ABSYNT-Report 6/88

REPRESENTING SEMANTIC KNOWLEDGE WITH 2-DIMENSIONAL RULES IN THE DOMAIN OF FUNCTIONAL PROGRAMMING

Claus Möbus & Olaf Schröder

August 1988

Project ABSYNT[1]
FB 10, Informatik
Unit on Tutoring and Learning Systems
University of Oldenburg
D-2900 Oldenburg, FRG

This research was sponsored by the Deutsche Forschungsgemeinschaft (DFG) in the SPP Psychology of Knowledge under Contract No. MO 292/3-2

REPRESENTING SEMANTIC KNOWLEDGE WITH 2-DIMENSIONAL RULES IN THE DOMAIN OF FUNCTIONAL PROGRAMMING

Claus Möbus & Olaf Schröder

Project ABSYNT[1]
FB 10, Informatik
Unit on Tutoring and Learning Systems[2]
University of Oldenburg
D-2900 Oldenburg, FRG

0. Abstract

One of the many difficult problems in the development of intelligent computer aided instruction (ICAI) is the appropriate design of instructions and helps. This paper adresses the question of optimizing instructional and help material concerning the operational knowledge for the visual, functional programming language ABSYNT (ABstract SYNtax Trees). The ultimate goal of the project is to build a problem solving monitor (PSM) for this language and the corresponding programming environment. The PSM should analyse the blueprints of the students, give comments and proposals (SLEEMAN & HENDLEY, 1982). First, we will explain our motivation for choosing this domain of discourse. Second, we will shortly present the programming environment of ABSYNT. Third, we represent the development of two alternative 2-D-rulesets (appendix A, B), which describe the operational semantics of the ABSYNT interpreter. The development of the 2-D-rules was guided by cognitive psychology and cognitive engineering aspects and results of an empirical study. The study showed that the rules were comprehensible even for computer novices.

1. Introduction

The main research goal of ABSYNT is the construction of a PSM. We chose the domain of computer programming because the main activity of the programmer is problem solving, a very relevant research area from a cognitive science point of view. Because our PSM will have to analyse the planning processes of novices in depth, we decided to use a simple programming language, the syntax and semantics of which can be learned in a few hours. We propose 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". Furthermore, there is some evidence that there is a strong correspondency between

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

[2]We are grateful that Gabi Janke & Klaus Kohnert implemented ABSYNT in INTERLISP and LOOPS, that Heinz-Jürgen Thole did part of the rule-based programming in LPA-PROLOG and that Klaus-Dieter Frank assisted in empirical research and did the first implementation of the help system in HYPERCARD.

programmer's goals and use of functions (PENNINGTON, 1987; SOLOWAY, 1986; JOHNSON & SOLOWAY, 1985, 1987). This correspondence helps to avoid the difficult problem of interleaving plans in the code which shows up in imperative programming languages. SOLOWAY(1986) has argued that this kind of interleaving makes the diagnosis of programmer's plans rather difficult and time consuming. If we take for granted that a goal can be represented by a function, we can gain a great deal of 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 corresponds to the development of the functional program.

There are similar psychological reasons for the use of a *visual* programming language. 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; LARKIN & SIMON, 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 (LARKIN & SIMON, 1987). "Good" diagrams enable automatic control of attention with the help of the location of objects. These are in our case object icons 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 (POMERANTZ, 1985; CHASE, 1986).

A very crucial point concerning the "intelligence" of a PSM lies in the quality of the feedback system design. In this paper we are restricting ourselves to an instructional and feedback system based on 2-D-rules describing the operational semantics of ABSYNT. Other more formal approaches specifying the semantics of a language (ALBER & STRUCKMANN, 1988; PAGAN, 1981) are suited for computer scientists but not for programming novices.

When should an ICAI system administer feedback? Our tutorial strategy is guided by "repair theory" (BROWN & VanLEHN, 1980) and follows the "minimalist design philosophy" (CARROLL, 1984a,b). The latter 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 impasse 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 follow-up error.

2. The Programming Environment of ABSYNT

The programming environment of ABSYNT was developed in our project, using ideas from the "calculation sheet machine" (BAUER & GOOS, 1982). The complete programming environment was implemented in INTERLISP and the object-orientated language LOOPS (JANKE & KOHNERT, 1988; KOHNERT & JANKE, 1988) to get a programming environment with direct manipulation capabilities. Following SHU's (1986) and CHANG's (1987) dimensional analysis, ABSYNT is a language with high visual extent, low scope and medium level. The ABSYNT-environment comprises three modes: a programming mode, a trace mode, and a prediction mode (KOHNERT & JANKE, 1988).

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 specialised 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

In the upper half of the screen the programmer sees the main worksheet and in the lower half a subordinated one. Each worksheet is called frame. Frames are 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 shaded grey node that there is something missing. The editor warns with flashes if unsyntactic programs are going to be constructed: crossing of connections, overlapping 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. Parameter names and values for constants are entered in the leaves of the head and the body by pop-up-menus, too.

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 is 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 regarding the operational knowledge of ABSYNT as a necessary prerequisite for the development of planning and programming knowledge. That is, features which are necessary for the understanding of the computational process are visualized in the programming mode as well as in the other modes of the programming environment.

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 (FIGURE 2). The design of the trace is a result

---

FIGURE 2: The trace mode of ABSYNT

---

of our iterative specification cycle in the development of abstract rules and process icons (MÖBUS & THOLE, 1988; MÖBUS & SCHRÖDER, 1988). In the case of recursive programs, the frame actually computed 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 that he can smoothly acquire the operational knowledge for ABSYNT.

3. Preliminary Instructional Material and Semantic Bugs

Our starting point for developing a functional visual programming language was a paper-and-pencil study where we did some explorations without any computer implementations. Part of this feasability study was a verbal specification of the syntax and the operational knowledge, illustrated by simple programs and trees (MÖBUS & SCHRÖDER, 1988). The goals of the pilot study were:

- to get suggestions for the design of the language and the interface
- to collect syntactic and semantic bugs in order to find reasons for bugs and conditions under which they occur
- to study the memory representations of example programs

A detailed description of the feasability study is provided in SCHRÖDER, FRANK & COLONIUS (1987) and COLONIUS, FRANK, JANKE, KOHNERT, MÖBUS, SCHRÖDER & THOLE (1987). Among other things the subjects had to compute the value of various programs, simulating the ABSYNT interpreter. Then we analysed the observable computational errors.

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

- It was unclear whether the bugs arose because of ambiguities in the instructional material (the verbal description of the operational knowledge). Therefore, we could not be certain whether this description could actually 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, 1987), 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.

Therefore, we decided to use a runnable specification (DAVIS, 1982) of the language as a foundation for constructing process icons. These process icons were then programmed in the HYPERCARD-system and used as instructional and help material for teaching purposes.

4. Construction of improved instructional material: process icons

The specification of the operational knowledge was achieved in an iterative specification cycle (MÖBUS & THOLE, 1988; MÖBUS & SCHRÖDER, 1988). 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 of a program was formalized by a set of PROLOG facts similar to an approach of GENESERETH & NILSSON (1987, ch. 2.5).

In the next step of the specification cycle we tried a 2-D-representation of the facts and Horn clauses of rule set A. Thereby, we kept in mind design principles which are motivated by

results of POMERANTZ (1985) and LARKIN & SIMON (1987). POMERANTZ made some careful studies about selective and divided attention in information processing. One consequence for our design was that time-indexed information had to be spatially indexed by locations, too. Information with the same time index should have the same spatial index, that is appear in the same location. In our design a location is a visual object. These insights were supported by the formal analysis of LARKIN & SIMON (1987). They showed under what circumstances a diagrammatic representation of information consumes less computational resources than an informational equivalent written representation.

In the course of time we realized that a visual representation of the facts and Horn clauses of rule set A according to the recommendations of POMERANTZ and LARKIN & SIMON was only possible if we "enriched" the 2-D structure. This means that we had to add 2-D 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 the set led to 2-D-representations with disjunctive rules. 2-D-rules with disjunctive conditions require selective attention, which causes matching errors and longer processing time (BOURNE, 1974; HAYGOOD & BOURNE, 1965; MEDIN, WATTENMAKER & MICHALSKI, 1987). Thus rule set A was modified in such a way such that

- any undesired perceptual grouping of information in operator nodes,
- 2-D-rules with disjunctive conditions,
  and
- visual hiding of dynamic successor frames already put on a stack

was avoided.

We came up with a relaxed rule set B with 14 main rules (plus operator-specific rules)(MÖBUS & THOLE, 1988; MÖBUS & SCHRÖDER, 1988). 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.

But these rules still had some defects from a cognitive point of view. 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 for humans, 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 were forced to relax the minimum assumption a second time and introduce even more visual redundancy. This was i.e. in accordance with the third principle of FITTER & GREEN (1979).

Another reason for a further modification of the rule set was the recursiveness of the rules. Instructional and help material derived from such rules should enforce a higher mental working memory load because of the maintenance of a goal stack with return points.

The third rule set C 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. We have included examples for abstract rules of rule set C in FIGUREs 3 and 4. They are represented in visual 2-D-rules 8 and 9 in the state-specific rule set (appendix B).

Then the computational behavior of rule set C was "frozen" in our INTERLISP/LOOPS-Implementation (KOHNERT & JANKE, 1988). This completed the specification cycle.

```
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)),
    assert(inverted_name_stripe(frame_name(Frame_name),frame_no(Frame_no),
        tree_type(Tree_type),instance_no(Instance_no))),
    copy_frame_on_top(frame_name(Name_stripe),top_frame_no(Top_frame_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),input_stripe(Input_stripe)),
    bind_parameter_of_top_frame(input_stripe(Input_stripe)),
    modify(frame_name(Name_stripe),frame_no(Top_frame_no),tree_type(head),
        instance_no(Instance_no_root_head),output_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) There is no node in the start tree with an inverted name stripe.
    (4) The output_stripe of the node contains a "?".
    (5) The input_stripe of the node contains all input_values.

    THEN
        Invert the name_stripe of the node.
        Create a frame with the operator`s name and place it on top of the frame stack.
        Determine it`s head root.
        Transfer the input_stripe of the node to the head root.
        Bind the parameters.
        Put a "?" into the output_stripe of the head root.  */
```

