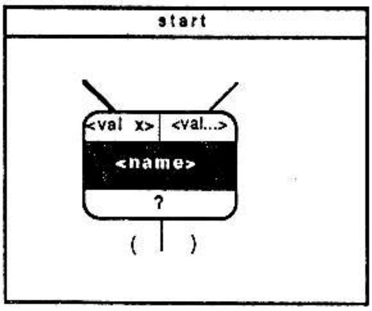
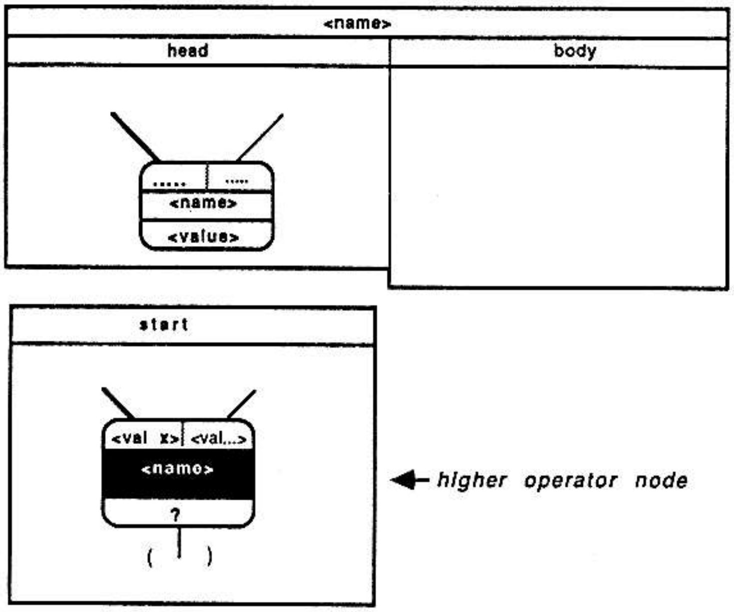


FIGURE 13
Iconic Rule 8 of Iconic Rule Set C on the Basis of Abstract Rule 8 in Figure 10.

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

Situation: A higher operator node is part of the start tree.
The name stripe of that node is inverted.
The output stripe of the node contains a "?".
It's input stripe contains only values.
There is a frame with the operator's name on top of the frame stack
The output stripe of the head root the upper visible frame contains a value.
There exists no lower visible frame.



Action: Write this value into the output stripe of the higher operator node of the start tree.
Delete the upper visible frame.
Undo the inversion of the name stripe of the higher operator node.

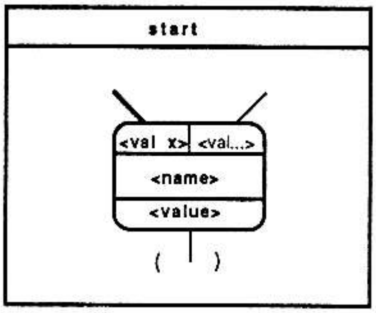


FIGURE 14
Iconic Rule 9 of Iconic Rule Set C on the Basis of Abstract Rule 9 in Figure 11.

5. SOME IMPLICATIONS

With the iconic rule sets at hand, we are now able to overcome the shortcomings of the verbal specification of the operational knowledge:

- There is precise and unambiguous instructional and help material concerning the operational knowledge.
- We can be sure that the operational knowledge acquired by the programmer is a solid base for programming and debugging.
- We have a framework for analyzing semantic bugs. They can be related to the rules.

Moreover, the rule sets allow a more fine-grained classification of programs. Figure 15 shows the partial order of programs based on the rule sets. As compared to the partial ordering used in the feasability study (Figure 3), it can be used for more elaborate task construction and sequencing.

