AD-A096 559 MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTE--ETC F/G 9/2
COHERENT BEHAVIOR FROM INCOHERENT KNOWLEDGE SOURCES IN THE AUTO--ETC(U)
JAN 81 R BROWN N00014-80-C-0505
UNCLASSIFIED AI-TR-610 NL

1 OF 3
AD A
096559

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 14 AI-TR-610	2. GOVT ACCESSION NO. AD-A096559	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 6 Coherent Behavior From Incoherent Knowledge Sources In The Automatic Synthesis of Numerical Computer Programs.		5. TYPE OF REPORT & PERIOD COVERED 9 Technical Report,
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) 10 Richard Brown	15	8. CONTRACT OR GRANT NUMBER(s) N00014-80-C-0505
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, Massachusetts 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 12 211
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd Arlington, Virginia 22209	11	12. REPORT DATE Jan[redacted]81
		13. NUMBER OF PAGES 211
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Office of Naval Research Information Systems Arlington, Virginia 22217		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Automatic Programming
Problem Solving
Numerical Programming

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A fundamental problem in artificial intelligence is obtaining coherent behavior in rule-based problem solving systems. A good quantitative measure of coherence is time behavior; a system that never, in retrospect, applied a rule needlessly is certainly coherent; a system suffering from combinatorial blowup is certainly behaving incoherently. This report describes a rule-based problem solving system for automatically writing and improving numerical computer programs from specifications.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED 407483

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0505.

ABSTRACT

COHERENT BEHAVIOR FROM INCOHERENT KNOWLEDGE SOURCES IN THE AUTOMATIC SYNTHESIS OF NUMERICAL COMPUTER PROGRAMS

by Richard Brown

A fundamental problem in artificial intelligence is obtaining coherent behavior in rule-based problem solving systems. A good quantitative measure of coherence is time behavior; a system that never, in retrospect, applied a rule needlessly is certainly coherent; a system suffering from combinatorial blowup is certainly behaving incoherently.

This report describes a rule-based problem solving system for automatically writing and improving numerical computer programs from specifications. The specifications are in terms of "constraints" among inputs and outputs. The system has solved program synthesis problems involving systems of equations, determining that methods of successive approximation converge, transforming recursion to iteration, and manipulating power series (using differing organizations, control structures, and argument-passing techniques).

The theory of coherent problem solving used by this system is based on syntactically restricting the rule language so that the effect of using a rule can be accurately predicted. Each rule is independently pre-analyzed and an abstract description of its possible effects is produced. These descriptions are combined with a local analysis of the current status of the deduction process to decide whether a rule should be applied. This report explains how to perform this analysis so that all known forms of combinatorial blowup are eliminated from the deduction process. Because this process depends only weakly on the nature of the task (program synthesis), there is hope that this theory of control can be adapted to other problem domains.

In addition to this theory of coherent rule-based deduction, results concerning the use of EL-like constraint networks as a programming language have been obtained (EL was a system written by Stallman, Sussman and others [S79] for electronic circuit analysis). In particular a new and powerful way to describe and manipulate looping control structures has been developed.

Thesis supervisor: Gerald J. Sussman.

Title: Associate Professor, Department of Electrical Engineering and Computer Science.

Contents

The tables are all collected at the end of each chapter.

TABLE OF DIAGRAMS

CHAPTER 1

INTRODUCTION and OVERVIEW

This report is about rule-based problem solving. It has a motto:

The less a rule can do, the better the effects of using it can be predicted.

This motto suggests the key to obtaining coherant behavior by avoiding exponential behavior (combinatorial explosion) in problem solving. The remainder of this report shows how to do this, at least in the domain of synthesizing numerical computer programs from their non-procedural specifications.

Nature and direction of the research

One of the central goals of research in Artificial Intelligence is to learn how to create automatic "expert problem solvers." Perhaps the fact that the author of an expert problem solver must have some expertise in the solver's domain explains the popularity (at least at MIT) of electrical circuits and computer programming domains. This research concerns the latter.

To construct an automatic programming system, several separate but related sub-domains and their corresponding experts are required. These include *program understanding*, *specification acquisition*, *program verification*, and *program synthesis*. The last is the domain of expertise of the system described by this report.

A program synthesis system takes a specification (for example, a predicate calculus description of the relationships between the inputs and the outputs) of a program and produces a program (in, for example, the programmming language LISP) that meets those specifications. Two quick examples of the kind of problem a program synthesis system should be able to solve (given a suitable rule library) are:

> 1. If told that the output times the output is the input, it should write a square root program. The system should be able to adopt Newton's method for finding zeros of functions to this problem.
> 2. If told that $F(0) = 0$ and that $F(n+1) = 2F(n)+1$, it should discover that $F(n) = 2^n - 1$ and write a program that computes this quantity.

This report describes a numerical program synthesis system. The kinds of problems it can currently solve (these include the two examples above) are described in the next section. Diagram "Bernoulli Example" shows an input specification (in mathematical notation) and the resulting code -- chapter 4 has the details.

The fundamental difficulty in constructing problem-solving systems is the so-called "combinatorial explosion" problem. That is to say, given a reasonable measure of how hard the problem at hand *should be*, a "combinatorially explosive" problem solver takes an amount of time proportional to an exponential (or worse!) function of that measure (chapter 3 is more explicit). The measure of "how hard the problem is" need not be the usual complexity-theoretic "size of input" measure! In general, the size of program (and therefore certainly the time needed to write it) is an unbounded function in the size of its specification. For purposes of measuring the coherence of a problem solver, a measure of "how hard a problem is" could be the number

DIAGRAM "BERNOULLI EXAMPLE"

$$\sum_{N=0}^{\infty} \frac{B(N)}{N!} t^N = \frac{t}{(e^t - 1)}$$

B(N) is the Nth Bernoulli number. The problem is:
Given N, find B(N).

The system wrote the following function:

```
(DEFUN BERNOULLI (N)
  (DO ((COUNT 0.0) (P (CREATE-VECTOR 0.0 N 1.0)))
      ((= COUNT N) (ACCESS-VECTOR P N))
   (SETQ COUNT (PLUS 1.0 COUNT) P
    (STORE-VECTOR
     P
     COUNT
     ((LAMBDA (G0065)
        (TIMES (COND ((= ((LAMBDA (G0041)
                            (COND ((= (RFACTORIAL G0041) 0.0) (ERROR))
                                  (T (QUOTIENT 1.0 (RFACTORIAL G0041)))))
                          (DIFFERENCE (PLUS COUNT 1.0) COUNT))
                         0.0)
                      (COND ((= G0065 0.0) (ERROR)) (T (ERROR))))
                     (T (QUOTIENT G0065
                                  ((LAMBDA (G0041)
                                     (COND ((= (RFACTORIAL G0041) 0.0) (ERROR))
                                           (T (QUOTIENT 1.0 (RFACTORIAL G0041)))))
                                   (DIFFERENCE (PLUS COUNT 1.0) COUNT)))))
               (RFACTORIAL COUNT)))
      (DIFFERENCE
       (COND ((= (PLUS COUNT 1.0) 1.0) 1.0) (T 0.0))
       (DO ((SUM 0.0) (M 0.0))
           ((= M COUNT) SUM)
        (SETQ SUM
         (PLUS SUM
               ((LAMBDA (G0068)
                  (TIMES (COND ((= (RFACTORIAL M) 0.0)
                                (COND ((= G0068 0.0) (ERROR)) (T (ERROR))))
                               (T (QUOTIENT G0068 (RFACTORIAL M))))
                         ((LAMBDA (G0041)
                            (COND ((= (RFACTORIAL G0041) 0.0) (ERROR))
                                  (T (QUOTIENT 1.0 (RFACTORIAL G0041)))))
                          (DIFFERENCE (PLUS COUNT 1.0) M))))
                (ACCESS-VECTOR P M)))
         M (PLUS 1.0 M)))))))))
```

of rule applications that were *needed* (as opposed to the larger number actually performed) in the *solution found*. *This is the measure* that will always be used in this report.

In order to be practical, a program synthesis system must take less than an exponential amount of time in the number of rule applications required, and in order to be usable it must not take more than a low-order polynomial amount of time in the number of rule applications required. Complexity theory tells us that in general there exist general recursive functions for which there is no "optimal" implementation. Even if there is for the particular function being synthesized, a practical implementation would generate one implementation fast, and then as a back-ground task be able to improve this function.

The *result of this research* is a collection of techniques (embedded in a synthesis system) that synthesize numerical computer programs. These techniques do *not* depend strongly on the fact that the task is synthesizing programs, so there is hope that they could be adapted to problem solving in other domains. I conjecture that these techniques in fact run in time proportional to a polynomial function of the number of rules required to solve a synthesis problem.

Key Results

What constitutes a "result" in the field of artificial intelligence? The goals of AI are to make machines smart and to understand human intelligence using the computational metaphore. A result in AI is therefore something that pulls that goal closer. Specifically, it is an insight into the nature of a task (possibly reformulating the description of the task) that makes the computational aspects of performing the task more tractable: a faster procedure or one requiring less memory, a method for performing the task on a larger "example space", or even a methodology making the programming of the task radically simpler.

Success in obtaining a result in AI comprises:

1. A task description, perhaps informal, with a sufficient number of dissimilar target examples to show the plausibility of covering the "example space."
2. A working program at least capable of solving the target examples. The program should not solve the examples by luck or accident.
3. A set of identifiable principles explaining why the working program functions. These identifiable principles constitute the result(s).

This research has been successful. The target examples (to be discussed in detail shortly) were selected to span the difficult subtasks involved in synthesizing numerical computer programs that do not use data-structures. A working program was obtained that (on a relatively unloaded PDP-10) can solve several examples faster than an expert human programmer.

There are three basic principles on which this research is based:

1. *Synthesis by analysis* of constraints. Since the first use of "networks of constraints" in Stallman and Sussman's EL system [S79], this formalism has become increasingly popular. This research uses the idea of propagating code fragments (in addition to the usual arithmetic and algebraic expressions) through a network of constraints in order to write code. Stated another way, the "data base" of constraints connected together to form a network is interpreted many different ways by the synthesis system.
2. *Diagrams provide insight*. In terms of the actual implementation, synthesis problems are of course presented as text strings. But what might be termed "topological configurations" seem to be the real key in deciding what deductions to perform next. By extensively using the diagrams of networks of constraints, much insight into the recognition and use of these "topological configurations" was obtained.
3. *Combinatorial Blow-up* should be the central concern. People can perform the task

> of writing code given specifications. What ever this task actually involves, people do *not* suffer from combinatorial blow-up This observation should be the starting point for an attempt to get computers to perform program synthesis: if a combinatorial amount of work is required to perform the task, then the task must be reformulated

I believe the three principles above can and should be applied to other research in artificial intelligence.

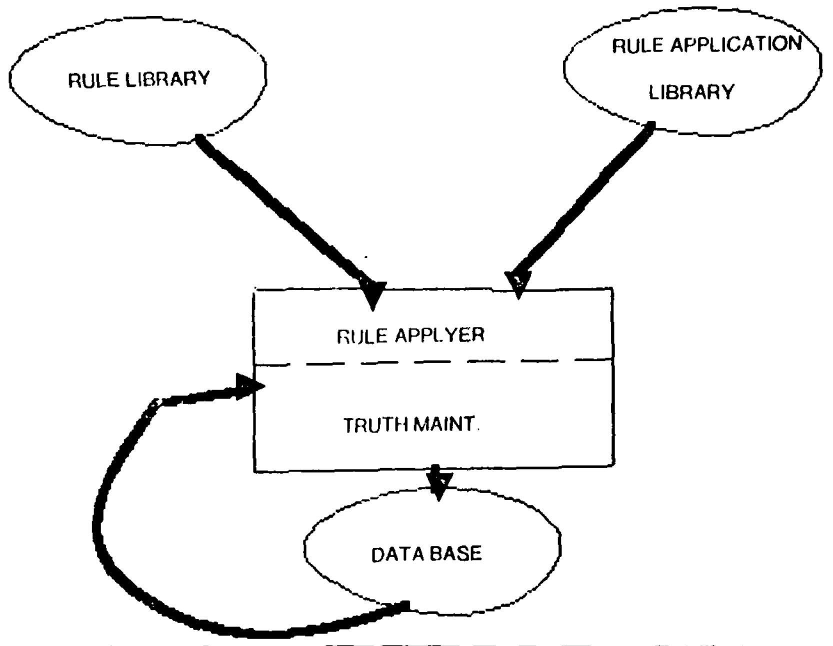
The nature of expert problem solving systems

At a very abstract level, a problem solver has four major components, as illustrated in diagram "problem solver". Rules may be assumed to be of the *situation/action* type, where the action is to (in general) produce a new data base by modifying the old one. Statements in the rule-application library tell when rules should (and more particularly should *not*) be applied Since the topic of statements in the rule-application library is different from the topic of the rules, there is reason to believe that these two libraries are written in different languages. (But see de Kleer *et al* [dK79]).

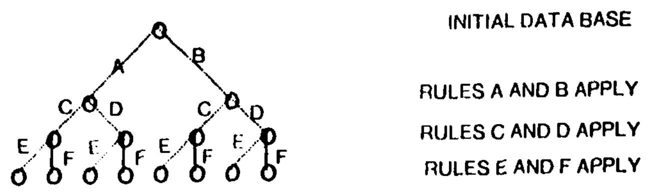
The deductive mechanism has two sub-components: a *matcher* or rule-applier, and a *truth-maintainer* that makes sure the updated data base is consistent.

The *data base* may be thought of as the current state of mind. In the absence of strong evidence to the contrary, it should be assumed that for any data base, several rules can be applied. Therefore, one must assume that the deduction mechanism gives rise to a tree of data bases as shown in diagram "tree of data bases". From this diagram it is evident that one must assume the number of data bases explored is exponential in the length of the solution derivation.

DIAGRAM "PROBLEM SOLVER"



"TREE OF DATA BASES"



The British Museum algorithm for program synthesis would generate (in order of increasing length) all possible programs and all possible program correctness proofs A proof checker would then examine each proof as it is generated. The first program whose proof is correct and shows that the specification has been satisfied would be returned. Although absurdly impractical, the British Museum algorithm takes an amount of time that is *only exponential* in the length of the program plus correctness proof.

The discussion above leads to the conclusion that in order to be better than the British museum algorithm, a rule-based expert problem solver must have some combination of the following properties:

1. For any realistic data base, only one rule is applicable (a special case of 4 below).
2. The data base tree is always shallow.
3. The rule-application library is good enough to guarantee that the *average* number of data bases explored is polynomial (in the depth of the tree).
4. The data-base "tree" converges so that the breadth at each level is bounded by a polynomial of the depth.
5. The form of rules allowed, and the deductive mechanism applying them, have been restricted sufficiently to guarantee that rules do not combine combinatorially.

Although a rigorous proof has not been found, the problem solver described in this report appears to have the fifth property. This means that as long as the rule library is consistent (not an unreasonable expectation), the problem solver is conjectured to expend only polynomial effort (in the number of rule applications actually required) in writing a program meeting the specifications (provided such a program exists and can be found using the rules in the library) regardless of how many rules are present and regardless of how much apparent potential there is for combinatorial explosion.

A problem-solving methodology

This report presents a problem-solving methodology (in the guise of a system that uses it) that is *not* based on the traditional goal/subgoal paradigm. It is a matter of speculation as to whether human problem-solving *is* goal-directed or only *appears* to be goal-directed. But as far as this program synthesis system is concerned, it only *appears* to be goal-directed, as will be evident when the theory of its innermost operation is revealed. That is to say, this system demonstrates (for at least the task of program synthesis) a workable substitute for an explicit goal/subgoal organization.

I believe a truely intelligent problem solving system will have both a *strategic* (goal/subgoal paradigm) and a *tactical* (this system's paradigm) component. The strategic component might have rules concerning topics like

* How to decompose a problem into (relatively) independent subproblems [Sa75a,b]
* How to form and debug an appropriate analogy [Br77], [U77], [W79].
* A sequence of steps to try performing to solve a class of problems.
* How to form and debug approximate solutions to the problem [Su75].

If the strategic component is (as I believe) inherently exponential, then combinatorial explosion can be prevented by limiting the depth of the strategy goal tree. Therefore one only wants the strategic component to operate at very high levels of abstraction.

The tactical component comes into play when the level of detail becomes too fine for the strategic component to deal with effectively (or when no strategic rules appear applicable). Given a relatively simple problem, the tactical component solves that problem without generating any new goals or subgoals.

The system described here is a prototype of a tactical component. It is surprisingly powerful in its domain considering its lack of global perspective. If the tactical component can take care of as wide a range of problems as this report suggests, then perhaps the strategic component can ignore details below a higher level of abstraction than previously believed. The tactical component could be thought of as the applier of "brute force." But the brute is a noble savage: street wise with animal cunning.

Both the tactical and strategic components might make use of specialists to do things like (in the domain of numerical programs) solve integrals. These specialists could ...

But enough speculation!

Examples

Naturally one must be concerned that a program synthesis system with a restrictive rule language (and specification language) nonetheless still be able to solve "interesting" problems and write "interesting" code. The following four problems are the central examples used in this report to illustrate the theory of operation of the program synthesis system. These examples concern the following general topics:

1. Systems of equations
2. Inverses and zeros of functions
3. Changing recursive control flow to iterative, and *vice versa*
4. A peek at algorithms involving data structures

The "example space" of numerical computer programming problems is very large; the list of example problems above is very small. Nonetheless, this small list will force the system to demonstrate competence in handling many (if not most) of the concepts of numerical computer programming. These concepts include: *convergence, power series, successive approximation,* symbolic differentiation (and other symbolic operations), transformations of program control structures, and limits. Since this system demonstrates "understanding" of these concepts, it is plausible that with a larger library of facts the system could solve any problem in the "example space."

Systems of equations: the "Linear equations problem"

The synthesizer is given the following three linear equations in three unknowns (the parsing of these equations has been shown explicitly):

$$((x + y) - z) = A$$
$$((x - y) + z) = B$$
$$((-x + y) + z) = C$$

It is told to write a program that takes A, B, and C as inputs and produces (for example) "x" as output.

Two observations about the synthesizer can be made concerning its ability to solve this problem. First, the system is *not* based on "refinement rules [Ba77]." The specification of the problem does not give any hint as to the eventual structure of the program. The system is not told it is solving a system of (linear) equations -- nor does it eventually recognize that it is doing so. Nor, for the same reason, is the synthesis system particularly "goal oriented" -- its goal is to get a program to compute the answer, but no other subgoals (in the traditional sense) are generated.

The second observation concerns the potential for combinatorial explosion. In solving this problem, the system only uses the standard arithmetical axioms of commutativity, associativity, identity (e.g. $q + 0 = q$), and the distributivity of multiplication over addition. If the system is to avoid exponential behavior, it must avoid discovering things like

$$2A - A = A.$$

That exponential behavior in this problem has been avoided will be demonstrated (in chapter 3) by the fact that the system needed (in retrospect) every deduction it made, with the exception of a few initial applications of commutativity. That is to say, the solution effort did not involve any unnecessary steps. Some statistics concerning this forthcomming solution may be of interest:

Number of rule pattern matches found: 148
Number of rules applied: 13
Number of rule applications required: 5
All unneeded rule applications concerned commutativity of addition.

Finding the inverse of a function: The SQRT problem

The synthesis system is told that

$$X * X = Y$$

and is asked to write a program that takes Y as input and produces X as output. Since this specification neither guarantees any real output (the case if Y is negative) or a unique output (the case if X is negative), the system is also told that

$$X > 0$$
$$Y > 0$$

Finally, the "absolute error" in X is bounded by a constant E. The system, using the fact about multiplication

"If $A > 0$ and $C > 0$ and if $A*B = C$, then $B > C$ if and only if $1 > A$"

deduces that

$$\text{if } Y > 1, \text{ then } 1 < X < Y$$
$$\text{if } Y < 1, \text{ then } Y < X < 1.$$

So the system knows upper and lower bounds for the value of X. The system also decides that the quantity X is a zero of the function

$$F(X) = Y - (X * X).$$

The system finally writes code implementing a bisection search for the zero of the auxiliary function *F(X)*.

A Digression

If my supervisor sent me a system message "Quick, write me a square root program" I would write a program using a bisection search because that is the program I'm able to write the fastest and with the least amount of thought. But I wouldn't be surprised to receive a second message "Thanks, I'll use that, but why don't you try to write a better program."

A practical automatic programming system (or, perhaps more accurately, a semi-automatic program development system, since even if programs won't need debugging, program specifications certainly will) should write programs as fast as it can so testing can proceed in a timely manner, and use its "spare time" to polish, hone, fine tune, and otherwise generally improve its code.

For both this practical reason and for complexity-theoretic reasons (various speedup theorems show that "optimal" programs do not exist for some recursive functions) the program synthesis system described in this report uses the same data base, rule library, and control mechanisms to write LISP programs, and to improve programs it has written.

SQRT example resumed: The "Newton's SQRT problem"

The synthesizer has written a square root function using a bisection search algorithm. It is now asked to improve that code.

The system continues using everything it discovered in the initial problem solving effort. One of the rules in the system's library says that if Newton's method converges, then on the average Newton's method will require many fewer iterations to obtain the same accuracy as the bisection method.

The rule says to use the following test (see [R69], p.1178, section 31.4.c) to check for convergence:

1. A zero of F(x) is in the range [a,b].
2. F'(x) is either always positive or always negative for x in [a,b].
3. F"(x) is either always positive or always negative for x in [a,b].
4. F(a) and F(b) have different signs.
5. Start with initial x = a or b, depending on which of F(a), F(b) has the same sign as F"(*root*). (The rule actually used ignores this restriction, but it could easily be

included.

In order to apply this test, the system must be able to take symbolic derivatives, and symbolically determine whether certain functions have zeros within specified ranges.

After succeeding in proving Newton's method converges, the system compares the time-costs of both methods, and decides that Newton's method is superior. In any future improvement efforts it will be the Newton's method computation that receives attention, not the bisection search computation.

The description above of the system's problem solving effort is in terms of goals and subgoals. It must be remembered that these terms are the author's. In fact, the system at no time announces to itself "now I will try to solve this subproblem".

In addition to showing the incremental nature of the synthesis system, this example also illustrates the system's approach to (worst case) time-cost analysis. There are two popular techniques of time-cost analysis:

1. Comparison of algebraic descriptions of the time-cost (the system can always obtain such an expression for an upper bound of the code's time cost).

2. Comparison of the numeric time-cost quantity for some "typical" set of inputs (this is the only method used by Kant's LIBRA system [K77]).

The synthesis system described here uses both techniques and notes an anomality if the two analyses do not agree in which computation is faster.

Iteration and Recursion: The "Factorial problem"

Notions of looping control control structure -- iteration and recursion -- are the most distinctive aspects of programming. These notions are absent in (if the reader will excuse the term) classical mathematics. A program synthesis system must therefore exhibit a facility in dealing with looping. Furthermore, the underlying theory of the system should include some description of recursion and iteration that is deeper than a purely syntactic distinction in the data-base representation.

The example used in illustrating the system's facility in dealing with loops starts out with a specification of the factorial function:

$$f(n) = n * f(n\text{-}1)$$
$$f(0) = 1 \text{ (specified as base step)}$$

When asked to write a program for "f", the system produces a recursive factorial (no interesting deductions are required) like the one below (1- is MacLISP's decrement function):

```
(DEFUN FACTORIAL (N)
       (COND ((= N 0) 1)
             (T (* N (FATORIAL (1- N))))))
```

This is an appropriate point to mention that program specifications can themselves be programs. In at least a limited sense a program synthesis system is also a program transformation system and a hairy optimizing compiler.

There are (at least) two iterative factorials that could presumably be derived from a recursive factorial function:

```
(DEFUN FACTORIAL1 (N)
       (DO ((M N (1-'M))
            (FACT 1 (* M FACT)))
```

```
                    ((= M 0) FACT)))

      (DEFUN FACTORIAL2 (N)
              (DO ((M 0 (1+ M))
                  (FACT 1 (* FACT M)))
                  ((= M N) FACT)))
```

Symbolically these three functions compute di'ferent expressions:

FACTORIAL: (N * ((N-1) * ((N-2) ...))) "counts down, computes up"

FACTORIAL1: (...((N * (N-1)) * (N-2)) ...) "counts and computes down"

FACTORIAL2: (N * ... * (3 * (2 * 1))) "counts and computes up"

The transformations from FACTORIAL->FACTORIAL1 is only legitimate because multiplication is associative. The transformation from FACTORIAL->FACTORIAL2 is only legitimate because the function "1-" has a functional inverse.

The synthesis system can perform both of these transfromations. The easier rule to state turns out to be the one resulting in FACTORIAL->FACTORIAL2. The other transformation can only be partially implemented because the system does not pursue subgoals of the form "prove the following (complex) operation is associatiative."

Data Structures and Bernoulli Numbers

A quick perusal of the literature on automatic programming, apprentice programmers, and program synthesis would suggest that the principal difficulty is associated with devising and manipulating the appropriate data structures for the particular program-task. So it must be surprising that, as yet, none of the examples have used a data structure more complex than "floating point" numbers, where even the number of digits of precision has not been relevant. In fact, the domain *numerical computer programs* was carefully selected to specifically avoid having to consider the problem of data structure design.

Why avoid worrying about data structures? There were several reasons. One was simply to limit the amount of work involved in the research (no one can fault the author for that!). Another reason is of more general interest. It has often been noted that there is "something funny" about the distinction between what might be called *temporal* organization and *spatial* organization within a computer program. Many computer programming languages use a "pun" in treating an array access in syntactically the same way as a function invocation. When one works at the assembly language level, one occasionally finds it faster to re-compute a quantity when ever it is needed than to store and recall it each time.