FIGURE 3: Abstract Rule 8 of Rule Set C (First part of Call-by-Value, call in start tree; corresponds to 2-D-rule 8 in the "state-specific" rule set in appendix B)

```
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_stripe_root_head),
    not_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 4: Abstract rule 9 of Rule Set C (Second part of Call-by-Value, call in start tree; corresponds to 2-D-rule 9 in the "state-specific" rule set in appendix B)

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.

On the basis of rule sets B and C we developed 2-D-rules to describe the operational behavior of the ABSYNT-interpreter so that it can be predicted by a student. We got two different 2-D-rule sets B and C with 8 respectively 16 2-D-rules. The 2-D-rules are visual representations of only the most important 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 a separate glossary. The glossary also contains a short introduction to the syntax of the 2-D-rules.

5. Empirical Evaluation of the two 2-D-rule Sets

We did a study in which programming novices computed ABSYNT-programs with the aid of earlier versions (MÖBUS & THOLE, 1988) of the two 2-D-rule sets. One of the aims of the study was to evaluate the learnability of the 2-D-rules. We wanted to detect rules or parts of rules which led to misunderstandings and errors.

Procedure: 12 programming novices (6 subjects working with each rule set) computed ABSYNT-Programs of increasing difficulty. This was done in the prediction mode of the ABSYNT-Environment (section 2.2 and KOHNERT & JANKE, 1988). In this mode the user computed. ABSYNT-Programs by himself without any help from the interpreter. The subjects worked in pairs (cf. MIYAKE, 1986). So three pairs of subjects worked with each rule set. Beside the 2-D-rule set, they were provided with the glossary, that is additional explanations of basic concepts mentioned in the rules. Therefore, complete instructional material was given.

Each pair of subjects computed 33 ABSYNT-programs. The sequence of programs was ordered by the number of 2-D-rules needed. So the most difficult program contained abstraction as well as recursion. The subjects computed each ABSYNT-Program once without being interrupted by the experimentator. In case of correct computation, the next program was presented. In case of a bug, the program was presented again. This time, if bugs occurred, the experimentator gave immediate feedback.

Preliminary results: The evaluation of the study is not completed as yet, but some results related to the aim of the study mentioned above will be presented.

First, some concepts are explained:

1. A *" computational step"* denotes the following actions:
   - changing the content of an input field or an output stripe
   - creating or deleting a frame (choosing the corresponding menu item)
   - typing a frame number

2. A *"rule-consistent computational step"* is any computational step which is part of a correct rule application. It is consistent with the part of an "action" description of a rule the "situation" description of which is satisfied. It is not regarded whether the computational step is made in the right context. So parts visible on the screen but not mentioned in the rule may be faulty.

3. A *"deviation"* is any computational step which is not part of a correct rule application. There are the following possibilities:

   3.1 *Faulty rule application:* The computational step is not consistent with any part of the action description of any rule.

   3.2 *Omission:* The computational step is consistent with a part of an action description of a rule the situation description of which is not yet satisfied, but is satisfiable by intermediate computational steps.

   3.3 *Interference:* The computational step is consistent with a part of an action description of a rule the situation description of which is not satisfied and not satisfiable.

   3.4 *Shortcut:* This is an *optimizing deviation* since it leads to the same result (or the same intermediate result) as the correct sequence of computational steps, while simultaneously saving computational steps. Shortcuts may occur because of the visual redundancy on the screen. So the visible results of sequences of earlier computational steps may be used for handling later situations.

   3.5 *Correction:* Recovering omissions, undoing computational steps, and replacing values by other values are corrections.

Faulty rule applications, interferences and omissions are *"bugs"*. Table 1 shows the absolute frequencies and percentages (in brackets) of types of computational steps for both rule sets:

		Computational steps			
		rule-consistent	bugs	shortcuts	corrections
Rule-set	operator-centered	7096 (97.82%)	71 (0.97%)	42 (0.58%)	45 (0.62%)
	state-centered	7815 (97.14%)	96 (1.19%)	38 (0.47%)	96 (1.19%)

Table 1: Absolute frequencies and percentages (in brackets) of types of computational steps for both rule sets

Within both rule sets, more than 97% of all computational steps were rule-consistent, and only about 1% were bugs. Although the subjects did not receive any feedback during the first computation of a program, the error rate was small. Moreover, there were no typical bugs. There are few examples of bugs for almost every 2-D-rule.

The results indicate that there is no need to redesign the 2-D-rule sets or to change specific rules. Moreover, the hypothesized differences between the two alternative 2-D-rule sets did not seem to show up in the behavior of the subjects. So they possibly used the rules to construct a mental representation which did not correspond to the different structure of the two 2-D rule sets.

Some more observations should be mentioned though, which initiated some slight changes of the rules:

- 25 bugs altogether (= 15%) consisted of typing a wrong frame number. This supported the decision to drop the frame number, which was possible because the interpreter uses a linear stack, and there is at most one pending call in function bodies and in the start tree.

- 40 bugs altogether (= 24%) were omissions occurring with rules containing several computational steps in their action description, (i.e., rules for creating and deleting frames). This motivated a clarification of the structure and an improvement of the readability of the action descriptions of these rules.

- 37 more bugs (= 22%) were interferences occuring when the subjects worked in the head of a newly made frame. This caused us to clarify the structure and improve the readability of the situation descriptions of the rules for creating a new frame.

6. Representing operational semantic knowledge of ABSYNT with 2-D-rules

We tried to make the 2-D-rules as self-explaining as possible. Appendix A shows the rules from the *"operator-specific"* 2-D-rule set which is based on the abstract rule set B. Furthermore Appendix B shows the rules from the *"state-specific"* 2-D-rule set which is based on the abstract rule set C. Two examples of state-specific rules (rule 8 and 9) are shown in FIGUREs 3 and 4.

The operator-specific rules in appendix A are to be interpreted according to the following rough guidelines. The thick arrows on the left side of the rules 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. The same is true for a third situation-action pair.

In contrast to this, the state-specific rules (appendix B) are individual situation-action pairs. Like production rules they are not reentered a second time.

7. Summary

With the 2-D-rule sets at hand, we are now able to overcome the shortcomings of purely verbal or example based instructions. Now there is precise and unambiguous instructional and help material concerning the operational knowledge. We can be confident that the student acquires very easily and rapidly operational knowledge as a solid base for his programming and debugging activities which will be a further topic in our research.

8. References

ALBER, K. & STRUCKMANN, W., Einführung in die Semantik von Programmiersprachen, Mannheim: BI-Wissenschaftsverlag, 1988

BAUER, F.L.& GOOS, G.: Informatik, 1.Teil. Berlin, Springer, 1982 (3. Edition)

BOURNE, L.E.: An Inference Model of Conceptual Rule Learning. In: SOLSO, R. (ed): Theories in Cognitive Psychology. WASHINGTON, D.C.: ERLBAUM, 1974, 231-256

BROWN, J.S.; van LEHN, K.: Repair Theory: A Generative Theory of Bugs in Procedural Skills. Cognitive Science, 1980, 4, 379-426

CARROLL, J.M.: Minimalist Design for Active Users. In: SHACKLE, B. (ed): Interact 84, First IFIP Conference on Human-Computer-Interaction. Amsterdam: Elsevier/North Holland, 1984a

CARROLL, J.M.: Minimalist Training. Datamation, 1984b, 125-136

CHANG, S.K., Visual Languages: A Tutorial and Survey, in: P.GORNY & M.J.TAUBER (eds), Visualization in Programming, Lecture Notes in Computer Science, Heidelberg: Springer , 1987, 1- 23

CHASE, W. G., Visual Information Processing, in: K.R. BOFF, L. KAUFMAN & J.P. THOMAS (eds), Handbook of Perception and Human Performance, Vol. II, Cognitive Processes and Performance, New York: Wiley, 1986, 28-1 - 28-71

COLONIUS, FRANK, JANKE, KOHNERT, MÖBUS, SCHRÖDER & THOLE, Stand des DFG-Projekts "Entwicklung einer Wissensdiagnostik- und Fehlererklärungskomponente beim Erwerb von Programmierwissen für ABSYNT", in: R. GUNZENHÄUSER & H. MANDL (Hrsgb), "Intelligente Lernsysteme", S. 80 - 90, 1987, Institut für Informatik der Universität Stuttgart & Deutsches Institut für Fernstudien an der Universität Tübingen

COLONIUS, FRANK, JANKE, KOHNERT, MÖBUS, SCHRÖDER & THOLE, Syntaktische und semantische Fehler in funktionalen graphischen Programmen, ABSYNT Report 2/87, 1987

DAVIS, R.E., Runnable Specification as a Design Tool, in: K.L. CLARK & s.A. TÄRNLUND (eds), Logic Programming, New York: Academic Press, 1982, 141 - 149

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

GENESERETH, M.R.; NILSSON, N..J.: Logical Foundations of Artificial Intelligence. Los Altos, California: Morgan Kaufman, 1987

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, 221-251

HAYGOOD, R.C.; BOURNE, L.E.; Attribute- and Rule Learning Aspects of Conceptual Behaviour. Psychological Review, 1965, 72, 175-195

JANKE,G. & KOHNERT,K., Interface Design of a Visual Programming Language: Evaluating Runnable Specifications According to Psychological Criteria, paper presented at MACINTER, 1988, Berlin/GDR, ABSYNT-Report 5

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

KOHNERT, K. & JANKE, G.: The Object-Oriented Inplementation of the ABSYNT-Environments. ABSYNT-Report 4/88, Project ABSYNT, FB 10, Unit on Tutoring and Learning Systems, University of Oldenburg, 1988

LARKIN, J.H.; SIMON, H.A.: Why a Diagram is (Sometimes) Worth More Than Ten Thousand Words. Cognitive Science, 1987, 11, 65-99

MEDIN, D.L.; WATTENMAKER, W.D.; MICHALSKI, R.S.: Constraints and Preferences in Inductive Learning: An Experimental Study of Human and Machine Performance. Cognitive Science, 1987, 11, 299-339

MIYAKE, N.: Constructive Interaction and the Iterative Process of Understanding. Cognitive Science, 10, 1986, 151-177

MÖBUS, C., Die Entwicklung zum Programmierexperten durch das Problemlösen mit Automaten, in: MANDL & FISCHER (Hrsgb), Lernen im Dialog mit dem Computer, München: Urban & Schwarzenberg, 1985, 140-154

MÖBUS, C. & SCHRÖDER, O., Knowledge Specification and Instructions for a Visual Computer Language, to appear in: F.KLIX, H.WANDKE, N.A.STREITZ & Y.WAERN (eds), Man-Computer Interaction Research, MACINTER II, 1988, Amsterdam: North-Holland (in press)

MÖBUS, C. & THOLE, H.J., Tutors, Instructions and Helps, ABSYNT-Report 3/88, to appear in: CHRISTALLER, Th. (ed), Künstliche Intelligenz. KIFS87, Heidelberg: Springer, Computer Science Lecture Series (in press)