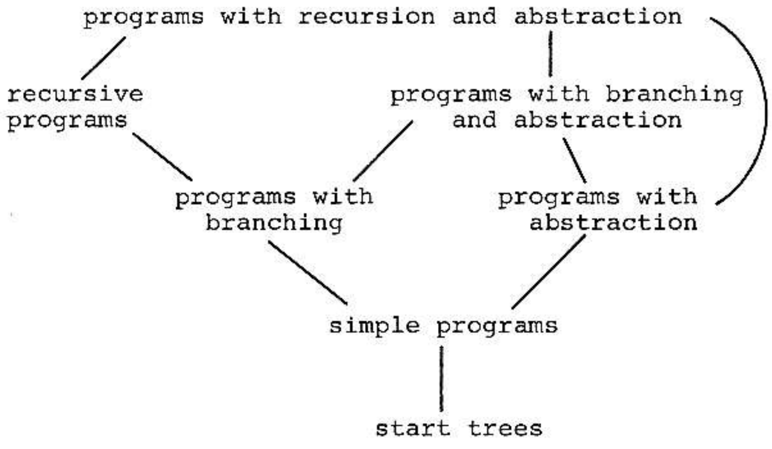

programs with recursion and abstraction
recursive programs
programs with branching and abstraction
programs with branching
programs with abstraction
simple programs
start trees

FIGURE 15
Partial ordering of ABSYNT programs.

The different structure of the two iconic rule sets is due to differences in the corresponding abstract rule sets. It raises some psychologically relevant questions. Since users of the iconic rule set B have to remember the points where they temporarily left rules, errors and/or the amount of search for the next computational step should increase when 1. there are many pending rules, and 2. many computational steps were necessary in order to continue work on a pending rule. That is, memory faults should increase in such computational states. In contrast, for users of the iconic rule set C there should be no differences in errors or amount of search in the same computational states. In contrast, for users of the iconic rule set C there should be no differences in errors or amount of search in the same computational states.

On the other hand one may suggest that rule set B enhances understanding of the structure of the computational process, since it is a more integrated representation as rule set C.

In a recent study we asked programming novices to compute programs (that is, to predict the trace) with the aid of the iconic rule sets (plus appendix). The subjects had no serious trouble with the iconic rules. Also, the predicted different effects of the iconic rule sets B and C seem to emerge.

As the evaluation of this study will be completed, the next step will be to implement the iconic rules for instructional and help purposes so that the interpreter becomes self explaining, and the student can get this information when he is uncertain about computational processes of the machine.

6. REFERENCES

(1) Attisha, M., Non-borrow Subtraction Algorithm. Working Paper W-119, Computer Science Dept. (University of Exeter, 1984).

(2) Attisha, M.; Yazdani, M., An Expert System for Diagnosing Childrens' Multiplication Erors. Research Report R-117, Computer Science Dept. (University of Exeter, 1983).

(3) Brown, J.S.; Burton, R.R., Cognitive Science (1978) 2.

(4) Bundy, A., Computer Modeling of Mathematical Reasoning (New York: Academic Press, 1983).

(5) Burton, R.R., Diagnosing Bugs in a Simple Procedural Skill. In: Sleeman, D.; Brown, J.S. (eds.): Intelligent Tutoring Systems (New York: Academic Press, 1982) pp. 157-183.

(6) van Lehn, K.; Brown, J.S., Planning Nets: A Representation for Formalizing Analogies and Semantic Models of Procedural Skills. In: Snow, R.E.; Federico, P.A.; Montague, W.E. (eds.): Aptitude, Learning and Instruction. Vol. II: Cognitive Process Analyses of Learning and Problem Solving (Hillsdale, N.J.: Erlbaum, 1980) pp. 95-137.

(7) Young, R.M.; O'Shea, T.; Cognitive Science (1981) 5.

(8) O'Shea, T., Self-Improving Teaching Systems (Basel: Birkhäuser, 1979).

(9) O'Shea, T., Int. Journal of Man-Machine Studies (1979) 11, and in: Sleeman, D.; Brown, J.S. (eds.): Intelligent Tutoring Systems (New York: Academic Press,1982).

(10) Sleeman, D.H., Assessing Aspects of Competence in Basic Algebra. In: Sleeman, D.; Brown, J.S. (eds.): Intelligent Tutoring Systems (New Academic Press,1982).

(11) Sleeman, D.H., Inferring Student Models for Intelligent Computer-Aided Instruction. In: Michalski,R.S.; Carbonell,J.G.; Mitchell,T.M. (eds.): Machine Learning: An Artificial Intelligence Approach (Palo Alto: Tioga Publ. Co.,1983) pp. 483-510.

(12) Sleeman, D.H., Cognitive Science (1984) 8.

(13) Sleeman, D.H., Int. Journal of Man-Machine Studies (1985) 22.

(14) Sleeman, D.H., Inferring (Mal)rules from Pupils Protocols. In: Steels, L.; Campbell, J.A. (eds.): Progress in Artificial Intelligence (Chichester, Sussex: Ellis Horwood Ltd., 1986).