A decision was made to avoid data-structure considerations. But then the problem of computing Bernoulli numbers came up, and rather than generate horribly inefficient code, the author decided to build into the system a little data-structure design capability. This example shows what resulted.

Bernoulli numbers (written as a function B(n) for n a positive number) are defined by the equation

$$\text{SUM from } n=0 \text{ to inf. } B(n)*(t^n/n!)) = t / (e^t - 1)$$

Suppose the system is given this equation, and asked to write a program for B(n). The definition of B(n) is "buried" inside an (infinite!) loop. One might expect that this would make the problem hard to state, and the rules needed to solve this problem very hard to write. In fact, the representation for loops makes no distinction between "inside" and "outside", and the rules can be written straightforwardly.

The system solves this problem, unsurprisingly, in a more or less standard way. The left side is (or can be viewed as) a polynomial series (power series) in the variable "t". Using the fact (contained in the set of rules provided to the system) that

$$e^t - 1 = \text{SUM from m=1 to inf } t^m/m!$$

the defining equation can be written as the multiplication of two power series. But there is a general rule for multiplying power series. The system derives

$$\text{SUM from L=1 to inf [SUM from p=0 to L-1 B(p)*(1 / (P! * (L-p)!))] * } t^L = t$$

But from this it can be deduced that, except for L=1, the equation

$$\text{SUM from p=0 to L-1 [B(p)*(1 / (P! * (L-p)!))] = 0}$$

holds. Looking at the equation above, it is clear that if B(0) up to B(L-2) are known, then B(L-1) can be computed. Specifically,

$$\text{B(L-1) = - SUM from p=0 to L-2 [B(p)*(1 / (p! * (L-p)!))] * (L-1)!}$$

For L=1, it must be that

$$\text{SUM from p=0 to L-1 [B(p)*(1 / (P! * (L-p)!))] = 1}$$

so B(0) = 1.

The analysis above can be used to write a program to compute B(n), see table "Bernoulli code" in chapter 4, and diagram "Bernoulli example". If a recursive program is read off of the equations directly, the resulting program will run exponentially in the size of its argument. On the other hand, an algorithm employing a dynamically built table starting with B(0) up to B(n) has a time-cost of about n^3.

There should be no surprise that the synthesis system is able to write the "exponential" Bernoulli number generator. The purpose of the fourth example is to show how the classical n^3 algorithm is found and illustrating how algorithm specification and derivation might lead directly to data-structure specification and realization.

THESIS

One can investigate expert problem solving either "in general" or in a particular domain. This report concerns expert problem solving in the particular domain of numerical program synthesis. Free use is made of particular (and perhaps unusual) features of this domain. Although I will argue that the results of this investigation are generally applicable, the thesis being presented only concerns numerical program synthesis.

The usual investigation of expert problem solving develops a deductive component and a knowledge-representation scheme that is (claimed) *sufficient* for the task of interest. Beginning with a selection of facts and rules about the domain, these investigations develop systems capable of applying these facts and rules. Attention then is centered on controlling the system (or, rather, had better be centered on ...!), which may be more powerful than is actually necessary.

In sharp contrast, this investigation focusses on what deductive capabilities are *necessary* for the task of program synthesis. This research approach begins with constraints on the capabilities of the deductive component. The knowledge representation scheme reflects these constraints by (initially) having limited expressive capabilities. From this starting point the research approach turns to encoding domain facts and rules.

What *a priori* limitations can be placed on the deductive capabilities of a program synthesis system? While one would expect a synthesizer to be able to *apply* Newton's method for finding zeros of a function, for example, it need not be able to *invent* Newton's method. That is to say, there are problems we would all agree are simply too hard for a synthesis system (*any* system) to solve. There must be a class of problems (humanly solvable) that one would not ask human programmers to solve -- so an automatic system need not be capable of solving problems in this class.

One class of problems (*writing* code) we all can agree to try avoiding could be characterized as

> *"problems whose solution time is exponential (or worse) in the length of the solution derivation."*

(*Solution time* and *length* refer to writing the code, not running the code). Notice this is not the usual time-complexity measure. There are several useful operations for program synthesis that are (on the face of it) exponential. These include register-allocation (Bruno and Sethi show this to be NP-complete [Br76]), and theorem proving to show some series converges (discussion below shows this has an NP-complete problem as a subproblem). As a detailed example of this kind of problem is that of proving a statement in the propositional calculus is not a tautology (this is known to be NP-complete). The statement "A or B implies A" is not a tautology because it is false if A is "false" and B is "true". A proof is an assignment of "true" and "false" to the variables in the statement such that the entire statement is false. The best (known) algorithm for finding proofs like this is exponential in the number of variables (the length of the "solution").

One way to limit a deductive system (hoping to make it run in polynomial time) is to limit the expressive power of the "rule language." The term "expressive power" refers not to (an absence of) syntactic sugaring, but to the existence of predicate calculus formulae that cannot (in any way) be translated into rules. As an example of such a restriction in expressive power, consider a modification to the task (above) of proving the non-tautologic status of certain propositional calculus expressions. Suppose the form of the expression is limited to implications of conjunctions of variables -- expressions without the "OR" connective *and* without the "NOT" operator (these are Horn clauses, [H74]). Such an expression (like "A and B and C implies A and D") can easily be shown not a tautology by finding a letter variable (D in this case) on the right hand side of the implication that is not among the conjuncts to the left of the implication. If this letter variable is assigned "false" and all others assigned "true", the entire expression evaluates to "false." Expressions of this restricted form can be proven non-tautologies in *linear* *time* (actually O(n*log(n)) because of the time taken to look up variable names).

With these preliminary remarks, the *thesis* can be succinctly stated:

By limiting the expressive power of a rule language and correspondingly limiting the capabilities of the deductive system, a problem solver can be developed in which any problem whose observed solution involved N steps (solution of length N) will be solved in steps numbering no more than a polynomial function of N.

If solution time is a polynomial in the total number of rules applied (and it is), then a convenient measure of solution length is the number of rule applications actually required to solve the code synthesis problem. The thesis says that although useless deductions might be made, the number of useless deductions will be kept reasonably low (bounded by a polynomial in the number of worthwhile deductions that will be required) *by virtue of the deduction algorithm.*

This thesis is still only a conjecture. It will be defended, but not formally proven, by exhibiting a rule language whose expressive power has been limited in a way very similar to the way expressions were limited in the propositional calculus example above. A deductive mechanism will be described and demonstrated on the four key programming problems described above.

A Sketch of the Synthesis System

Here is a brief sketch of the synthesis system. The reader should not worry much about the details; all will be explained in chapters two and three.

In writing axioms for arithmetic, one often finds instead of, say, addition being treated as a function of two arguments, it is written as a predicate of three arguments. To say, for example, that A=X+Y and B=X-Y, one would write something like

(+C X Y A) and (+C B Y X).

The logician (ignoring the problem of how the variables should be quantified) would then interpret these expressions as declaring that certain relations hold among the variables X, Y, A, and B.

On the other hand, a computer scientist might interpret a statement

A = X + Y

as a computational specification -- it tells how to compute the value of a variable A from the values of variables X and Y.

The rule language for the program synthesis system is based on a representation combining both the declarative and computational interpretations. Specifically, a +C *device* would be defined as having three *terminals* (arguments) named SUMMAND1, SUMMAND2, and SUM. Some *constraint rules* would be written telling that some terminals can be computed on the basis of others, and giving details about how that computation is effected:

terminal computed	terminals needed	expression
SUM	SUMMAND1,SUMMAND2	(+ SUMMAND1 SUMMAND2)
SUMMAND1	SUM,SUMMAND2	(- SUM SUMMAND2)
SUMMAND2	SUM,SUMMAND1	(- SUM SUMMAND2)

The expressions can be thought of as LISP code, although the system actually uses a separate constraint rule language and interpreter system so that expressions can be evaluated not only to form code, but to provide time-costs, upper and lower bounds, and symbolic expressions.

The separation of "what terminals are needed" from "how to do it" is both due to efficiency considerations and for theoretical reasons. If I were to tell you that

$$C + D = E \text{ and } C + D = F$$

you would immediately conclude that E and F were the same. But if I claimed that

$$D = E^2 \text{ and } D = F^2$$

you would not feel safe in asserting that E = F. (E=3, F=-3, D=9 for example). In the next chapter it will be shown that whether identity can be deduced or not in the two cases above depends entirely on the "terminal-needed" parts of the constraint rules.

Stating the Problem

A programming problem is given to the synthesis system as a series of devices connected (at their terminals) to various variables (represented by *nodes*). When devices are connected to nodes, a *network* is formed. In addition to a network, the programming problem also specifies which nodes are to be inputs, and which is the output.

Although these networks are given to the system as a sequence of LISP expressions, several important insights can be obtained by examining a network drawn out in a diagram. The network corresponding to

(+C X Y A) and (+C B Y X)

is shown in diagram "two-linear-equations". This network formalism is based on Stallman and Sussman's electronic circuit analysis program EL [S79].

Suppose that nodes A and B are specified as inputs, and node Y is specified as the output. A process called *propagation* (a form of local deduction) would examine the constraint rules of the two +C devices (named D1 and D2) to see if any other nodes can be assigned values (see diagram "Value Propagation Example" in chapter 2). In this example no new values can be obtained, so some other, more global, deductions apparently must be made.

The diagram for this example contains a very important configuration called a *circuit*. This circuit starts at node X, goes to device D1 (direction is *not* important), then to node Y, then to device D2, and finally back to node X. None of the nodes in the circuit have been given values (which is to say that the system doesn't know how to compute them yet). A key observation is that if a problem can be solved, but is not solved by propagation, then the output node must be in a circuit (or can be determined by propagation from nodes that are in a circuit).

A second key observation is that to solve a problem one wants, somehow, for circuits to

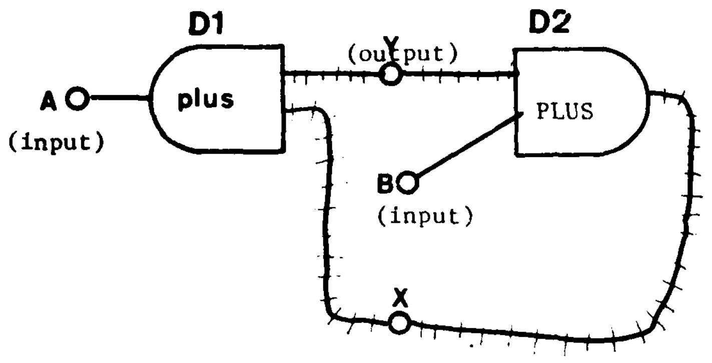


DIAGRAM "TWO LINEAR EQUATIONS"

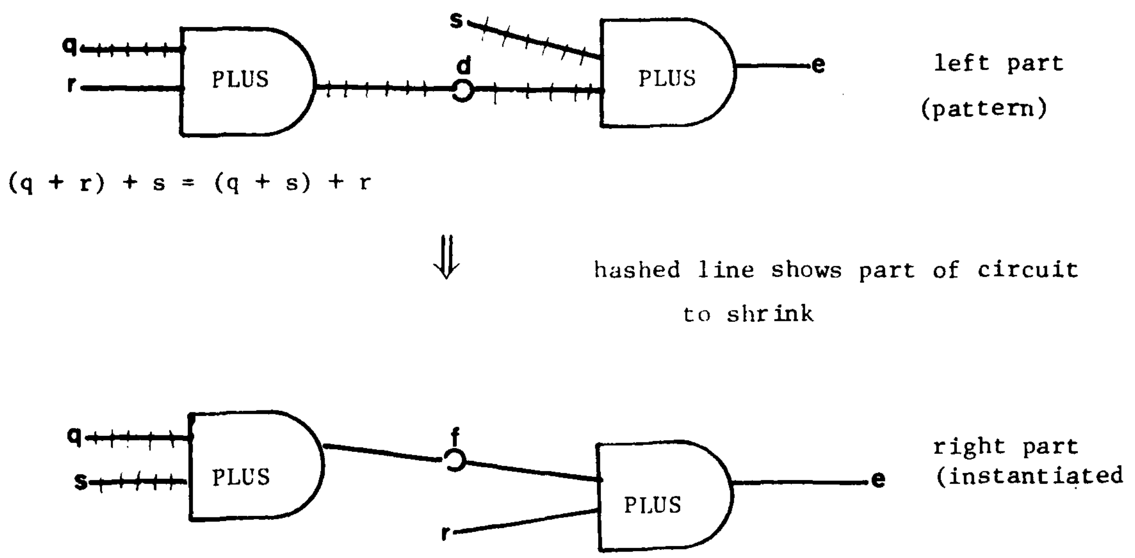


DIAGRAM "FUNNY ASSOCIATIVITY"

get smaller (see "Two Linear Equations Solution"). The system uses a library of *transformation rules* (or simply *transforms*) to make this happen.

Transformation rules

It is the language for writing these transformation rules that has been restricted. Recall that a network is a collection of interconnected devices and nodes. A network can also be considered as a conjunction of constraints (a *constraint* is a device interpreted as a "predicate" of all its terminals). By their nature (e.g., usage in this system) constraints must all be satisfiable (be "true") in the sense that there must exist some assignment of values to nodes so that all constraints (considered to be predicates) evaluate to "true." A transformation rule is a statement to the effect that one network can be transformed into another (called the "instantiation network"). Of course the entire transformation must be "true" when considered as a mathematical statement. But to prevent expressing a disjunction, a stronger restriction is imposed: the instantiation network, considered in isolation, must have the property that all its constraints (expressed as devices) are satisfiable.

For example, one transformation the system has in its library says that r+r=2r:

((+C R R S)) => ((*C 2 R S)).

This transform could be drawn as shown in diagram "doubling." An important observation to make concerning this diagram is that if S is known and node R is not, then the left side of the diagram contains a circuit (with only one node R) but the right side does not. The "doubling" transformation apparently "removes" a circuit when S is known and R is not.

Another transform, expressing the mathematical fact that

$$(Q + R) + S = (Q + S) + R$$

is shown in diagram "funny associativity." There is nothing special about this transformation, other than it happens to help solve the current problem. Written as constraints, this transform is

((+C R Q D) (+C D S E)) => ((+C S Q F) (+C R F E)).

The left part of this diagram (the *pattern*) matches the problem in the diagram "two linear equations" with the correspondence

Q => Y, R => B, S => Y, E => A.

The transform node D is also matched, but this doesn't matter because D does not appear on the right-hand side. The result of applying the "funny associativity" transform is to *add* a copy of the instantiation network to the problem network (of course, this may trigger other rules -- combinatorial explosion is a danger the system controls, see chapter 3). A new node must be created for the node F because it does not appear in the left-hand pattern part of the transform. The solution to this problem is found in the data base after applying "funny associativity" followed by "doubling." The state of the data base after all this is shown in diagram "two linear equations solution."

An important observation to make concerning the "funny associativity" diagram is that, in the left (pattern) part, there are exactly two circumstances in which using this transform will result in shrinking a circuit. These are first if nodes "Q" and "S" are known, and nodes "R"

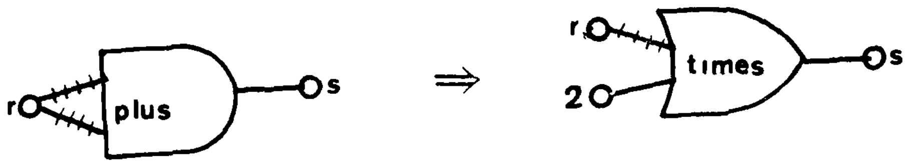


DIAGRAM "DOUBLING"

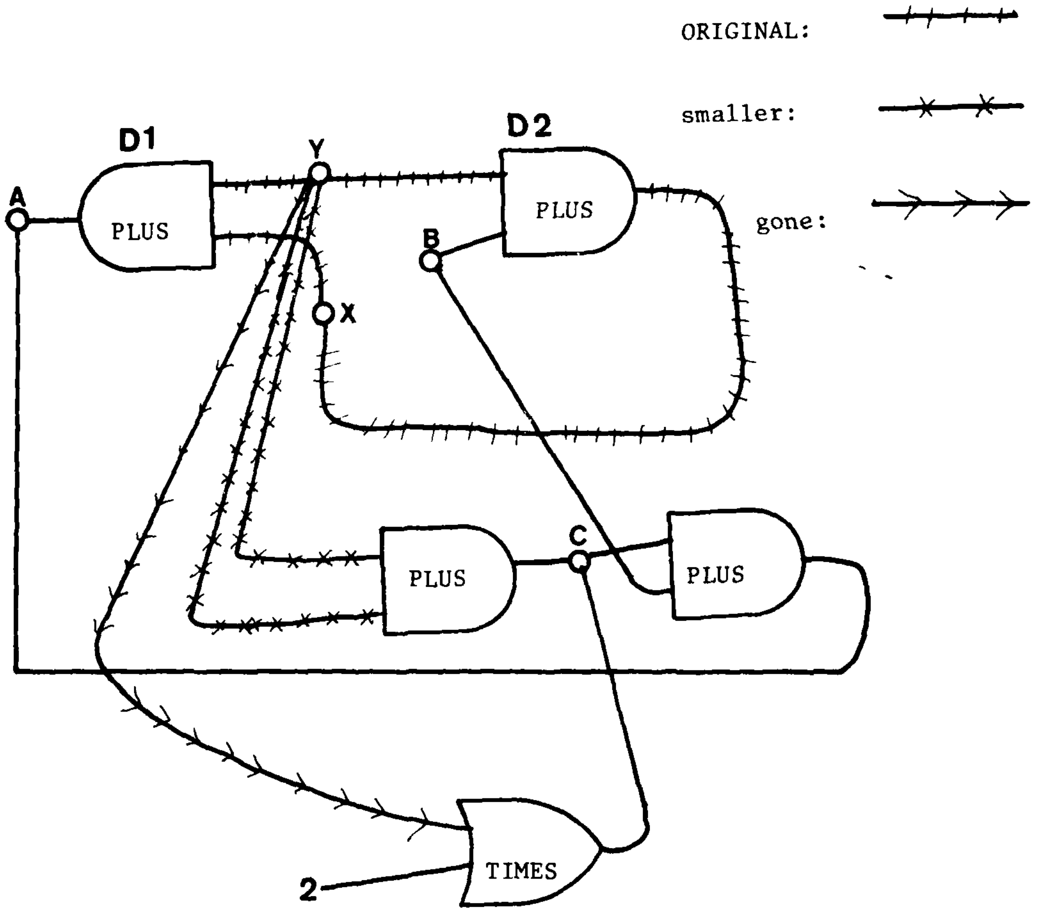


DIAGRAM "TWO LINEAR EQUATIONS SOLUTION"

and "E" are unknown, and second if nodes "R" and "E" are known and nodes "Q" and "S" are unknown. The second situation is extant in the initial problem.

In the diagram "two linear equations solution" the node C can be assigned a value by propagation (the value is A - B). Then Y can be assigned a value by the "+C" device via propagation. If code fragments are propagated instead of symbolic values, then code will be written for computing Y from inputs A and B, as was requested.

Limiting expressive power and combinatorial blowup

The mathematical facts contained in the "funny associativity" and "doubling" are of the form that the left side (a conjunction) *implies* the right side (another conjunction). All transforms have this simple form. Although all examples have been *equivalences* this is not generally true. Disjunctions (use of OR) and negations (use of NOT) are not allowed. The expressive power of the transformation rule language has been limited. Furthermore, it should be remembered that propositional statements of this restricted form (implications of conjunctions) are precisely those for which the non-tautology proof was easy.

To see that the expressive power *has* been limited, notice that even though implication can be rewritten as shown, it cannot be used to encode disjunction:

A => B is equivalent to ~A OR B

Trying to encode a disjunction using negation doesn't work because negations are not allowed:

(~A) => B (equivalent to A OR B).

Direct use of negation is not allowed. But maybe indirectly one could encode negation:

C => "FALSE" (equivalent to ~C OR "FALSE", equivalent to ~C).

But consider what such a transformation rule would have to say: From C one can deduce that some constraint holds among some nodes when in fact it can never be satisfied! This violates the imposed condition that all constraints in a network are satisfiable. That is, the transformation rule would necessarily be a lie. So a disjunction cannot be constructed in this (or any other) way. Besides, since the really important feature of the restricted form is that disjunctions cannot appear on the right-hand side of implications, the simplest "bad" form looks like:

A => [(C => "FALSE") => B] (equivalent to A => (B OR C)).

The form above is not an implication of conjunctions as required.

The restrictions to the transformation rule language may look familiar to those working with resolution theorem provers. A Horn clause has the form

(OR A ~T1 ~T2 ... ~TN).

It is a disjunctive clause with no more than one positive literal. Any implication of conjunctions of positive literals can be translated to Horn clauses (this is not true for more general forms). Resolution becomes more efficient if only Horn clauses are involved, see [H74].

By using this restricted form of transformation rule two separate kinds of combinatorial explosion (exponential behavior) have been avoided. The first concerns the general problem of proof by contradiction and data base splits. The second concerns legitimate but useless deductions.

If a problem-solver is faced with a fact like "x implies y or z" a common reaction is (if x is true in the current data base) to assume y and see if a contradiction results. If one does, then (one way or another) the deductive system "backs up" and decides that "not y" and "z" are both the case. Since in effect the problem solver is thus searching a tree, there is a potential for exponential behavior -- an exponential number of data bases must be searched (in general) as a function of the depth of the tree.

In sharp contrast, the transformation rule language has been restricted so that it is impossible to state a rule that might give rise to a tree of data bases. Reasoning by contradiction cannot be used by the deductive system because the rules are not expressive enough to say what should be done if a contradiction is found.

Another kind of exponential behavior arises in the problem of finding the value of a variable Vf given the values of V1 through Vn when the following is true:

$$(Vf + (V1 + (V2 + \ldots + (Vf + \ldots)))) = 0.$$

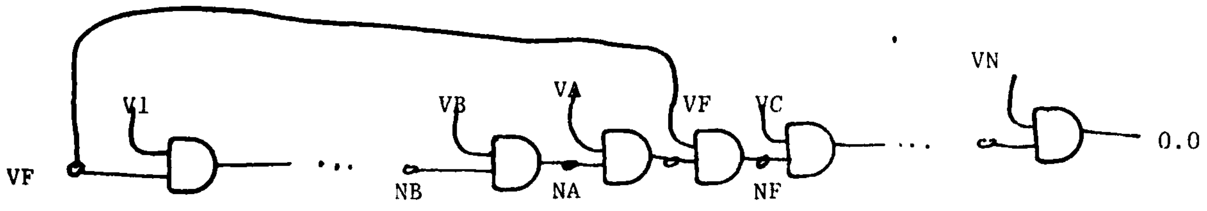
Suppose that the system can interchange two variables without changing the nesting. By applying the interchange operation, all sequences of Vf and V1 through Vn can be generated, and there are $0.5*(n+2)!$ such sequences. If the system can solve the problem after getting the two Vf variables next to each other, then in fact no more than n interchanges actually need to be performed. Since the factorial function grows worse than an exponential, there is a kind of exponential behavior to avoid. See diagram "Interchange Problem" for a demonstration of how knowledge of *circuits* solves this combinatorial problem.

Because the form of transformation rule is so simple, one can determine *in advance* the effect of applying a transformation with respect to shrinking or removing circuits. The doubling transformation, for example, removes the circuit containing the node R. Before an applicable transformation is actually used, the system performs a few simple tests to see if the transformation will shrink or remove a circuit of current interest (these tests involve examining nodes to see if they are known or unknown). If it won't, then the transform's use is postponed. In this way, the form of exponential behavior described above is avoided. *If disjunctions were allowed on the right of implications, this style of analysis would be impossible.*

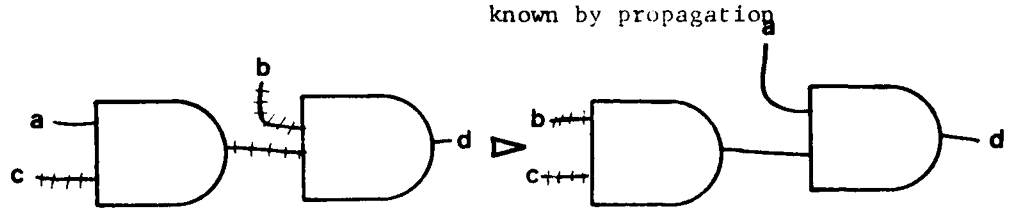
The mechanisms sketched out above are, of course, not sufficient for solving more interesting problems in program synthesis. Nothing has been said about representing and making deductions about iterative and recursive procedures, for example. The next sections hints at how the system can reason about iteration. There are other forms of combinatorial explosion to be avoided. To this end, attention-focusing mechanisms and local approximation techniques are used. There are philosophical questions like "How does one prove a rule library correct and consistent?" that must be addressed. These are the topics of the chapters that follow.