PAGAN, F.G., Formal Specification of Programming Languages, Englewood Cliffs, N.J.: Prentice-Hall, 1981

PAYNE, S.J.; SIME, M.E.; GREEN, T.R.G.: Perceptual Structure Cueing in a Simple Command Language. Int. Journal of Man-Machine Studies, 1984, 21, 19-29

PENNINGTON, N.: Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. Cognitive Psychology, 1987, 19, 295-341

POMERANTZ, J.R.: Perceptual Organization in Information Processing. In: AITKENHEAD, A.M.; SLACK, J.M. (eds): Issues in Cognitive Modeling. Hillsdale: Erlbaum, 1985, 127-158

SCHRÖDER, O., FRANK, K.D. & COLONIUS, H., Gedächtnisrepräsentation funktionaler, graphischer Programme, ABSYNT-Report 1/87, Universität Oldenburg, 1987

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, 11-34

SLEEMAN, D.H. & HENDLEY, R.J., ACE: A system which Analyses Complex Explanations, in: D.SLEEMAN & J.S.BROWN (eds), Intelligent Tutoring Systems, New York: Academic Press, 1982, 99 - 118

SOLOWAY, E.: Learning to Program = Learning to Construct Mechanisms and Explanations. Communications of the ACM, 29, 9, 1986, 850-858

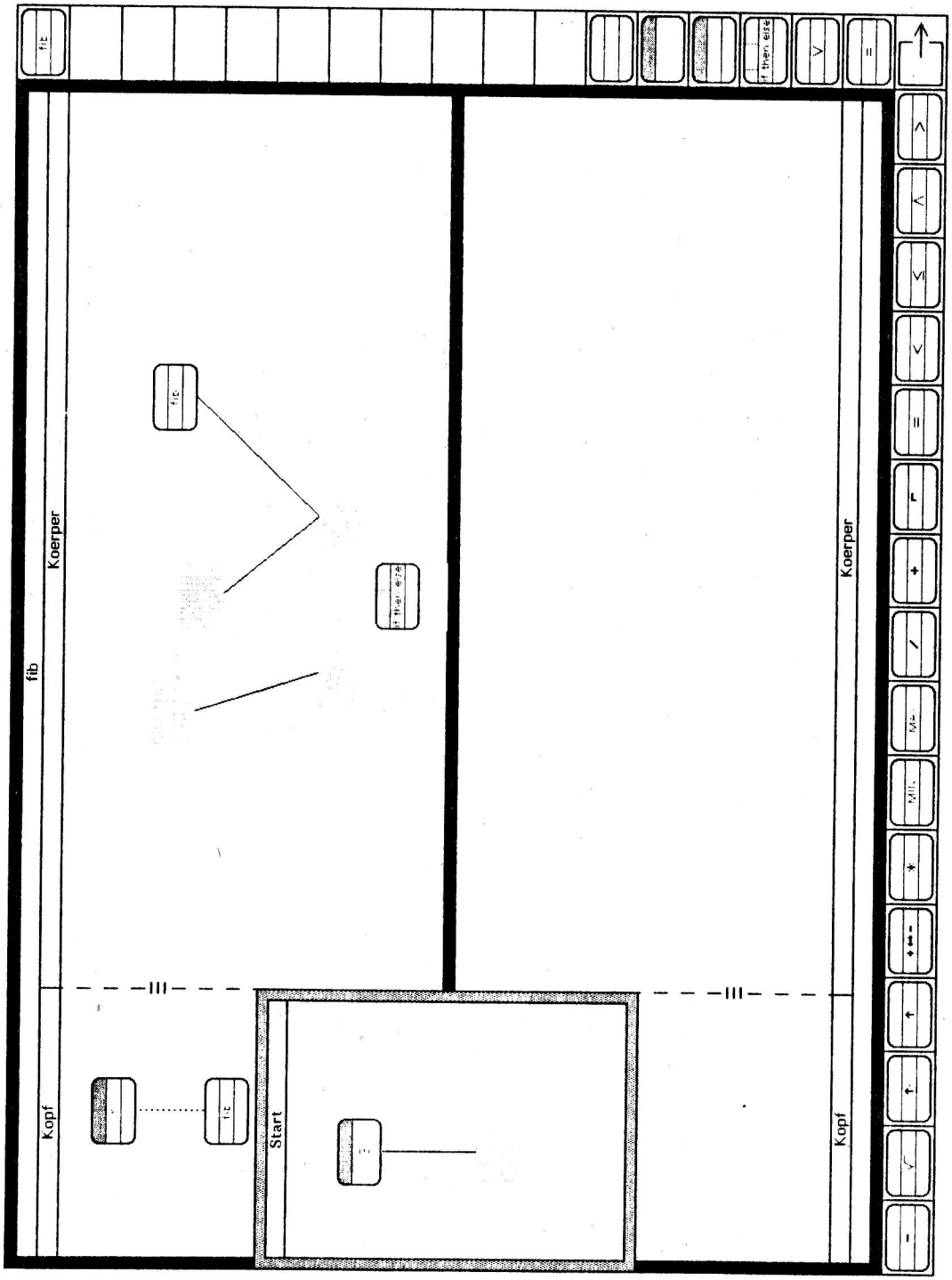


FIGURE 1: The programming mode of ABSYNT

FIGURE 2: The Trace Mode of ABSYNT

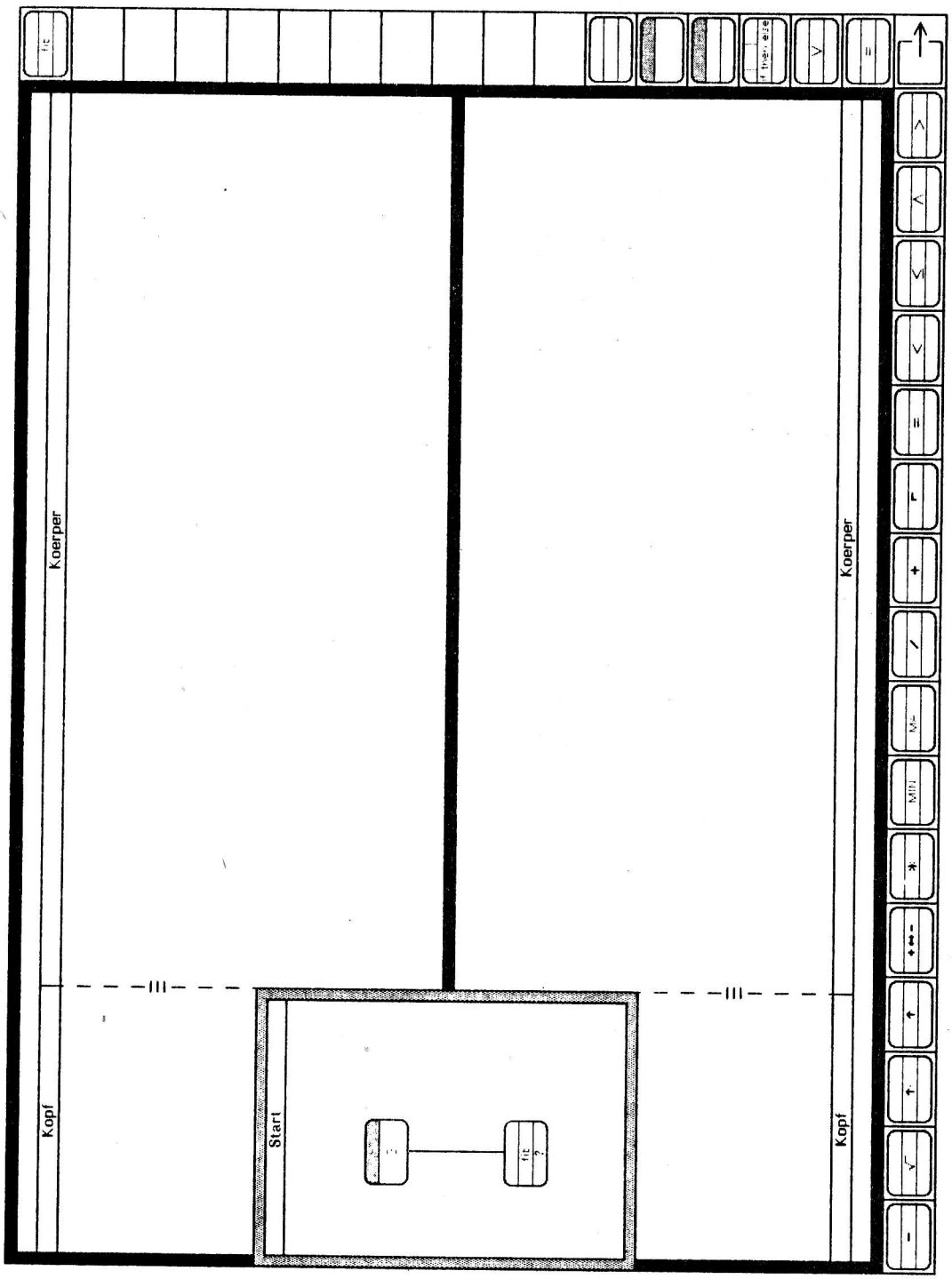


FIGURE 2a: The Initial Computation Step

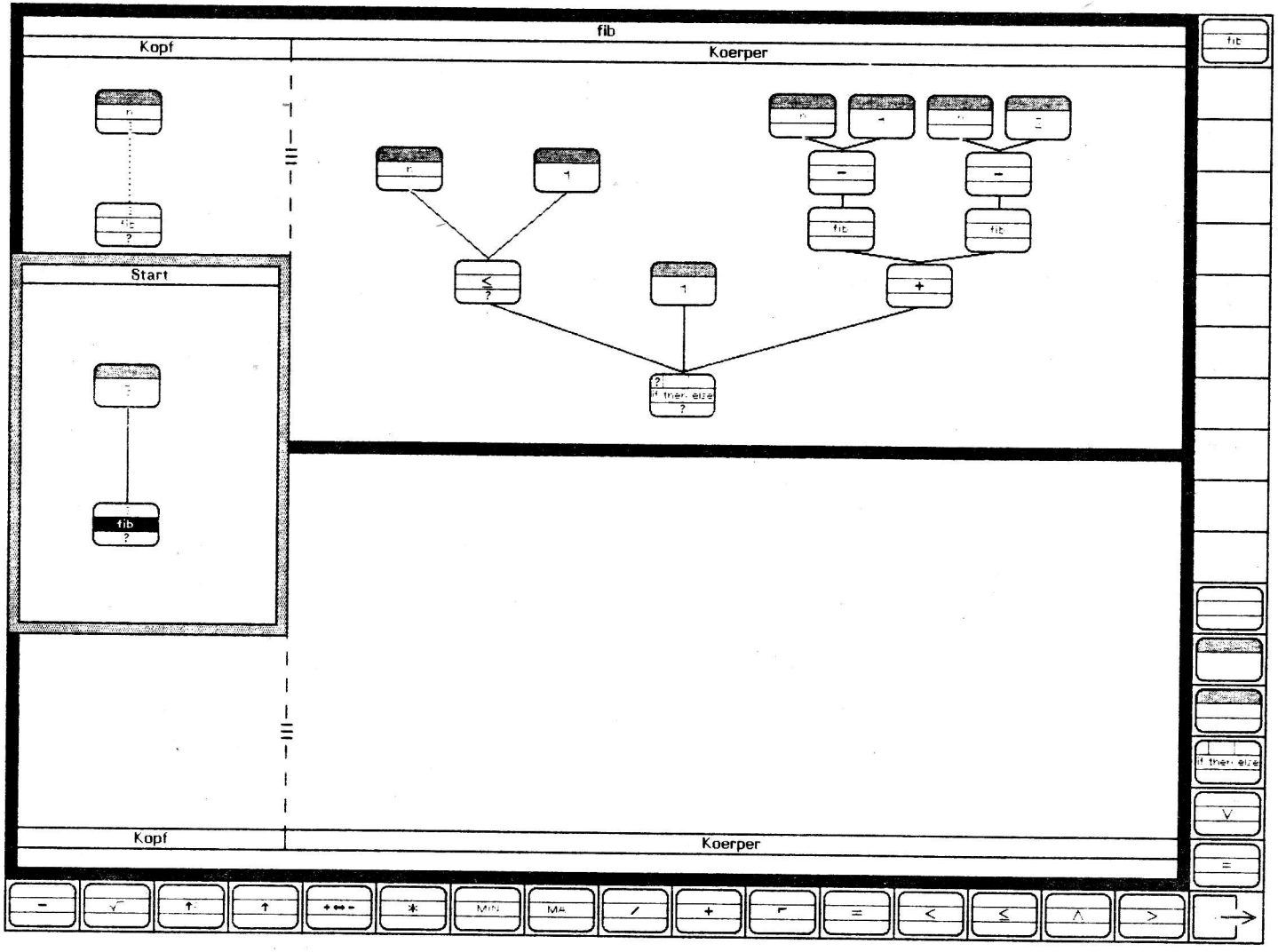


FIGURE 2b: Last computation step corresponds to the 1st action of "operator-specific" rule 2, resp. to the action of "state-specific" rule 3

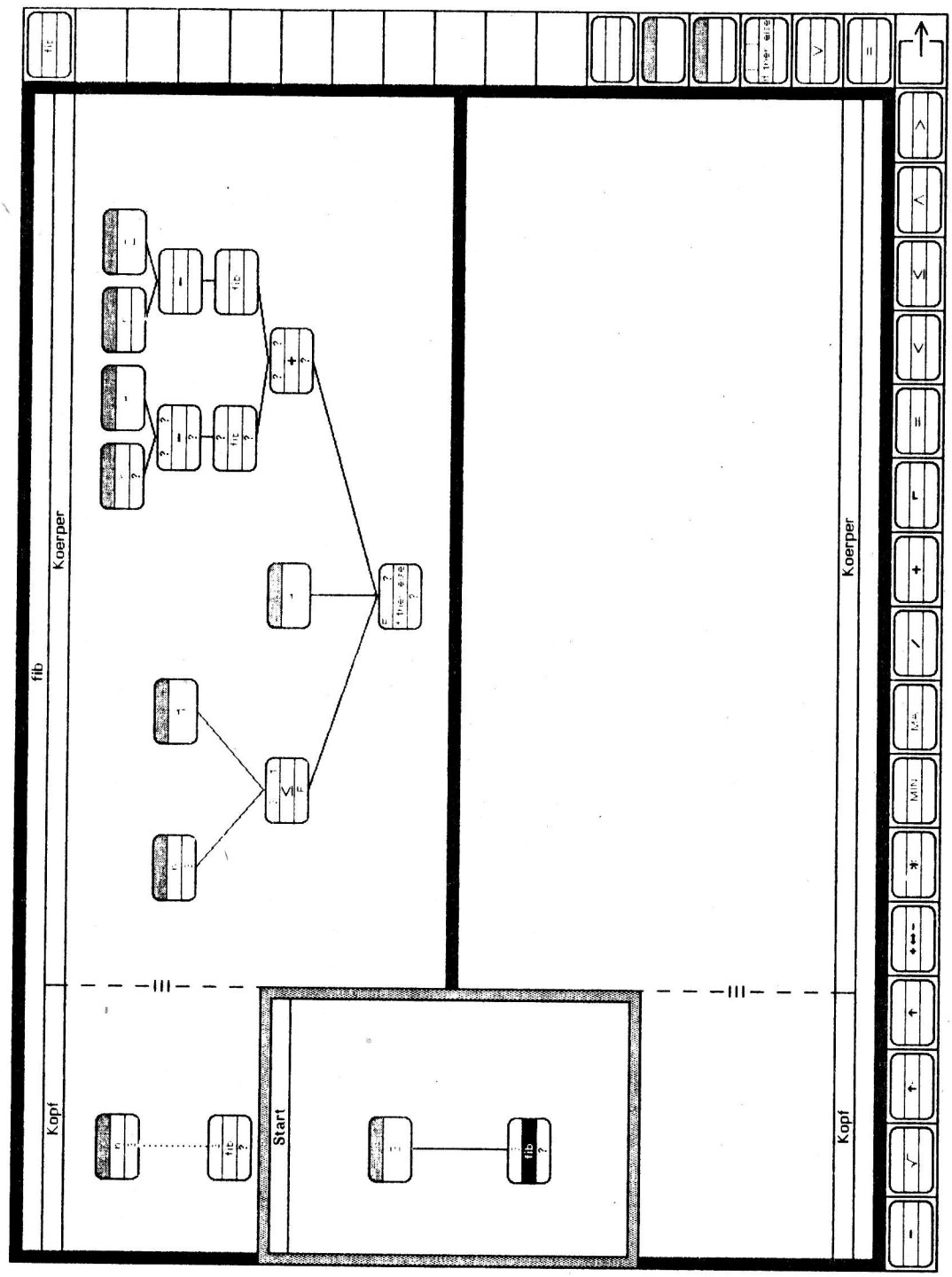


FIGURE 2c: Last computation step corresponds to the 1st action of "operator-specific" rule 2, resp. to the action of "state-specific" rule 3

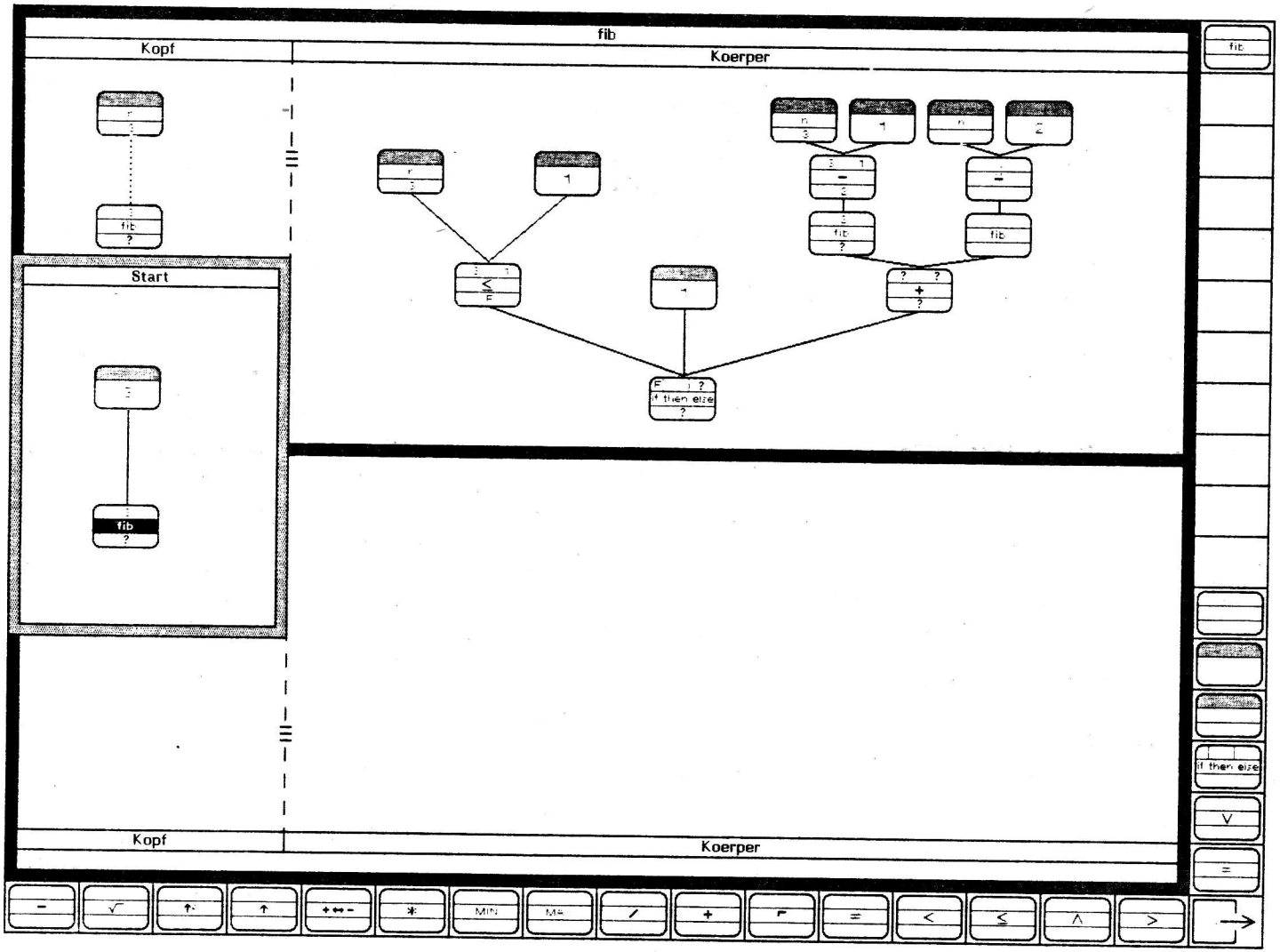


FIGURE 2d: Last computation step corresponds to the 2nd action of "operator-specific" rule 2, resp. to the action of "state-specific" rule 4

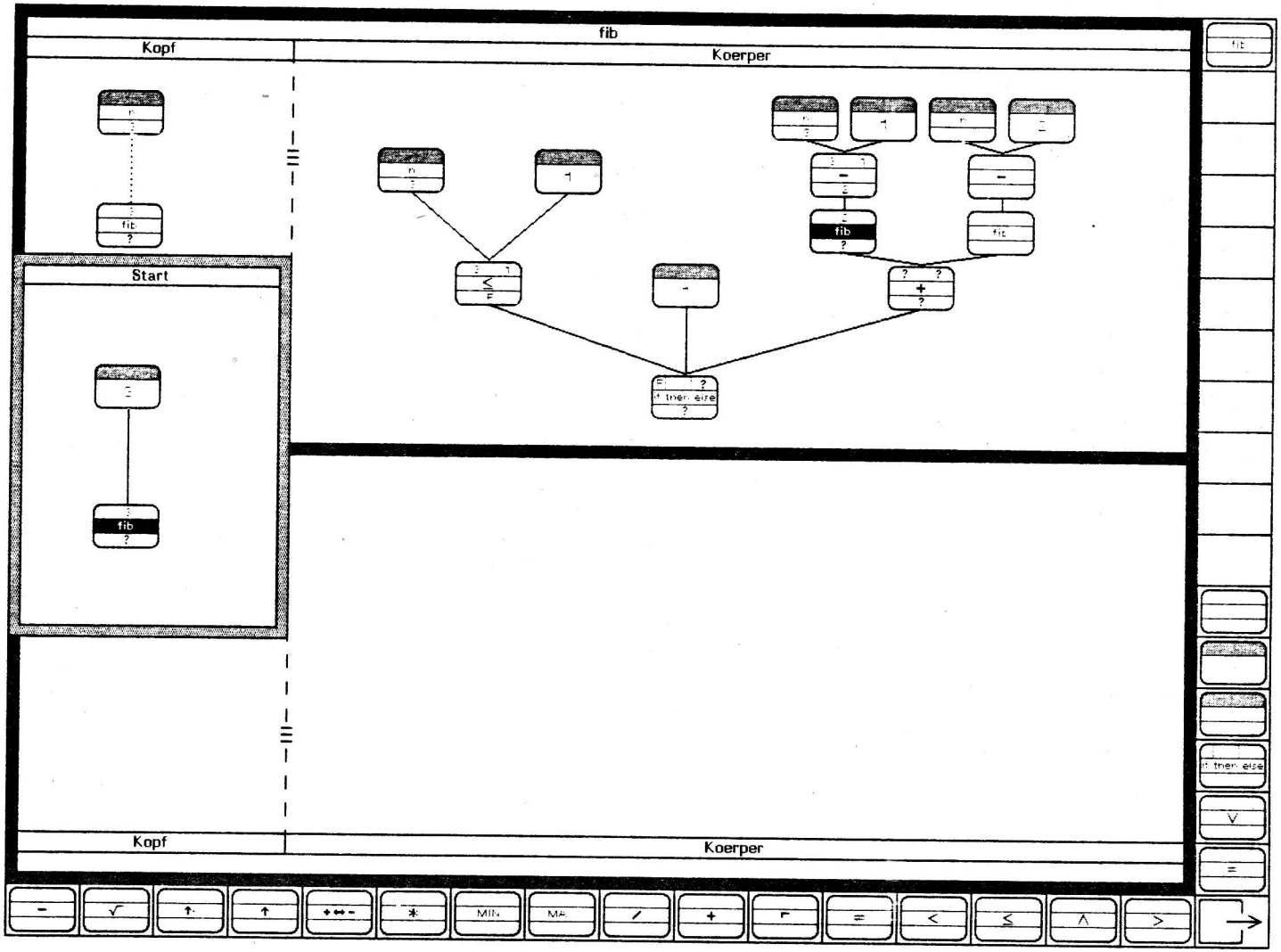


FIGURE 2e: Last computation step corresponds to part of the 2nd action of "operator-specific" rule 8, resp. to part of the action of "state-specific" rule 15

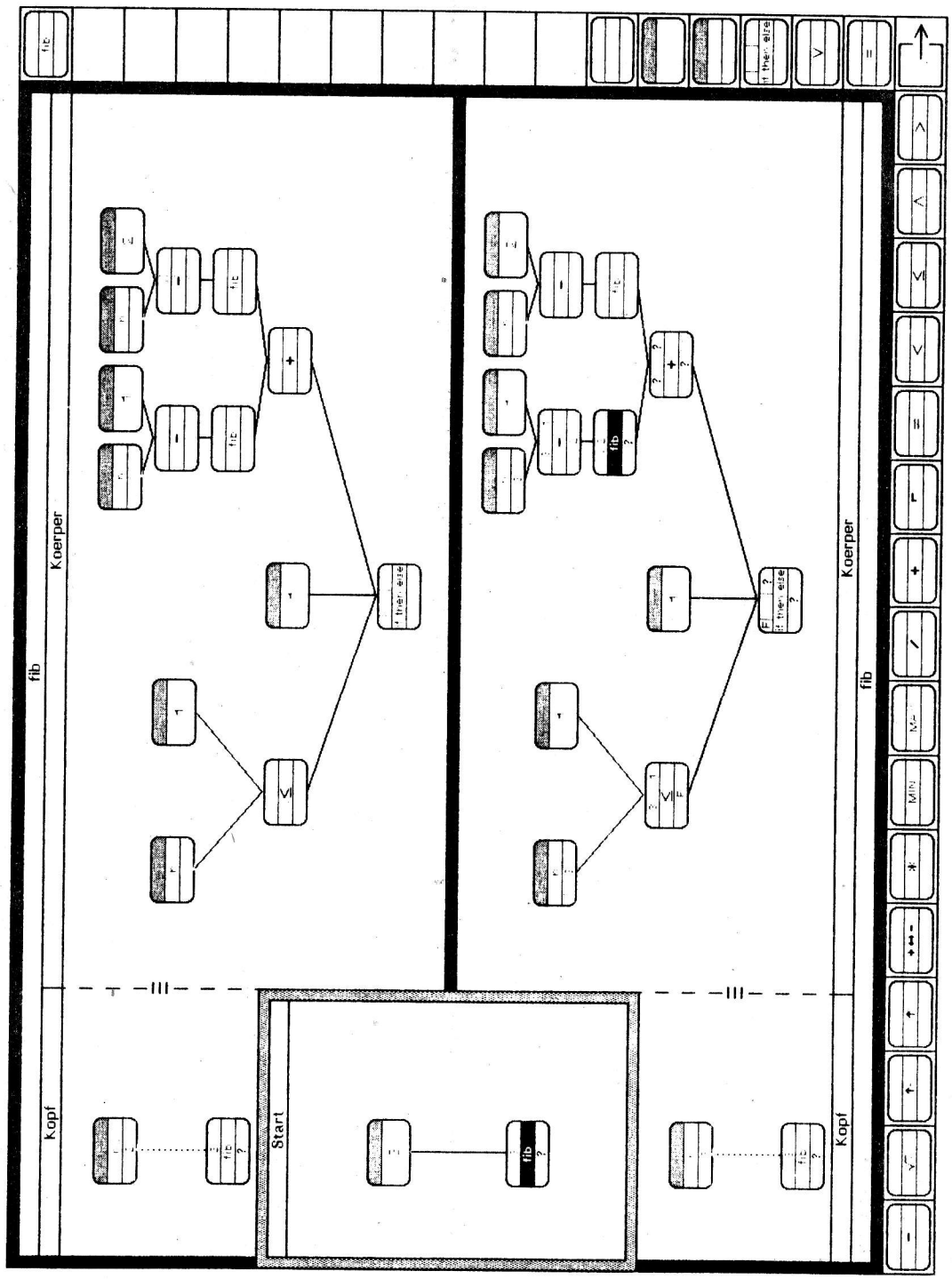


FIGURE 2f: Last computation step corresponds to part of the 2nd action of "operator-specific" rule 8, resp. to part of the action of "state-specific" rule 15

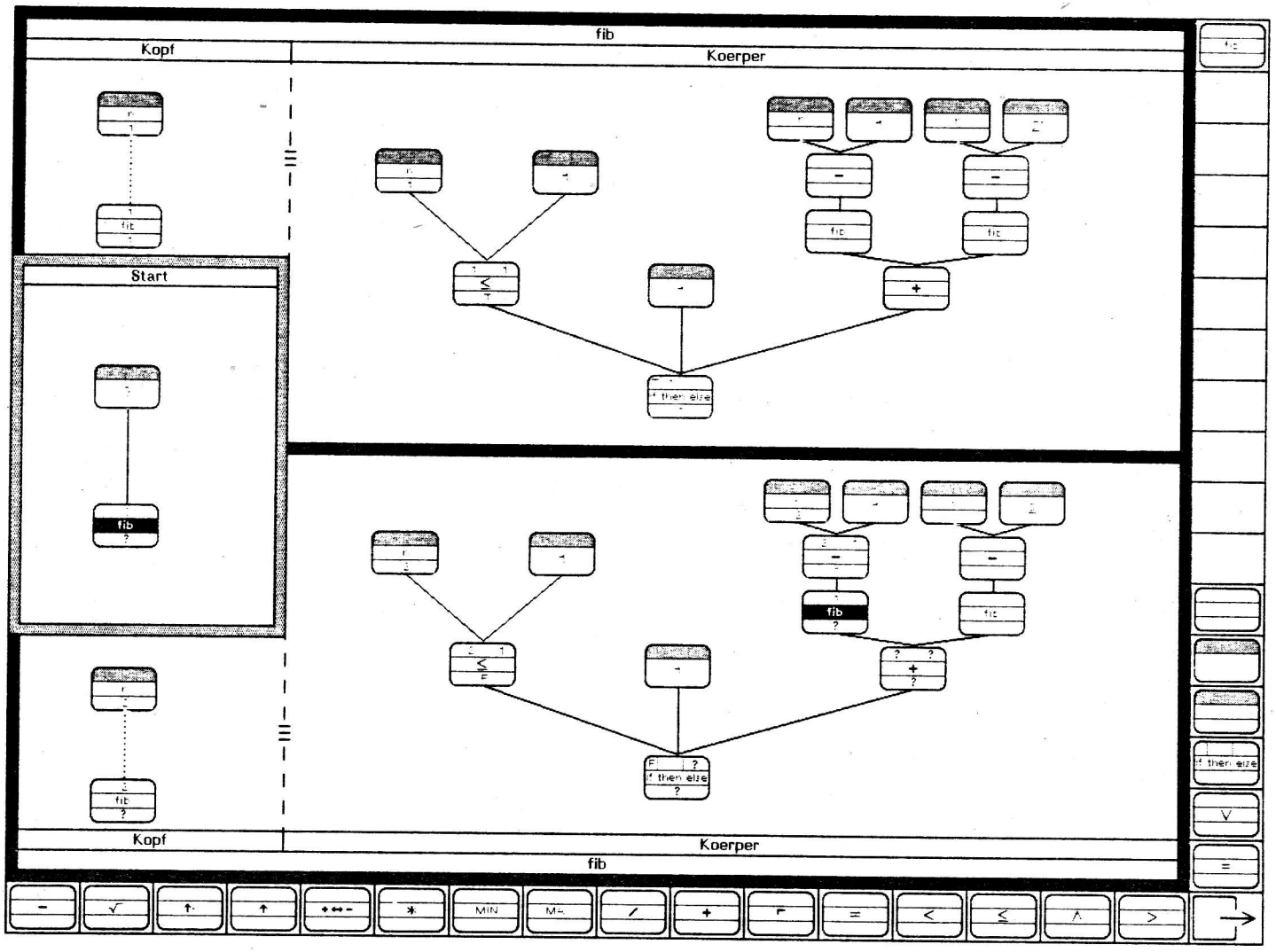


FIGURE 2g: Last computation step corresponds to the 2nd action of "operator-specific" rule 3, resp. to the action of "state-specific" rule 6

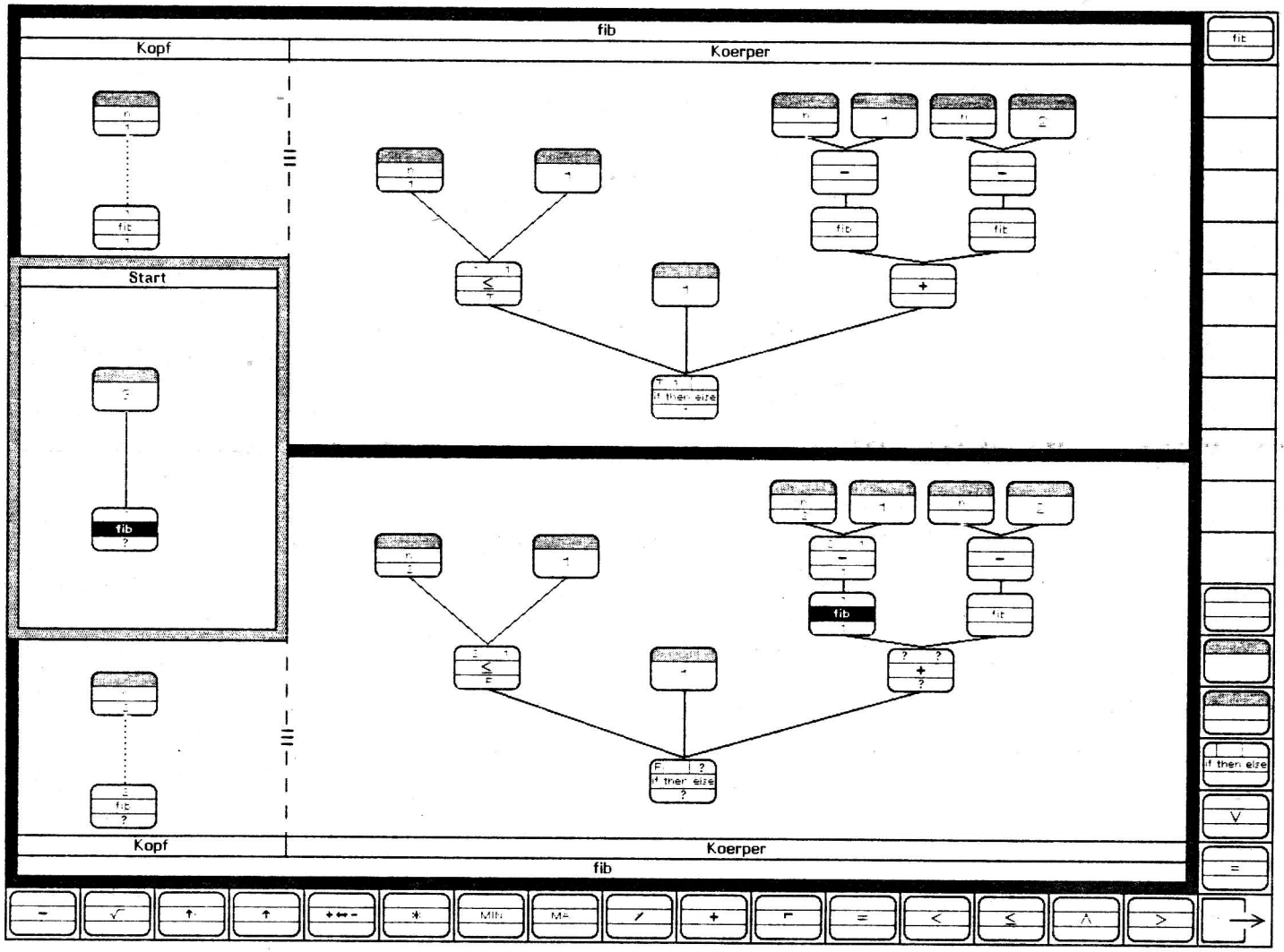


FIGURE 2h: Last computation step corresponds to part of the 3rd action of "operator-specific" rule 8, resp. to part of the action of "state-specific" rule 16

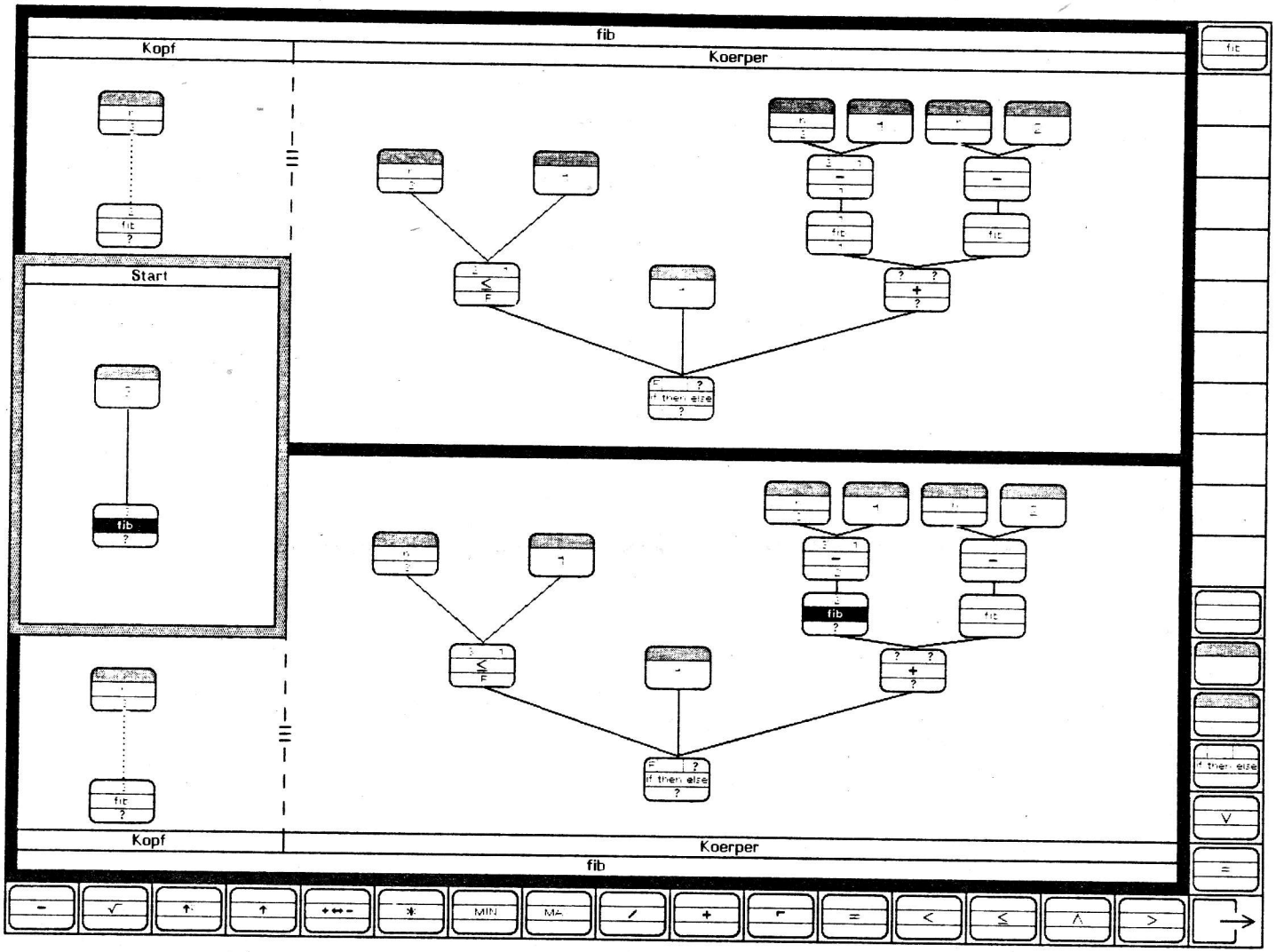


FIGURE 2i: Last computation step corresponds to part of the 3rd action of "operator-specific" rule 8, resp. to part of the action of "state-specific" rule 16

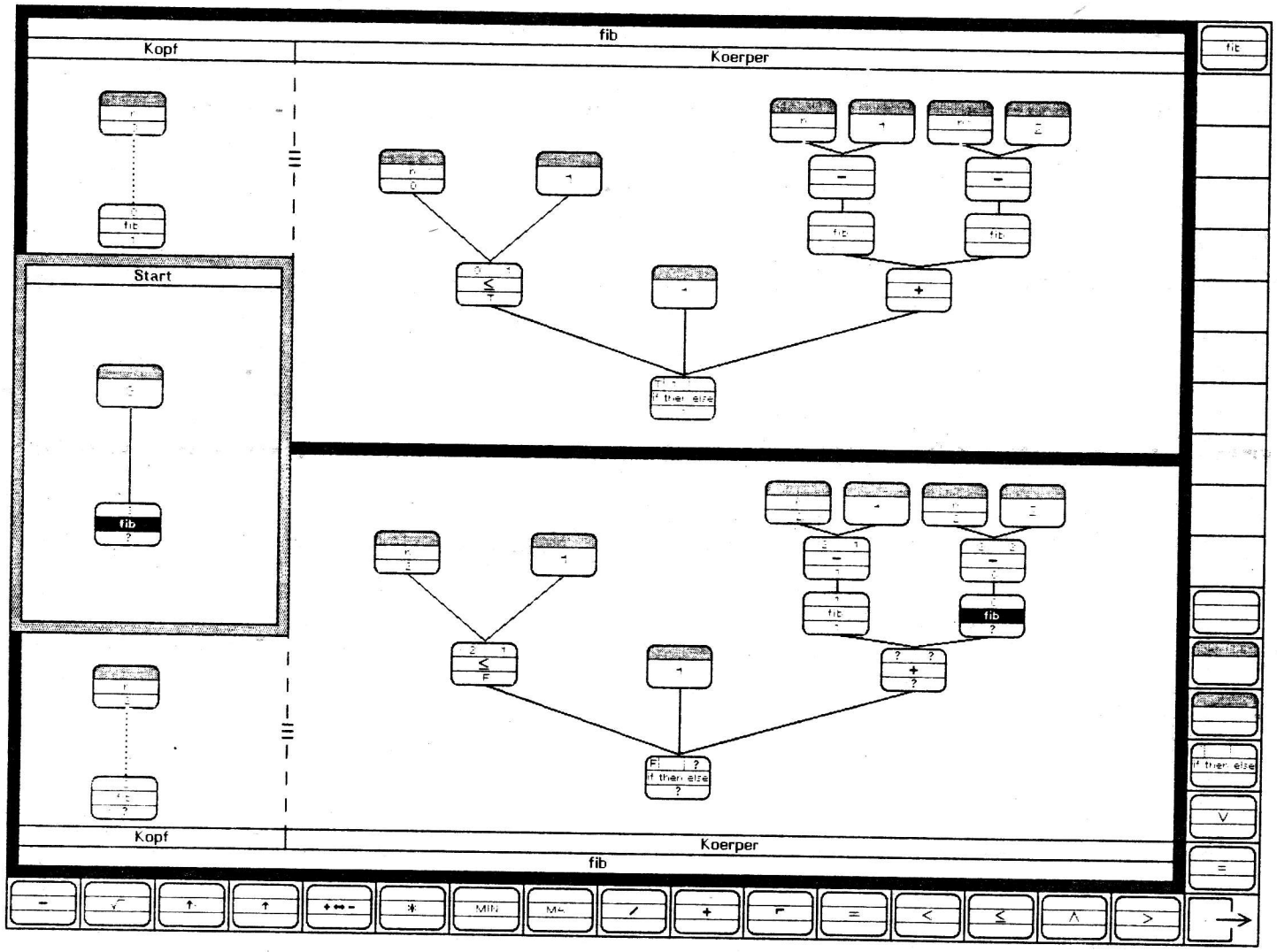


FIGURE 2j: Last computation step corresponds to the 2nd action of "operator-specific" rule 3, resp. to the action of "state-specific" rule 6

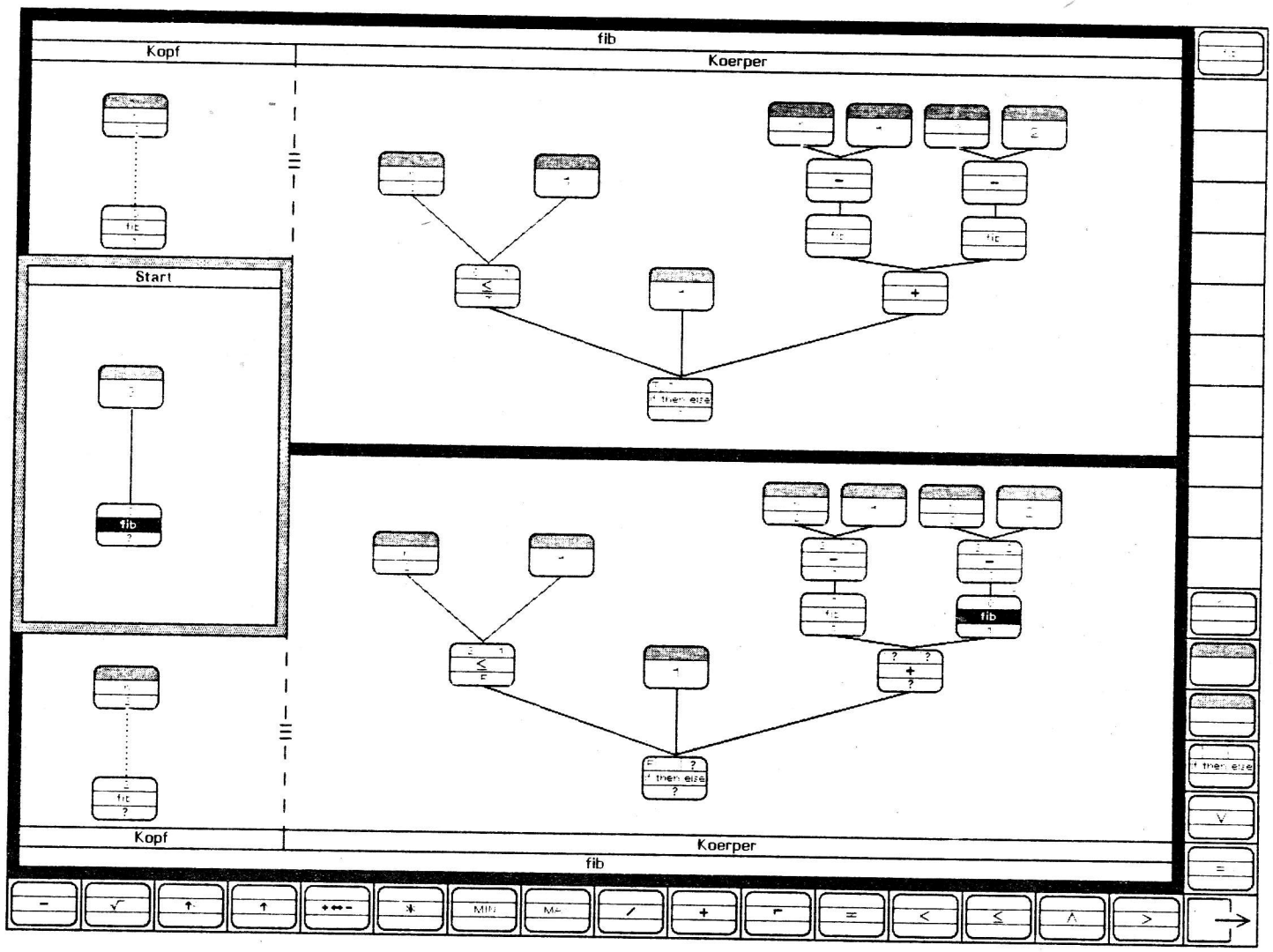


FIGURE 2k: Last computation step corresponds to part of the 3rd action of "operator-specific" rule 8, resp. to part of the action of "state-specific" rule 16

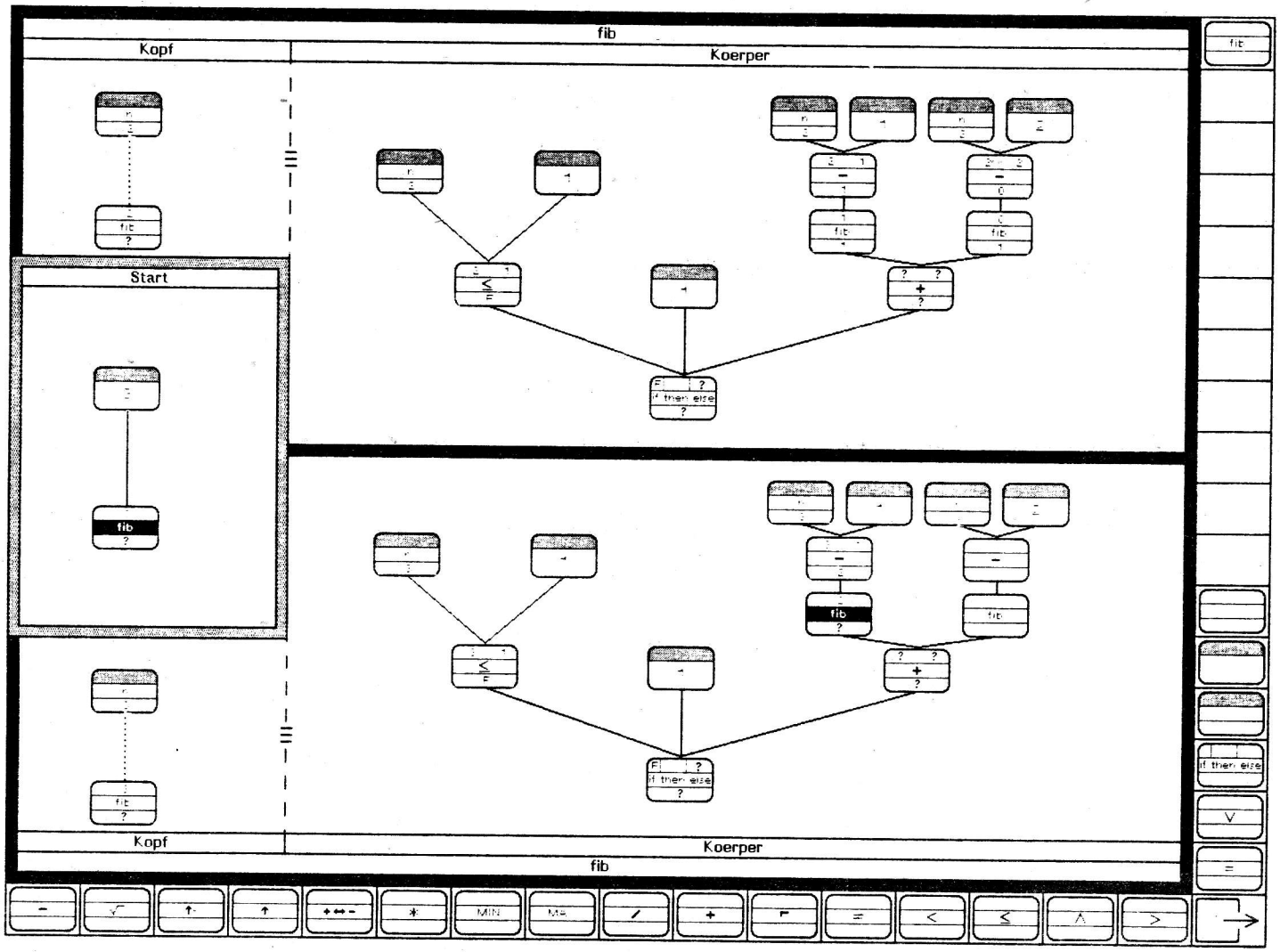


FIGURE 2l: Last computation step corresponds to part of the 3rd action of "operator-specific" rule 8, resp. to part of the action of "state-specific" rule 16

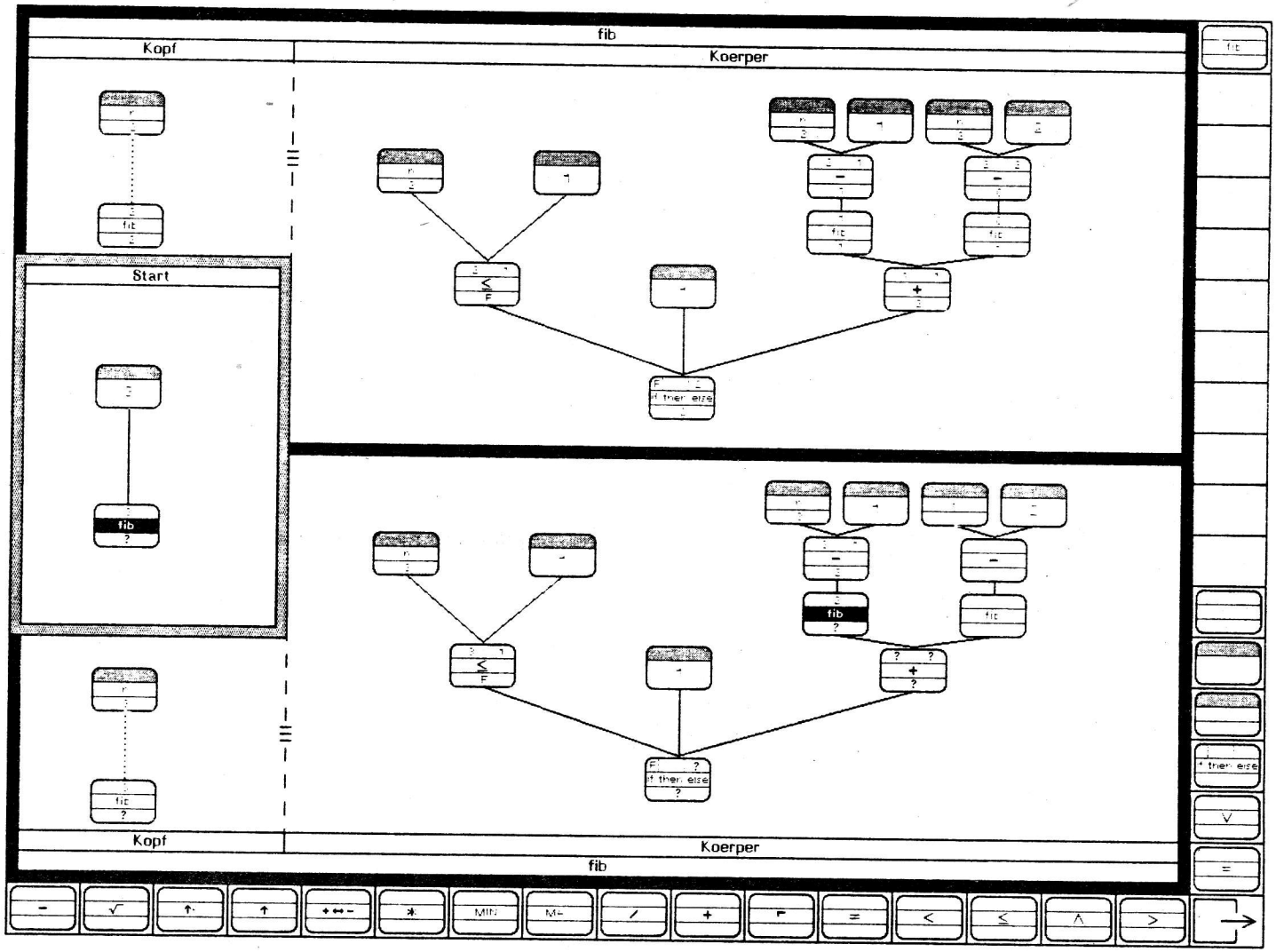


FIGURE 2m: Last computation step corresponds to the 2nd action of "operator-specific" rule 3, resp. to the action of "state-specific" rule 6

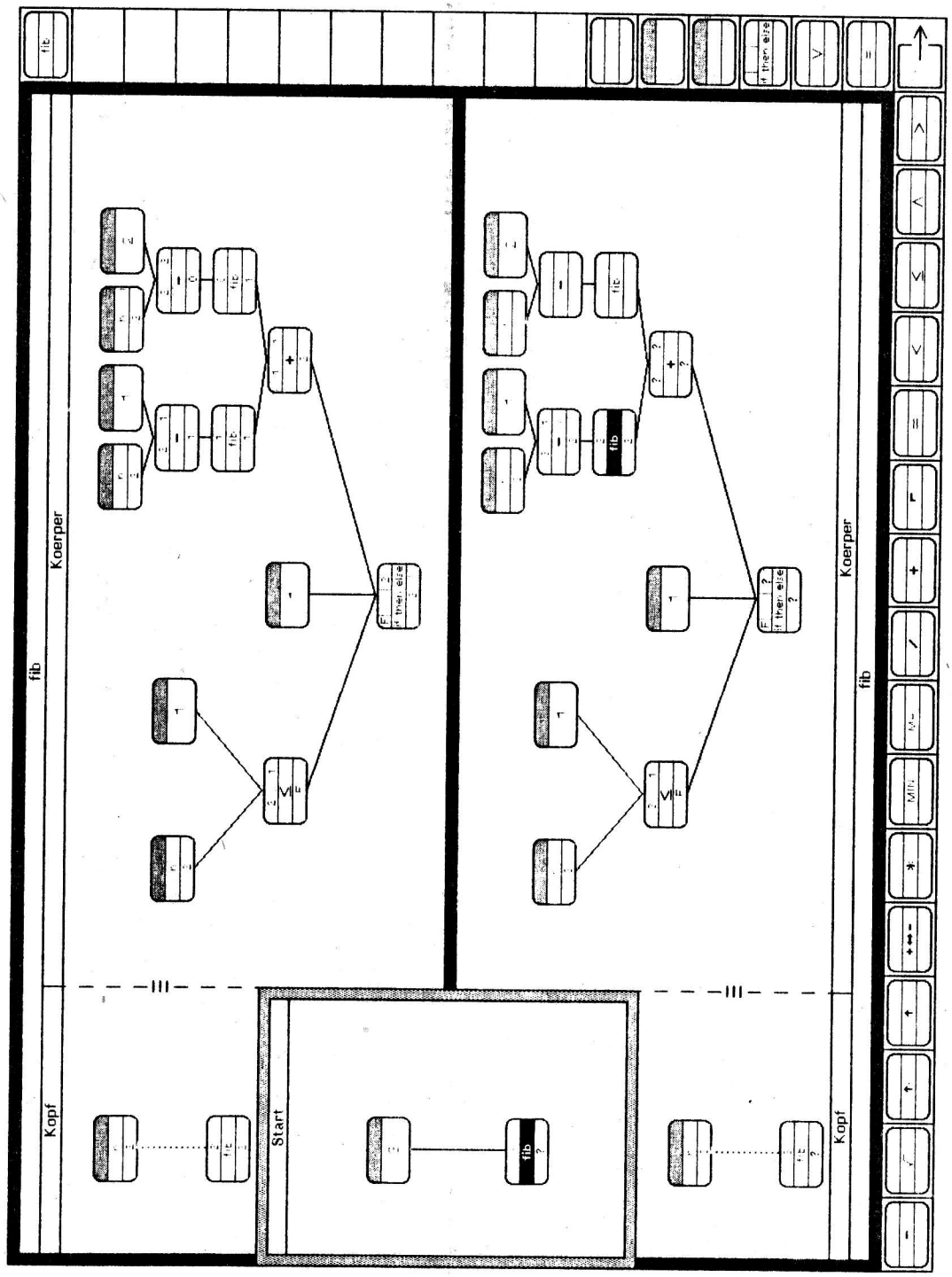


FIGURE 2n: Last computation step corresponds to part of the 3rd action of "operator-specific" rule 8, resp. to part of the action of "state-specific" rule 16

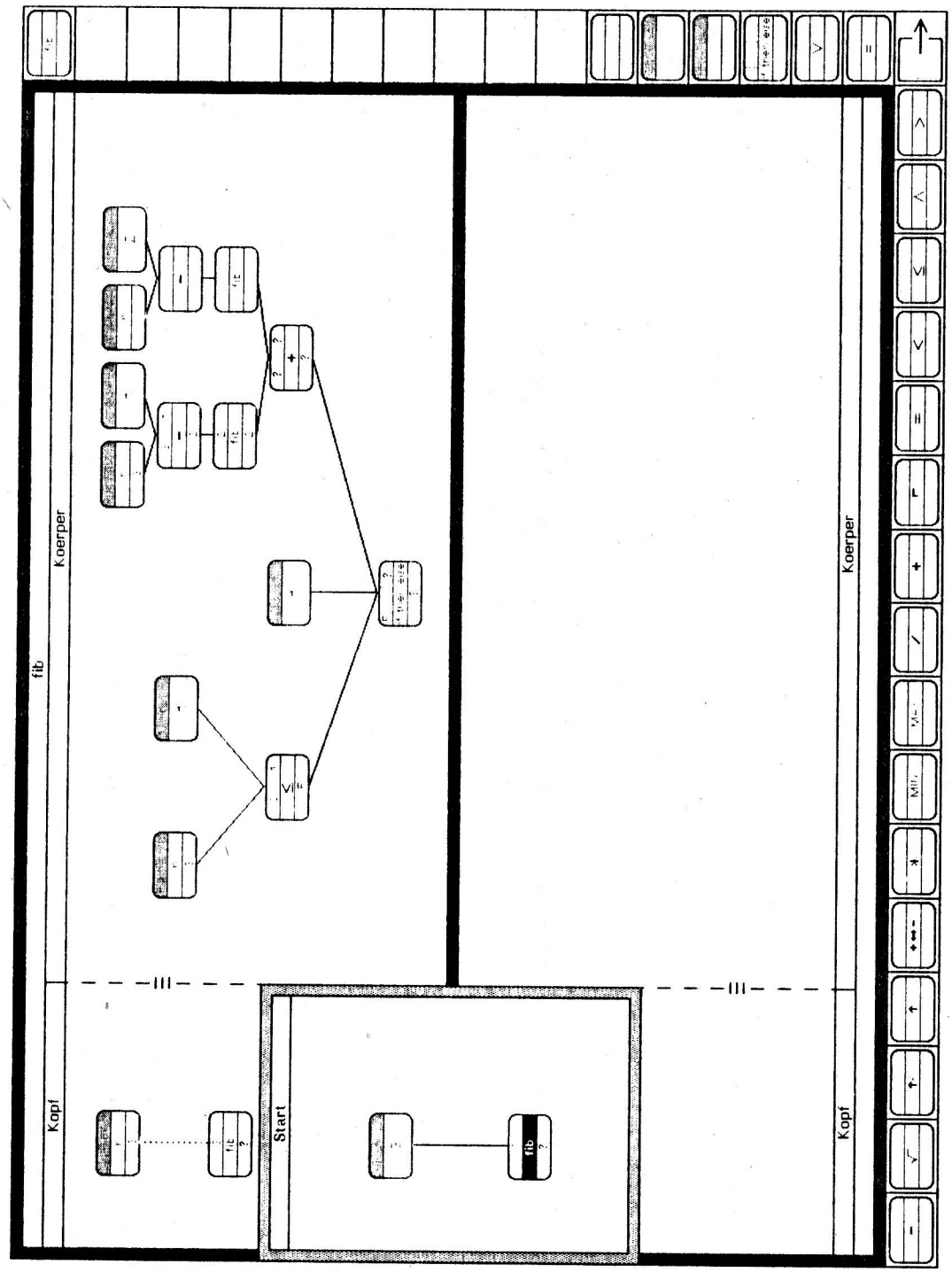


FIGURE 2o: Last computation step corresponds to part of the 3rd action of "operator-specific" rule 8, resp. to part of the action of "state-specific" rule 16

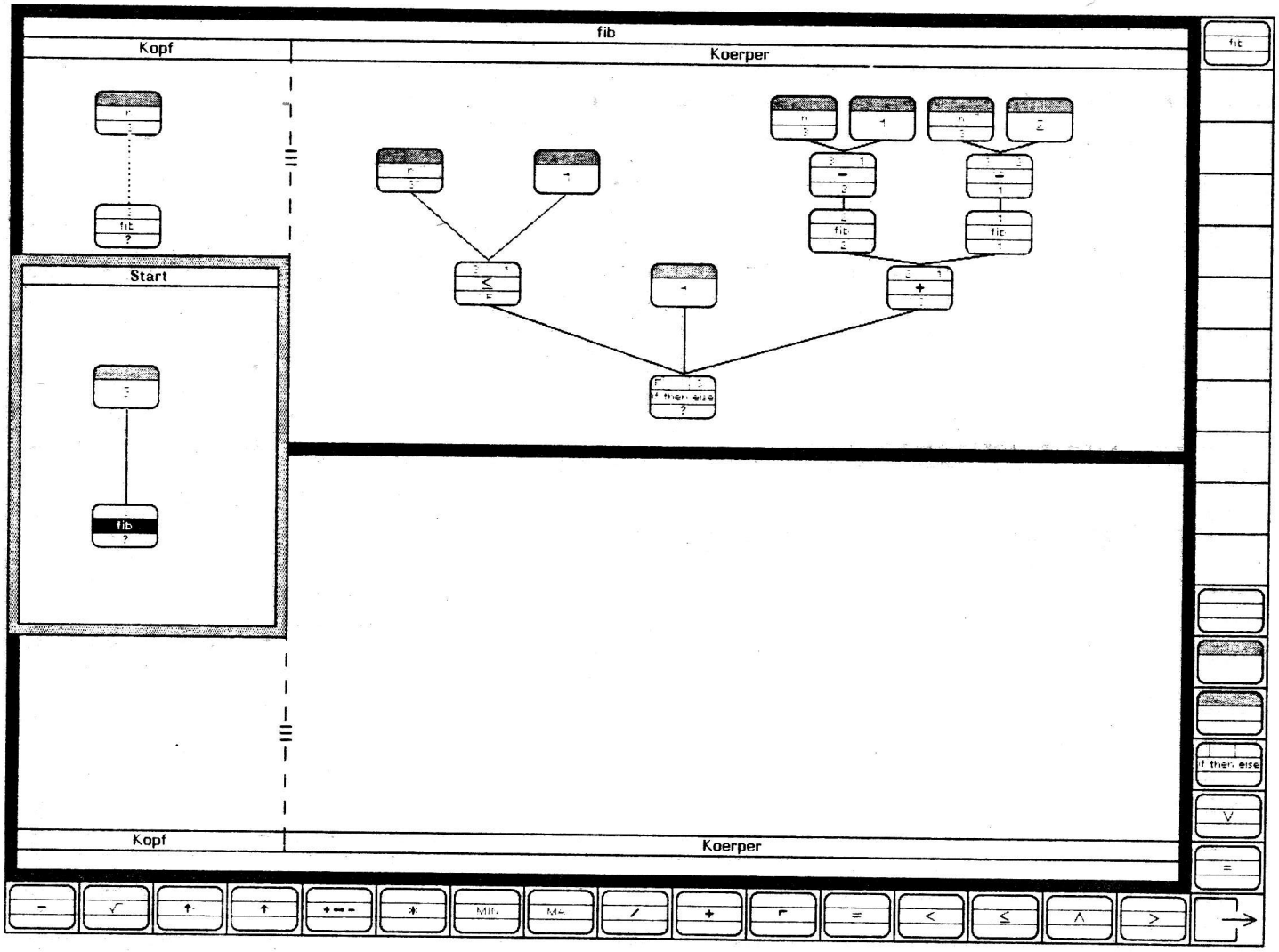


FIGURE 2p: Last computation step corresponds to the 2nd action of "operator-specific" rule 2, resp. to the action of "state-specific" rule 4

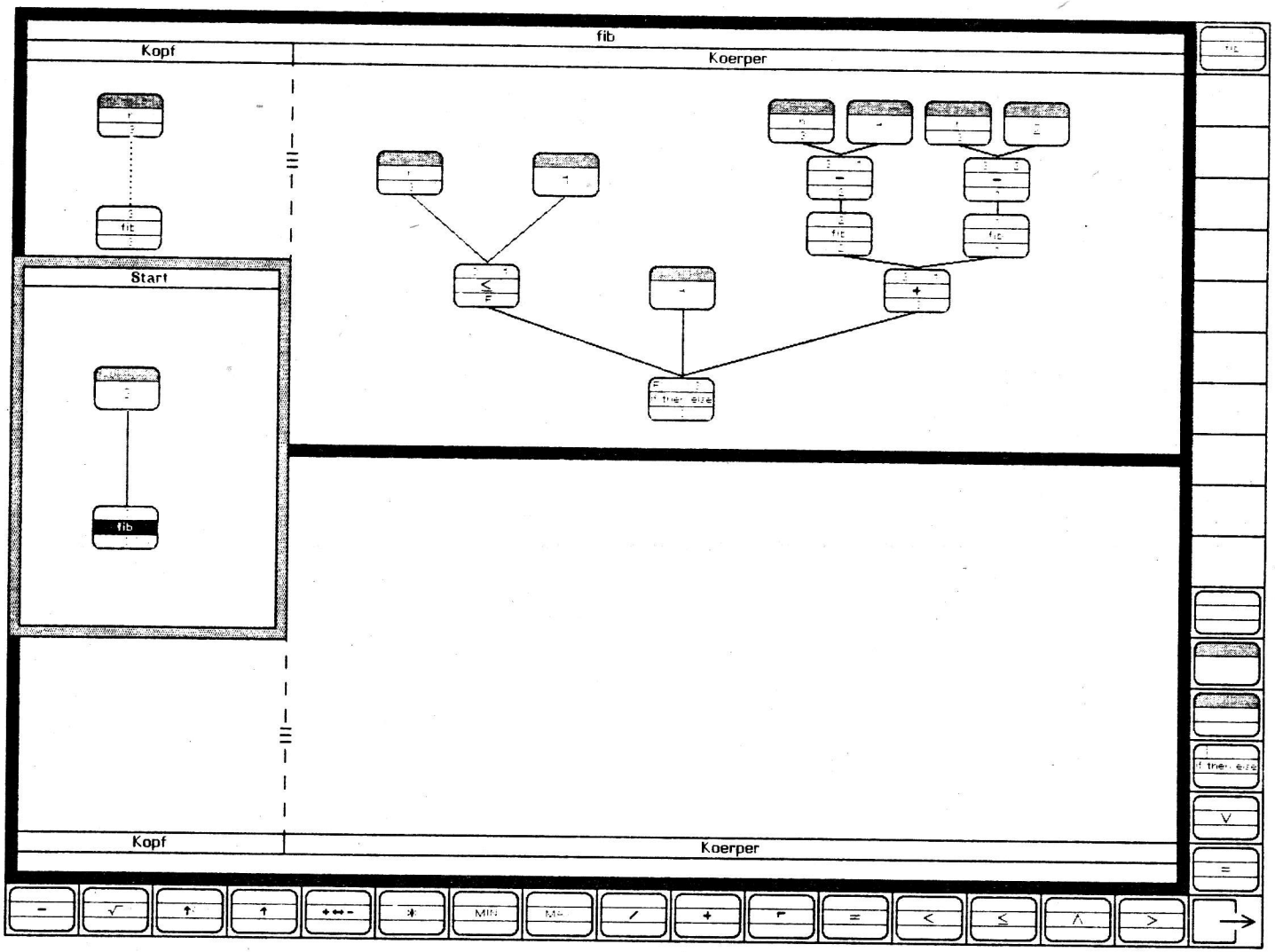


FIGURE 2q: Last computation step corresponds to part of the 3rd action of "operator-specific" rule 5, resp. to part of the action of "state-specific" rule 9

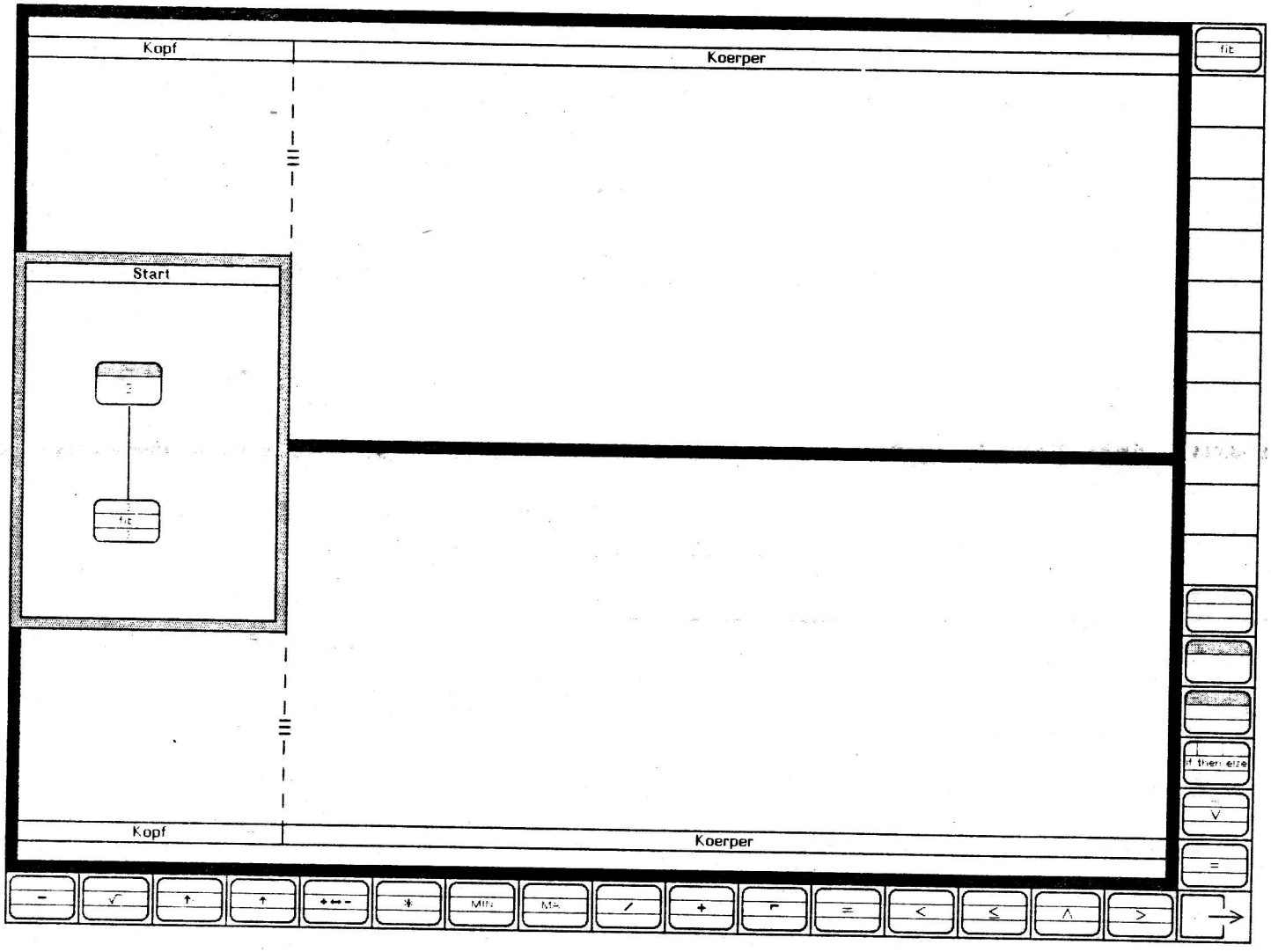


FIGURE 2r: Last computation step corresponds to part of the 3rd action of "operator-specific" rule 5, resp. to part of the action of "state-specific" rule 9

Appendix A: Representing Computational Knowledge for ABSYNT with Operator Specific 2-D-ruleset

Rule 1: To compute the outputvalue of primitive operator nodes (except IF-THEN-ELSE-nodes !)

Rule 2: To fetch the input value for an operator node

Rule 3: To compute the outputvalue of the root of head

Rule 4: To fetch parameter bindings from the head for leafs in the body

Rule 5: To compute the outputvalue of a higher operator node (= user defined function) in the start tree

Rule 6: To compute the outputvalue of the IF-THEN-ELSE node in case of a true predicate

Rule 7: To compute the outputvalue of the IF-THEN-ELSE node in case of a false predicate

Rule 8: To compute the outputvalue of a higher operator node in a body tree

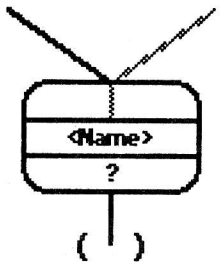
Rule 1: Computing of primitive operator node (No IF-THEN-ELSE-node !).

1st Situation

1) The output stripe of a primitive operator node contains a "?".

2) The primitive operator node is not an IF-THEN-ELSE-node.

3) The input stripe of the primitive operator node is empty.

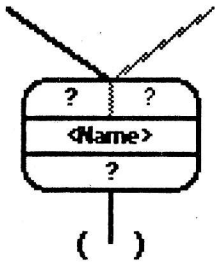


Instruction | Overview | Action

Rule 1: Computing of primitive operator node (No IF-THEN-ELSE-node !).

1st Action

Write a "?" in every input field of the primitive operator node.