(15) Anderson, J.R., Acquisition of Proof Skills in Geometry. In: Michalski, R.S.; Carbonell, J.G.; Mitchell, T.M. (eds.): Machine Learning: An Artificial Intelligence Approach. (Palo Alto: Tioga Publ. Co., 1983) pp. 191-219.

(16) Anderson, J.R.; Boyle, C.F.; Farrell, R.; Reiser, B.J., Cognitive Principles in the Design of Computer Tutors. In: Morris, P. (ed.): Modelling Cognition (Chichester, Sussex: J. Wiley, 1987) pp. 93-133.

(17) Anderson, J.R.; Greeno, J.G.; Kline, P.J.; Neves,D.M., Acquisition of Problem-Solving Skill. In: Anderson,J.R. (ed.): Cognitive Skills and their Acquisition (Hillsdale, Erlbaum, 1981) pp. 191-230.

(18) Anderson, J.R., Learning to Program. Proceedings of the Eighth International Joint Conference on Artificial Intelligence (Los Altos, California: Morgan Kaufman, 1983).

(19) Anderson, J.R., Production Systems, Learning and Tutoring. In: Klahr, D.; Langley, P.; Neches, R. (eds.): Production System Models of Learning and Development (Cambridge, Mass.: MIT Press 1987) pp. 437-458.

(20) Anderson, J.R.; Farrell, R.; Sauers, R., Learning to Plan in LISP. Technical Report ONR-82-2, Department of Psychology, Carnegie-Mellon University (Pittsburg, PA, 1982).

(21) Anderson, J.R.; Farrell, R.; Sauers, R., Cognitive Science (1984) 8.

(22) Anderson, J.R.; Reiser, B.J., BYTE (1985) 4.

(23) Anderson, J.R.; Swarecki, E., Communications of the ACM (1986) 29 (9).

(24) Johnson, W.L., Intention-Based Diagnosis of Novice Programming Errors (Los Altos, California: Morgan Kaufman Publ., 1986).

(25) Johnson, W.L.; Soloway, E., PROUST: An Automatic Debugger for PASCAL Programs. BYTE (1985) April, pp. 179-190, and in: Kearsley,G.P. (ed.): Artificial Intelligence and Instruction (Reading, Mass.: Addison Wesley, 1987), pp. 49-67.

(27) Soloway, E., Communications of the ACM, 29 (1986) 9.

(28) Wertz, H., Journal of Man-Machine Studies (1982) 16.

(29) Wertz, H., Intelligence Artificielle: Application â l'Analyse de Programmers (Paris: Masson, 1985).

(30) Wertz, H., Automatic Correction Improvement of Programs (Chichester, West Sussex: Ellis Horwood Ltd.,1987).

(31) Abelson, H.; Sussman, G.J.; Sussman, J., Structure and Interpretation of Computer Programs (Cambridge, Mass.: MIT Press, 1985).

(32) Ghezzi, C.; Yazayeri, M., Programming Language Concepts 2/E. (New York: Wiley, 1987).

(33) Henderson, P., Functional Programming: Application and Implementation. (Englewood Cliffs. N.J.: Prentice Hall, 1980).

(34) Henderson, P., IEEE Transactions on Software Engineering SE-122 (1986) pp. 241-250.

(35) Pennington, N., Cognitive Psychology (1987) 19.

(36) Fitter, M.; Green, T.R.G., When Do Diagrams Make Good Computer Languages? Int. Journal of Man-Machine Studies, (1979) 11 pp. 235-261, and in: Coombs, M.J.; Alty, J.L. (eds.): Computing Skills and the User Interface (New York: Academic Press, 1981) pp. 253-287.

(37) Green, T.R.G.; Sime, M.E.; Fitter, M.J., The Art of Notation. In: Coombs, M.J.; Alty, J.L. (eds.): Computing Skills and the User Interface (New York: Academic Press, 1981) pp. 221-251.

(38) Payne, S.J.; Sime, M.E.; Green,T.R.G., Int. Journal of Man-Machine Studies (1984) 21.

(39) Larkin, J.H.; Simon, H.A., Cognitive Science (1987) 11.

(40) Pomerantz, J.R.: Perceptual Organization in Information Processing. In: Aitkenhead, A.M.; Slack, J.M. (eds.): Issues in Cognitive Modeling (Hillsdale: Erlbaum, 1985) pp- 127-158.