Specifications involving iteration

Iteration and recursion are fundamental features of computer programming. They become



VA,VB,V1,...,VN ARE KNOWN, SOLVE FOR VF
nodes NA, NB, ... are unknown, node NF and nodes to right are known by propagation

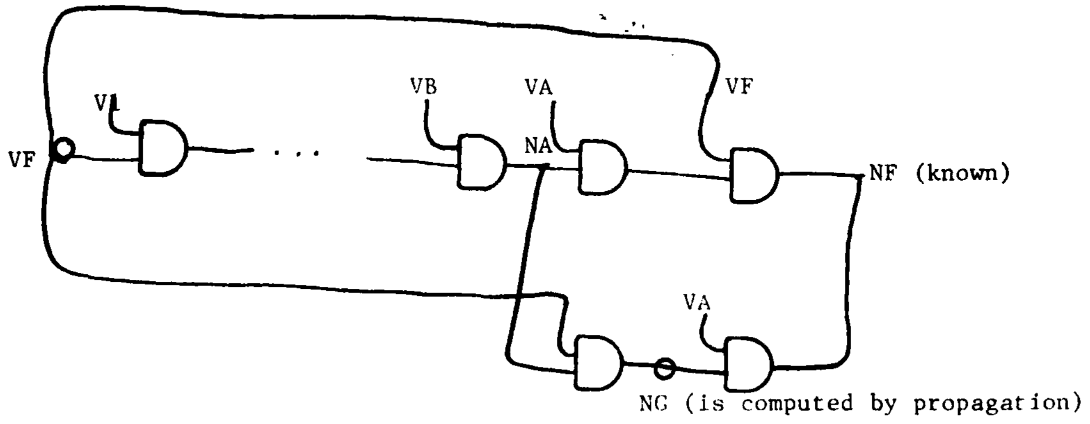


Above is the "interchange rule". A priori, this shrinks a circuit c--b, so system only (but see chapter 3) applies this rule if nodes c,b are unknown and nodes a,d are known.

The only site meeting application requirements has matching:

c - NA, a - VA, b - VF, d - NF

AFTER APPLICATION AND PROPAGATION, CIRCUIT WILL LOOK LIKE THIS:



NOW THE ONLY SITE FOR "INTERCHANGE RULE IS:

c - NB, a - VB, b - VF, d - NG.

DIAGRAM "INTERCHANGE PROBLEM"

manifest in an actual program's control structure. They also occur in specifications. One would like to be able to reason directly about looping control structures, but the undecidability of almost any question concerning looping procedures makes the situation hopeless. Since one cannot in general reason about looping control structures, the approach taken in this research has been to "bury" looping in device rules.

For example, there is a SIGMA device (see diagram "Sigma Device" in chapter 4) that contains prepackaged looping control structures for computing SUM from C, N, and a function F:

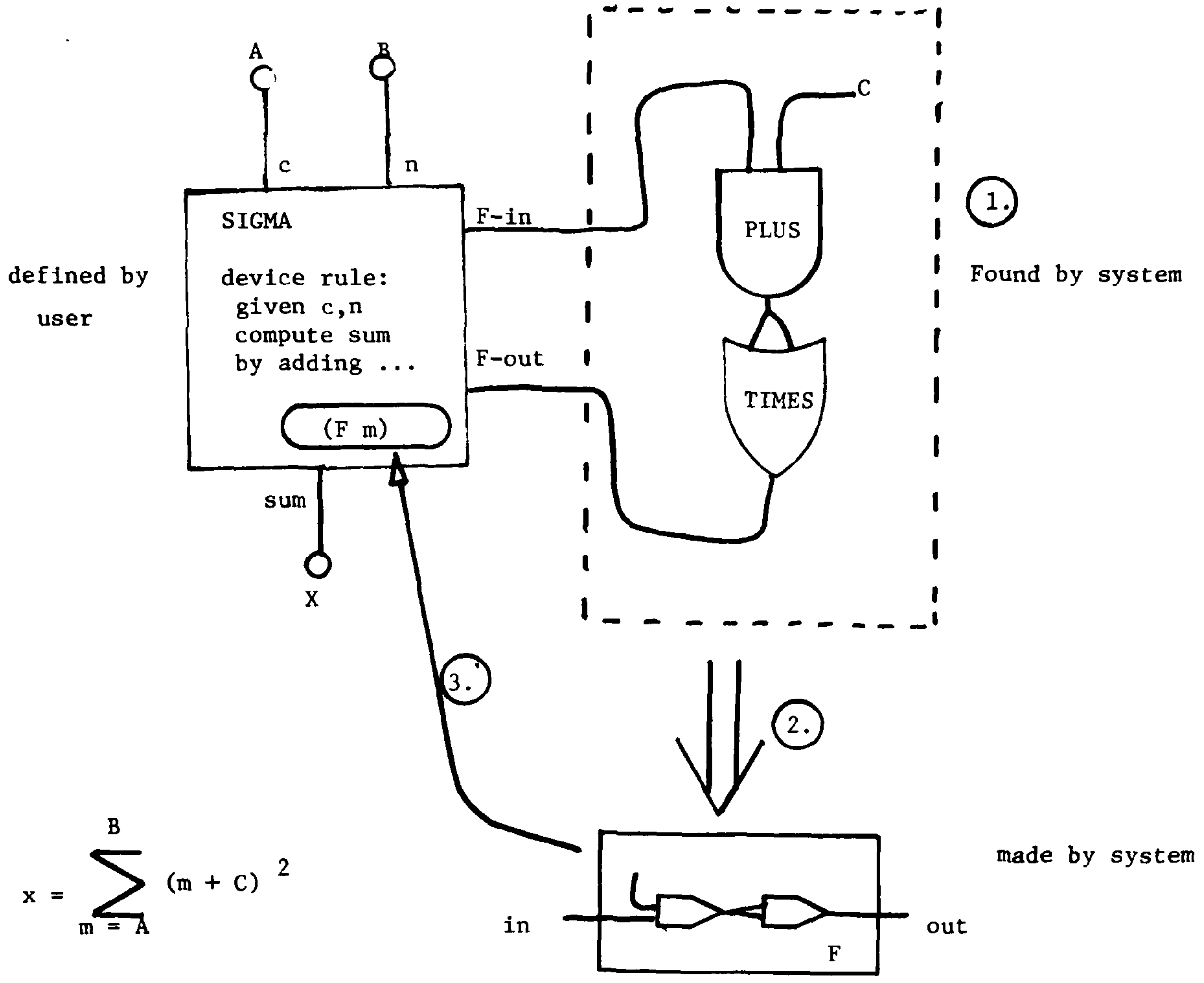
SUM = sum M from C to N of F(M).

The device rule for the SIGMA device contains a "blank" to be filled in with "code" for the function called F. This function F is to be found by the synthesis system. When the system finds how to compute F, that knowledge is packaged into a structure called a *macro-device*.

The key to being able to "bury" all iteration and recursion into prepackaged control structures is being able to tell the synthesis system how to find macro-devices.

When the user writes a device rule that uses a macro-device, the user must also tell the system how to find the macro-device when it is needed. This is always done by telling the system that the macro-device has certain nodes as inputs, and others as outputs. In the case of SIGMA, terminal F-in is macro-device F's input, and terminal F-out is F's output.

Diagram "Introduction to Macro Devices" summarizes how the system responds to a device rule in SIGMA asking for a macro-device F. Essentially, three steps are involved: first



$$x = \sum_{m=A}^{B} (m + C)^2$$

SYSTEM ACTIVITY

1. Find portion of network. In this case, portion that computes F-out from F-in.

2. Construct macro-device "containing" a copy of the portion of network found in step 1.

3. Use macro-device in interpretation of device rule.

DIAGRAM "INTRODUCTION TO MACRO-DEVICES"

find the portion of the network that can perform the required computation, then package this portion of the network into a macro-device, and finally use the macro-device in device rule interpretation. Later chapters will give algorithms for accomplishing these steps.

By using the macro-device mechanisms, devices for various kinds of iteration and recursion, as well as devices like SIGMA, devices for doing bisection searches, and devices for power serries manipulation have been written. Transformation rules involving these devices can ignore the fact that the devices' definitions involve looping control structures.

Related Work

Several other investigators have worked on the problem of program synthesis. There are three dimensions in which to classify these efforts: specification technique, rule form, and control mechanisms.

One can divide specification techniques into roughly five catagories:

1. Pre- and Post- conditions [M79].
2. I/O relations (this work)
3. Very High level languages [Ba77], [Bu77].
4. Examples [S77], [H75] and traces [Bi76]
5. Analogy [Br77], [U77]

By providing pre-conditions and post-conditions, side effects can be specified -- I/O relations can only specify what the output should be. Very high level languages (weaker still) are specifications that can (however inefficiently) be interpreted on specific inputs to obtain the desired output.

As an alternative to describing the program, catagories 4 and 5 are potentially handy specification techniques to incorporate into a large automatic programming system.

A second dimension concerns the form of rules used by the system. There are about three broad catagoiies:

1. Axioms/Theorems (this work), [E76], [G69]
2. Rewrite rules [M79], [Ba77]
3. Others [R80]

It is amusing that Emden and Kowalski's [E76] approach also restricts itself to Horn clauses Barstw [Ba77] and Rich [R80] have concentrated in building comprehensive libraries of rules

There are three levels of interest in controlling the synthesis system:

1. Completely manual [M79]
2. Semi-automatic Apprentice approach [R79], [R80], [Bu77]
3. Completely automatic (this work) [K77]

Kant's [K77] LIBRA system was the controller in Barstow's [Ba77] PECOS synthesizer component of the PSI [G76] automatic programming system.

Program synthesis is on the fringe of several other areas of current research. It can be viewed as the task accomplished by a smart compiler for very high level languages like CLU [L75] and Alphard [Wu76]. Weigbreit [We76] discusses such a smart compiler.

Traditionally, programming languages have tried to isolate the innocent programmer from details concerning storing data structures (early examples are ALGOL's dynamic array allocation and LISP's garbage collection). Low [L78] discusses the problem of efficient data structure selection.

The subjects of program specification and proving programs correct could be viewed as the "inverse" of the subject of program synthesis. Synthesis goes from specification to programs, while proving a program correct goes the other way. The "verification" literature is vast. Shrobe [Sh79] suggests that understanding code, expecially when the understanding can be modified as code is modified, is much more valuable than proving it correct.

CHAPTER 2

INCOHERENT SOURCES OF KNOWLEDGE

Local vs. Global

The term "incoherent" in the title is not intended to refer to the expository style of the chapter but to the "local" nature of descriptions. That is, the various descriptions (embodied in "rules") don't know about each other, and so they cannot negotiate cooperation in any way other than their effects.

One way to describe the meaning of a relation like +C or *C (corresponding to operations of addition and multiplication) uses a computational model. Another employs axiomatic methods.

Using the +C relation introduced in chapter 1, if the constraint

(+C A B C)

holds, then by using a computational model one could determine a value (symbolic or numeric) for C given values for A and B. The system uses ***constraint rules*** as a source for this kind of knowledge.

The synthesis system also uses axiomatic descriptions. One would state the standard associative axiom for addition as a transformation rule:

(+C A B G) and (+C G C D)

=> (+C A E D) and (+C B C E).

Both of these types of rules are *local* in the sense that they describe behavior of structures (devices and networks of devices) in isolation. One might think that information concerning when these local rules should be applied must be non-local in nature. Chapter 3 will show how this knowledge can be (mechanically) derived from these local, incoherent sources.

Table "Knowledge Sources Summary" lists the basic knowledge sources for the system. This chapter explains these sources in the order shown in the table. It concludes with a complete solution to the square root problem.

A glimpse of coherent behavior

The system's deductive capabilities center on two different procedures: *value propagation* (uses *constraint rules*) and *transformation application* (primarily uses *transformation rules*).

The system's representation of problems is primarily relational. This means that the system must have a simple and straightforward way to extract computational representations from relational ones. The procedure used for this purpose is *value propagation*. Looking at the diagram "value propagation example", suppose the node A has as a value the system's expression

(*VARIABLE A)

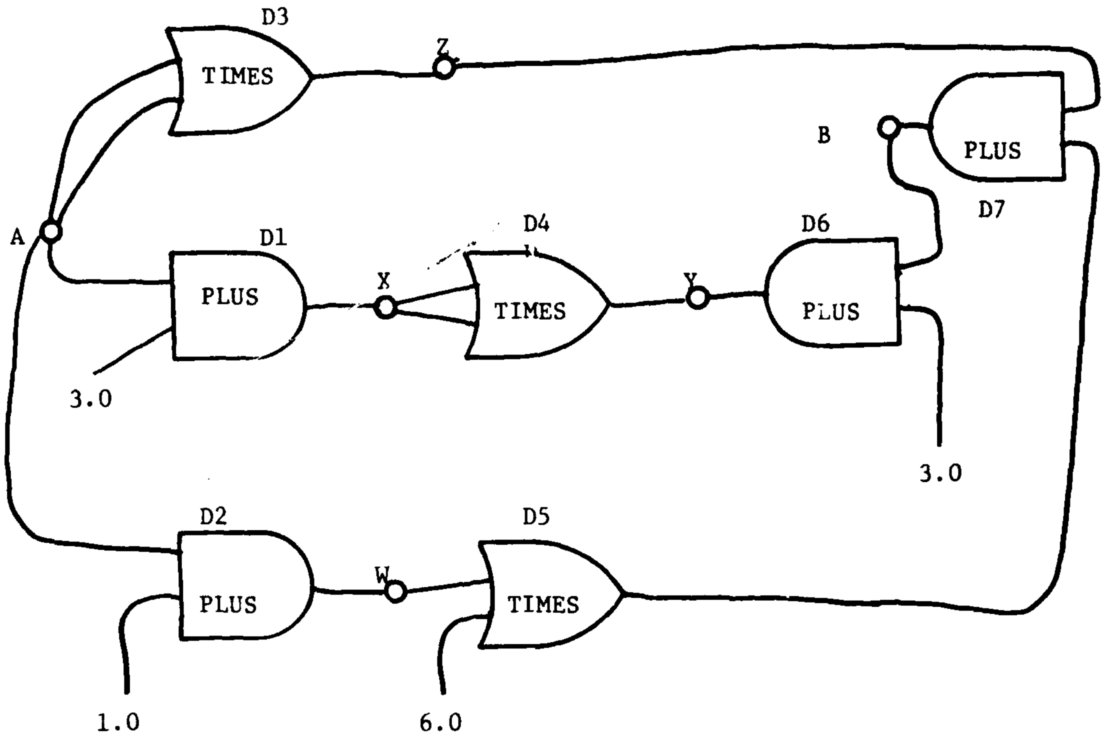
and that nodes X, Y, Z, W, and B do not have values. The value propagation (or simply *propagation*, though the term will also be used for the matching process) procedure finds all the devices attached to node A. These are devices D1, D2, and D3. Each device's list of constraint rules is then searched to find rules with values in all the nodes corresponding to terminals in the rule's terminal-needed list. Each device happens to have such a rule. The rules are then interpreted. The rule for device D1 under the interpretation for algebraic expressions yields for node X (notation will be explained later):

```
(*EXPRESSION (PLUS (*VARIABLE A) (*CONSTANT 3.0))).
```

Similarly expressions will result for nodes W and Z. Finally, *propagation* checks to see if the resulting expression is an improvement over the expression (if any) already in the node. If it is, the node's value is updated. This process is repeated until either the output node (say node B) receives a value, or until no more propagation can take place. In Stallman and Sussman [S79], other propagation techniques are used ("plunking"). These techniques are compatible with the techniques used here, but have not been incorporated into the system.

Value propagation does not lead to exponential behavior. The same wonderful accolade cannot be given to the transform application process. Obtaining coherent behavior in applying transforms will be a topic of the next chapter. Briefly, the steps in applying a transform are:

1. Match the pattern of the transform to the problem's network (the *datum* network).
2. Bind any variable-nodes. (This is probably incomprehensible, but don't worry, an explanation is coming.)
3. See if the problem context is one that agrees with the network's applicability (for example, can the transform reduce the size of a circuit). This step (and some of the following) constitutes the main topic of the next chapter. Assumptions may need to be made.
4. Match any macro-device specifications . (Ditto).
5. Check any *node-independence requirements*. These arise in, for example, rules for taking derivatives (the derivative of something that doesn't depend on the variable under consideration is zero, etc.).
6. If everything matches up properly, add the *instantiation* part of the transform to



If A is given, propagation gives values as shown in the table:

<u>Device responsible</u>	<u>node</u>	<u>value assigned</u>
D3	Z	A^2
D1	X	3*A
D2	W	A+1

D5	C	w+6 = a + 7
D4	Y	x^2 (= $(c*a)^2$)

D7	B	z+c
D6	B	3*y

DIAGRAM "VALUE PROPAGATION EXAMPLE"

the datum network.
7. Instantiate any required macro devices (see next chapter).

This chapter explains how to provide the system with all the information needed to perform transform application. The algorithms and control mechanisms will be found in the next chapter.

DEVICES

Problem: To say anything about computation, I need objects and something that acts on those objects.
Solution: Nodes "contain objects, and devices act on those objects according to rules.

The basic relational and computational building block in the system is the *(constraint) device*. The term "constraint" will be used to emphasize the relational nature, and "device" to emphasize the computational nature of the concept. This section explains how devices are defined. For purposes of illustration, this discussion will center on three devices used in the SQRT example. The device named "*C" is used to express multiplication and division. The device "GREATER" expresses inequality. The device "BSFZ" finds the zero of a function by using bisection search. Table "Three Device Definitions" shows how these devices are described to the system.

The statement

(MAKE-DEVICE name-of-device <name-of-terminal>)

causes the internal data structures for a device type to be set up. These structures include device templates and empty rule lists for both weak and strong constraint rules. Terminals are like LISP lambda variables in that they are a way for something conceptually inside a device to refer to a node outside. In this report phrases like "the terminal's value" usually are shorthand for "the value of the node attached to the terminal" (where "value" is again shorthand for some expression of a facet in the node's value structure). The casual reader can safely ignore the distinction between "terminal" and "node."

The device type named "*C" has only *strong constraint rules*. These rules claim that a value can be computed for one terminal on the basis of values on other terminals (actually the value structure of a node attached to a terminal can be constructed on the basis of the value structures of nodes attached to other terminals). The statement for making a rule has the form

```
(RULE-OF name-of-device terminal-computed
        (<terminal-needed>) expression [(<macro-device-specification>)]).
```

The purpose of having macro-device specifications will be explained in the next section. BSFZ uses a *macro-device it calls* F.

If one knows that X > Y, expressed as

```
(GREATER X Y (*MODE-CONSTANT *TRUE))
```

and if one also knows the value of X, then the value of Y can be constrained even though it cannot be computed. In particular, *if* X = *3.0, then* Y*'s upper bound can be computed to* be 3.0 (or less). Similarly, from Y's value, X's lower bound can be determined. These facts about the meaning of the GREATER constraint are expressed by weak constraint rules:

```
(WEAK-RULE-OF name-of-device terminal-computed
```

(<terminal-needed>) expression).

The term "weak" is used to indicate that the rule is not strong enough to constrain the *value* facet of the node of the terminal computed.

Expressions are written in the device-rule language. The form of these expressions is generally one of the following:

(type-of-construction expression)

(type-of-construction (<expression>)).

The types of constructions used by the system are shown in the table "constraint-rule constructions." If the type is *EXPRESSION, then the expression itself can be used "bare" without the tag "*EXPRESSION". Some constructions require mode expressions in certain positions (for example, a *CASES construction does a dispatch on the result of evaluating a mode expression). In these circumstances the tag "*MODE-EXPRESSION" can similarly be left off. This use of modes was suggested by the representation in EL of transistor (devices) as operating in different modes (specifically in "active", "saturated", and "cutoff" modes).

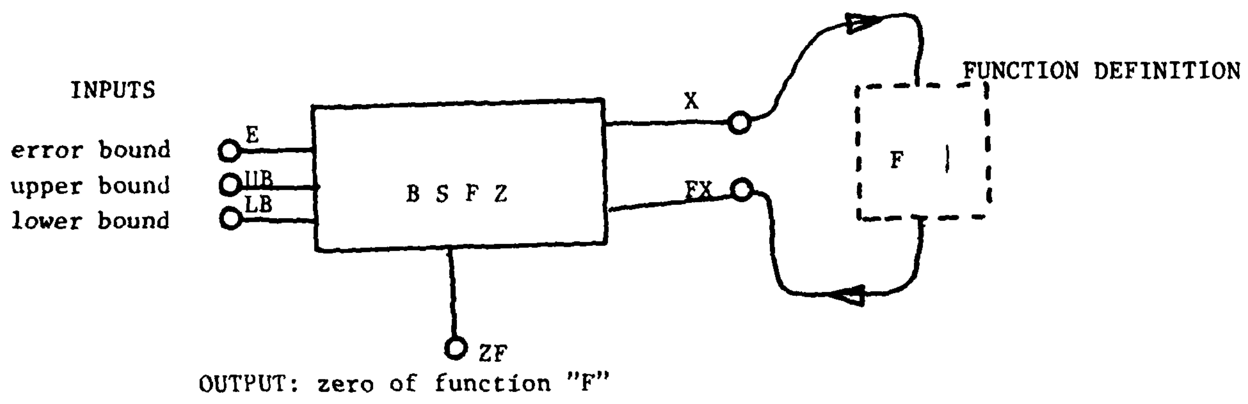
It is important that all devices in a network be satisfiable. In the initial problem specification this means that there must exist some assignment of numeric, mode, or "non-existant" (*NOT-EXIST constructions, see table "constraint rule constructions") values such that if any device rule is interpreted, the "new" value and existing values agree.

Macro-devices and Loops

Problem: To solve the square root problem I want to use a bisection-search schema. This schema says "code for finding a zero of some function F looks like this, with blanks that need to be filled in with code for the function F." The device rules will let me write such a schema, except for specifying the blanks and for finding the function F.
Solution: I have an algorithm (Chapter 3) for finding, coding, and packaging up functions given a specification of the function's input and output nodes. The output of this algorithm is called a macro-device.

Subroutines and looping control structures (whether using GOTOs, DO loops, or recursion) are fundamental to programming. The synthesis system brings a collection of novel and powerful techniques to bear on the problem of reasoning about looping control structure. The basic idea is to "package" the computational relationship between two (or more) nodes of a network into a structure called a *macro-device.* Having the computation path in this easily digested form lets the constraint rule interpreter use that computation to build code (and other facets) involving looping control structures. Manna and Waldinger [M79] use an alternative technique first presented in Burstall and Darlington [Bu77]. The technique is based on the simple idea of noting when a subgoal is an instance of the top-level goal.

As an example, the bisection solution to finding square roots is in diagram "BSFZ SQRT solution". The BSFZ device wants to write a code-fragment containing a DO loop to find the zero of some function (see diagram "BSFZ"). Although the device rule doesn't really know what the function is, the rule says where to find the function (internally called F): it is the function that, given a value in terminal X (node NX), will compute a value for terminal FX (node NFX). The device rule only tells *where* to find the function; *how* to find it is a topic of the next chapter.



LB is less than ZF is less than UB

BLANKS TO FILL IN WHEN "F" IS FOUND

BISECTION SEARCH SCHEMA

```
(make-device-type bsfz e ub lb zf fx x)
(rule-of bsfz zf (ub lb e)
        (*do ((bit-length (prim-div (prim- ub lb ) e))
              (prim-less  (prim- (*do-variable lub) (*do-variable llb))  e)
              (*do-variable L)
              (l (prim-div (prim+ ub lb) (*constant 2.0))
                 (prim-div (prim+ (*do-variable lub) (*do-variable llb))
                           (*constant 2.0)))
              (sgn (prim-sign ((*macro-device f fx) ub)) (*do-variable sgn))
              (lub ub (*cases ((prim=
                                 (prim-sign ((*macro-device f fx) (*do-variable l)))
                                 (*do-variable sgn))
                                (*true (*do-variable l))
                                (*false (*do-variable lub)))))
              (llb lb (*cases ((prim=
                                 (prim-sign ((*macro-device f fx) (*do-variable l)))
                                 (*do-variable sgn))
                                (*true (*do-variable llb))
                                (*false (*do-variable l)))))))
        ((f (x) (fx))) )
```

where to look for "F"

DIAGRAM "BSFZ"

A macro device is specified (within a device rule) by giving a name (meaningful only for the particular device instance), a list of terminals to be considered as *inputs* to the macro device, and a list of terminals considered as *outputs*:

(macro-device-name (<input-terminal>) (<output-terminal>)).

Occasionally, one needs a macro-device with multi-directional functionality. For example, one might need a macro-device with two terminals that can compute either terminal from the other (two directional functionality). For obscure and unimportant reasons, a separate format is used. The first direction is defined using the form above, and subsequent directions are defined using the form

((original-name new-name) (<input-terminal>) (<output-terminal>)).

In any such construction some of the original input terminals will necessarily become output terminals, and *vice versa*. An example of this construction can be found in the rules for a device named SIGMA2, discussed in chapter 4.

The macro-device specification for F in the rule for BSFZ is

(F (X) (FX)).

If such a macro-device can be found (as it can in diagram "BSFZ SQRT solution" as shown by the dotted lines), then the device's constraint rule can be interpreted. The computational relationship of the inputs to an output contained in a macro-device can be accessed in the rules by a construction:

((*MACRO-DEVICE device-name output-terminal-selected) <input-expression>).

This construction is little more than a subroutine call, where the subroutine was written as a result of finding the macro-device. The macro-device found for F has an extra input terminal for node SQ (the number to find the square root of). It will turn out that these extra inputs play several important roles.

The algorithm for finding what nodes and devices to include in a macro-device is a major topic of the next chapter. The computation path discovered for a macro-device (both for constraint rules above and for *transform macro specifications* below) is always packaged up in its own network.

Rule Closures

Problem: Propagating time-costs and value expressions separately causes a technical problem of synchronization.
Solution: Rule closures are a technical fix for this problem.

When a rule of a particular instance of a device type is used to update some facet of a node's value structure, the rule is recorded along with the new value. It is recorded in the form of a *rule closure* that contains the rule, the device having that rule, and any macro-devices the rule used. As with a rule, a rule-closure can be interpreted for various facets. The system insures that, for example, the TIME-COST facet in a node's value structure is the cost corresponding to the code-fragment in the CODE-EXP facet of that node by using the rule-closure for one facet to get the expression for the other.