Instruction | Overview | Situation

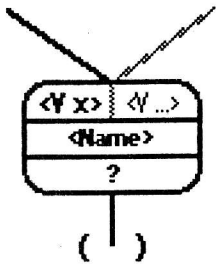
Rule 1: Computing of primitive operator node (No IF-THEN-ELSE-node !).

2nd Situation

1) The output stripe of a primitive operator node contains a "?".

2)The primitive operator node is not an IF-THEN-ELSE-node.

3)The input stripe of the primitive operator node contains values only.



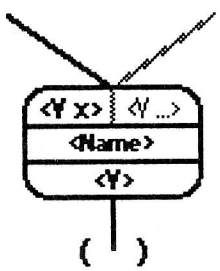
Instruction | Overview | Action

Rule 1: Computing of primitive operator node (No IF-THEN-ELSE-node !).

2nd Action

1) Compute the primitive operator node.

2)Write the value into the output stripe of the primitive operator node.



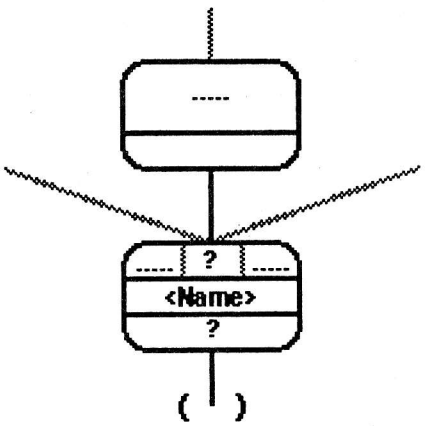
Instruction | Overview | Situation

Rule 2: Fetching an input value for an operator node.

1st Situation

1)Any input field of an operator node contains a "?".

2)The input field of the operator node is connected with another node whose output stripe is empty.

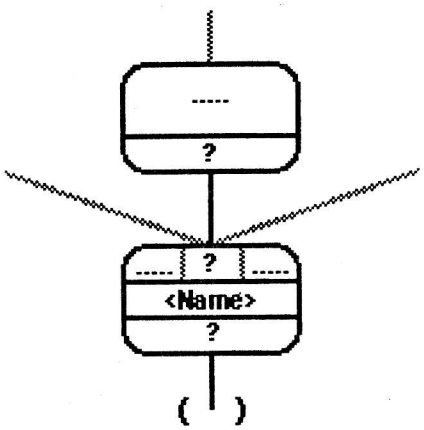


Instruction | Overview | Action

Rule 2: Fetching an input value for an operator node.

1st Action

Write a "?" into the output stripe of the node connected with the input field.



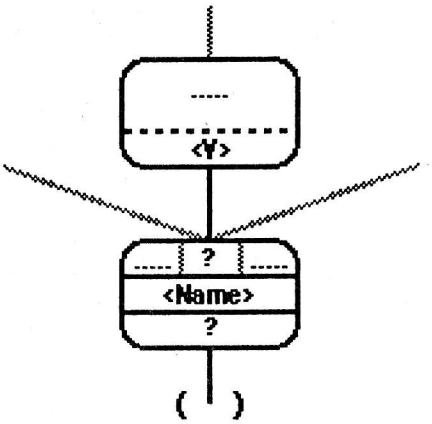
Instruction | Overview | Situation

Rule 2: Fetching an input value for an operator node.

2nd Situation

1)Any input field of an operator node contains a "?".

2)The input field of the operator node is connected with another node whose output stripe contains a value.

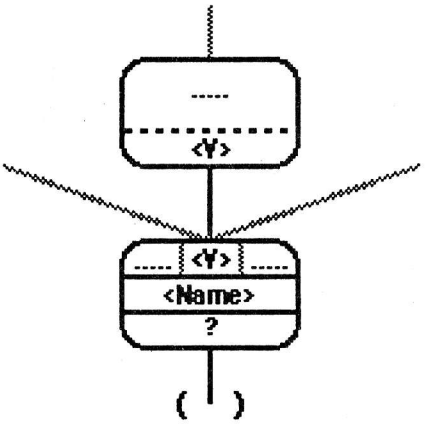


Instruction | Overview | Action

Rule 2: Fetching an input value for an operator node.

2nd Action

Write the output value of the node connected with the input field into the input field.



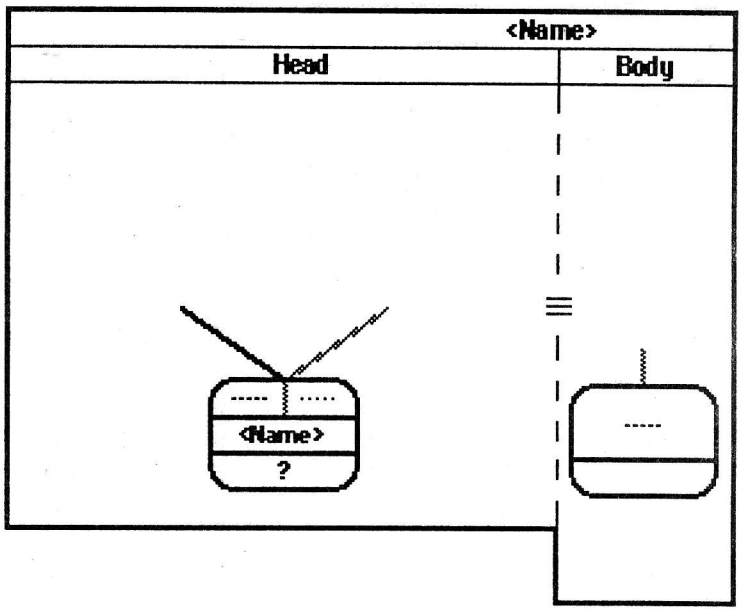
Instruction | Overview | Situation

Rule 3: Fetching output value for head root.

1st Situation

1)The output stripe of the head root of a frame contains a "?".

2)The output stripe of the body root of the frame is empty.

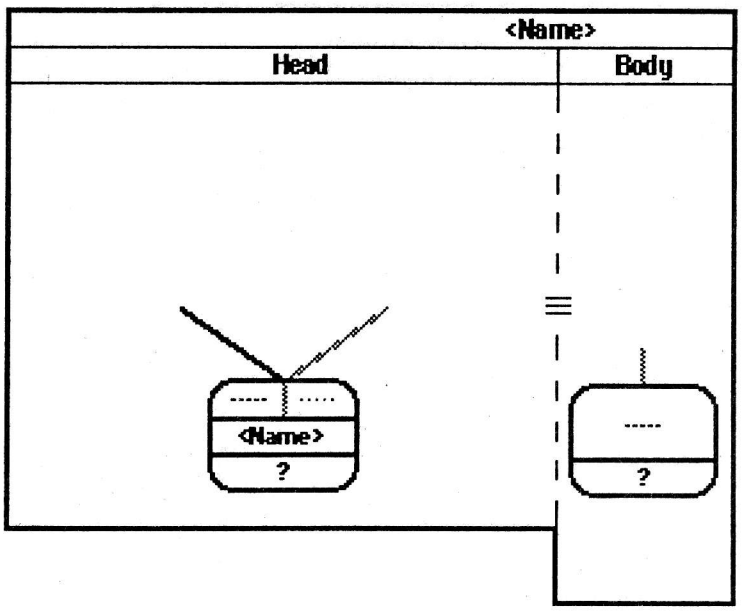


Instruction | Overview | Action

Rule 3: Fetching output value for head root.

1st Action

Write a "?" into the output stripe of the body root of the frame.



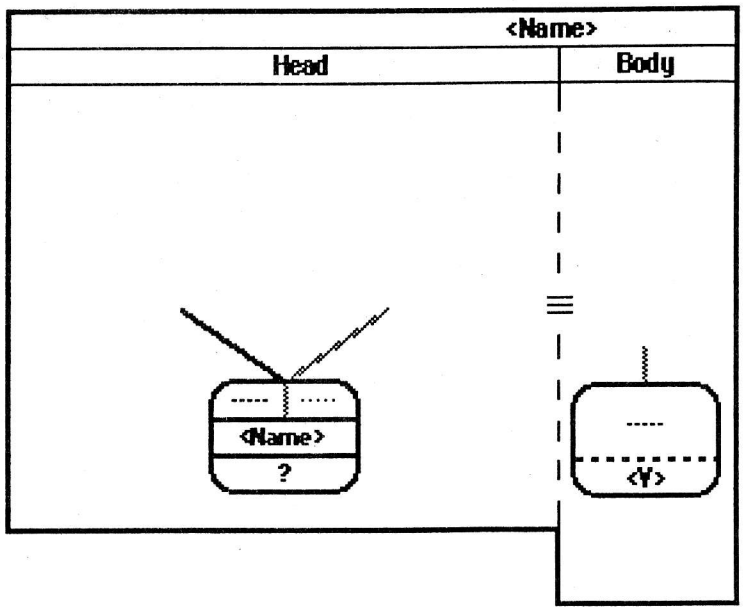
Instruction | Overview | Situation

Rule 3: Fetching output value for head root.

2nd Situation

1)The output stripe of the head root of a frame contains a "?".

2)The output stripe of the body root of the frame contains a value.

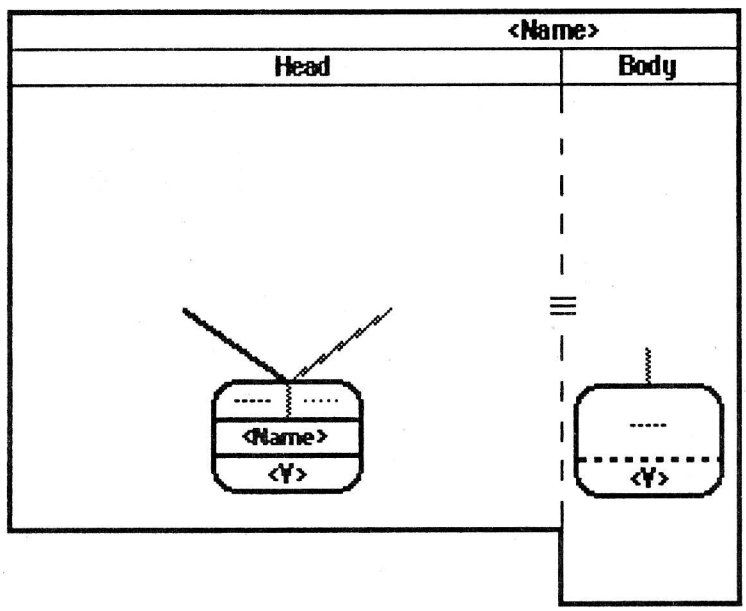


Instruction | Overview | Action

Rule 3: Fetching output value for head root.

2nd Action

Write the output value of the body root of the frame into the output stripe of the head root.

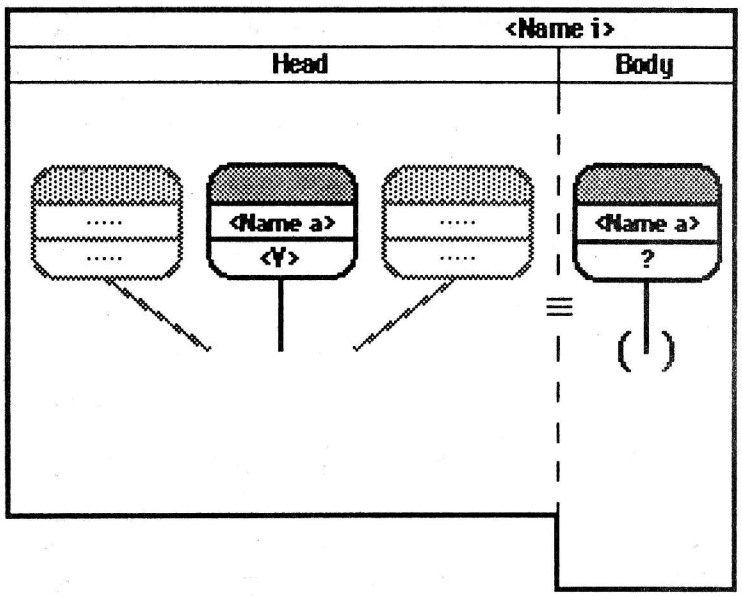


Instruction | Overview | Situation

Rule 4: Fetching parameter value from head for body.

Situation

The output stripe of a bodyleaf of a frame contains a "?".

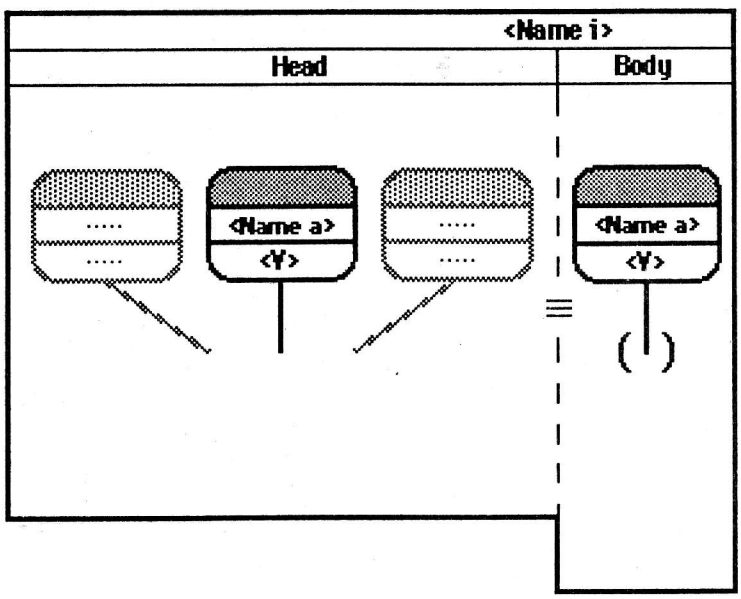


Instruction | Overview | Action

Rule 4: Fetching parameter value from head for body.

Action

Write the output value of the headleaf with the same name into the output stripe of the bodyleaf.



Instruction | Overview | Situation

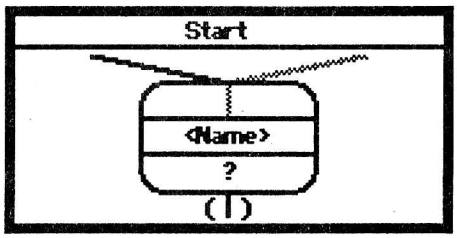
Rule 5: Computing of higher operator node in start tree.

1st Situation

1)The output stripe of a higher operator node in the start tree contains a "?".

2)There is no node with inverted name stripe in the start tree.

3)The input stripe of the higher operator node in the start tree is empty.

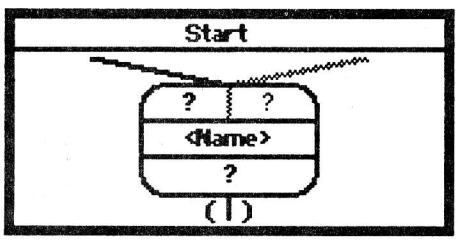


Instruction | Overview | Action

Rule 5: Computing of higher operator node in start tree.

1st Action

Write a "?" into every input field of the higher operator node.



Instruction | Overview | Situation

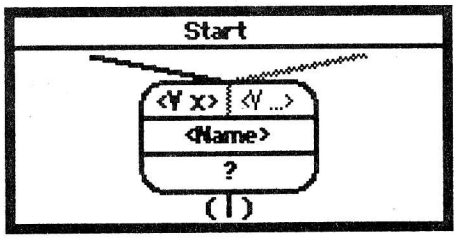
Rule 5: Computing of higher operator node in start tree.

2nd Situation

1)The output stripe of a higher operator node in the start tree contains a "?".

2)There is no node with inverted name stripe in the start tree.

3)The input stripe of the higher operator node in the start tree contains values only.



Instruction | Overview | Action

Rule 5: Computing of higher operator node in start tree.

2nd Action

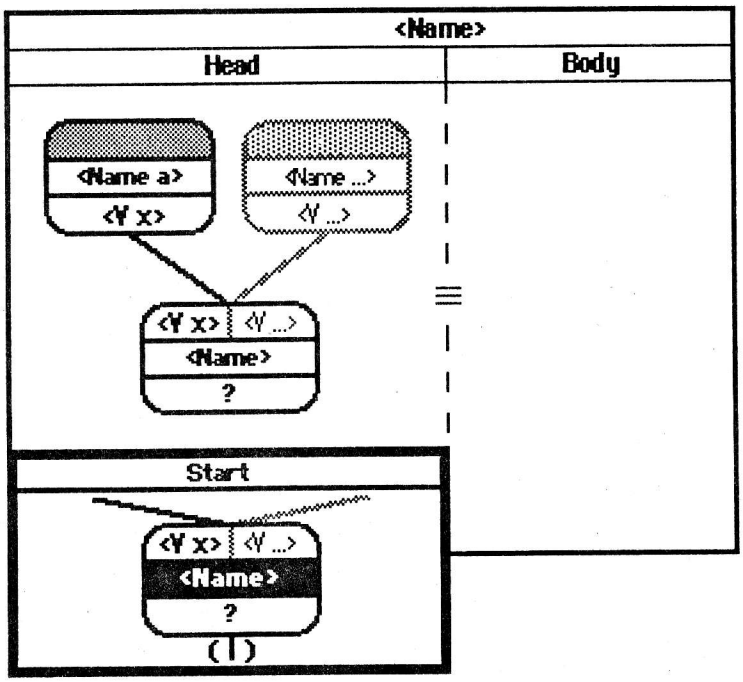
1)Invert the name stripe of the higher operator node.

2)Create a frame at the top with the name of the higher operator node.

3)Write each input value of the higher operator node into the corresponding input field of the head root of the frame.

4)Write each input value of the head root into the output stripe of the connected head leaf.

5)Write a "?" into the output stripe of the head root.



Instruction | Overview | Situation

Rule 5: Computing of higher operator node in start tree.

3rd Situation

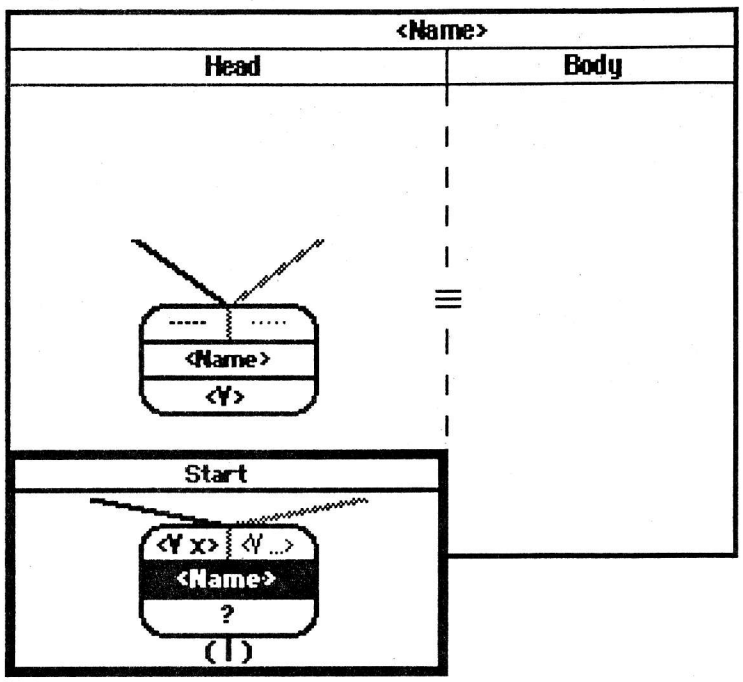
1)The output stripe of a higher operator node in the start tree contains a "?".

2)The name stripe of the higher operator node is inverted.

3)The input stripe of the higher operator node contains values only.

4)There is a frame at the top with the name of the higher operator node.

5)The output stripe of the head root contains a value.



Instruction | Overview | Action

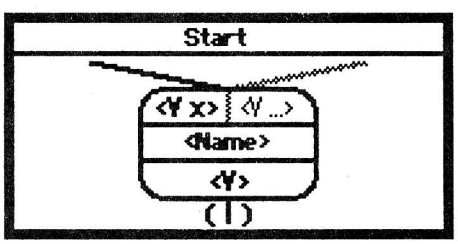
Rule 5: Computing of higher operator node in start tree.

3rd Action

1)Write the output value of the head root of the frame into the output stripe of the higher operator node with the inverted name stripe.

3)Delete the upper frame.

2)Undo the inversion of the name stripe of the higher operator node.



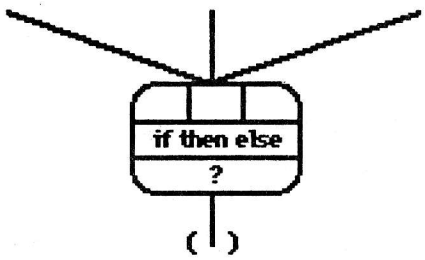
Instruction | Overview | Situation

Rule 6: Computing of IF-THEN-ELSE-node (1st rule).

1st Situation

1)The output stripe of an IF-THEN-ELSE-node contains a "?".

2)The input stripe of the IF-THEN-ELSE-node is empty.

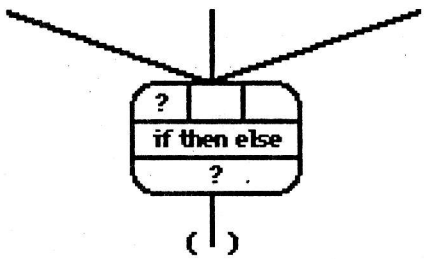


Instruction | Overview | Action

Rule 6: Computing of IF-THEN-ELSE-node (1st rule).