(41) Chase, W.G., Visual Information Processing, in: Boff, K.R.; Kaufmann, L. and Thomas, J.P. (eds.): Handbook of Perception and Human Performance, Vol. II. Cognitive Processes and Performance (New York: Wiley, 1986) pp. 28-1 - 28-71.

(42) van Lehn, K., Bugs are not Enough: Empirical Studies of Bugs, Impasses and Repairs in Procedural Skills. XEROX Parc. Cognitive and Instructional Science Group (1981) CIS-11 (SSL-81-2) and Journal of Mathematical Behavior (1982) 3.

(43) Egan, D.E.; Greeno, J.G., Theory of Rule Induction: Knowledge Acquired in Concept Learning, Serial Pattern Learning, and Problem Solving. In: Gregg, L.W. (ed.): Knowledge and Cognition (Potomac: Erlbaum, 1974)pp. 43-103.

(44) Simon, H.A.; Lea, G., Problem Solving and Rule Induction: A Unified View. In: Gregg, L.W. (ed.): Knowledge and Cognition (Potomac: Erlbaum, 1974) pp. 105-127.

(45) Anderson, J.R., The Architecture of Cognition. Cambridge Mass.: Harvard University Press 1983.

(46) van Lehn, K., Artificial Intelligence (1987) 31.

(47) van Lehn, K., Towards a Theory of Impasse-Driven Learning. ONR Tech. Rep., CMU-University (Pittsburgh, USA, 1987).

(48) McKendree, J., Feedback Content During Complex Skill Acquisition. In: Salvendy, G.; Sauter, S.L.; Hurrell, J.J. (eds.): Social, Ergonomic and Stress Aspects of Work with Computers (Amsterdam: Elsevier Science Publ., 1987) pp. 181-188.

(49) Brown, J.S.; van Lehn, K., Cognitive Science (1980) 4.

(50) Carroll, J.M., Minimalist Design for Active Users. In: Shackle, B. (ed.): Interact 84, First IFIP Conference of Human-Computer-Interaction (Amsterdam: Elsevier/ North Holland, 1984).

(51) Carroll, J.M., Minimalist Training (Datamation, 1984).

(52) Bauer, F.L.; Goos, G., Informatik. 1. Teil (Berlin, Springer, 1982 (3. Edition)).

(53) Janke, G. and Kohnert, K., Interface Design of a Visual Programming Language: Evaluating Runnable Specifications According to Psychological Criteria (this volume).

(54) Kohnert, K. and Janke, G.: Object-oriented Implementation of the ABSYNT Environment, ABSYNT Report (1988) 4.

(55) Fähnrich, K.P. and Ziegler, J., Workstation Using Direct Manipulation as Interaction Mode, in: Proceedings of INTERACT '84, Vol. II, 1985a pp. 203-208 (in german: Direkte Manipulation als Interaktionsform an Arbeitsplatzrechnern, in: Bullinger, H.J. (Hrsg.), Software-Ergonomie '85-Mensch-Computer-Interaktion (Stuttgart: Teubner, 1985) pp. 75-85.

(56) Hutchins, E.L.; Hollan, J.D. and Norman, D.A., Direct Manipulation Interfaces, in: Norman, D.A. and Draper, S.W. (eds.): User Centered System Design - New Perspectives on Human Computer Interaction (Hillsdale, N.J.: Lawrence Erlbaum Ass., 1986) pp. 87-124.

(57) Shneiderman, B., IEEE Computer (1983) 16 (8).

(58) Shneiderman, B., Designing the User Interface: Strategies for Effective Human-Computer Interaction (Reading, Mass.: Addison-Wesley, 1987).

(59) Shu, N.C., Visual Programming Languages: A Perspective and a Dimensional Analysis, In: Chang, T.; Ichikawa, Ligomenides, P.A. (eds.): Visual Languages (New York, Plenum Press, 1986) pp. 11-34.

(60) Möbus, C. and Thole, H.J., Tutors, Instructions and Helps ABSYNT Report 3/88 (1988), to appear in: Christaller, Th. (ed.): Künstliche Intelligenz KIFS 1987, Heidelberg, Springer, Lecture Notes in Computer Science (in print).

(61) Colonius, Frank, Janke, Kohnert, Möbus, Schröder and Thole, Syntaktische und semantische Fehler in funktionalen graphischen Programmen, ABSYNT Report 2 (1987).