Networks

Problem: I need to talk about collections of nodes and devices for a host of purposes.
Solution: Networks are a bookkeeping structure for such collections.

A network is simply a collection of instances of devices and nodes all connected together. Certain nodes of a network are singled out as corresponding internally to the network's *terminals*. For example, the following creates a network that serves as one form of the problem statement for the SQRT example:

```
(define-network sqrtnet (sq sqrt erb)
        (*c sqrt sqrt sq)
         (greater sq (*constant 0.0) (*mode-constant *true))
         (greater sqrt (*constant 0.0) (*mode-constant *true))
         (error-bound sqrt erb (*mode-constant *true))
         (*c (*constant 10000.0) erb sq))
```

Another statement (the one for the example to be solved) can be found in diagram "SQRT Problem Statement." The statement above has the form

(DEFINE-NETWORK network-name (<terminal-names>) <device-specification>)

where a device-specification has one of the following two forms:

(device-type <node-reference>)

((device-type <node-reference>) device-name).

The sequence of node-references corresponds to the sequence of terminal names in the device-type declaraction. Naturally I have written a parser to convert "algebraic forms" into networks, but the details of this parser are not interesting.

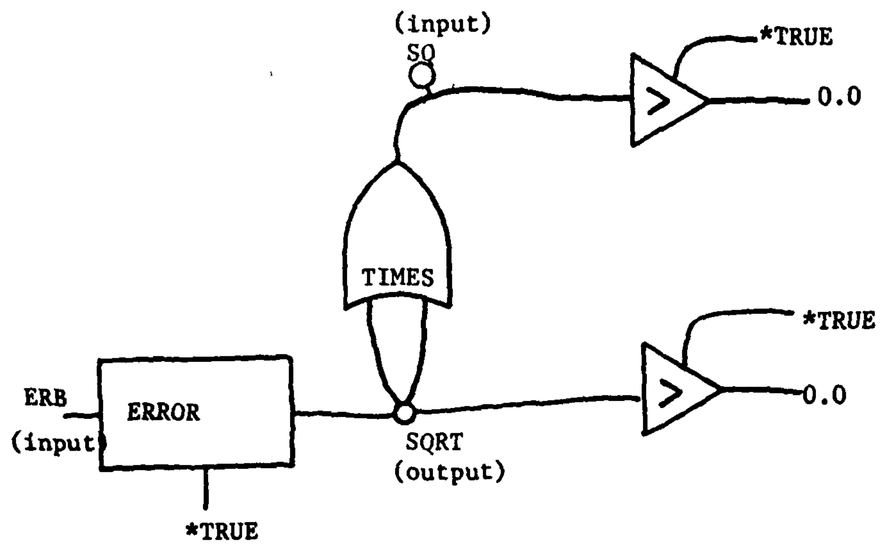


diagram "SQRT PROBLEM STATEMENT"

```
;;Here is the square-root example.

(defun testsqrt ()
 (define-network sqrt (sq sqrt erb)
 (*c sqrt sqrt sq)
  (greater sq (*constant 0.0) (*modal-constant *true))
  (greater sqrt (*constant 0.0) (*modal-constant *true))
  (error-bound sqrt erb (*modal-constant *true)))
  (encode-network *csprobs 'sqrt-erb '(sq erb) 'sqrt))
```

Networks are basically a convenience for bookkeeping. For example, the map from atomic names of devices and nodes to the structures named is recorded on the property of the network.

The system creates a network whenever a macro-device is found. This network then serves as part of the definition of the macro-device. Similarly, when the synthesis system creates a (LISP) function to compute one network terminal from others, the relevant information concerning how that function was written is recorded on the defining network.

Within the synthesis effort, there is always a *current-network*. This is the network created by the problem statement. It serves as a data base for the synthesis system's two deductive procedures. In particular, *transform application* may add devices and nodes to the current-network.

Complex Devices

Problem: I need to express hierarchies. I can build a network out of devices, but I cannot then use that network in building a larger network.
Solution: Complex devices are devices with networks that describe their "internal structure."

The devices discussed thus far have all been "simple" devices. They are given rules, but they have no internal structure. It is possible to create another kind of device with an internal structure specified by a network. The system in fact creates these *complex devices* whenever it finds a macro-device.

The role of complex devices in the synthesis process is two-fold. First, they are added to the problem network when it is necessary to copy a computation path. By using complex devices rather than copying the constituent nodes and devices of the computation path, the amount by which the network's size increases can be limited. The second role of complex devices is as a form of "grey box." One of the first things the system does when trying to improve code is to find all the complex devices actually used and *expand* them by creating copies of the nodes and devices in the defining network and adding them to the problem network

It is also possible to create a complex device manually, although it is extremely painful to do so.

TRANSFORMS

Problem: Deduction via value propagation is not sufficient to solve some synthesis problems. For example, the SQRT problem cannot be solved using propagation alone.
Solution: Transforms change one synthesis problem to a (hopefully) simpler one.

A *transformation rule* (or *transform*) is a type of situation/action rule. The "situation" part of the transform is specified by a *pattern network* and certain other auxiliary specifications. The "action" that may follow detecting the specified situation is always the addition of devices and nodes to the current network. This action is accomplished by instantiating the *instantiation network* of the transform.

A transform is defined by a statement of the form

(DEFINE-TRANSFORM name (<common-terminals>) (<pattern-device-spec>)
(<instantiation-device-spec>)
[(<add-break-spec>)])

The first pattern device is always considered the "seed" device (see chapter 3). The first important transform to be used in the SQRT example is the following:

> "If A and C are known to be positive, and if A * B = C, then if A > 1, then B (which must be positive because both A and C are) must be less than C. Similarly, if A < 1, then C < B." That is, the range of B can be determined by a range test on A.

This transform (see diagram "Add-break in SQRT") is written as follows:

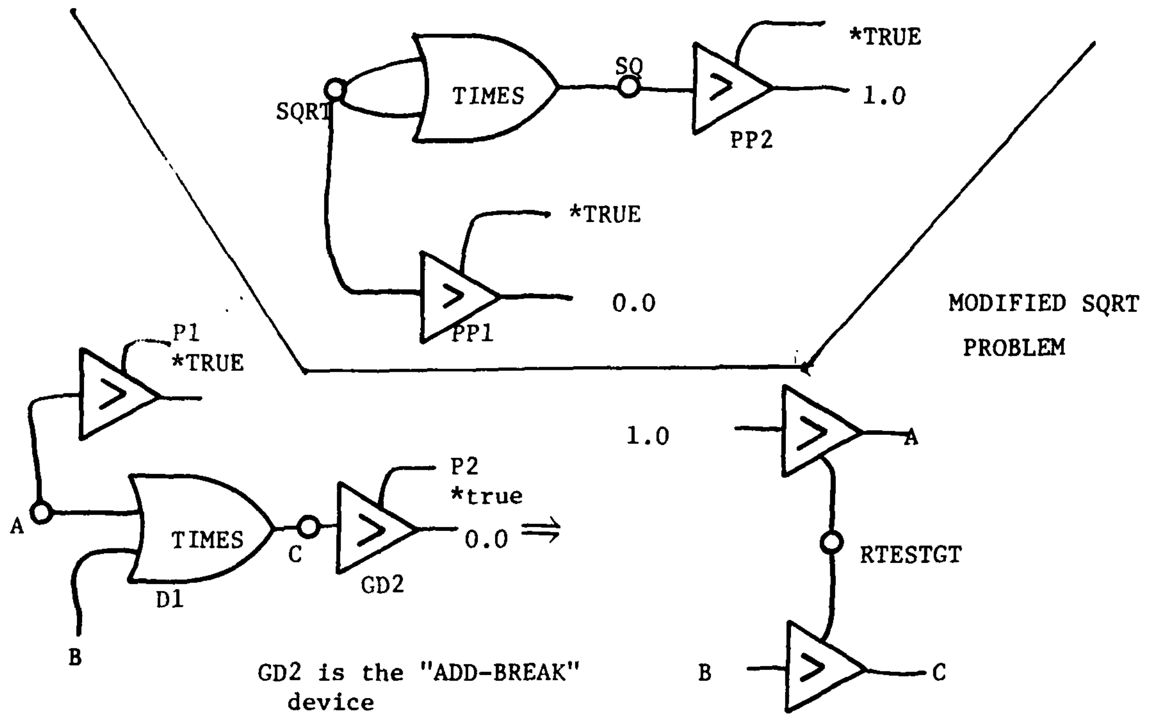
```
(define-transform mult-sign (a b c)
                  ;;Pattern network:
                  ((*c  a b c)
                   ((greater a (*constant 0.0) ((*mode-constant *true) p1)) gd1)
                   ((greater c (*constant 0.0) ((*mode-constant *true) p2)) gd2))
                  ;;Instantiation network:
                  ((greater (*constant 1.0) a rtestgt)
                   (greater b c rtestgt))
                  ;;Here are the add-break declarations
                  ((gd1 p1) (gd2 p2))  )
```

The two devices named GD1 and GD2 are flagged as add-break devices, to be explained in the next section. Notice that the node named RTESTGT (which is a mode valued node) could be computed by either of the GREATER devices.

Both the pattern network and the instantiation network have the same terminal names (A, B, and C). When compared to the network in diagram "SQRT problem statement", these terminals match up as shown:

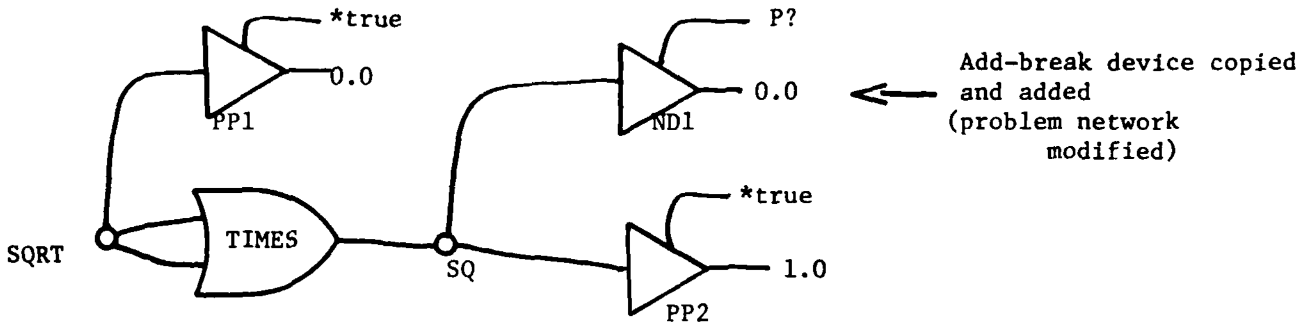
A => SQRT; B => SQRT; C => SQ.

Instantiating the instantiation network results in creating two new devices, both of type



GD2 is the "ADD-BREAK" device

MULT-SIGN TRANSFORMATION RULE



*true
0.0
PP1
P?
0.0
ND1
Add-break device copied and added (problem network modified)
*true
1.0
SQRT
TIMES
SQ
PP2

In applying MULT-SIGN, D1 → PD1, P1 → PP1, C → SQ, A → SQRT, B → SQRT GD2 fails to match. ADD-BREAK restrictions are met, so a new device ND1 is created. Now propagation gives:

P? = *TRUE

Now GD2 matches ND1, P2 → P?, etc.

DIAGRAM "ADD_BREAK IN SQRT"

GREATER, and creating two new nodes, one for RTESTGT, and the other for a constant value 10

Since each device in a network can be interpreted as a predicate, it is possible to prove each rule correct, in the same way one could prove a theorem about mathematics. Because iteration, recursion, and data structures (see chapter 4) are packaged inside devices, these issues can be ignored when proving a transformation rule correct. Naturally, one must also prove that device rules "do the right thing."

Having proven device rules and transformation rules correct, then *assuming* the rule application mechanism performs as advertised, one can conclude that all code synthesized is correct (it meets the specifications presented to the synthesis system). Better test it anyway -- just because something has been "proven" doesn't make it true!

Add-Break facility

Problem: Sometimes the system says to itself "I could match this transform to the problem network if only I could find out if 'X?' were true."
Solution: The Add-break mechanism lets the matcher ask "X?" and take a break while value propagation tries to find the answer.

Suppose in the SQRT example that instead of declaring that the input (SQ) was greater than the constant 0.0, I had said it was greater than 1.0 as shown in diagram "Add-break in SQRT." Then the transform MULT-SIGN above would not have matched - a problem! The solution (discussed below) involves adding (under certain circumstances) a copy of the pattern device that didn't match to the problem network. The pattern device that can be copied is flagged as an "add-break" device (the copy is added during a "break" in the matching process).

There are several approaches to this problem, including attaching "arbitrary" predicates to match nodes or using data-type predicates like POSITIVE, and pushing around devices to check these predicates. For example,

```
(define-transform positive-declaration (x y)
        ((greater x y (*mode-constant *true)))
        ((POSITIVE y mv1) (POSITIVE x mv2) (IMPLIES mv1 mv2)))
```

This transform (not in the system!) could be used to deduce that if the input SQ is greater than 1.0 then it must be positive.

There is a basic problem underlying this issue having to do with the interaction between deduction via computational models and deduction via axioms. Under certain circumstances a relation in an axiom can be replaced by a check with the computational model.

The *add-break* facility is the synthesis system's way of allowing the matcher to perform computational checks by invoking all its capabilities of value propagation (and incidentally all other deductive mechanisms as well). Certain devices in a pattern network can be flagged as "add-break" devices. If the matcher cannot find one of these devices, it creates a new device of the proper type, and any new nodes required (see below), and adds the device and nodes to the datum network. Then value propagation is attempted for the device added. If propagation takes place, then the match is retried (this is a very limited type of subgoal -- limited because success of the goal does not really depend on the success of the subgoal, and the validity of the subgoal deductions does not depend on assumptions made for the goal). In diagram "Add-break in SQRT" propagation does take place, and when the match is retried it succeeds.

There are some restrictions on what devices in a pattern can be flagged as add-break. At the time of a match, the nodes on the terminals of the add-break device will fall into three catagories (these can be determined *a priori*):

MATCHED: nodes that have been matched in the datum network.

INITIALIZED: unmatched nodes that should be initialized if they are created and added to the datum network by the add-break facility.

UNINITIALIZED: unmatched nodes that should not be initialized.

Matching add-break devices is left until last by the matcher. This means that unmatched pattern nodes are not connected to pattern devices other than perhaps to other add-break devices in the pattern. The add-break device must have strong constraint rules for all the uninitialized terminals, and these rules should need only terminals with matched or initialized nodes on them. The add-break device must not have a *strong constraint rule computing* matched nodes. If these restrictions were not satisfied, then possible contradictions would be created as a result of adding the device.

In principle those pattern devices that could be flagged as add-break devices could be found and flagged automatically. Currently they must be specified manually. An add-break specification has the form

(add-break-device-name <uninitialized-node-name>).

All instances of devices meeting the criteria above have been so flagged (currently only equality and ordering predicates are affected)

Macro-devices for Transforms

Problem: Usually I want to modify a problem network by adding devices. But sometimes simply adding structure will cause "global" contradictions. I need to specify a portion of network to copy, and then make the additions to the copy.
Solution: Macro-device specifications for transforms specify where to find the portion of network to copy, and where to copy it.

Just as with constraint rules for devices, macro-device specifications can be associated with transforms. Again, a macro-device is specified by giving a set of terminals to be considered inputs and a set of output terminals. Just as for a constraint rule, the system creates a network containing the macro-device's computation path. But since transform rules can add devices to the datum network, a transform's macro-device specification , in addition to instructions on where to find the macro-device, has instructions telling where to add extra copies of it. The specification is written:

```
(DEFINE-TRANSFORM-MACRO transform-name
  (name-of-macro-device (<input-terminal>) (<output-terminal>))
  <(<instantiation-network-node-names>)>)
```

The instantiation instructions are simply lists of instantiation-network nodes. A regrettable punning is always in effect between the (atomic) terminal names of a transform, the pattern network terminals, the instantiation network terminal names, and the names of the nodes corresponding to those terminals in the respective networks.

Here's an example of a macro-device specification for a transform named SINGLE-REC-GEN that transforms singly recursive "programs" to "programs" using iteration on two variables. Refering to diagram "SINGLE-REC-GEN" may help clarify what is going on, but for the moment don't worry about what the transform "really does."

```
(define-transform single-rec-gen (input tst output base-value depth-limit
                                   recurse-on recurse-return nbtm base-return)
                    ((recrl depth-limit tst input output recurse-on recurse-return nbtm base-return)
                     (eq? input base-value tst))
                    ((do2 base-value do-variable2 depth-limit tst noutl nout2 output nin2init)
                     (eq? base-value input tst)))

(define-transform-macro single-rec-gen
                        (pop (recurse-on output) (recurse-return))  (base-value nout2 do-variable2))
```

The name of the macro-device (mostly ignored by the system) is POP. It takes two inputs (specified as the nodes attached to transform terminals RECURSE-ON and OUTPUT) and produces a single output for the node attached to the transform terminal RECURSE-RETURN. Altogether POP has three terminals.

The remaining macro-device specifications for SINGLE-REC-GEN are shown below:

```
(define-transform-macro single-rec-gen
                        (bumpinv (input) (recurse-on)) ( noutl  base-value))
;;nbtm is the first value "appearing" on input satisfying the test. In this
;;example, it will always have the value of base-value.
(define-transform-macro single-rec-gen
                        (s (nbtm) (base-return)) (base-value nin2init))
```

The system's action on a transform rule's macro device specification is to package the computation path it discovers in the datum network from the input datum nodes (specified by

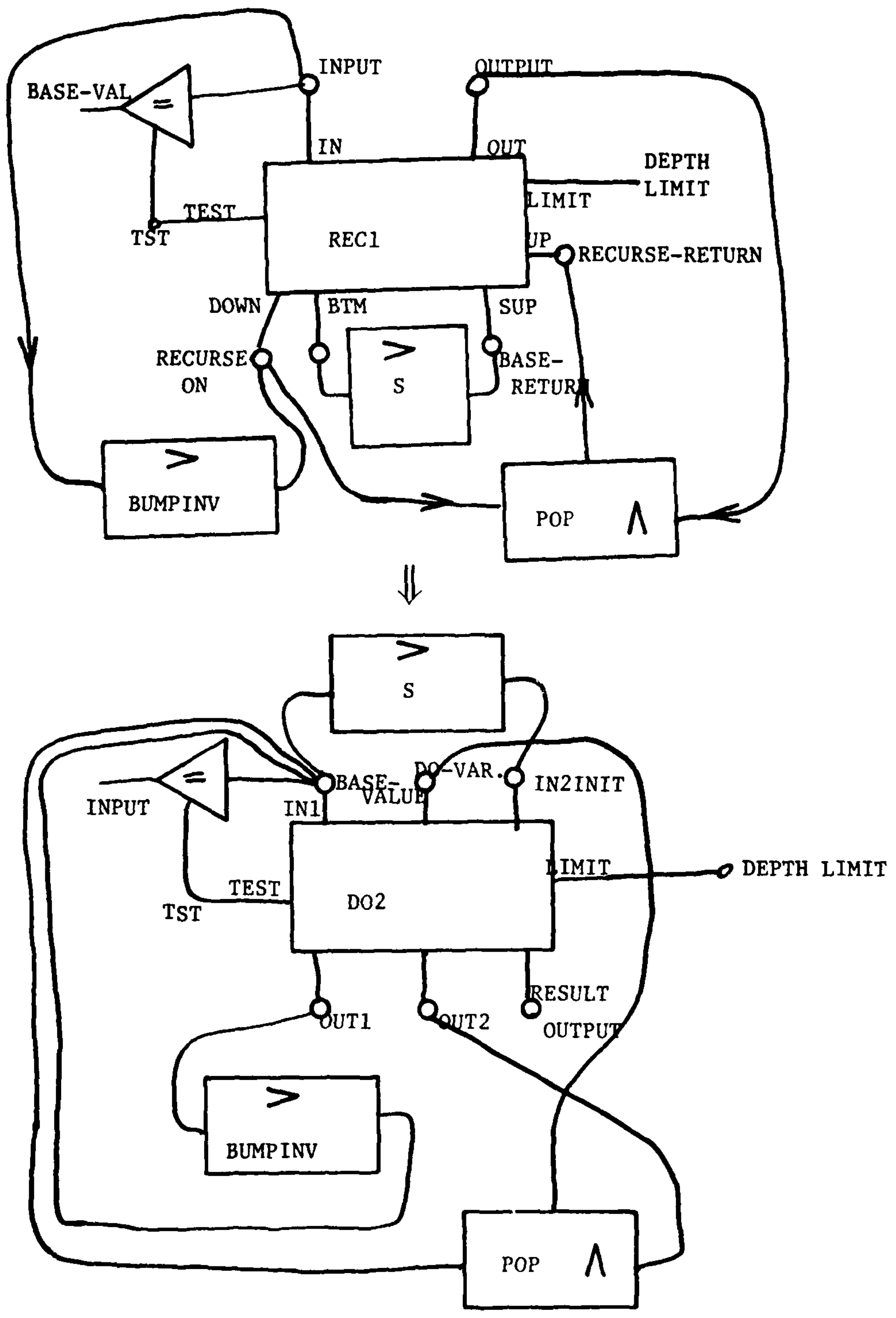


DIAGRAM "SINGLE-REC-GEN"

the transform terminals) to the output datum nodes into a new network. This network serves as the definition of a newly created complex device type: the type of the macro device found. This *find macro device* algorithm will be explained in full detail in the next chapter.

If everything goes well the instantiation network is eventually *expanded* so that the datum network will have a new DO2 device (a primitive for iteration on two variables), a commuted EQ? device, and some additional complex devices defined by networks created when macro-devices were found. One of these results from the POP macro-device specification. Recall that this device has three terminals. Originally these terminals could have been connected to nodes •RECURSE-ON, •OUTPUT, and •RECURSE-RETURN:

(POP •RECURSE-ON •OUTPUT •RECURSE-RETURN)

where POP is the type of a complex device found to satisfy the macro-device specification (in the system this name will always be a *gensym*). The instantiation instructions say that a new device of the type of the POP macro-device should be created and attached to the newly expanded datum network as follows:

(POP •BASE-VALUE •NOUT2 •DO-VARIABLE2)

where •BASE-VALUE is the node in the datum network corresponding to the instantiation network's node BASE-VALUE. Similarly, •NOUT2 and •DO-VARIABLE2 are nodes in the datum network that were created to correspond to the instantiation network's nodes NOUT2 and DO-VARIABLE2, respectively.

<u>*Solution to the "Factorial" problem*</u>

Before continuing with the sources of knowledge, let's look at a short example showing how recursive definitions can be transformed into iterative procedures.

What does the SINGLE-REC-GEN transformation above actually do? For non-tail-recursive functions there are two well known ways to convert recursion to iteration. The view to take is that going "down," elements of some set are being generated, coming "up" they are being accumulated. If the generation order can be reversed, then the function only needs to come "up", and this can be done via an iteration involving two variables: one for the current element; the other for accumulation. This is the transformation performed by SINGLE-REC-GEN. The other way to convert recursion to iteration is to reverse the order of accumulation. Some optimizing compilers have special cases of both of these transformations built into them.

The RECR1 device is a primitive for singly recursive control structures. It is defined as:

```
(make-device-type recrl limit test in out down up btm sup)
(rule-of recrl out (in limit)
        (*recursive (((invar in))
                     limit
                     ((*macro-device stest test) (*do-variable invar))
                     ((*macro-device start sup) (*do-variable invar))
                     ((*macro-device pop up)
                      (*do-variable invar)
                      (*do-variable out))
                     (out ((*macro-device bump in) (*do-variable invar)))))
        ;;continued: here are macro-device specs
        ((stest (in) (test))
          (bump (down ) (in))
          (start (btm) ( sup))
          (pop (down out) (up))) )
```

The DO2 device is a primitive for doubly-indexed iteration:

```
(make-device-type do2 in1 in2 limit test out1 out2 result in2init)
(rule-of do2 result (in1 in2init limit)
         (*do (limit
               ((*macro-device tester test) (*do-variable lv1) (*do-variable lv2))
               (*do-variable lv2)
               (lv1 in1 ((*macro-device bump in1) (*do-variable lv1) (*do-variable lv2)))
               (lv2 in2init ((*macro-device bump in2) (*do-variable lv1)
                                                      (*do-variable lv2)))))
         ((tester (in1 in2) (test))
          (bump (out1 out2) (in1 in2))  ))
```

Any relation between the two iteration variables (network structure between nodes on terminals IN1 and IN2) is a loop invariant. The loop invariants may not (in this example *are* not) be strong enough to constrain IN2's initial value. For this reason the DO2 device has a separate terminal for initializing the second iteration variable.

A recursive definition of the familiar FACTORIAL function is expressed by the network FPROB (see diagram "Single-rec-application"):

```
(define-network fprob1 (n fact)
                ((eq? n ((*constant 0.0) c0node) nt) d1)
                ((+c n ((*constant 1.0) c1node1) nd) d2)
                ((+c nb ((*constant 1.0) c1node2) nsv) d3)
                ((*c nd fact nv) d4)
                ((recr1 n nt n fact nd nv nb nsv) d5))
```

Asked to write code for computing FACT from N, the system comes up with:

```
(DEFUN FACTORIAL (N) (INTERNAL N))

(DEFUN INTERNAL (IN)
      (PROG (OUT)
            (COND ((= IN 0.0)
                   (RETURN (PLUS IN 1.0)))
                  (T (SETQ OUT
                          (INTERNAL (DIFFERENCE IN 1.0)))
                     (RETURN (TIMES IN OUT)))))).
```

In this example, the system's gensymed atoms have been replaced with italic mnemonic names. This code works, of course (if it weren't true, it wouldn't be published!) but it can be improved. If it could be converted to an iterative procedure then it might run a little faster because the code could avoid using the (Lisp) stack.