1st Action

Write a "?" into the 1st input field of the IF-THEN-ELSE-node.



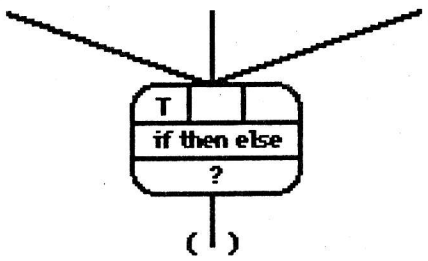
Instruction | Overview | Situation

Rule 6: Computing of IF-THEN-ELSE-node (1st rule).

2nd Situation

1)The output stripe of an IF-THEN-ELSE-node contains a "?".

2)The first input field of the IF-THEN-ELSE-node contains the value "T" (= true).

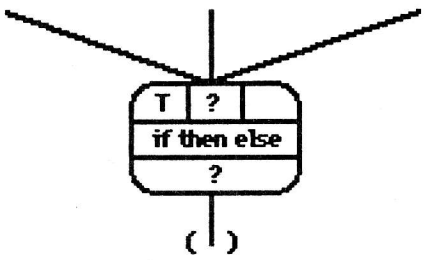


Instruction | Overview | Action

Rule 6: Computing of IF-THEN-ELSE-node (1st rule).

2nd Action

Write a "?" into the 2nd input field of the IF-THEN-ELSE-node.



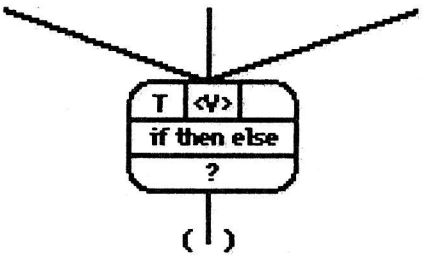
Instruction | Overview | Situation

Rule 6: Computing of IF-THEN-ELSE-node (1st rule).

3rd Situation

1)The output stripe of an IF-THEN-ELSE-node contains a "?".

2)The 2nd input field of IF-THEN-ELSE-node contains a value.

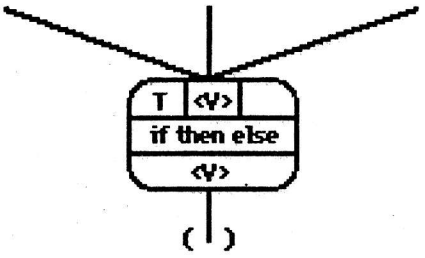


Instruction | Overview | Action

Rule 6: Computing of IF-THEN-ELSE-node (1st rule).

3rd Action

Write the value into the output stripe of the IF-THEN-ELSE-node.



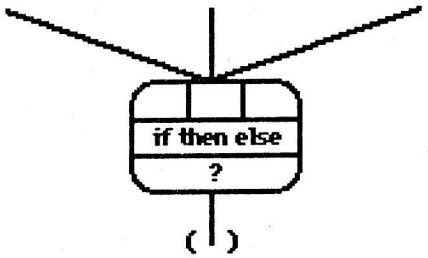
Instruction | Overview | Situation

Rule 7: Computing of IF-THEN-ELSE-node (2nd rule).

1st Situation

1)The output stripe of an IF-THEN-ELSE-node contains a "?".

2)The input stripe of the IF-THEN-ELSE-node is empty.

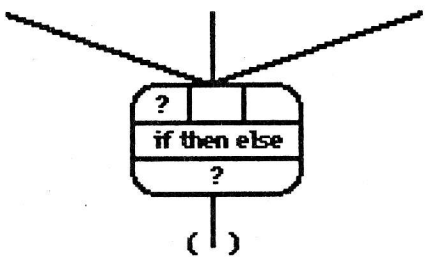


Instruction | Overview | Action

Rule 7: Computing of IF-THEN-ELSE-node (2nd rule).

1st Action

Write a "?" into the 1st input field of the IF-THEN-ELSE-node.



Instruction | Overview | Situation

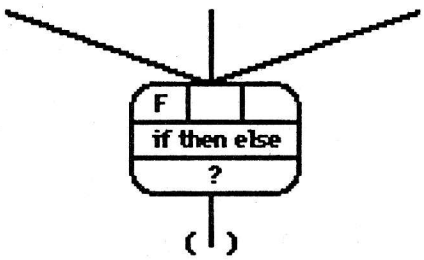
Rule 7: Computing of IF-THEN-ELSE-node (2nd rule).

2nd Situation

1)The output stripe of an IF-THEN-ELSE-node contains a "?".

2)The first input field of the IF-THEN-ELSE-node contains the value "F" (= false)

.

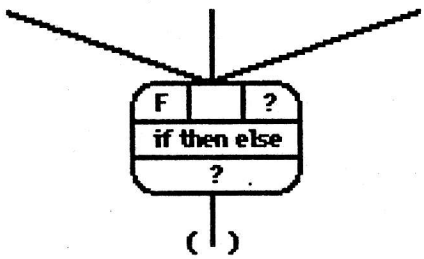


Instruction | Overview | Action

Rule 7: Computing of IF-THEN-ELSE-node (2nd rule).

2nd Action

Write a "?" into the 3rd input field of the IF-THEN-ELSE-node.



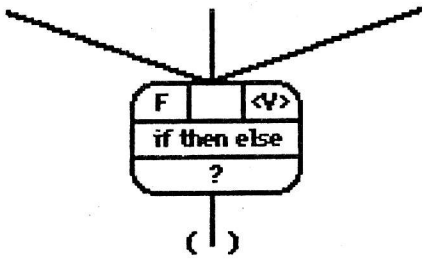
Instruction | Overview | Situation

Rule 7: Computing of IF-THEN-ELSE-node (2nd rule).

3rd Situation

1)The output stripe of an IF-THEN-ELSE-node contains a "?".

2)The 3rd input field of IF-THEN-ELSE-node contains a value.

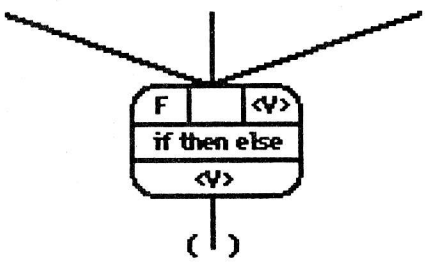


Instruction | Overview | Action

Rule 7: Computing of IF-THEN-ELSE-node (2nd rule).

3rd Action

Write the value into the output stripe of the IF-THEN-ELSE-node.



Instruction | Overview | Situation

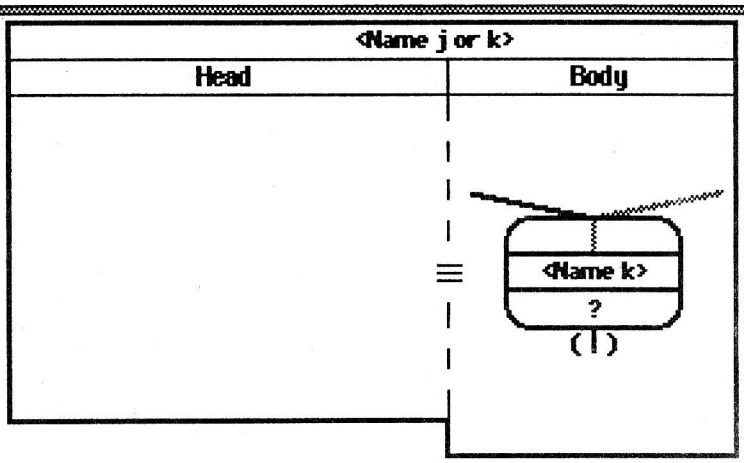
Rule 8: Computing of higher operator node in body tree.

1st Situation

1)The output stripe of a higher operator node of a body tree contains a "?".

2)There is no higher operator node with inverted name stripe in the body tree.

3)The input stripe of the higher operator node is empty.

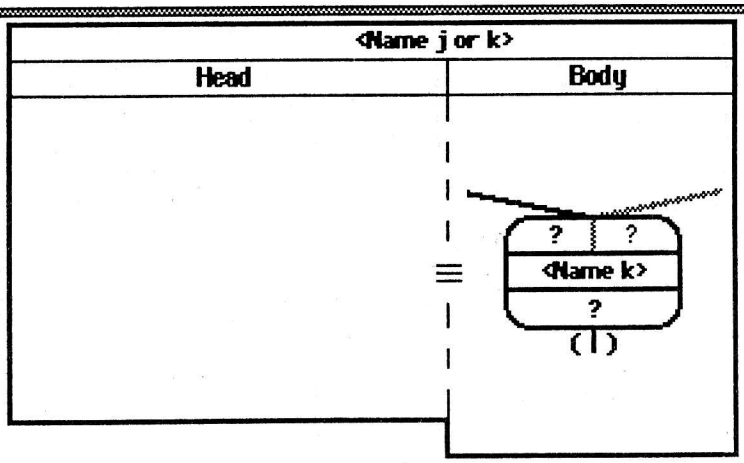


Instruction | Overview | Action

Rule 8: Computing of higher operator node in body tree.

1st Action

Write a "?" into every input field.



Instruction | Overview | Situation

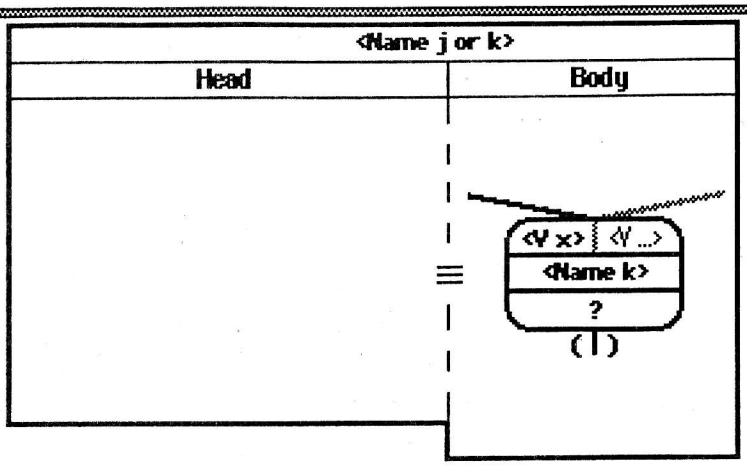
Rule 8: Computing of higher operator node in body tree.

2nd Situation

1)The output stripe of a higher operator node in a body tree contains a "?".

2)There is no higher operator node with inverted name stripe in the body tree.

3)The input stripe of the higher operator node contains values only.



Instruction | Overview | Action

Rule 8: Computing of higher operator node in body tree.

2nd Action

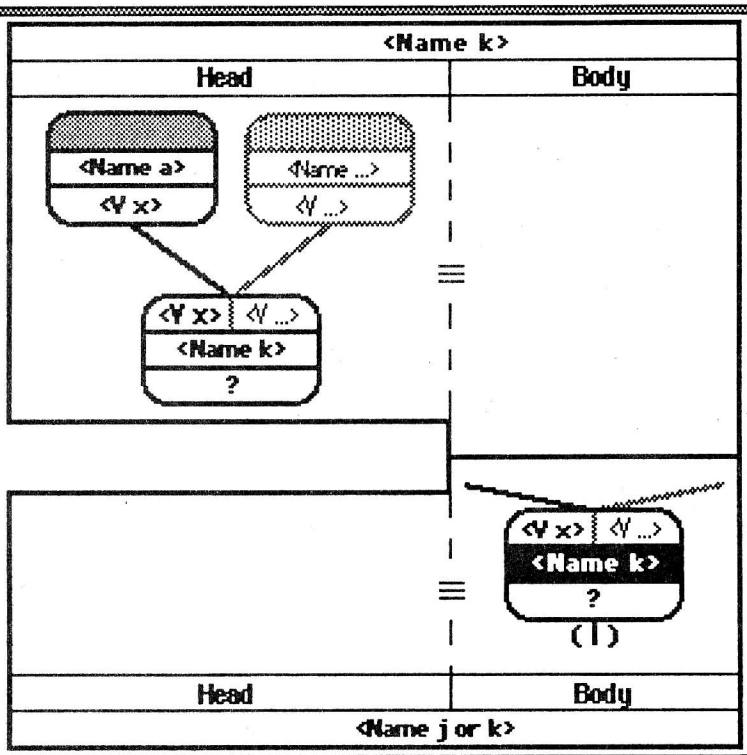
1)Invert the name stripe of the higher operator node.

2)Create a frame at the top with the name of the higher operator node.

3)Write each input value of the higher operator node into the corresponding input field of the head root of the new frame.

4)Write each input value of the head root into the output stripe of the connected head leaf.

5)Write a "?" into the output stripe of the head root of the new frame.



Instruction | Overview | Situation

Rule 8: Computing of higher operator node in body tree.

3rd Situation

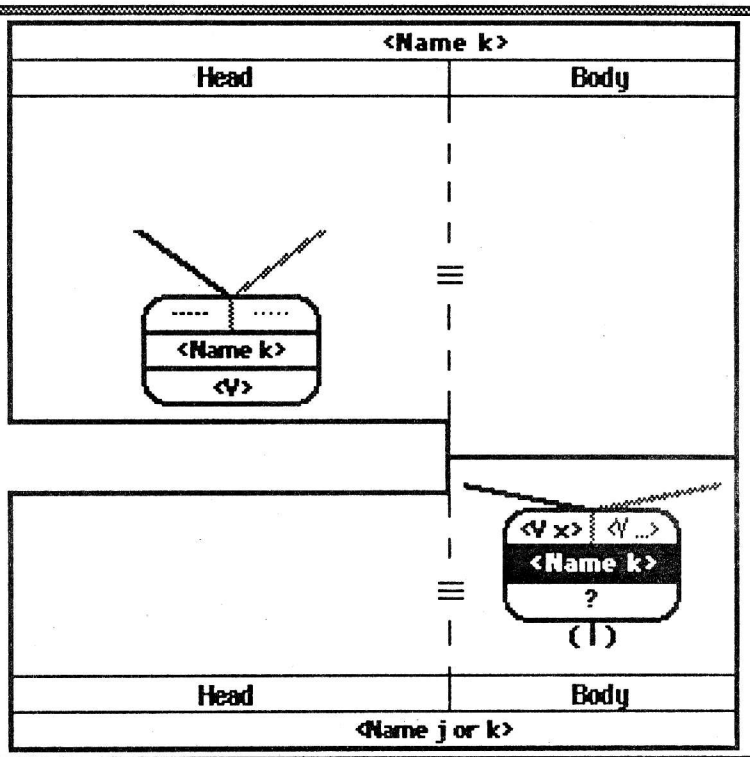
1)The output stripe of a higher operator node in a body tree contains a "?".

2)The name stripe of the higher operator node is inverted.

3)The input stripe of the higher operator node contains values only.

4)There is a frame at the top with the name of the higher operator node.

5)The output stripe of the head root of the frame contains a value.



Instruction | Overview | Action

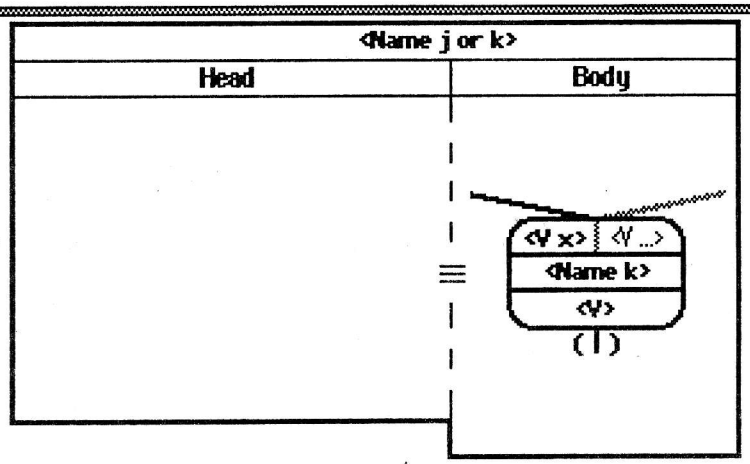
Rule 8: Computing of higher operator node in body tree.

3rd Action

1)Write the output value of the head root of the frame into the output stripe of the higher operator node with the inverted name stripe.

2)Delete the upper frame.

3)Undo the inversion of the name stripe of the higher operator node.



Instruction | Overview | Situation

Appendix B: Representing Computational Knowledge for ABSYNT with State-Specific 2-D-ruleset

Rule 1: To move computation goals in an operator node to the inputstripe of the node (except IF-THEN-ELSE-node !)

Rule 2: To compute the outputvalue of a primitive operator node (except IF-THEN-ELSE-node !)

Rule 3: To move a computation goal to an outputstripe of a connected node

Rule 4: To fetch an outputvalue from a connected node for the corresponding input field

Rule 5: To move a computation goal from the root of the head to the root of the body of a function

Rule 6: To fetch the outputvalue from the root of the body for the root of the head of a function

Rule 7: To fetch the binding of a parameter from the head for a leaf in the body of a function

Rule 8: To compute the outputvalue of a higher operator node in the start tree

Rule 9: To fetch the outputvalue of a higher operator node in the start tree from the root of the head of the called function