(62) Möbus, C., Die Entwicklung zum Programmierexperten durch das Problemlösen mit Automaten. in: Mandl and Fischer (Hrsg.), Lernen im Dialog mit dem Computer (München: Urban & Sehwarzenberg, 1985) pp. 140-154.

(63) Hoc, J.M., Int. J. of Man-Machine Studies (1977) 9.

(64) Adelson, B., Memory and Cognition (1981) 9.

(65) Brooks, R., Int. J. of Man-Machine Studies (1983) 18.

(66) Letovsky, S., Cognitive Processes in Program Comprehension. In: Soloway, E.; Iyengar, S. (eds.): Empirical Studies of Programmers (New York, Ablex, 1986) pp. 58-80.

(67) Rist, R.S., Plans in Programming: Definition, Demonstration, and Development. In: Soloway, E.; Iyengar, S. (eds.) Empirical Studies of Programmers (New York, Ablex, 1986) pp. 28-47.

(68) Colonius, Frank, Janke, Kohnert, Möbus, Schröder and Thole, Stand des DFG-Projekts "Entwicklung einer Wissensdiagnostik- und Fehlererklärungskomponente beim Erwerb von Programmierwissen für ABSYNT", In: Gunzenhäuser, R. und Mandl, H. (Hrsg.): Intelligente Lernsysteme (1987) pp. 80-90, Institut für Informatik der Universität Stuttgart und Deutsches Institut für Fernstudien an der Universität Tübingen.

(69) Chang, S.K., Visual Languages: A Tutorial and Survey. In: Gorny, P.; Tauber, M.J. (eds.): Visualization in Programming. Lecture Notes in Computer Science, No. 282 (Berlin: Springer, 1987).

(70) Davis, R.E., Runnible Specification as a Design Tool, in: Clark, K.L.; Tärnlund, S.A. (eds.): Logic Programming (New York: Academic Press, 1982) pp. 141-149.

(71) Norman, D.A., Notes Toward a Theory of Complex Learning. In: Lesgold, A.M.; Pellegrino, J.W.; Fokkema, S.D.; Glaser, R. (eds.): Cognitive Psychology and Instruction (New York, Plenum Press, 1978) pp. 39-48.

(72) Goldstein, I.P., The Genetic Graph: A Representation of the Evolution of Procedural Knowledge, in: Sleeman, D. and Brown, J.S. (eds.): Intelligent Tutoring Systems (New York: Academic Press, 1982).

(73) Möbus, C., Logic Programs as a Specification and Description Tool in the Design Process of an Intelligent Tutoring System, in: Salvendy, G. (ed.): Abridged Proceedings of the HCI International '87 (1987) p. 119.

(74) Möbus, C., Specifications of Instructions and Helps for an ICAI-System in the Field of Graphical Programming. Paper Presented at the First European Seminar on Intelligent Tutoring Systems. Commission of the European Communities (Rottenburg, 25. - 31. October 1987).

(75) Genesereth, M.R.; Nilsson, N.J., Logical Foundations of Artificial Intelligence (Los Altos, California: Morgan Kaufman, 1987).

(76) Bourne, L.E., An Inference Model of Conceptual Rule Learning. In: Solso, R. (ed.): Theories in cognitive psychology, (Washington,D.C.: Erlbaum, 1974) pp. 231-256.

(77) Haygood, R.C.; Bourne, L.E., Psychological Review (1965) 72.

(78) Medin, D.L.; Wattenmaker, W.D.; Michalski, R.S., Cognitive Science, 1987) 11.

REPRINTED FROM

Man–Computer Interaction Research MACINTER–II

Selected Papers of the
Man–Computer Interaction Research Network of
The International Union of Psychological Science (IUPsyS)

Edited by

F. KLIX
Department of Psychology
Humboldt University
Berlin, German Democratic Republic

N. A. STREITZ
Integrated Publication and
Information Systems Institute
Gesellschaft für Mathematik und Datenverarbeitung
Darmstadt, Federal Republic of Germany

Y. WAERN
Department of Psychology
University of Stockholm
Stockholm, Sweden

H. WANDKE
Department of Psychology
Humboldt University
Berlin, German Democratic Republic

1989

NORTH-HOLLAND
AMSTERDAM · NEW YORK · OXFORD · TOKYO