Tail-recursive programs have the property that the input arguments are not used after the program's single recursive call. The tail-recursive to iterative transformation algorithm should be part of every compiler's "bag of tricks." The factorial program above is not tail-recursive, so if one wants to convert it to an iterative form, one needs a more complex transformation. The system's response to a request to improve the factorial function:

(IMPROVE-FUNCTION 'FACTORIAL)

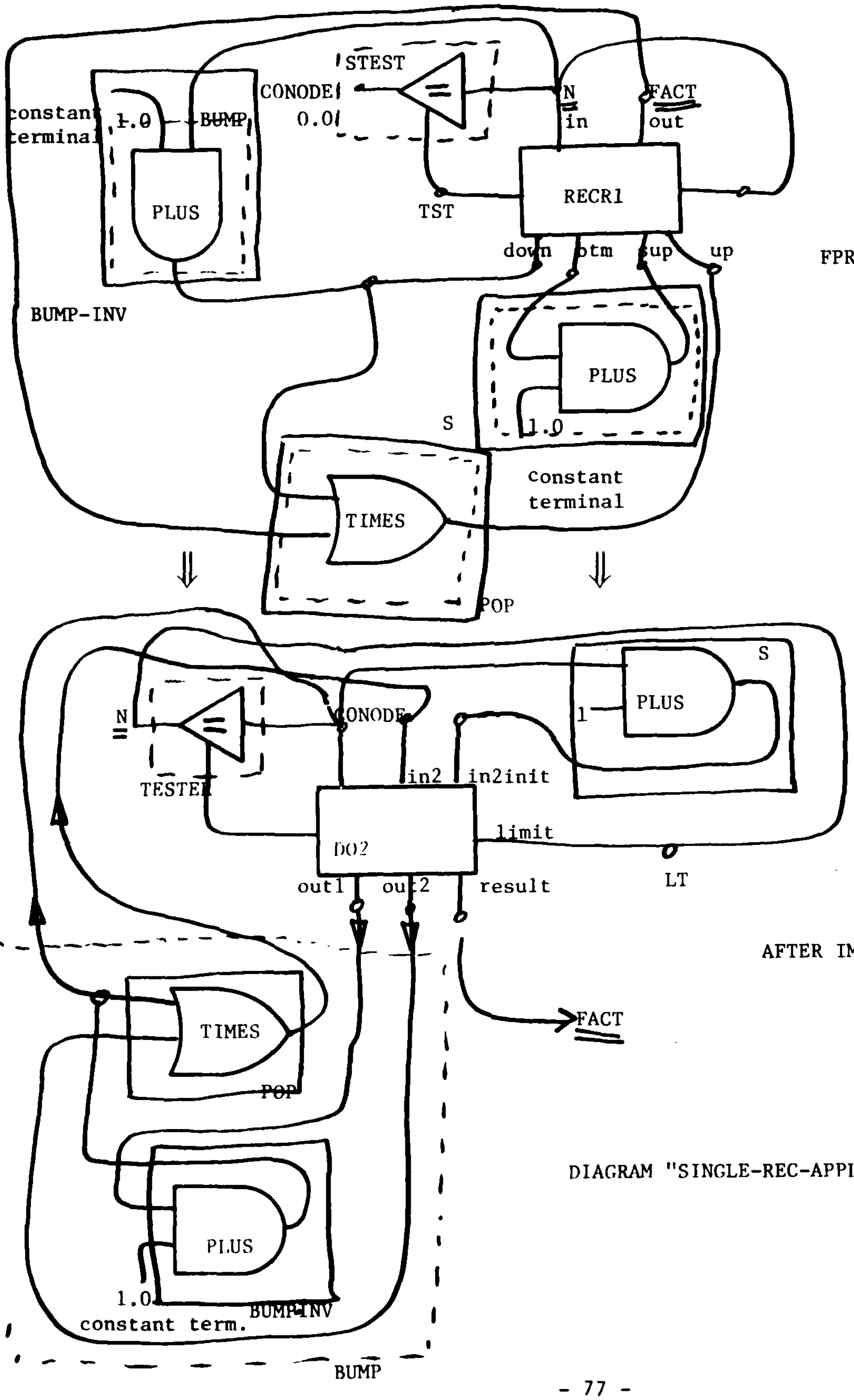
is to commute all the +C and *C devices. The system's library also contains other transformations that apply, but they are all rejected in favor of using the SINGLE-REC-GEN transformation. The diagram "SINGLE-REC-application" shows what happens when this tranform is applied to the network FPROB. Solid boxes with other devices inside are complex devices. They result from the transform's macro-device specifications, and in the code sometimes show up as subroutine calls (actually function applications). Dashed boxes are macro-devices found by constraint rule macro-device specifications. They also show up in the code as system-written functions.

The improved code (the system looks at the time cost to see if improvement takes place) is shown below. It is slightly faster, and much less readable, than the recursive FACTORIAL.

```
(DEFUN FACTORIAL (N)
       (PROG (TEMP)
             (RETURN
              (DO ((LV1 0.0
                        (PROG (NEW-LV1)
                              (SETQ NEW-LV1 (PLUS LV1 1.0))
                              (SETQ TEMP (TIMES LV2 NEW-LV1))
                              (RETURN NEW-LV1)))
                   (LV2 (PLUS 1.0 0.0) TEMP))
                  ((= LV1 N) LV2)))))
```

Discussion of the multi-return processing (responsible for *TEMP* directly, and *NEW-LV1* indirectly) will be postponed. The system failed to perform "constant folding" because the addition was hidden inside a system-created macro-device. In fact, normally the system *will* perform "constant folding" as a post-processing step.

The diagram "SINGLE-REC-application" also illustrates finding and using so-called *constant terminals* A macro-device specification (both constraint rule and transformation rule specifications) says certain nodes (refered to by terminal names) are to be inputs. The system may decide that in addition to those nodes, other nodes are also required and (hard to



FPROB

AFTER IMPROVEMENT

DIAGRAM "SINGLE-REC-APPLICATION"

determine) safe to use. These are added to the macro-device's definition (a network) as constant terminals -- so called because they are guaranteed to be "constant" with respect to the specified input/output computation path. The *free test* is used to determine whether this is true. It is discussed further in the next section.

If one of the outputs of a macro-device is itself *free*, then with regard to that output the macro is a constant macro-device. The synthesis system knows about these sorts of things. This means that in the original FPROB network, the device named D3 could be eliminated and the node BASE-RETURN could be initialized as a constant 1.0 and everything would work more-or-less the same (this time the constant would be folded in because the addition would never be refered to). In initially writing code, the RECRI's START macro-device would be a constant macro-device.

The Free-With-Respect-To Test

Problem: In order to show Newton's method converges, I need to take symbolic derivatives. I could write a transformation rule for Derivative[G(x) + c] = Derivative[G(x)] provided I could tell the matcher to make sure "c" does not depend on "x."
Solution: I have an algorithm (chapter 3) for determining answers to questions like that. The matcher uses "free-with-respect-to" specifications to invoke this algorithm.

In the previous section it was mentioned that with respect to a given computation path the system may need to decide whether a certain node is free or not. Examples are pointed out in diagram "SINGLE-REC-application." The idea behind the free test is that if some nodes are inputs, and other nodes are needed as outputs, then sometimes the computation cannot take place unless certain (free) nodes' values are also used, and these nodes cannot be determined on the basis of only the input nodes.

The system uses a conservative algorithm to determine if a node is free with respect to a specified set of inputs and outputs. This algorithm, which can err by claiming a node is *not* free when it really is, will be discussed in the next chapter. For now, suppose that such an algorithm exists.

If F(x) = Ax then the derivative of "F(x)" with respect to "x" is the constant "A." The condition one wishes to impose on this "transform" is that A be free with respect to the computation path from "x" to "F(x)"

Similarly, if

$$F(Y) = G(y) - NFY$$

and if G(y) depends on Y but NFY does not, then the derivative of F(y) with respect to "y" is simply the derivative of G(y) with respect to "y". Again, one wants to insure that NFY is free with respect to the computation path from "y" to F(y). The synthesis system's transform and additional information to express this fact is:

```
;;The constant is in node NFY.
(define-transform d-minus-constant (x fx y fy nfy rfy er)
                  ((deriv x fx y rfy er) (+c rfy nfy fy))
                  ((deriv x fx y fy er)))
(define-free-wrt d-minus-constant nfy (y) (rfy))
```

The DERIV device computes its second terminal by evaluating the derivative of the function computing its fourth argument from the third at its first argument (to within some error bound supplied by its fifth argument). The syntax of the "free with respect to" declaration is:

```
(DEFINE-FREE-WRT transform-name terminal-to-be-checked
        (<input-terminal>) (<output-terminal>))
```

where terminals are for the transform.

VARIABLE-NODES

Transforms are essentially a kind of pattern/action rule. The additional specifications concerning finding macro-devices and test for Free-WRT only refer to nodes already matched. Furthermore, a match sequence can be written (see chapter 3) so that at every point in the matching process all nodes already matched can be reached from a device matched earlier. This section introduces other mechanisms for matching nodes. The synthesis system has two kinds of what might be called "variable node" matching processes. Both have the property that if a match can be found for the "variable node," then that match is unique. This avoids all potential problems with combinatorial blowup involving "trying all nodes."

K-variable node specifications

Problem: If I can write code for $G^{-1}(x)$, then I can use a bisection search to write code for $G(x)$. Suppose I say $H(x)^2 = x^3$. Neither the code for $H(x)$ nor for $H^{-1}(x)$ are self-evident, but the problem of writing $H(x)$ is the same (almost) as writing square-root.

Solution: I have an algorithm that can discover that $G^{-1}(H(x)) = H(x)^2$. The matcher uses K-variable node specifications to invoke this algorithm.

In the course of solving the SQRT problem the system uses a transformation rule

SEARCH-INVERSE stating approximately

> "if you want to find a value FND between two known values U and L to within some specified error bound E and you happen to know a way of computing some known value OT on the basis of FND, then a new function can be constructed that has FND for a zero. Furthermore, a bisection search can be used on this new function to find FND."

The rule goes on to specify how this new function can be constructed. See diagrams "Search-inverse" and "SQRT ready for Search-inverse."

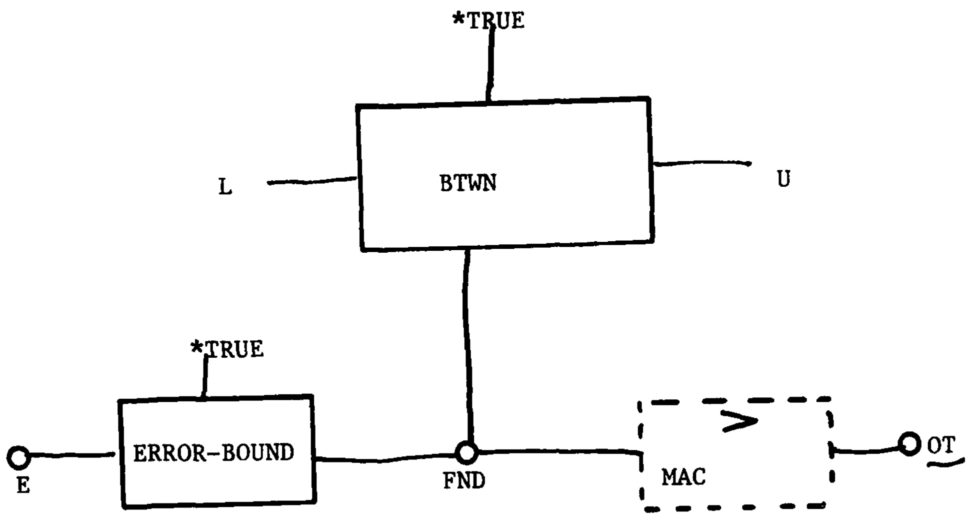
The difficulty in stating this transform concerns getting one's hands on the node the transform wants to call "OT." The pattern network, macro-device specification, and instantiation network are defined as shown:

```
;;The meaning of (BTWN  a b c pred) is that pred is true iff a < b < c.
;;The meaning of (ERROR-BOUND x e pred) is
;; that x has an absolute error less than E if pred is true.

(define-transform search-inverse  (u l ot fnd e)
                  ;;Pattern network:
                  ((btwn l fnd u (*mode-constant *true))
                   (error-bound fnd e (*mode-constant *true)))
                  ;;Instantiation network:
                  ((bsfz e u l fnd fxnode xnode)
                   (+c fxnode ot xint)))
(define-transform-macro search-inverse (mac (fnd ) (ot))  (xnode xint))
```

The macro-device could be found if the system knew which node OT were, but since it isn't mentioned in the pattern network some further specification will be needed.

The situation is as follows:

> MAC's input node (FND) has been found (e.g., matched) but it doesn't have a value yet. MAC's output OT node has not been found, but the intent is that it should have a known value



OT is a K-variable

⇓

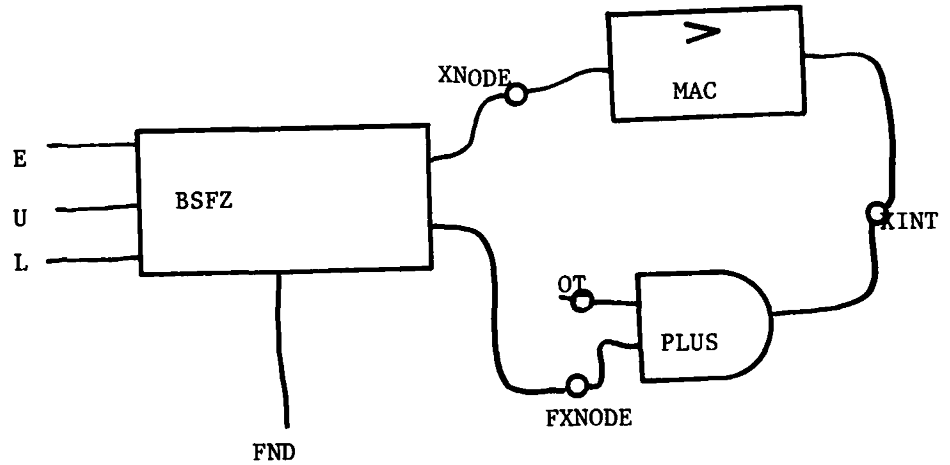


DIAGRAM "SEARCH-INVERSE"

In this kind of situation OT is a K-variable node (the "K" is for "Known", to be contrasted with a U-variable ("U" for "Unknown") below). These variables are specified by:

```
        (DEFINE-K-VARIABLE-NODE transform-name
              (k-variable-terminal (<assume-known-terminal>)))
```

The "assume-known-terminals" must all be unknown (meaning that the nodes connected must not have NVALUE facets) for the specification to be met. The K-variable specification for the SEARCH-INVERSE transform is shown below:

```
(define-k-variable-node search-inverse (ot (fnd)))
```

The result of applying transform SEARCH-INVERSE is shown in diagram "BSFZ SQRT Solution." This, by the way, completes the problem of writing the initial version of the code for the SQRT problem.

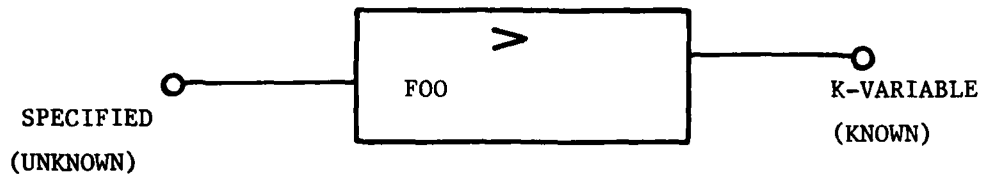
U-variable node specifications

Problem: In chapter 4, during the solution of the Bernoulli problem, a situation arises in which a very powerful solution technique can be used. This technique essentially solves a polynomial reversion problem -- but it needs to know what to solve the problem for.

Solution: None of the previous specifications really works, but there is an algorithm (chapter 3) that can find the right node to solve for. The U-variable specification is used by the matcher to invoke this algorithm.

The previous section told about one kind of "variable" node in a transform. This section is about another kind of variable: the U-variable. Glancing at diagram "variable node comparison" shows that in a sense U-variables and K-variables are symetric cases.

The idea is to find a potential computation path from some "inputs" to some "outputs"



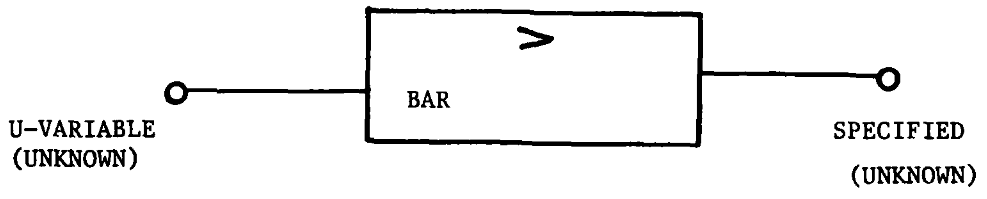


DIAGRAM "VARIABLE NODE COMPARISON"

that is being blocked because some node is unknown. The blocking node is the U-variable. An example of how the need for such a pattern-matching capability could arise, and how the U-variable meets this need can be found in the diagram "Transform TPS-MULT-U" in chapter 4.

The specification of a U-variable has the form

```
    (DEFINE-U-VARIABLE-NODE transform-name
            (U-variable-terminal (<input-terminal>) (<output-terminal>)))
```

The nodes matched by the transform's terminals (actually nodes corresponding to the specified terminals) in the output terminal list must be currently unknown. If a U-variable node is found, it will also be currently unknown and not one of the inputs or outputs.

The U-variable node may in fact depend (only) on the input terminals, but at the time the transform is being applied the nature of this dependency will not be known. It will be the case that the outputs can be computed if the inputs *and* the U-variable node are known, but *not* if just the inputs are known.

DOUBLETS

All sources of knowledge have now been presented. This section explains a transformation applied to transformation rules.

Suppose that one knows

$$X = A + B \ ; \ Y = A + B.$$

What enables one to deduce that $X = Y$? Furthermore, is it just that the algebraic expressions for X and Y are the same, or is the relationship more intimate?

To answer this question without bias, the problem must be stated without an implicit computation direction. In terms of constraints, suppose for an arbitrary constraint FOO:

(FOO X A B) ; (FOO Y A B).

Under what circumstances can one conclude that X and Y are *identically* the same node?

The two FOO devices mentioned above form a peculiar configuration. If one were to call the constraint "+C" rather than "FOO", then one could record the following observations

1. X can be uniquely computed on the basis of A and B.

2. Y can be uniquely computed on the basis of A and B.

3. The relation X has to A and B is the same as the relation Y has to A and B.

From the three facts above (known to be true for +C), one can conclude that X and Y are identically the same node. In the event that A or B (or both) are not in the number system, then the number system can be extended (for the purposes here) so that facts 1 and 2 remain true and the values do exist (a +NOT-EXIST form could be used to form such an extension). Then clearly X and Y are identical. Therefore one might as well assume they are identical in the unextended system (though they might be vacuously identical in that neither can receive a numeric value).

The above is a special case of the *doublet theorem*. This theorem is invoked by the system whenever a device terminal is attached to a node. Stated in terms of device rules:

> *Doublet Theorem:* If a pair of devices of the same type are found for which a rule exists whose terminal-needed list agrees (i.e., each terminal in the needed-list goes to the same node in each device), then the terminal-computed nodes (the nodes at the respective terminals) are identical and can be merged.

There are a few additional conditions that need to be imposed. If the devices in question use macro-devices, then these macro-devices need to be computationally equivalent. Currently the system avoids the question of computational equivalence by simply not looking at device rules using macro-devices. Some device rules are "special case" rules. These are also detected and ignored when applying the doublet theorem.

Devices can be merged if they argee as to type, and all terminals are identical.

For an example of the way the doublet theorem gets used, consider the network for the pair of non-linear equations (see diagram "Doublet Example"):

$$(R1 * R2) / (R1 + R2) = A$$

$$R2 / (R1 + R2) = B$$

This network would initially contain four devices:

```
((+c r1 r2 pl) d1)
((*c r1 r2 pr) d2)
((*c pl b r2) d3)
((*c pl a pr) d4))
```

To solve this pair of non-linear equations for R1 given values for A and B, the system first commutes the four devices D1 through D4. It then rather blindly tries to commute the commuted devices, but each time the new device is merged with one of the originals. So after

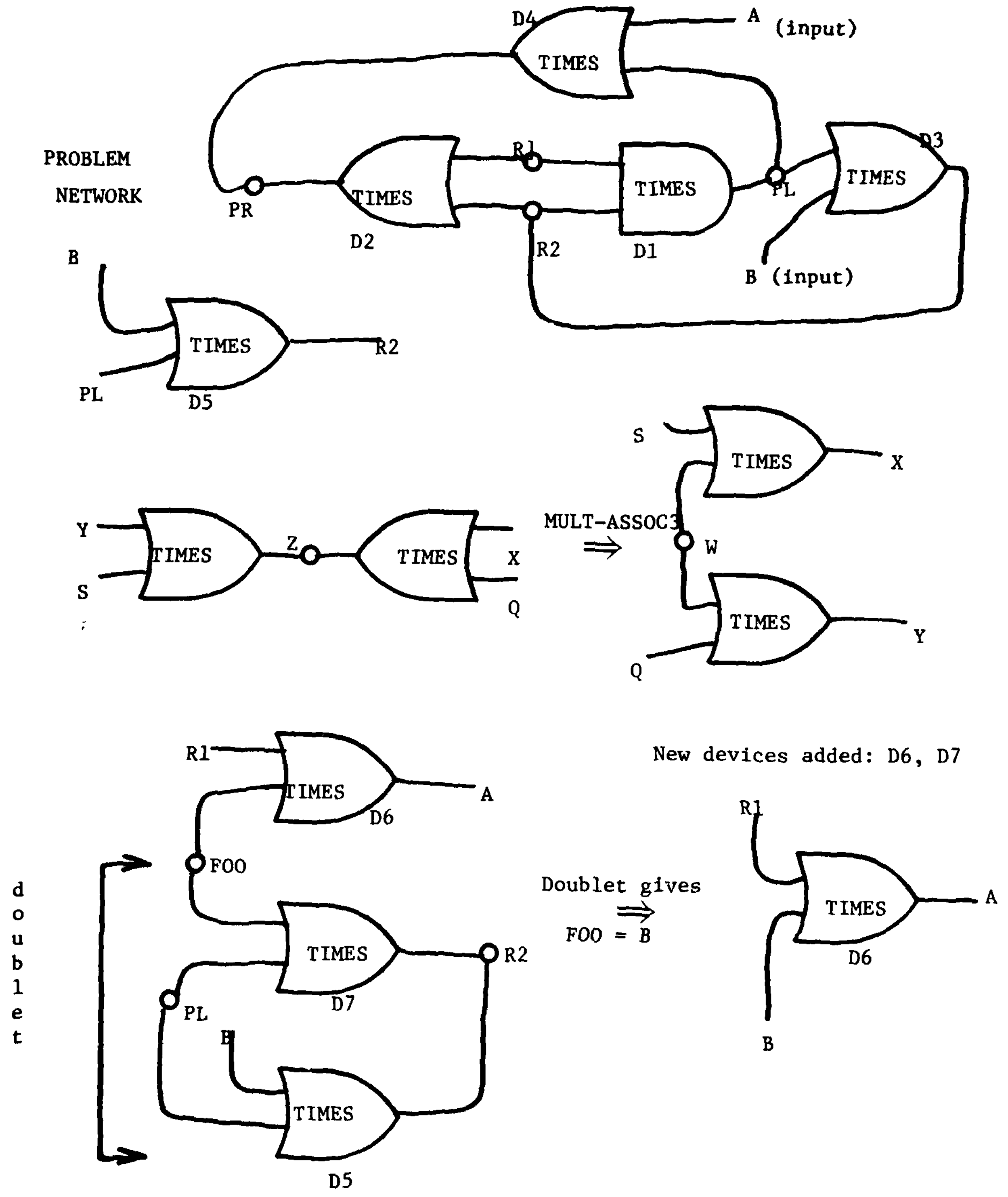


DIAGRAM "DOUBLET EXAMPLE"

the initial spat of commutations, no new ones really take place. One of the commuted devices comes from device D3:

((*c b pl r2) d5).

At this point, the system finds 30. new things to do. One of them is to apply the transform MULT-ASSOC3:

```
(define-transform mult-assoc3 (s y q x)
                  ((*c s y z) (*c q x z))
                  ((*c s w x) (*c w q y)))
```

with the correspondence

X->A; Q-> Pl; Y->R2;S->Rl.

The transform adds two new devices to the network:

((*c rl foo a) d6) ((*c foo pl r2) d7)

Now notice that devices D7 and D5 form a doublet. Therefore nodes FOO and B are identical. This means that device D6 is really

((*c rl b a) d6)

and simple propagation will give node Rl the symbolic expression A/B. Devices D5 and D7 are duplicates. Duplicates are automatically detected and merged. The code written by the system is shown below:

```
(DEFUN SYS1 (A B)
       (COND ((= B 0.0)
              (COND ((= A 0.0) (ERROR))
                    (T (ERROR))))
             (T (QUOTIENT A B)))).
```

It is amusing that the system distinguishes between the error of trying to compute 0/0 (which could be any value, after all), and division by 0.