Rule 10: To move a computation goal to the predicate field in the IF-THEN-ELSE operator

Rule 11: To move a computation goal to the THEN-inputfield in the case of a true predicate

Rule 12: To fetch the outputvalue in the IF-THEN-ELSE operator in the case of a true predicate

Rule 13: To move a computation goal to the ELSE-inputfield in the case of a false predicate

Rule 14: To fetch the outputvalue in the IF-THEN-ELSE operator in the case of a false predicate

Rule 15: To move a computation goal from the outputstripe of a higher operator node to the outputstripe of the root in the head of the called function

Rule 16: To compute the outputvalue of a higher operator node in the body of a function

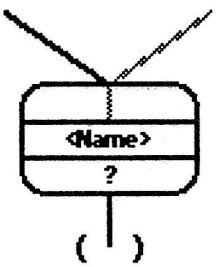
Rule 1: Passing goals to input stripe of operator node (No IF-THEN-ELSE-node!).

Situation

1) The output stripe of an operator node contains a "?".

2) The operator node is not an IF-THEN-ELSE-node.

3) The input stripe of the operator node is empty.

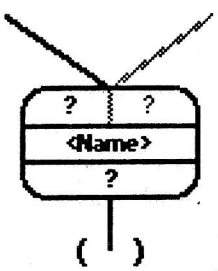


Instruction | Overview | Action

Rule 1: Passing goals to input stripe of operator node (No IF-THEN-ELSE-node!).

Action

Write a "?" in every input field of the operator node.



Instruction | Overview | Situation

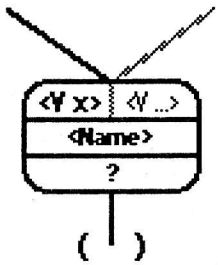
Rule 2: Computing primitive operator node (No IF-THEN-ELSE-node!).

Situation

1) The output stripe of a primitive operator node contains a "?".

2)The primitive operator node is not an IF-THEN-ELSE-node.

3)The input stripe of the primitive operator node contains values only.



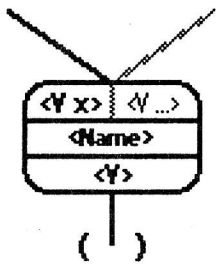
Instruction | Overview | Action

Rule 2: Computing primitive operator node (No IF-THEN-ELSE-node!).

Action

1) Compute the primitive operator node.

2)Write the value into the output stripe of the primitive operator node.



Instruction | Overview | Situation

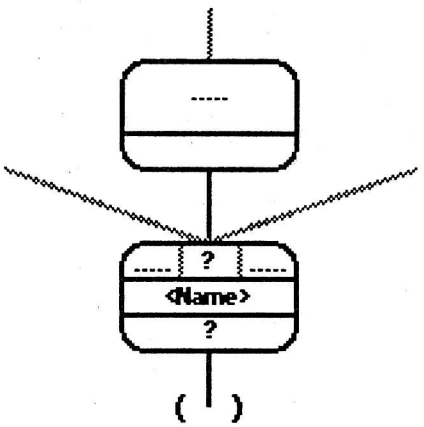
Rule 3: Passing goal to output stripe of connected node.

Situation

1)The output stripe of an operator node contains a "?".

2)Any input field of the operator node contains a "?".

3)The input field of the operator node is connected with another node whose output stripe is empty.

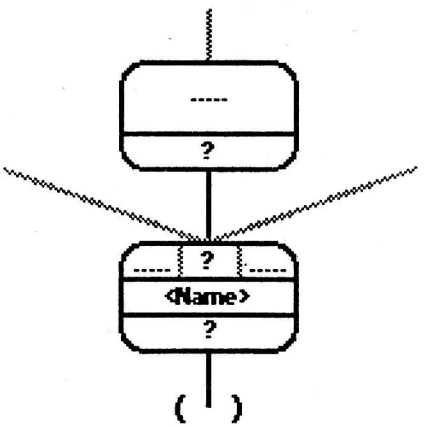


Instruction | Overview | Action

Rule 3: Passing goal to output stripe of connected node.

Action

Write a "?" into the output stripe of the node connected with the input field.



Instruction | Overview | Situation

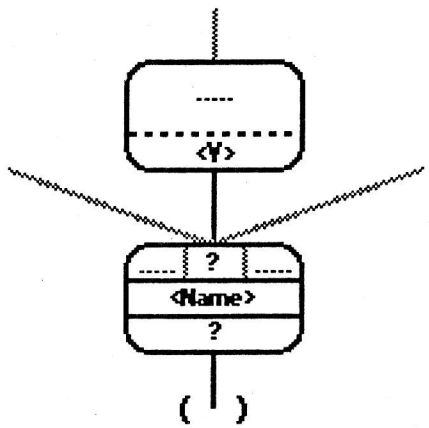
Rule 4: Fetching input value for operator node.

Situation

1)The output stripe of an operator node contains a "?".

2)Any input field of the operator node contains a "?".

3)The input field of the operator node is connected with another node whose output stripe contains a value.

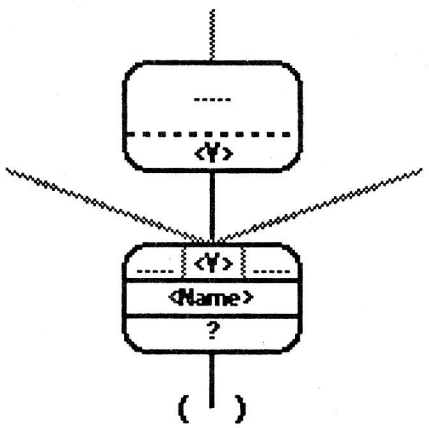


Instruction | Overview | Action

Rule 4: Fetching input value for operator node.

Action

Write the output value of the node connected with the input field into the input field.



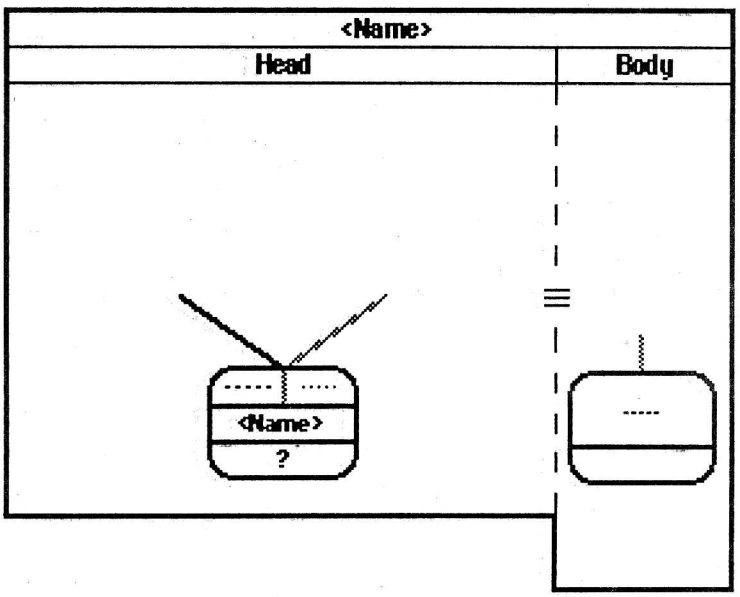
Instruction | Overview | Situation

Rule 5: Passing goal to body root.

Situation

1)The output stripe of the head root of a frame contains a "?".

2)The output stripe of the body root of the frame is empty.

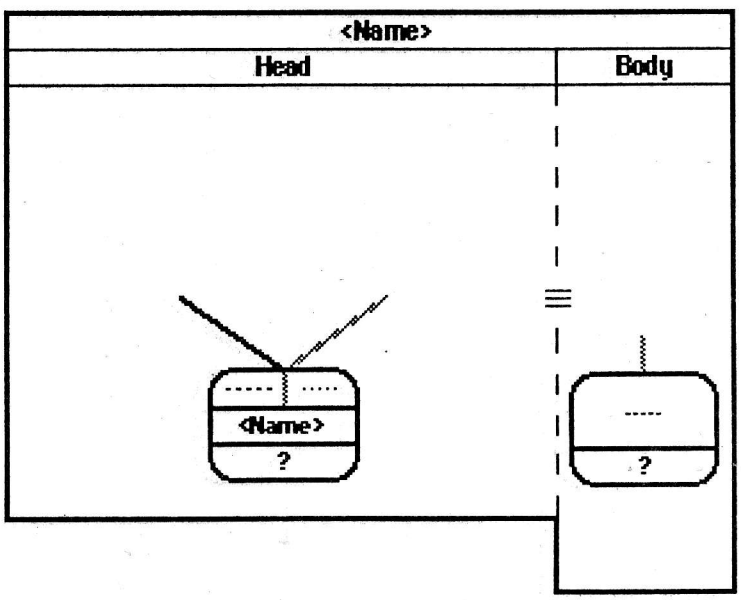


Instruction | Overview | Action

Rule 5: Passing goal to body root.

Action

Write a "?" into the output stripe of the body root of the frame.



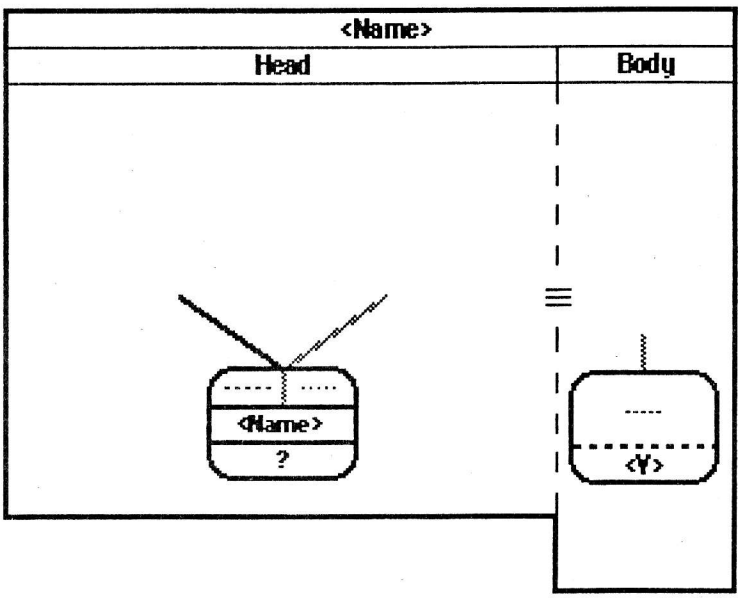
Instruction | Overview | Situation

Rule 6: Fetching output value for head root.

Situation

1)The output stripe of the head root of a frame contains a "?".

2)The output stripe of the body root of the frame contains a value.

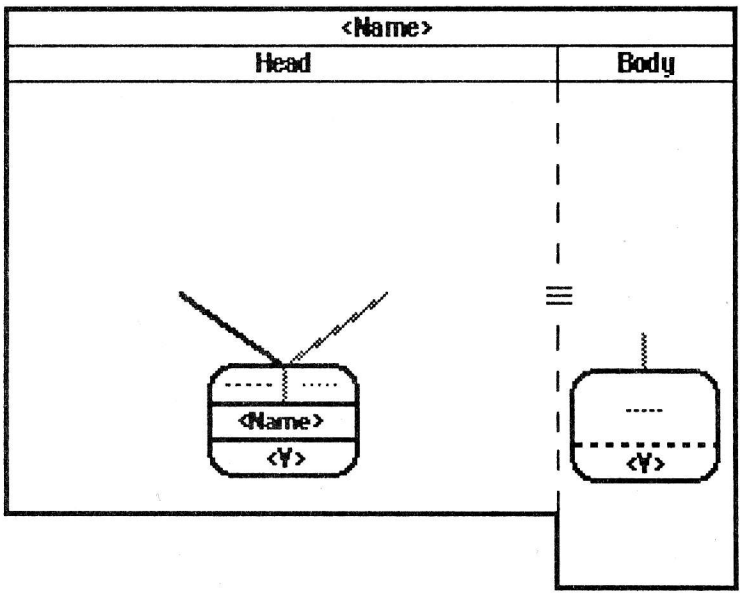


Instruction | Overview | Action

Rule 6: Fetching output value for head root.

Action

Write the output value of the body root of the frame into the output stripe of the head root.

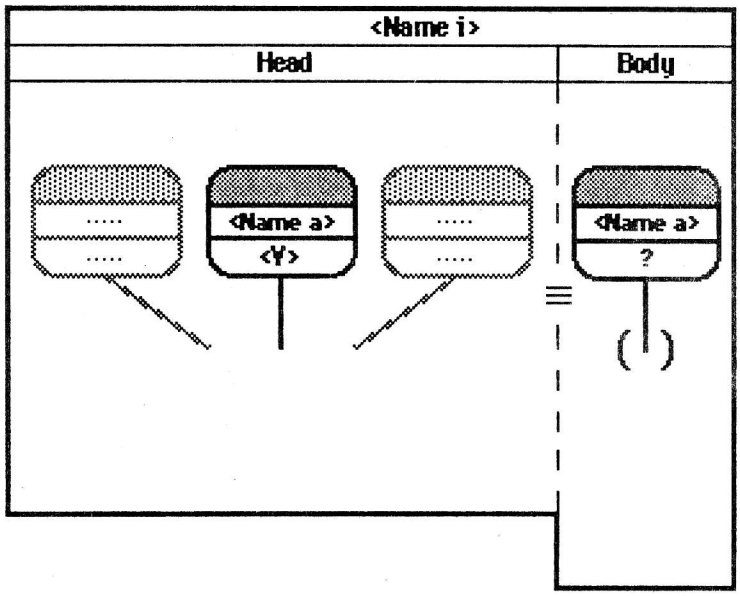


Instruction | Overview | Situation

Rule 7: Fetching parameter value from head for body.

Situation

The output stripe of a bodyleaf of a frame contains a "?".

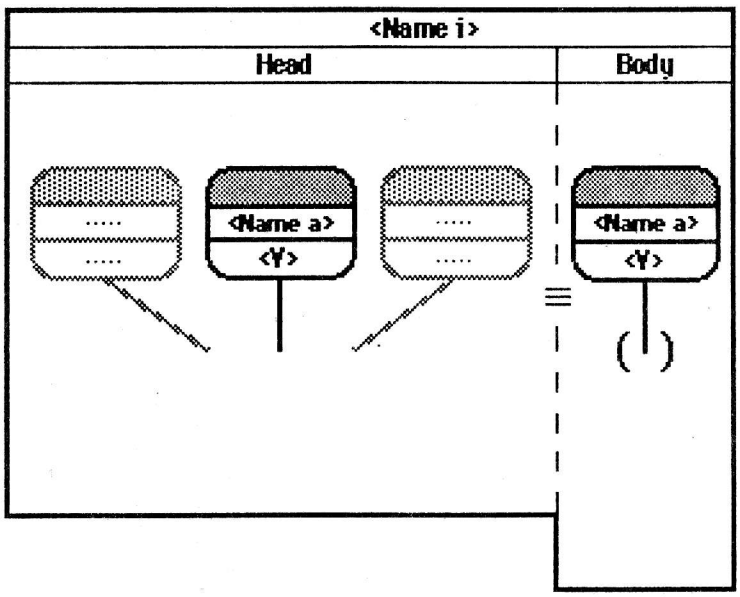


Instruction | Overview | Action

Rule 7: Fetching parameter value from head for body.

Action

Write the output value of the head leaf with the same name into the output stripe of the bodyleaf.



Instruction | Overview | Situation

Rule 8: Computing higher operator node in start tree.

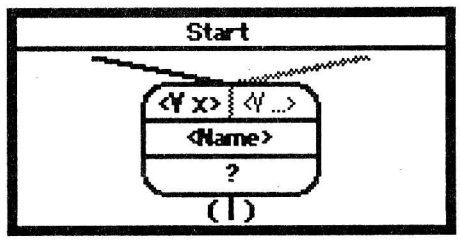
Situation

1)A higher operator node is part of the start tree.

2)There is no node with inverted name stripe in the start tree.

3)The output stripe of the higher operator node contains a "?".

4)The input stripe of the higher operator node contains values only.



Instruction | Overview | Action

Rule 8: Computing higher operator node in start tree.

Action

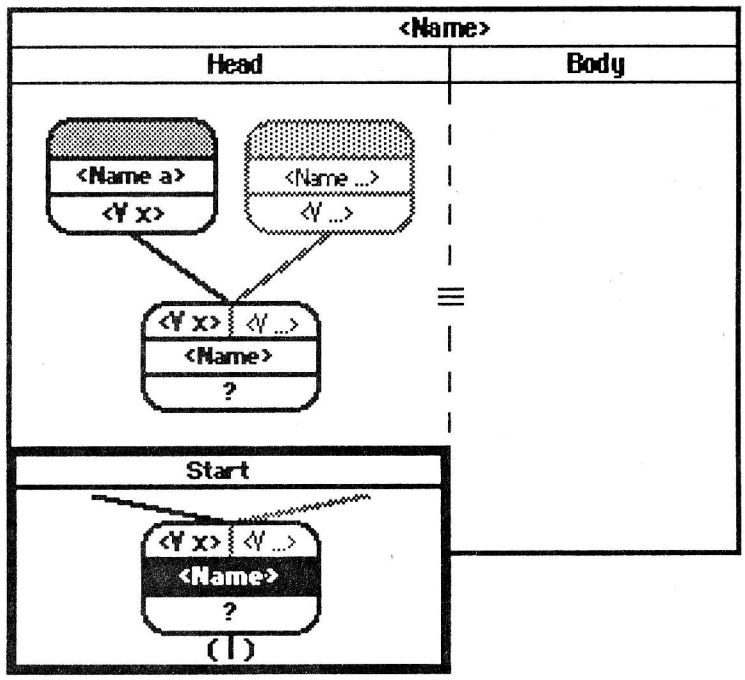
1)Invert the name stripe of the higher operator node.

2)Create a frame at the top with the name of the higher operator node.

3)Write each input value of the higher operator node into the corresponding input field of the head root of the frame.

4)Write each input value of the head root into the output stripe of the connected head leaf.

5)Write a "?" into the output stripe of the head root of the frame.



Instruction | Overview | Situation

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

Situation

1)A higher operator node is part of the start tree.

2)The name stripe of the higher operator node is inverted.

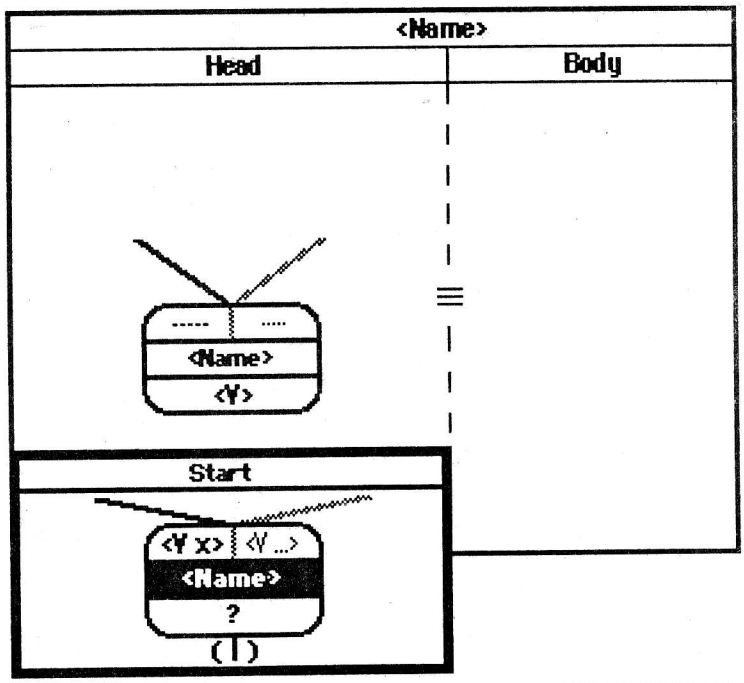
3)The output stripe of the higher operator node contains a "?".

4)The input stripe of the higher operator node contains values only.

5)There is a frame at the top with the name of the higher operator node.

6)The output stripe of the head root contains a value.

7)There is no frame at the bottom.