Doublets and Variations of Transforms

The previous section had an example involving the transform named MULT-ASSOC3 Where did this come from? This transform is just the familiar multiplicative axiom of associativity:

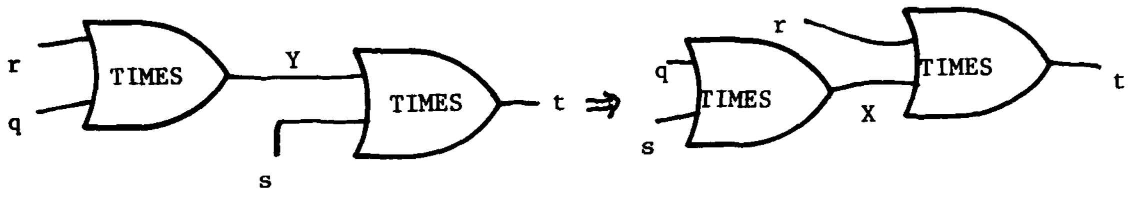
Associativity: ((a*b)*c) = (a*(b*c))

Assoc3: ((s*y)/x) = (y/(x/s))

There is an automatic way to obtain all such variants of an axiom. The operation (an operation on transforms) involves imagining what would happen if a doublet were added to the datum network, and then the transform were applied, and finally after the transform were applied a newly created doublet were removed. Although this sounds like something that should be part of the matching process, it really is a pre-processor operation on transforms. Diagram "Derivation using Doublets" shows the steps involved.

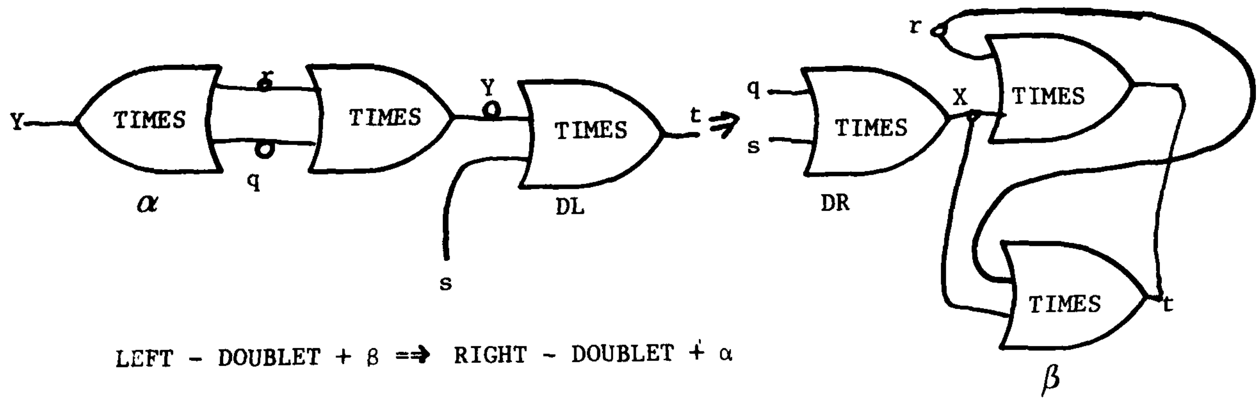
Although this process could be automated, the current system expects it to be done by hand. All rules have been so permuted.

Solution to SQRT Problem

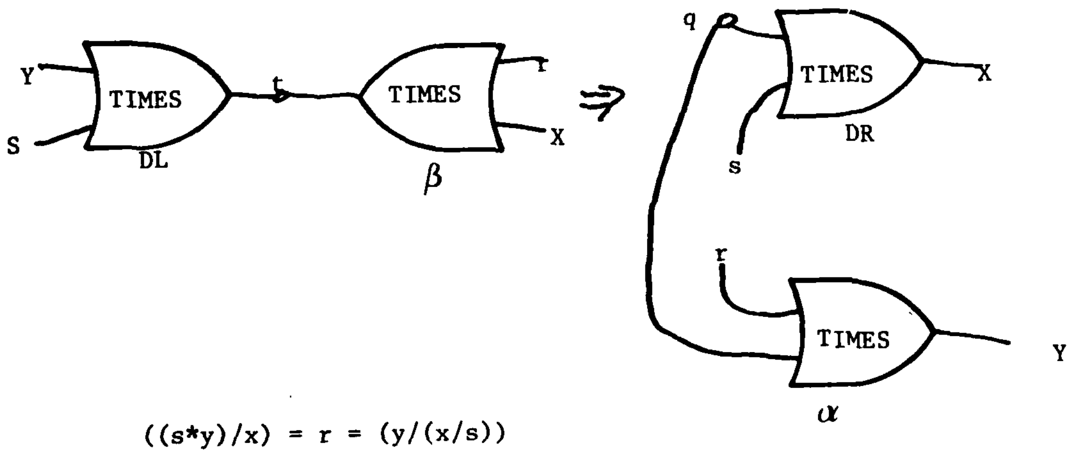


((r*q)*s) = t = (r*(q*s))

ADD α TO LEFT PART, AND ADD α TO RIGHT



LEFT - DOUBLET + β ⇛ RIGHT - DOUBLET + α



((s*y)/x) = r = (y/(x/s))

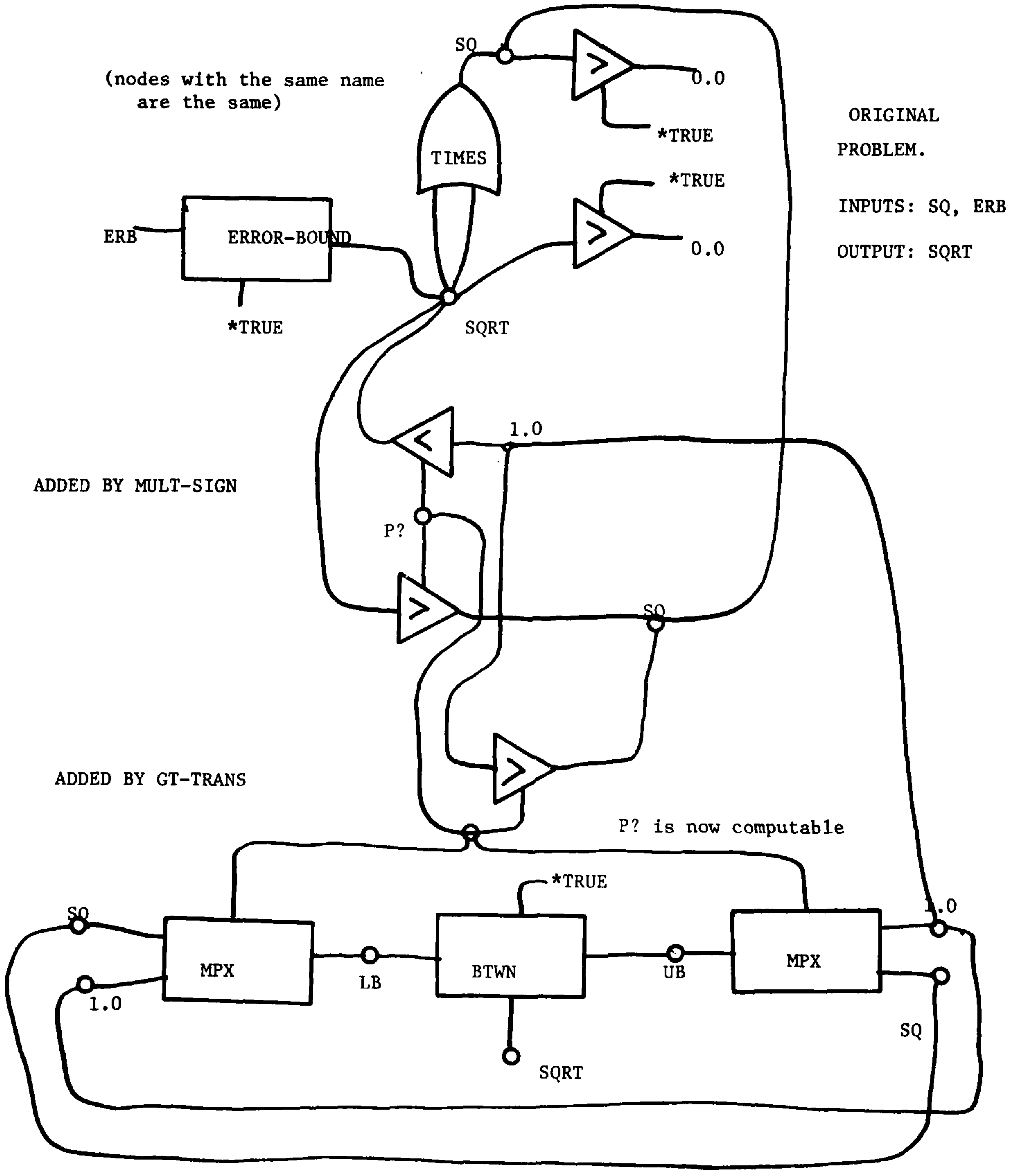
DIAGRAM "DERIVATION USING DOUBLETS"

This section ties together the sources of knowledge presented above.

In the preceeding sections the original problem network and several required transforms have been given for the problem of finding square roots. Diagram "SQRT ready for SEARCH-INVERSE" shows the complete network after transforms MULT-SIGN, GT-TRANS, and BTWN-DEDUCE have been applied in that sequence. The later two transforms are shown in table "Order Transformations." The bisection solution makes use of a MPX (for "multiplex") device. This is used to select one of two values depending on whether a predicate is true or false. Its definition is included in table "Order Transformations."

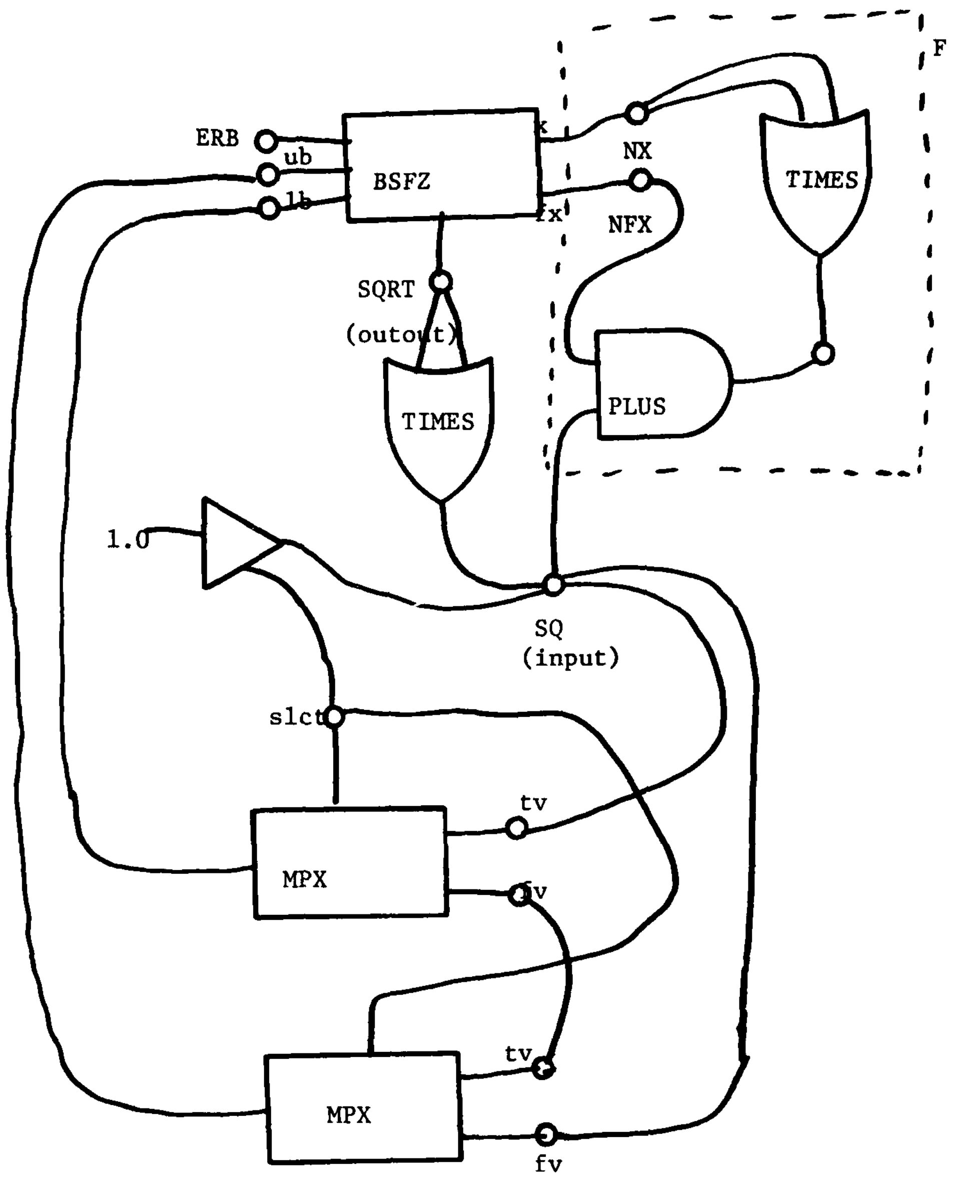
The bisection code is obtained by propagation from the portion of the solution network shown in diagram "BSFZ SQRT solution." The code generated is shown below

```
(DEFUN SQRT-ERB (SQ ERB)
  (PROG (DIR)
   (SETQ DIR (LESSP SQ 1.0))
   (RETURN (DO ((L (QUOTIENT (PLUS (COND (DIR 1.0)
                                         (T SQ))
                                   (COND (DIR SQ)
                                         (T 1.0)))
                             2.0))
                (SGN (SIGNUM ((LAMBDA (TEMP) (DIFFERENCE (TIMES TEMP TEMP)
                                                          SQ))
                              (COND (DIR 1.0)
                                    (T SQ)))))
                (LUB (COND (DIR 1.0) (T SQ)))
                (LLB (COND (DIR SQ) (T 1.0))))
               ((LESSP (DIFFERENCE LUB LLB) ERB) L)
               (SETQ L (QUOTIENT (PLUS LUB LLB) 2.0)
                     SGN SGN
                     LUB (COND ((= (SIGNUM (DIFFERENCE (TIMES L L) SQ)) SGN) L) (T LUB))
                     LLB (COND ((= (SIGNUM (DIFFERENCE (TIMES L L) SQ)) SGN) LLB) (T L)))))))
```



SEARCH-INVERSE MATCHES WITH E → ERB, FND → SQRT, OT → SQ,
L → LB, U → UB

DIAGRAM "SQRT READY FOR SEARCH-INVERSE"



If 1 is greater than sq, then lb = sq, ub = 1.

DIAGRAM "BSFZ SQRT SOLUTION"

AD-A096 559 MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTE--ETC F/G 9/2
COHERENT BEHAVIOR FROM INCOHERENT KNOWLEDGE SOURCES IN THE AUTO--ETC(U)
JAN 81 R BROWN N00014-80-C-0505
UNCLASSIFIED AI-TR-610 NL

2 OF 3
AD A
096559

There are several ways the code above could be improved. Some of them, like getting rid of the repeated test to see which way to update the bounds, could easily be accomplished by rewriting the BSFZ rule. Others, like eliminating the tests in the initialization of L, would involve making BSFZ a complex device, and then distributing addition over multiplexing.

Newton's method -- the basic device

The system can be asked to try improving the SQRT-ERB function it wrote:

(IMPROVE-FUNCTION 'SQRT-ERB).

This will cause the system to eventually write code using Newton's method. Newton's method finds a zero of a function F(x), from successive approximations X(0), X(1), etc., as shown:

$$X(0) = (UB + LB)/2$$

$$X(n+1) = X(n) - F(X(n))/F'(X(n)).$$

The device with a "canned" Newton's method is NEWTON-FZ defined in table "Newton's Method." Note that its only device rule is a special case rule. The device will find a zero *only* if CONV? is known to be true.

Currently, CONV? must receive a symbolic constant *TRUE in order for NEWTON-FZ's rule to be used. This is a limitation, but it can be overcome. The idea is to write code with a computational convergence test and a branch to Newton's method or bisection depending on the outcome. The system has a partially installed facility for combining a special case with another expression so that a "run-time" check could be used for convergence, but that will not be needed. An even more interesting idea stems from the observation that for some problems Newton's method does not converge in the initial region. Then one might use bisection until a region is found in which Newton's method *does* converge. The problem of synthesizing code incorporating that scheme has not been investigated.

The number of iterations is claimed to be

$$\log_2 (\log_2 (\text{number of intervals of size E between bounds})).$$

It turns out that if Newton's method converges, then within its region of convergence there is a sub-region in which Newton's method converges quadratically, doubling the number of bits of precision per iteration.

The NEWTON-ZF device rule uses a macro-device with two outputs specified, so one would expect the resulting code to show evidence of the multi-return facility's operation.

Newton's Method -- proving convergence

There are a number of tests that can be used to tell if Newton's method converges. The system uses the one mentioned in chapter 1, embedded in the transformation named TRY-NEWTON shown in diagram "Transform TRY-NEWTON" and also in table "Newton's Method."

The AND-C device simply performs a logical conjunction. This transform uses two other devices that have not been seen before: DERIV and ZERO-FREE. These are defined in table "Operator-like Devices."

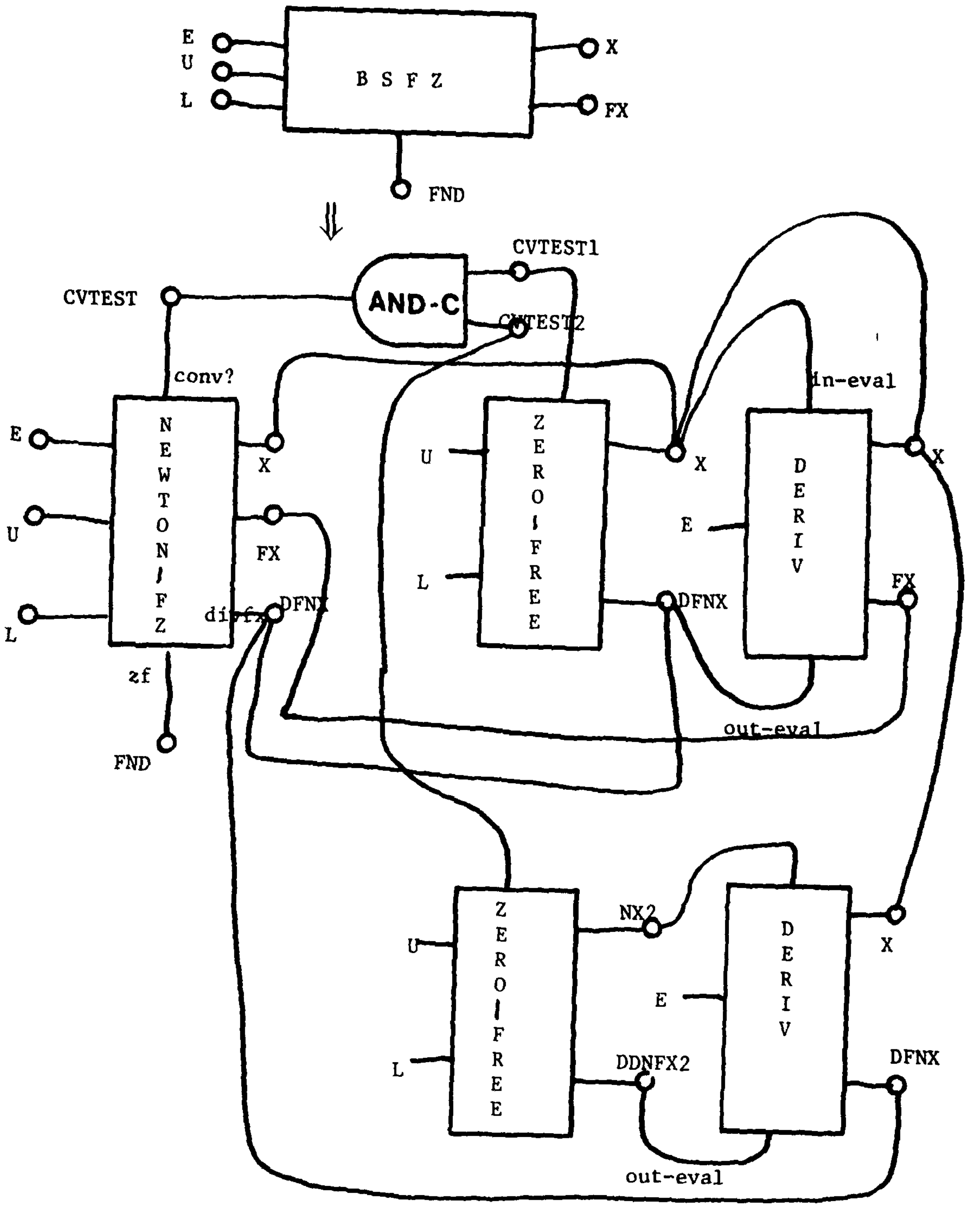


DIAGRAM "TRANSFORM TRY-NEWTON"

The rule for ZERO-FREE is rather interesting because it asks explicitly for the upper and/or lower bound of one of its terminals. The weak rules are used to move bounds around. As soon as those bounds can be propagated, the two convergence test predicates can be assigned values. The DERIV device is willing to do a simple (but not particularly accurate) numerical differentiation, but that will not be required for this problem.

Three transforms are used to perform symbolic differentiation in this problem. In order of application they are: D-MINUS-CONSTANT, D-SQUARE-SPECIAL, and D-CONST-MULT. They are given in table "Derivative Transforms." Note in particular the way Free-WRT specifications are used.

The Newton's method solution is produced after these transforms are applied. The path through which bounds information is propagated is shown in "Newton SQRT Solution." The code produced is shown below. The two loop variables are CRCT, the "correction term", and RSLT, the most recent approximation. ABS is a standard LISP function that takes the absolute value of its argument. Notice again the multi-return facility operation.

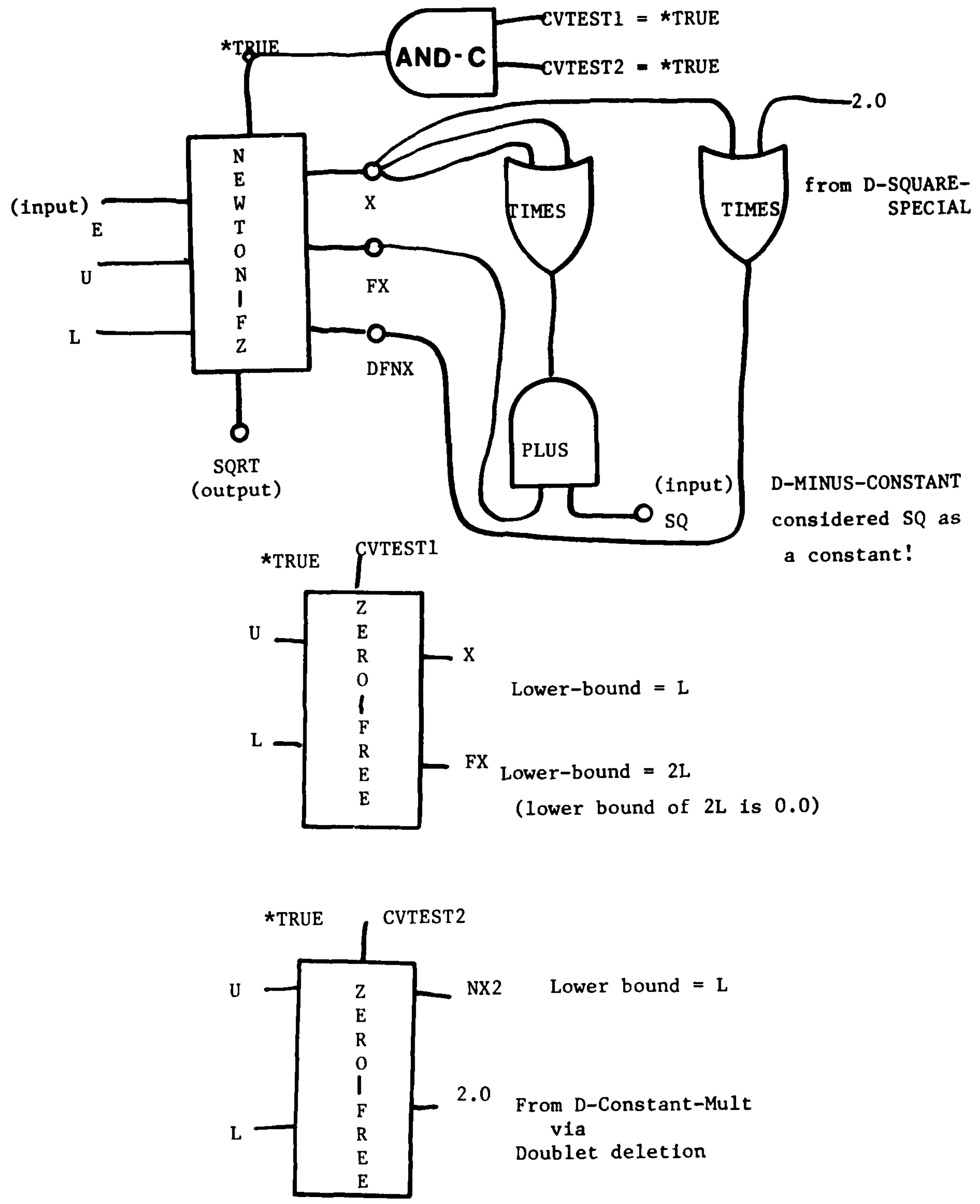


DIAGRAM "NEWTON SQRT SOLUTION"

```
(DEFUN SQRT-ERB (SQ ERB)
 (PROG (MR-TEMP DIR)
   (SETQ DIR (LESSP SQ 1.0))
   (RETURN (DO ((CRCT (DIFFERENCE (COND (DIR 1.0)
                                        (T SQ))
                                  (COND (DIR SQ)
                                        (T 1.0))))
                (RSLT (QUOTIENT (PLUS (COND (DIR SQ)
                                            (T 1.0))
                                      (COND (DIR 1.0)
                                            (T SQ)))
                                2.0)))
               ((LESSP (ABS CRCT) ERB) RSLT)
               (SETQ CRCT (QUOTIENT (PROG NIL
                                          (SETQ MR-TEMP (TIMES 2.0 RSLT))
                                          (RETURN (DIFFERENCE (TIMES RSLT RSLT)
                                                              SQ)))
                                    MR-TEMP)
                     RSLT (DIFFERENCE RSLT CRCT))))))
```

This code is fairly efficient. The multi-return temporary variable MR-TEMP didn't make very much difference because the two returns didn't use any intermediate results in common. Again, there is a need to distribute addition and subtraction over the multiplexing.

Summary

The principle structural elements in this report's theory of program synthesis have been described. This section brings them all together.

Nodes serve as a repository for various kinds of information (*facets*) about values. A node's *value structure* is an association between facets and expressions. The various facets and meanings of the associated expressions can be found in table "Facets". Two distinct nodes can be identical in any or all of their facets, and still represent distinct values.

Devices are the basic relational and computational element for the system. A device type is defined by a device name, and a list of the device's *terminal* names. In order for a device to be properly connected in a network, each device terminal must be attached to some node. Each device type may have constraint rules associated with it. There are two classes of constraint rule: *strong* constrant rules that can compute values, and *weak* constraint rules that cannot compute values but can still compute other facets (like bounds).

Constraint rules are written in a special *rule language* in which only one rule is needed to express a "computation." The meaning of this rule is changed from facet to facet by the interpreter. The structure of the interpreter is two-tiered. One tier is structural (expressions, conditionals, iteration, etc.) and the other is operational (operations like addition and multiplication). Each operation has a handler for each facet. While new structural expressions are hard to add, a new operation is easy: simply write a handler for each facet.

In addition to refering to values at terminals, constraint rules may also refer to computations between terminals via a *macro-device* facility. Besides allowing the description of iterative and recursive devices, the *macro-device* facility is used in describing devices for doing things like summation and bisection searches (see diagram "BSFZ").

A collection of devices and their connecting nodes forms a *network*. Networks have three major uses in the system. First, LISP programs are only written for networks. Second, *transformation rules* are written by saying that if one network (the *match pattern*) is present as a subnetwork (of the *datum network*), then another network (the *instantiation pattern*) may be added (by copying) to the datum network. The third use of networks is as an implementation of the notion of a *grey box*. Any device (usually considered a "black box" by the system) may have associated with it a *defining network* that reveals its internal structure.

Like a device, a network has *terminals*. Associated with each network terminal is a node inside the network. A network may have *encoded-functions* associated with it for computing one terminal (the output of the function) from others (the inputs). These are usually written by the system. Unlike device rules, networks have a *multi-return* facility; some encoded-functions set up registers to be used by other encoded-functions (the LISP machine multiple-value return is another way to accomplish the same thing). Code resulting from the multi-return facility's operation has an appearance that will annoy structured programmers.

Transforms are objects expressing *transformation rules*. Fundamentally, a transform claims that the existence of the pattern network as a subnetwork of the datum network implies that the instantiation network should also be a subnetwork of the datum network. The result of successfully applying a transform is to copy the instantiation network into the datum network.

Five kinds of suplemental information may be associated with a transform:

1. ADD-BREAK flags
2. U-variables
3. K-variables
4. Macro-devices (both pattern and instantiation)
5. "Independence" checks

Certain devices in the pattern network may be flagged by an add-break property saying "if this device isn't present, create these nodes (some with initial values) and add a device of the proper type." In addition to nodes in the pattern network, certain auxiliary nodes may also be matched with nodes in the datum. There are two kinds of these variable-nodes, with correspondingly different selection algorithms. *U-variables* are matched with nodes in the datum network whose values are currently unknown. *K-variables* are matched with nodes whose values are known. Finally, *macro-devices* can be found and copied as a result of applying a transform, and can be used in determining if some nodes are "independent" of others.

This completes the brief (but complete) list of the system's sources of knowledge. Currently the system needs to be told certain other things having to do with controlling transform application, but the next chapter will explain how this information could be automatically derived by a pre-processor.

TABLE 'FACETS'

Facet-name	type of exp.	meaning
NVALUE	symbolic:	description of the value
CODE-EXP	code-fragment:	If executed in proper environment this fragment would compute the numerical value for this node
TIME-COST	symbolic:	Time executing the code-fragment would take (including time-cost of arguments)
NODE-USED	special:	Nodes whose value was used in obtaining this node's value. Divided into definitely-used and possibly-used parts.
LBOUND	symbolic:	Lower bound for value
UBOUND	symbolic:	Upper bound for value
ERROR-BOUND	symbolic:	Maximum amount by which "true value may differ from "computed value."
TYPICAL-VALUE	numeric:	Example of a typical value this node might have. (not implemented)

These facets are relative to a particular set of inputs

TABLE 'CONSTRAINT-RULE CONSTRUCTIONS'

(*CONSTANT numeric-value) This is obvious, the TIME-COST interpretation is (*CONSTANT 0.0). The numeric value must be a floating-point number.

(*MODE-CONSTANT mode-value) Modes are things like *TRUE and *FALSE. The distinction between modes and numbers is used by several algorithms. The notion of a "mode" was suggested by EL's transistor model.

(*EXPRESSION expression) The expression is interpreted by refering to the handlers of the function names for each facet.

(*MODE-EXPRESSION expression) Just like *EXPRESSION except a mode value (either a constant or an expression) is returrned.

(*VARIABLE atom) This symbol is passed around as a value. The TIME-COST interpretation is (*CONSTANT 0.0). Generally, an atom in an argument position to a function will mean the device's terminal.

(*NOT-EXIST atom) This is something like a *VARIABLE except it is not a value. The atom is an indication of why the value doesn't exist. For example, division by zero will generate one of these symbolically. This is not used very often.

(*ANOMALOUS atom) Similar to a non-existant value. The difference is illustrated by considering what it would mean for a node to be assigned both an *EXPRESSION and a *NOT-EXIST construction (a contradiction!), as compared to being assigned both an *EXPRESSION and an *ANOMALOUS construction (not a contradiction).

(*CASES (mode-expression <(mode-symbol expression)>)) The interpretation of this is that the result of evaluating the mode-expression is a mode-symbol (like *TRUE or *FALSE). The value represented by a *CASES is then the value represented by the expression corresponding to this mode-symbol. Naturally if the symbol is not known because the mode-expression resulted in a *MODE-EXPRESSION construction, then a *CASES construction results. Some *CASES expressions are said to be *special-cases* because the expression for one of the listed mode-symbols is NIL.

((*MACRO-DEVICE device-name terminal-selected) <expression>) This construction is described in the text. The significance of the sequence of expressions is that they correspond to the sequence of device terminals declared as "inputs" to the macro-device. The terminal-selected above is one of the terminals declared as an output terminal in the macro-device specification.

(*EXTERNAL-NODE node) This should never occur in a rule. It is used by the system to process extra terminals in macro-devices.

(*DO (limit test result <(variable initial iteration)>)) where *limit* is an expression whose value is an upper bound on the number of times the variables will need to be updated before *test* becomes *TRUE. *Test* is a mode-expression. *Variables* are do-variables, and may be used in the expressions for *result* and *iteration*. The *initial* expressions may not refer to do-variables. The interpretation is quite similar to the LISP DO construction, except that there is no provision for anything "inside" the loop other than incrementing do-variables. All initializations are assumed to take place "in parallel," but iterations take place sequentially. By clever use of multi-return macro-devices, any combination of parallel/sequential iterative assignments can be obtained.

(*DO-VARIABLE atom) This can only occur inside a *DO or a *RECURSIVE construction, and refers to the most recent value assigned to the variable named. There is a problem if an expression like this is used in a TIME-COST facet. The NVALUE interpretation of this form is the same *DO-VARIABLE form. Time-cost interpretation is 0.0. Although not completely satisfactory, these forms are replaced by the "limit's" NVALUE interpretation whenever they occur in a time-cost expression (see chapter 4 for brief discussion).

(*RECURSIVE ((<(input-variable expression)>) limit test start-up upward <(variable <expression>)>)) This is the most complex construction.The idea is that if *test* is true, then *start-up* is returned. Otherwise recursive calls are used to set all the *variables* (simultaneously), and then *upward* is used to obtain the value to return. *Limit* is the limit to the depth of the recursion. Each entry *(variable <expression>)* is a recursive call to the code described by the *RECURSIVE construct. The number of these entries is the order of the recursion. *Input-variables* and *variables* are atoms, and are refered to within the *limit, test, start-up, upward,* and *expression* as (*DO-VARIABLE atom). The *expressions* for the input-variables are the values those input variables should have at the outermost call. The *start-up* and *expressions* cannot use the atoms in the *variable* slot.

TABLE "KNOWLEDGE SOURCES SUMMARY"

(MAKE-DEVICE name-of-device <name-of-terminal>)

(RULE-OF name-of-device terminal-computed
 (<terminal-needed>) expression [(<macro-device-specification>)]).
(WEAK-RULE-OF name-of-device terminal-computed
 (<terminal-needed>) expression).
See table "constraint rule construction" for how device rules are written.
Macro-devices are always specified by a name and a list of terminals:
 (macro-device-name (<input-terminal>) (<output-terminal>)).
Multi-directional capabilities are added with specifications like:
 ((original-name new-name) (<input-terminal>) (<output-terminal>)).

(DEFINE-NETWORK network-name (<terminal-names>) <device-specification>)

(DEFINE-TRANSFORM name (<common-terminals>) (<pattern-device-spec>)
 (<instantiation-device-spec>) [(<add-break-spec>)])
Add-break devices are specified by:
 (add-break-device-name <uninitialized-node-name>)

(DEFINE-TRANSFORM-MACRO transform-name
 (name-of-macro-device (<input-terminal>) (<output-terminal>))
 <(<instantiation-network-node-names>)>)

(DEFINE-FREE-WRT transform-name terminal-to-be-checked
 (<input-terminal>) (<output-terminal>))

(DEFINE-K-VARIABLE-NODE transform-name
 (k-variable-terminal (<assume-known-terminal>)))

(DEFINE-U-VARIABLE-NODE transform-name
 (U-variable-terminal (<input-terminal>) (<output-terminal>)))

Chapter 3 explains one more (derivable!) source of knowledge
See table "Transform Types" (chapter 3) for meaning of "type":
(DEFINE-ENABLEMENT transform-name type
 <((<outline-terminal>) (<input-terminal>) (<output-terminal>))>)

Table "Three Device Definitions"

```
(MAKE-DEVICE-TYPE *C A B C)
(RULE-OF *C C (A B) (PRIM* A B))
(RULE-OF *C A (B C)
         (*CASES ((PRIM= B (*CONSTANT 0.0))
                  (*TRUE (*CASES ((PRIM= C (*CONSTANT 0.0))
                                  (*TRUE (*ANOMALOUS Z-DIV-Z))
                                  (*FALSE (*NOT-EXIST DIV-BY-ZERO)))))
                  (*FALSE (PRIM-DIV C B)))))
(RULE-OF *C B (A C)
         (*CASES ((PRIM= A (*CONSTANT 0.0))
                  (*TRUE (*CASES ((PRIM= C (*CONSTANT 0.0))
                                  (*TRUE (*ANOMALOUS Z-DIV-Z))
                                  (*FALSE (*NOT-EXIST DIV-BY-ZERO)))))
                  (*FALSE (PRIM-DIV C A)))))

(make-device-type bsfz e ub lb zf fx x)
(rule-of bsfz zf (ub lb e)
         (*do ((bit-length (prim-div (prim- ub lb ) e))
               (prim-less  (prim- (*do-variable lub) (*do-variable llb))  e)
               (*do-variable L)
               (l (prim-div (prim+ ub lb) (*constant 2.0))
                  (prim-div (prim+ (*do-variable lub) (*do-variable llb))
                            (*constant 2.0)))
               (sgn (prim-sign ((*macro-device f fx) ub)) (*do-variable sgn))
               (lub ub (*cases ((prim=
                                  (prim-sign ((*macro-device f fx) (*do-variable l)))
                                  (*do-variable sgn))
                                 (*true (*do-variable l))
                                 (*false (*do-variable lub)))))
               (llb lb (*cases ((prim=
                                  (prim-sign ((*macro-device f fx) (*do-variable l)))
                                  (*do-variable sgn))
                                 (*true (*do-variable llb))
                                 (*false (*do-variable l)))))))
         ((f (x) (fx))) )
```

```
(make-device-type greater bg ltl p)
(rule-of greater p (bg ltl)
        (*mode-expression (prim-less ltl bg)))
(weak-rule-of greater bg (p ltl)
              (*cases (p
                        (*true (lower-bound-op ltl))
                        (*false nil))))
(weak-rule-of greater ltl (p bg)
              (*cases (p
                        (*true (upper-bound-op bg))
                        (*false nil))))
```

Syntax explained in table "constraint rule constructions."

Table "Order Transformations"

```
(define-transform gt-trans (a c gtest)
                 ((greater a b gtest)
                  (greater b c gtest))
                 ((greater a c gtest)))

(define-enablement gt-trans remove ( nil (a c) (gtest)))

(define-transform btwn-deduce (a b c dirtest)
                 ((greater a b dirtest)
                  (greater b c dirtest))
                 ((mpx a c dirtest upper)
                  (btwn lower b upper (*mode-constant *true))
                  (mpx c a dirtest lower)))

(define-enablement btwn-deduce range ( nil (a c) (b)))

;;HERE IS THE DEFINITION OF THE MPX DEVICE:

(make-device-type mpx tv fv slct rslt)

(rule-of mpx rslt (tv fv slct)
         (*cases (slct (*true tv)
                       (*false fv))))



(rule-of mpx tv (rslt slct)
         (*cases (slct (*true rslt)
                       (*false nil))))

(rule-of mpx fv (rslt slct)
         (*cases (slct (*true nil)
                       (*false rslt))))
```

TABLE "NEWTON'S METHOD"

```
(make-device-type newton-fz conv? e ub lb zf fx x divfx)

(rule-of newton-fz zf (conv? ub lb e)
         (*cases
          (conv?
           (*true
            (*do ((bit-length (bit-length (prim-div (prim- ub lb) e)))
                  (*mode-expression (prim-less (prim-abs (*do-variable crct)) e))
                        (*do-variable rslt)
                  (crct  (prim- ub lb)
                        (prim-div ((*macro-device f fx) (*do-variable rslt))
                                  ((*macro-device f divfx) (*do-variable rslt))))
                  (rslt (prim-div (prim+ lb ub) (*constant 2.0))
                         (prim- (*do-variable rslt) (*do-variable crct))))) )
           (*false nil)))
         ;;Macro-device
         ((f (x) (fx divfx))) )

(define-transform try-newton (e u l fnd x fx)
                  ((bsfz e u l fnd fx x))
                  ((newton-fz cvtest e u l fnd fx x dfnx)
                   (and-c cvtest1 cvtest2 cvtest)
                   (zero-free cvtest1 u l x dfnx)
                   (zero-free cvtest2 u l nx2 ddnfx2)
                   (deriv x dfnx x fx e)
                   (deriv nx2 ddnfx2 x dfnx e)))

(define-enablement try-newton speedup (nil (u l e) (fnd)))
```

TABLE "Operator-like Devices"

```
(make-device-type zero-free pred upper lower x-in fx-out)

(weak-rule-of zero-free x-in (lower)
              (lower-bound-op lower))

(weak-rule-of zero-free x-in (upper)
              (upper-bound-op upper))

(rule-of zero-free pred (fx-out)
         (*mode-expression
          (prim-or (prim-less (*constant 0.0) (lbound* fx-out))
                   (prim-less (ubound* fx-out) (*constant 0.0)))))

(make-device-type DERIV in-eval out-eval x-def fx-def abserr)

(rule-of deriv out-eval (in-eval abserr)
         (prim-div
          (prim-
           ((*macro-device fun fx-def) (prim+ in-eval abserr))
           ((*macro-device fun fx-def) (prim- in-eval abserr)))
          (prim* (*constant 2.0) abserr))
         ;;Macro definition
         ((fun (x-def) (fx-def))))
```

TABLE "Derivative Transforms"

```
;;RFY = F(y) - constant
;;The constant is in node NFY.

(define-transform d-minus-constant (x fx y fy nfy rfy er)
                  ((deriv x fx y rfy er) (+c rfy nfy fy))
                  ((deriv x fx y fy er)))

(define-free-wrt d-minus-constant nfy (y) (rfy))

(define-enablement d-minus-constant reduce (nil (x) (fx)) )

(define-transform d-square-special (x fx)
                  ((deriv x fx y ysq err) (*c y y ysq))
                  ((*c (*constant 2.0) x fx)))

(define-enablement d-square-special remove (nil nil nil))

;;The instantiation portion of this rule is unfortunate because what I really want to say
;;is simply "merge nodes FX and NFY". But I cannot (for no particularly good reason), so I
;;need to go around the bush a bit

(define-transform d-const-mult (x fx y fy nfy)
                  ((deriv x fx y fy err) (*c nfy y fy))
                  ((*c fx y fy)))

(define-free-wrt d-const-mult nfy (y) (fy))

(define-enablement d-const-mult remove (nil nil nil))
```

CHAPTER 3

COHERENT BEHAVIOR

In chapter 2 the system's sources of knowledge were presented and explained. The problem addressed by this chapter is to obtain coherent behavior from these isolated, local sources of knowledge.

All researchers have an intuitive notion of what "coherent behavior" means: if the problem solving system "does it like I might," then it is coherent. A more objective quantitative measure of coherence is the ratio of the time taken by the system to the observed difficulty of the problem. As in chapter 1, the observed difficulty of a problem is the number of rule applications that, in retrospect, were actually required. In what follows, terms like "polynomial behavior", "exponential behavior," etc., will always refer to this ratio.

The principle effort of this research has been to eliminate exponential behavior from rule-based problem solving, at least as applied to numerical program synthesis. For the most part, this effort has been successful, in that exponential behavior does not occur in any of the test cases. The remainder of this chapter explains how this has been accomplished.

The Top Level

The basic top level for both initially writing code and later improving it looks like

```
        TOP LEVEL
        (INIT) List of previous matches is empty
    LOOP
        (PROP) propagate all value structures
        (TEST) if outputs have values (improved values if
                improving code), then write code and quit
        (FIND) Find all new matches, and add any found to front
                of list of previous matches
        (SELECT) Search list of previous matches for best one to apply
        (APPLY) Apply it.  Go to LOOP
```

The three steps PROP, FIND, and APPLY can all be shown to be polynomial in the number of devices in the problem network (see individual sections following). Except for add-break devices, the problem network's size is only increased by the APPLY step. Since there is some maximal size of transformation rule instantiation network, the total number of devices added to the original problem network is bounded by a linear function of the number of rules applied. (One reason transformation rule macro-device specs create a new complex device rather than causing the computation path found to be copied is so that this observation will hold). Add-break devices unfortunately make a more precise argument difficult.

A lower bound on the system's time cost is provided by the FIND step. If N is the number of devices in the solution network (this is the original problem network with all the devices added by all the rules applied), and M is the maximum size of any transformation rule's pattern network, and there are K transformation rules, then step FIND will take (for each time around the loop) time bounded by (see below):

$$K*N^M$$

This is, of course, a polynomial in N, which is in turn a polynomial in the number of rules applied. This worst case never occurs in practice.

As one might have suspected, the critical problem is how to select, from a number of applicable rules, the correct rule to actually apply (or, rather, how to avoid making poor choices). The next section gives a solution to this problem. But first, a more detailed description of the top level is needed.

Matches and Psuedo-devices

The matching process will be completely described later in this chapter. But some facts about it are needed now.

Transforms are typed according to a scheme to be described later (see table "Transform Types"). Whether an effort is made to find a match for a transform's pattern is determined (in part) by this type. These types are ranked in a strict sequence. In general, the matching process doesn't go further in this sequence than necesssary.

A transform's pattern network is examined by a pre-processor. Starting with a "seed" device (manually selected now), a match sequence is created with the property that each device (except the "seed") in the pattern is reached from a node already matched.

This match sequence is then used to control the way an object called a *partial match* is propagated in the datum (problem) network. This propagation procedure may add devices to the network corresponding to those flagged as add-break devices in the pattern network. When all devices in the match sequence have been matched, then the partial match is said to be completed. At this point, macro-device specs, U-variable nodes specs, K-variable node specs, and "free-wrt" specs still need to be processed.

All completed partial-matches are further processed by being *instantiated*. This process creates a structure called a *psuedo-device* (used instead of real devices for efficiency) that is hooked into the problem network in a way similar to the way a device is connected. Like a device, a psuedo-device has terminals. These are the same as the transform's terminals, and are connected to the nodes matched by the appropriate pattern nodes. This instantiation of partial matches also makes sure all U-variable and K-variable specifications have been satisfied. After instantiation all transform terminals have been matched.

The FIND step of the top level involves both partial match propagation and intantiation of partial matches. The SELECT step actually examines a list of psuedo-devices, and only looks at the terminals of these psuedo-devices.

To apply a transformation rule, the psuedo-device created for the completed partial match corresponding to the transform's pattern network is expanded. Expanding a psuedo-device involves satisfying all macro-device specs, performing all Free-wrt tests, and finally adding all the transform's instantiation network's devices and nodes to the problem network.

To summarize, the system's top level looks like

```
        TOP LEVEL (detail)
(INIT)  Initialize list of psuedo-devices to nil
        for initial writing, and to old list for
        improving code.
        Set remembered type to "RESTATE."
    LOOP
        (PROP) propagate all value structures.
        (TEST) if all outputs have values (improved values if
            improving code), then write code and quit.
        (FIND) For all new devices, try starting partial matches if
            transform's seed matches.
            Propagate partial matches according to schedule,
                    down to and including remembered type.
                Possibly create and add "add-break" devices
            Instantiate all completed partial matches:
                Satisfy K-variable specs.
                Satisfy U-variable specs.
            Add new psuedo-devices to front of list.
        (SELECT) Search list of psuedo-devices to find best one to apply.
            Only the terminals of the psuedo-devices are examined.
        (APPLY) Expand selected psuedo-device:
                Apply Free-wrt tests.
                Satisfy macro-device specs.
                Add instantiation-network nodes and devices.
                Add required macro-devices as complex devices.
                Remember type of transform responsible for
                    expanded psuedo-device.
            Go to LOOP.
```

Deciding which rule to apply

In order to decide which rule to apply (or, more specifically, which psuedo-device to expand), the system uses the results of a preprocessing analysis of the possible effects of the transformation rules. The results take the form of a classification of types of effect, and a list (called the *enablement*) of the circumstances under which this effect may be obtained.

There are two different kinds of problems the synthesis system solves: initially writing code and improving code already written. Although the principles behind the rule selection mechanisms are the same for both kinds of problems, the mechanisms themselves are different The emphasis in this section is on the initial writing problem.

At any point in the deduction process, certain nodes will have values (be known), and others will be unknown. In addition to this division among nodes in the problem network, the selection mechanism maintains three lists of nodes as *assumptions*. One result of deciding to apply a rule (that is, expanding a psuedo-device) is to add nodes to the variaous assumption lists.

The selection mechanism looks like

```
                SELECT (but see detail below)
   For each psuedo-device, see if the enablement is satisfied
          If so, record the number of additions to the assumptions required.
   Select the psuedo-device requiring the fewest additions. In case of tie, pick the
  earlier in the psuedo-device list.
```

The tie-breaker above encourages depth-first search. The following sections discuss the selection mechanism in complete detail.

Circuits

As was mentioned in chapter I, the main thing one wants to do in trying to solve a problem is reduce the size of circuits, and remove them when possible. Globally, a circuit is a directed sequence of devices (see diagram "circuit definition") and nodes whose values are unknown such that

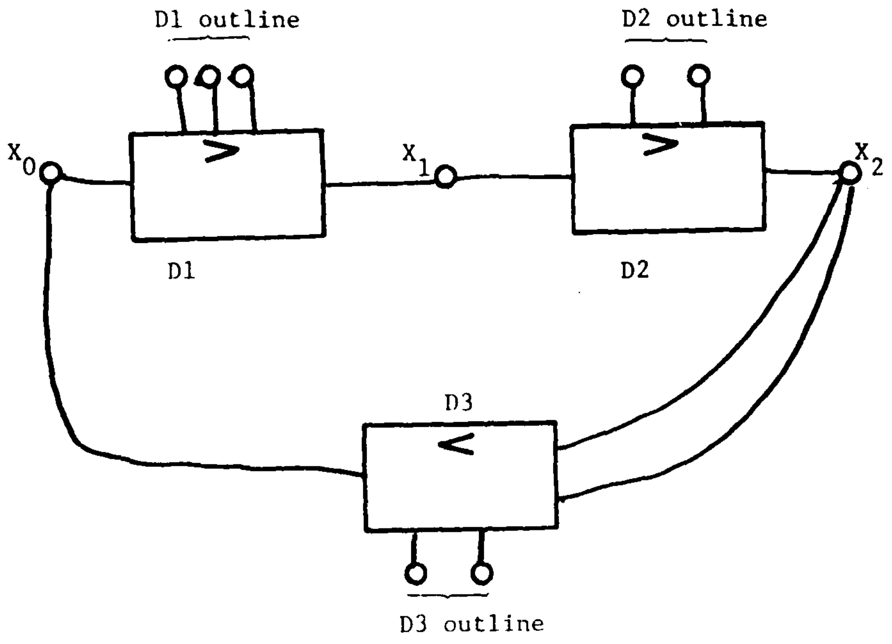


DIAGRAM "CIRCUIT DEFINITION"

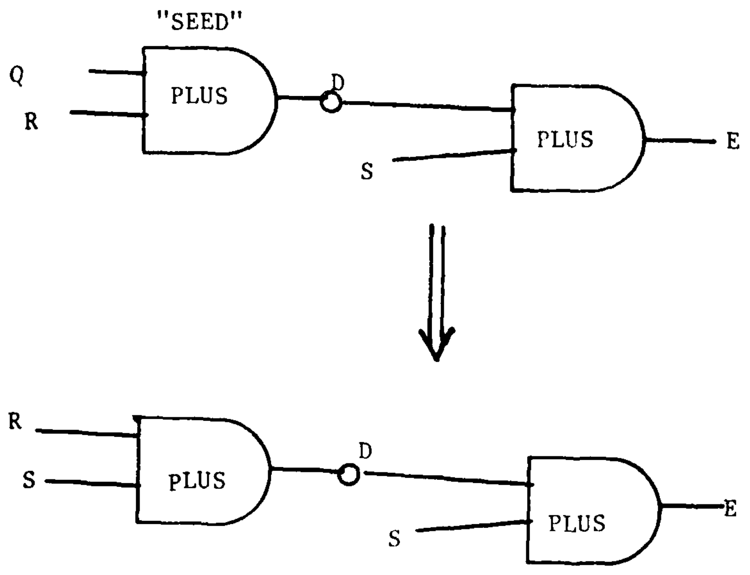


DIAGRAM "FUNNY-ASSOC"

1. Each circuit node appears as the node computed by a device rule of a circuit device.
2. Each rule above has as nodes in the needed list only circuit nodes and outline nodes.
3. All outline nodes are known (and only nodes needed for 2 are outline nodes).