Instruction | Overview | Action

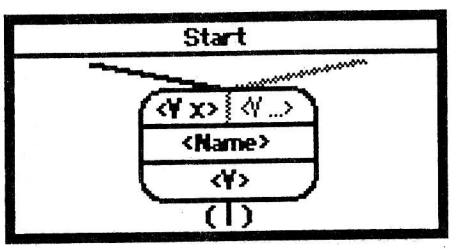
Rule 9: Fetching output value for higher operator node in start tree.

Action

1)Write the output value of the head root into the output stripe of the higher operator node with the inverted name stripe in the start tree.

2)Delete the upper frame.

3)Undo the inversion of the name stripe of the higher operator node.



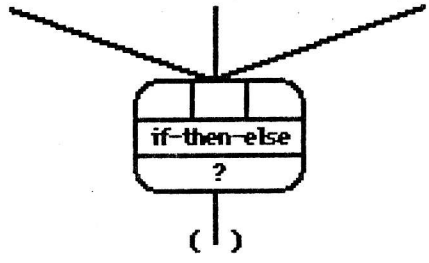
Instruction | Overview | Situation

Rule 10: Passing goal to 1st input field of IF-THEN-ELSE-node.

Situation

1)The output stripe of an IF-THEN-ELSE-node contains a "?".

2)The input stripe of the IF-THEN-ELSE-node is empty.

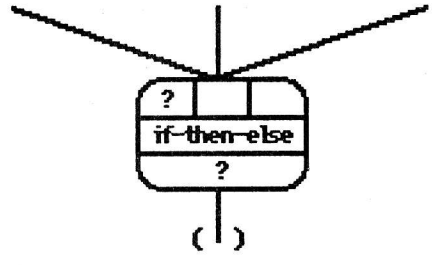


Instruction | Overview | Action

Rule 10: Passing goal to 1st input field of IF-THEN-ELSE-node.

Action

Write a "?" into the 1st input field of the IF-THEN-ELSE-node.



Instruction | Overview | Situation

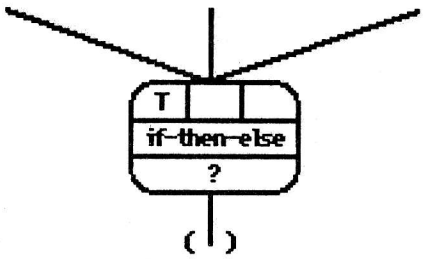
Rule 11: Passing goal to 2nd input field of IF-THEN-ELSE-node.

Situation

1)The output stripe of an IF-THEN-ELSE-node contains a "?".

2)The 1st input field of the IF-THEN-ELSE-node contains the value "T" (= true).

3)The 2nd input field of the IF-THEN-ELSE-node is empty.

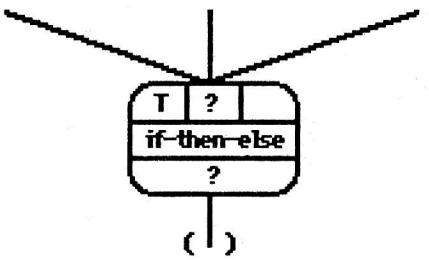


Instruction | Overview | Action

Rule 11: Passing goal to 2nd input field of IF-THEN-ELSE-node.

Action

Write a "?" into the 2nd input field of the IF-THEN-ELSE-node.



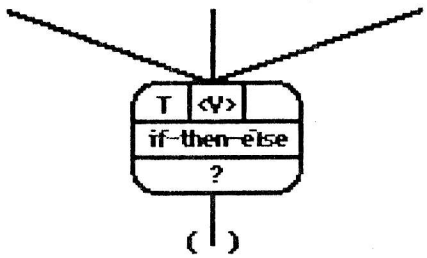
Instruction | Overview | Situation

Rule 12: Fetching output value for IF-THEN-ELSE-node from 2nd input field.

Situation

1)The output stripe of an IF-THEN-ELSE-node contains a "?".

2)The 2nd input field of the IF-THEN-ELSE-node contains a value.

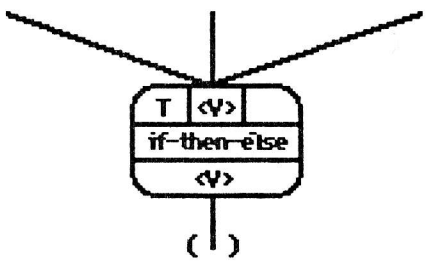


Instruction | Overview | Action

Rule 12: Fetching output value for IF-THEN-ELSE-node from 2nd input field.

Action

Write the value into the output stripe of the IF-THEN-ELSE-node.



Instruction | Overview | Situation

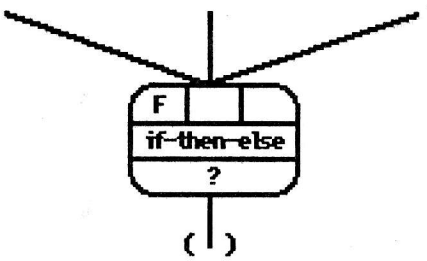
Rule 13: Passing goal to 3rd input field of IF-THEN-ELSE-node.

Situation

1)The output stripe of an IF-THEN-ELSE-node contains a "?".

2)The 1st input field of the IF-THEN-ELSE-node contains the value "F" (=false).

3)The 3rd input field is empty.

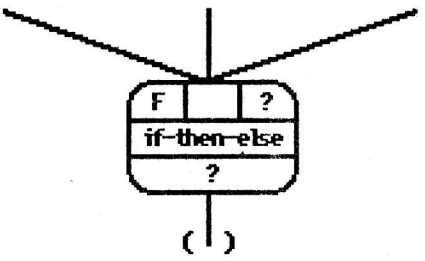


Instruction | Overview | Action

Rule 13: Passing goal to 3rd input field of IF-THEN-ELSE-node.

Action

Write a "?" into the 3rd input field of the IF-THEN-ELSE-node.



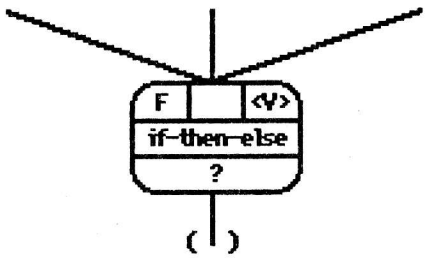
Instruction | Overview | Situation

Rule 14: Fetching output value for IF-THEN-ELSE-node from 3rd input field.

Situation

1)The output stripe of an IF-THEN-ELSE-node contains a "?".

2)The 3rd input field of IF-THEN-ELSE-node contains a value.

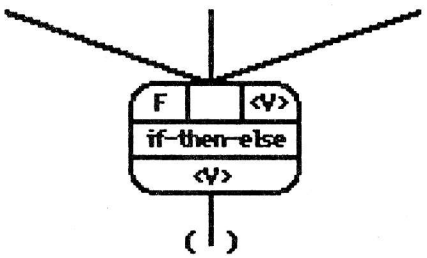


Instruction | Overview | Action

Rule 14: Fetching output value for IF-THEN-ELSE-node from 3rd input field.

Action

Write the value into the output stripe of the IF-THEN-ELSE-node.



Instruction | Overview | Situation

Rule 15: Computing higher operator node in body tree.

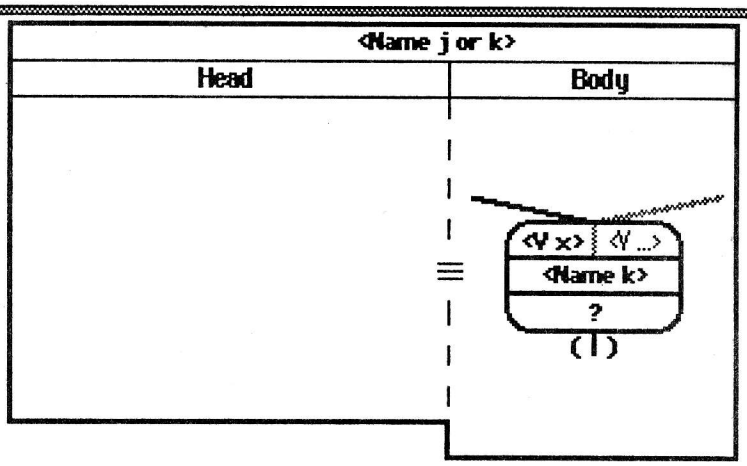
Situation

1)A higher operator node is part of the body tree of the frame at the top.

2)There is no inverted name stripe in the body tree.

3)The output stripe of the higher operator node contains a "?".

4)The input stripe of the higher operator node contains values only.



Instruction | Overview | Action

Rule 15: Computing higher operator node in body tree.

Action

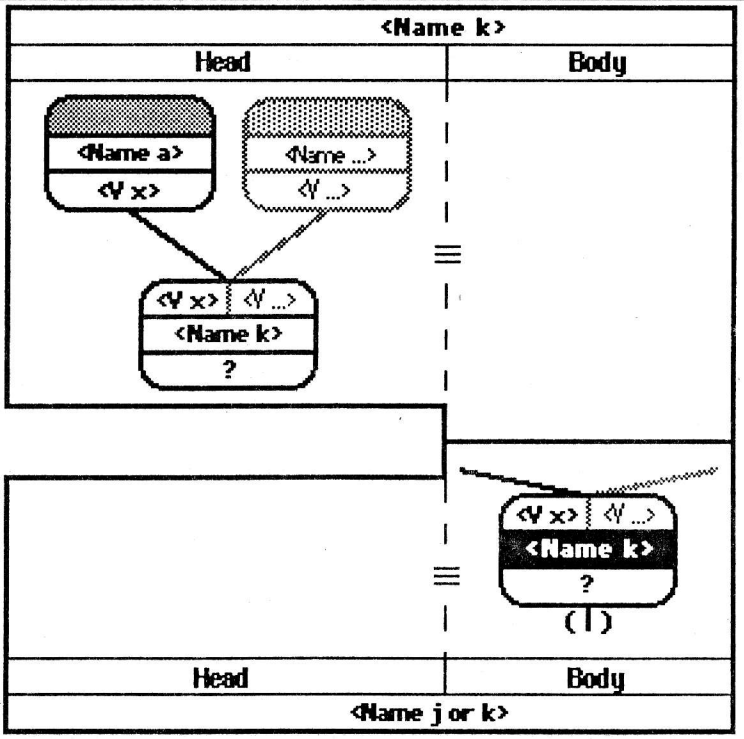
1)Invert the name stripe of the higher operator node.

2)Create a frame at the top with the name of the higher operator node.

3)Write each input value of the higher operator node into the corresponding input field of the head root of the new frame.

4)Write each input value of the head root into the output stripe of the connected head leaf.

5)Write a "?" into the output stripe of the head root of the new frame.



Instruction | Overview | Situation

Rule 16: Fetching output value for higher operator node in body tree.

Situation

1)A higher operator node is part of the body tree in the frame at the bottom.

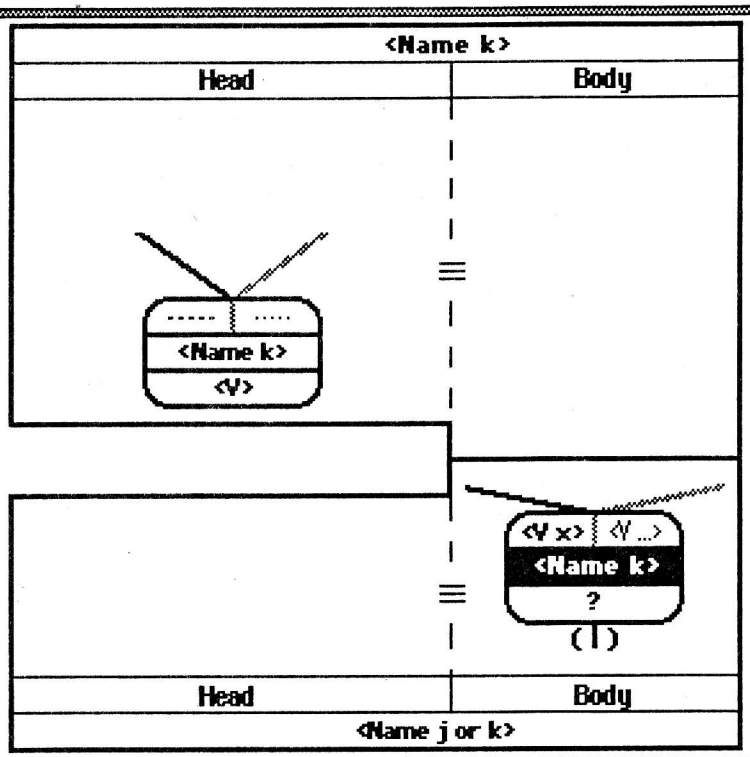
2)The name stripe of the higher operator node is inverted.

3)The output stripe of the higher operator node contains a "?"

4)The input stripe of the higher operator node contains values only.

5)There is a frame at the top with the name of the higher operator node.

6)The output stripe of the head root of the frame contains a value.



Instruction | Overview | Action

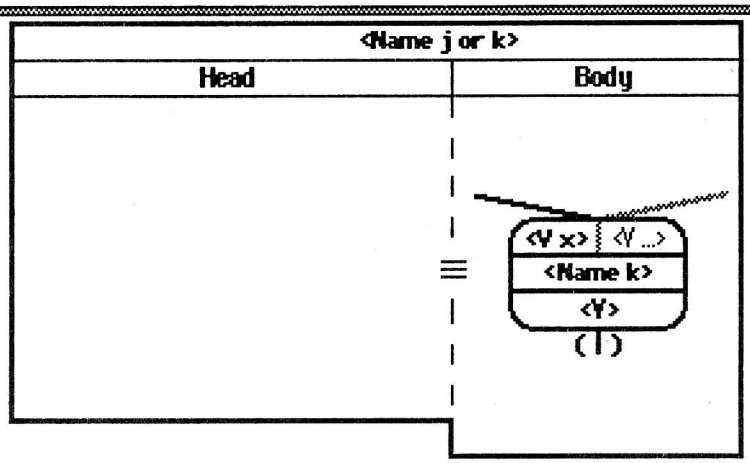
Rule 16: Fetching output value for higher operator node in body tree.

Action

1)Write the output value of the head root in the upper frame into the output stripe of the higher operator node with the inverted name stripe in the frame at the bottom.

2)Delete the frame at the top.

3)Undo the inversion of the name stripe of the higher operator node.



Instruction | Overview | Situation

ABSYNT-Reports:

1/87 SCHRÖDER, O., FRANK, K.D. & COLONIUS,H., **Gedächtnisrepräsentation funktionaler graphischer Programme**, Oktober 1987

2/87 COLONIUS, H., FRANK, K.D., JANKE, G., KOHNERT, K., MÖBUS, C., SCHRÖDER, O. & THOLE, H.J., **Syntaktische und semantische Fehler in funktionalen, graphischen Programmen**, Oktober 1987

3/87 MÖBUS, C. & THOLE, H.J., **Tutors, Instructions and Helps**, February 1988; to appear in: CHRISTALLER, Th. (ed), Künstliche Intelligenz: KIFS 87, Computer Science Lecture Series, Heidelberg: Springer (in press)

4/88 in preparation

5/88 JANKE, G. & KOHNERT, K., **Interface Design of a Visual Programming Language: Evaluating Runnable Specifications**, to appear in: F.KLIX, H.WANDKE, N.A.STREITZ & Y.WAERN (eds.), Man-Computer Interaction Research, MACINTER II, Amsterdam: North-Holland (in press)