The diagram illustrates a typical circuit. Note that outline nodes can be shared, and that several of the nodes needed by a rule can be circuit nodes.

This definition cannot actually be used because the third restriction (outlines must be known) is too strong. In operation it is replaced by a weaker criteria: outline nodes should be known or assumed known, or otherwise singled out (details follow).

The system does not actually find circuits, both because that would be a global operation and because the number of circuits grows exponentially with the number of devices (among other things, the assumptions cause this exponentiallity). Fortunately, on the basis of local evidence one can determine whether a portion of network could *potentially* be part of a circuit. This forms the basis of the selection mechanism.

Classification of Transforms

The "funny-associativity" transformation rule was mentioned in chapter I. It is reproduced below, and also shown in diagram "funny-assoc":

```
(DEFINE-TRANSFORM FUNNY-ASSOC (Q R S E)
                 ((+C Q R D) (+C D S E)) ((+C R S F) (+C F Q E)))
```

There are four ways this transform could reduce the size of a circuit. The outline nodes could be Q and E, or they could be R and S. For each of these outline node sets, either of the remaining terminals of the transform could be the input of the directed circuit.

The above analysis is typical of the preprocessing required for each transform. The type of effect of the transform is to REDUCE the size of a circuit, and the four collections of *outline*, *input*, and *output* nodes are the ***enablements***. In principle, this analysis could be automated, but the current system requires these to be entered by hand (automated analysis would be at least exponential in the number of terminals). This is done once as part of the transform's definition.

The form for stating an enablement is

```
(DEFINE-ENABLEMENT transform-name type
        <((<outline-terminal>) (<input-terminal>) (<output-terminal>))>)
```

For example, the enablement for FUNNY-ASSOC is:

```
(define-enablement funny-assoc REDUCE ((r s) (q) (e))
                   ((r s) (e) (q))
                   ((q e) (r) (s))
                   ((q e) (s) (r))).
```

There are six possible types of effect a transformation rule may have. The type of a transform, as defined in table "Transform types," is used to decide what partial matches to try propagating. The sequence used for initially writing code is:

REMOVE SUMMARIZE REDUCE RANGE RESTATE SPEEDUP.

For improving code the sequence is very similar; the SPEEDUP type is moved to between REDUCE and RANGE. These sequences are quite reasonable considering the semantics of the types.

Types of transformation rule application

As explained in the previous section, enablements have the form of a triplet of lists of transform terminals:

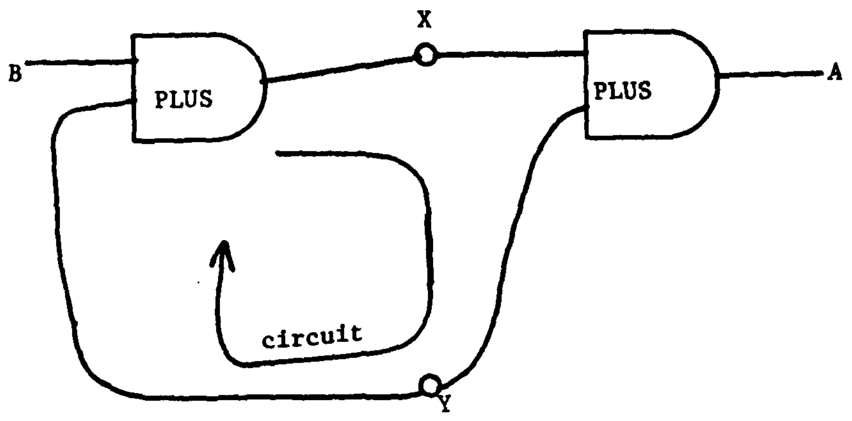
((<outline>) (<input>) (<output>))

where the computation path being affected or resulting from the application of the rule is specified as going from the inputs to the outputs. Although not apparent from the discussion thus far, this computation path is not *necessarily* actually usable because, for example, a device rule might require a macro-device that cannot actually be found. However, the general idea is that the computation path also uses the (presumed) known values of the outline terminals in the course of propagating a value from the inputs to the outputs.

There are two very different ways to use a transformation rule. The distinctions are most easily seen by considering a REDUCE transform. Supposing one could take a global view of the problem network, one would notice that there are two kinds of circuit (assuming certain outline nodes are known when they aren't really): those containing the desired output or nodes leading to immediately being able to compute it, and those not containing it or any node immediately leading to being able to compute it. In fact, both of these circuits should be reduced in size, but the local circumstances under which they should be reduced are different.

Diagram "Two linear equations problem" contains a circuit, and the circuit contains the desired output Y. The pattern for FUNNY-ASSOC matches this problem with

Q -> B, R-> Y, S -> Y, E -> A.



INPUTS: B, A
OUTPUT: Y

diagram "TWO LINEAR EQUATIONS PROBLEM"

One of FUNNY-ASSOC's enablements describes a situation where the transform's output is "helpful" in computing the desired output:

((Q E) (S) (R))

(reversing S and R doesn't matter, so that enablement also works). This kind of application is unimaginatively called a TYPE-B application; it is distinguished (basically) by all of the outline nodes being known, and the outputs leading to the desired problem solution.

Diagram "Three linear equations problem" is more complex. Firstly, with only nodes A, B, and C assigned values, there are no circuits in the diagram by the definition given above. However, if the single node Y (the desired output) is allowed as an outline node, then the circuit shown in the diagram does meet the definition.

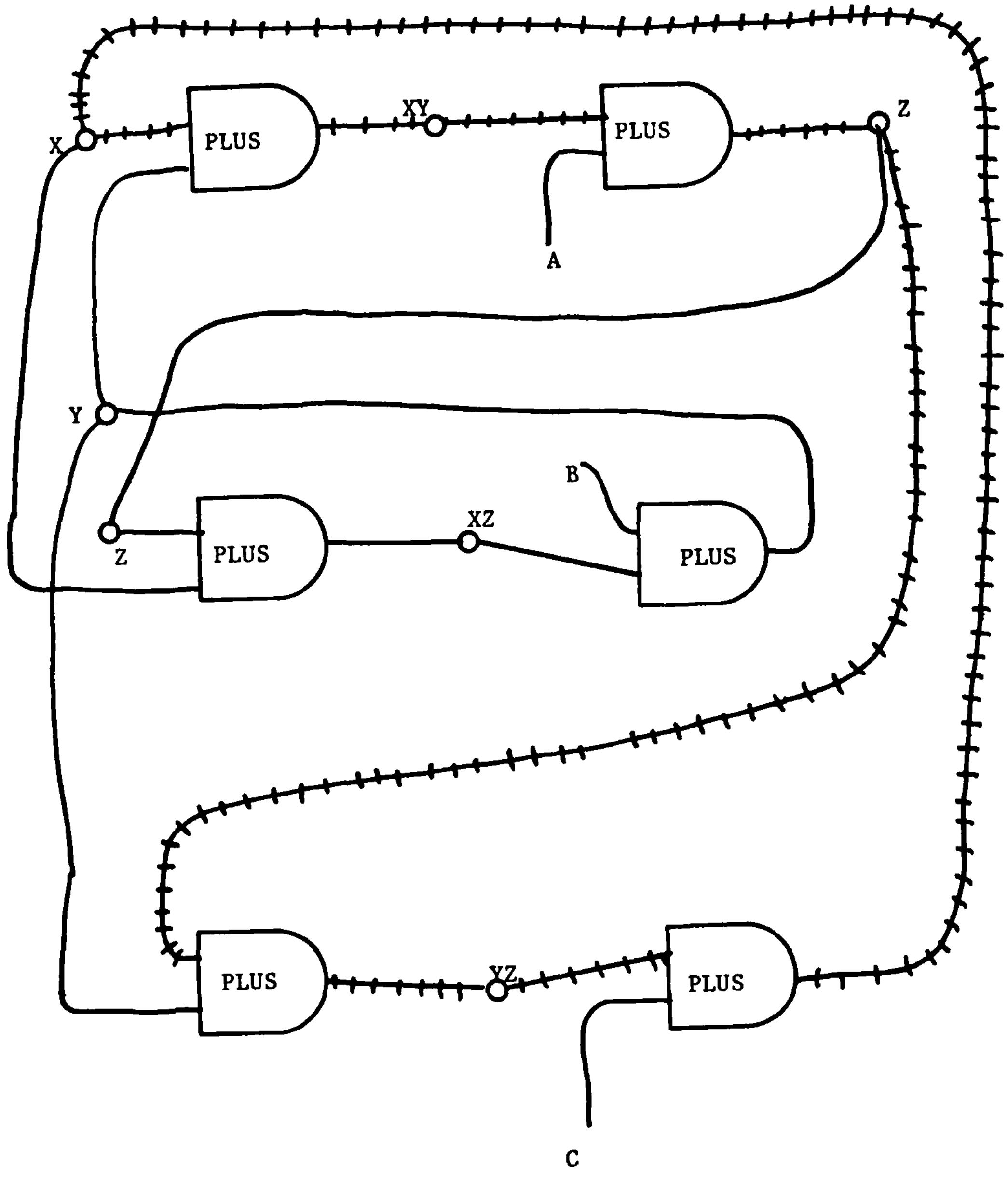
The transform FUNNY-ASSOC matches the problem diagram several ways, but the two of interest are when the seed matches device A1, and when it matches A5 (respectively):

Q -> X, R -> Y, S -> A, E -> Z

Q -> Z, R -> Y, S -> C, E -> X

In both of these matches, the global view containing the circuit in the diagram, the desired output (Y) is in the outline. This is the distinguishing feature of a TYPE-A rule application.

To summarize: there are two ways to apply a transformation rule. In terms of its enablements, the outline can contain the desired output (TYPE-A), or the circuit can contain the output (TYPE-B). The next section gives the details of the tests used to determine if a transformaton rule should be applied.



INPUTS: A , B, C   OUTPUT: Y
circuit: x, xy, a, yz, x (there are others)

DIAGRAM "THREE LINEAR EQUATIONS PROBLEM"

Assumptions and Enablements

The system uses three lists of nodes to help determine if a particular transform should be applied, and whether the application is TYPE-A or TYPE-B. These assumption lists are modified as a side-effect of the selection algorithm. Basically, these assumption lists are used as an attention focussing mechanism. They are:

- ASSUME-KNOWN -- These nodes don't have values, but the selection algorithm has previously treated them as if they were known.
- ASSUME-UNKNOWN -- These nodes are thought to be computationally "close" to the desired output, in the sense of leading immediately to being able to compute the output if they are known.
- REDUCE-GROUP -- These nodes are thought to belong to a circuit for which TYPE-A rule applications are appropriate.

Both TYPE-A and TYPE-B applications have several conditions in common. One is that all outline nodes should be distinct (that is, no duplications should occur in the outline). The reasoning behind this restriction is that if outline nodes (that is, nodes attached to the psuedo-device's terminals specified as outline terminals in the transform's enablement under consideration) *are* duplicated, then there ought to be a special case rule to deal with it. Also see diagram "Why no duplicates in outline" for an example of the kind of trouble this restriction eliminates.

Another restriction both TYPE-A and TYPE-B applications have in common is that the output nodes must in fact be unknown. If they were known, then the path could not be part of a circuit. This restriction must be altered when improving code.

The complete list of restrictions can be found in table "Enablement Tests for Writing Code." Notice that if the enablement has an empty outline, then TYPE-A fails and TYPE-B becomes simply "outputs are unknown."

The restrictions in table(s) "Enablement tests for writing (improving) code" were derived empirically. Although they "work" for all the examples in this report, they might need further refinement.

The assumption lists are initialized as shown below:

*REDUCE-GROUP, *ASSUME-KNOWN are initially empty

*ASSUME-UNKNOWN initially the desired output nodes

Extending and cutting back on assumptions

There are two ways the assumption list can be consistently extended. If one assumes that some collection of nodes is known, then one can determine what nodes are also known. This ALSO-KNOWN extension is the basis of both assumption extension mechanisms (rules involving macro-devices and special case rules are *not* used in finding this kind of extension). For example, if "x" is assumed known, then "y"$=x^2+13$ would be in the also-known extension of "x".

Whenever, as a result of analyzing an enablement and on that basis applying a rule, the system adds some nodes to the *assume-known assumption list, the also-known extension is also added, provided it does not intersect *assume-unknown.

Similarly, when trying to minimize the number of new nodes being added to *assume-unknown, if two enablements both would cause the same number of additions, then if one addition constitutes *wishful thinking* (described below), it is prefered. A perhaps useful refinement of this would be to compute the "wishful thinking distance," but this has not been tested.

Wishful-thinking is straight-forward to test for: if the also-known extension of the addition to *assume-unknown would result in nodes already in *assume-unknown being known, then the addition constitutes wishful thinking. It would be simple to also measure the smallest number of devices involved in obtaining any such node already in *assume-unknown.

The sequence of additions to the assumption lists is retained. Since it is inconsistent for a node in *assume-unknown or *reduce-group to have a value, these lists are trimmed back to the state they were in before the node (now) having a value was added. No such inconsistency occurs when an *assume-known node receives a value. But to simplify the code, these known nodes are removed from the *assume-known list.

The complete criteria for which rule to select (which psuedo-device to expand) can now be given:

```
                    SELECT (detailed)
Pick the rule with the highest type (in the transform type order being used).
Among rules of the same type:
    Rules that add to *reduce-group are WORSE than those that do not.
    The number of additions to *reduce-group is minimized.
    Rules that add to *assume-known are WORSE than those that do not.
    Rules that add to *assume-unknown are WORSE than those that do not. Among
      rules that add to *assume-unknown:
```

The fewer the number of additions to *assume-unknown, the better.
Among rules adding the same number of nodes to *assume-unknown, wishful-thinking additions are better.
All things being equal, pick the rule most recently matched (depth-first preference).

The selection algorithm above is used to compare two potentially applicable matches (more correctly, the psuedo-devices resulting from instantiating the completed partial match). After sweeping through all applicable psuedo-devices, a psuedo-device to expand is thus selected.

Special handling for Macro Devices

The discussion thus far has ignored the presence of macro-device specifications in device rules. Under certain circumstances these specifications can affect the assumption lists.

Device rules assert that some computational relation exists between the needed terminals (inputs) and the terminal determined (output) of a rule. But there is also some kind of connection between a device rule's needed terminals, and the terminals specified as inputs to a macro-device used by that rule.

Suppose, as in the square-root example, the system knows

$$X^2=Y; X > 0, Y > 0$$

and that given Y, it is to write a program to compute X. Then the node for X would be on the *assume-unknown assumption list, and Y would have a value. This problem (for which a solution has already been given) can be modified so that instead of simply computing the square-root of Y, the system is asked to compute the sum of square-roots of the numbers 1.0 to Z, inclusive. This problem is illustrated in the diagram "sum of square roots." In this diagram it should be obvious that if the system were to add node NSX to *assume-known and node NSFX to *assume-unknown then as far as the rule-selection mechanisms are concerned, this problem is about the same as the original square-root problem.

It should come as no surprise that the system has a rule:

> *Inward Macro-device Assumption Extension:* If all a rule's needed-node list is known, and the node determined by that rule is unknown, and that rule uses macro-devices, then add the macro-device's input nodes to *assume-known, and the macro-device's output node(s) to *assume-unknown.

Perhaps more surprising is that the system also extends assumptions in the other direction:

> *Outward Macro-device Assumption Extension:* If any of a rule's macro-device input nodes are known, then add the rule's needed nodes to *assume-known. Furthermore, if the macro-device's output node(s) are unknown, add the rule's output to *assume-unknown.

The Bernoulli example (in chapter 4) shows a case where this extension is needed. Diagram "Bernoulli problem statement" shows the initial situation. The result of using the rule above on the device TPS (term-wise Power Series) is to add NT to *assume-known, and add PSX to *assume-unknown. This allows the system to become interested in expanding the exponentiation device EXP. The complete solution to this problem is the backbone of chapter 4.

Example: Three linear equations Solution

Diagram "Three linear equations" shows the network expressing the following system of linear equations:

$$(x + y) * A = z$$
$$(x + z) * B = y$$

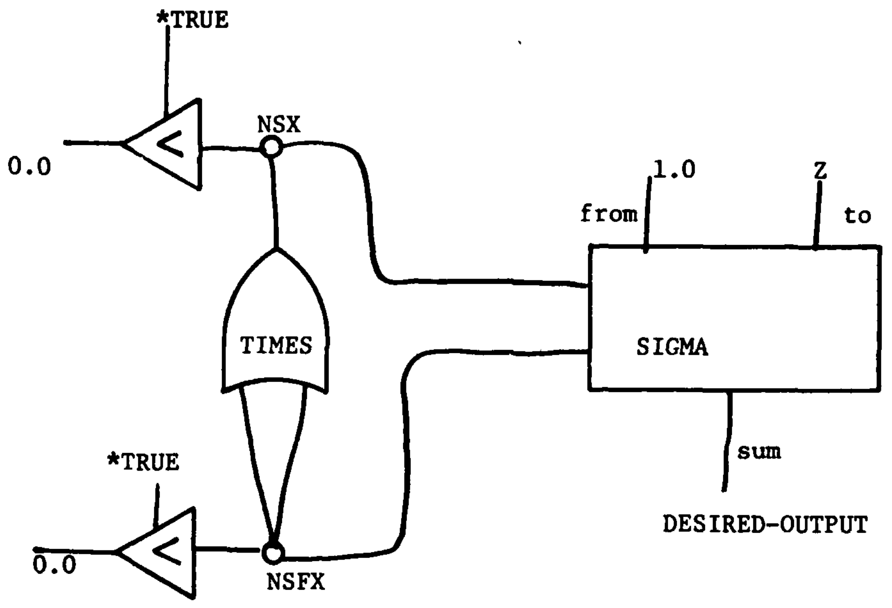
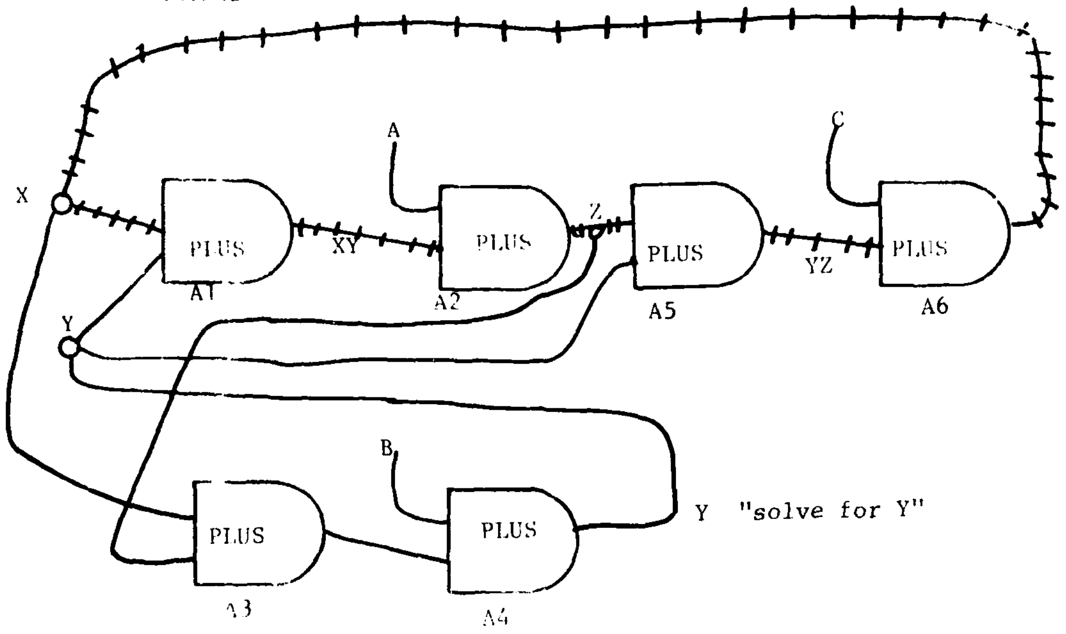
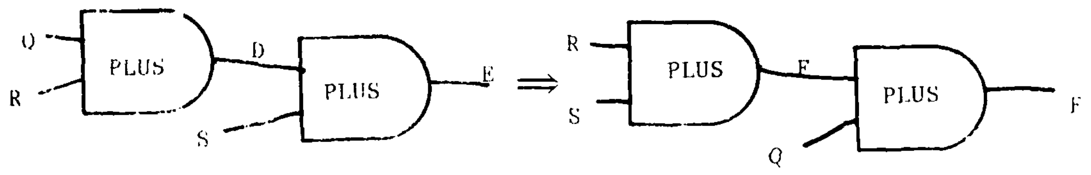


DIAGRAM "SUM OF SQUARE ROOTS"

DIAGRAM "THREE LINEAR EQUATIONS"

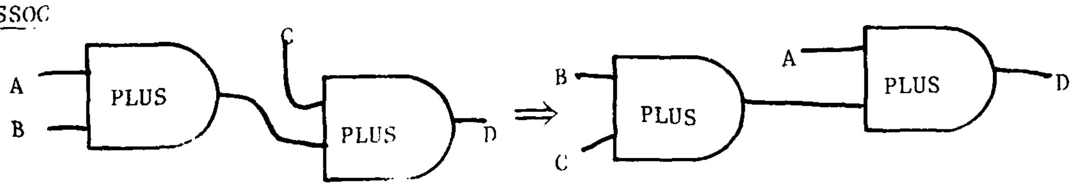


FUNNY-ASSOC



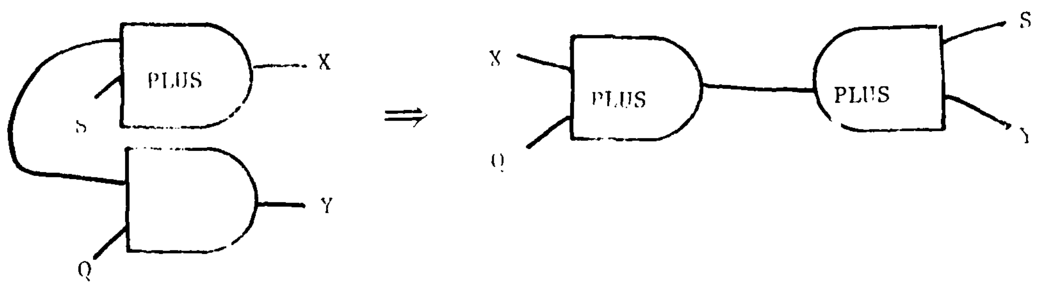
Enablements.	((R S) (E) (Q))	((Q E) (R) (S))
REDUCE	((R S) (Q) (E)	((Q E) (S) (R))

+ASSOC



Enablements:	((A D) (B) (C))	((B C) (A) (D))
REDUCE	((A D) (C) (B))	((B C) (D) (A))

+ASSOC4



Enablements:	((Q X) (S) (Y))	((Q X) (Y) (S))
REDUCE		

$(z + y) + C = x$

This diagram and set of equations will be used for two different problems. One (let's call it LE3) is "Write code for computing Y given A, B, and C." The other (LE2) is "Write code for computing Y given A, B, C, and X." The second (LE2) is a simpler problem, but it has some redundant facts in it. Most problem-solving systems blow up when given too much information; this system does not.

The diagram "Three linear equations" also shows three associativity rules. FUNNY-ASSOC and +ASSOC are almost identical -- the only differences involve commuting the arguments. Most problem-solving systems blow up when given redundant rules; this system does not. Each of these transforms are of type REDUCE and have four enablements. +ASSOC4 is derived from +ASSOC by doublet introduction and deletion (see diagram "derivation using Doublets" in chapter 2). Since the rule is symetric, only two enablements (instead of four) are required.

Both problems use the same initial assumptions: *assume-known and *reduce-group are empty, and *assume-unknown has only node Y. In problem LE2, node YZ's value can be determined (x-c).

No REMOVE transforms match. Commutation of addition is performed by a SUMMARIZE transform. For this, 6 matches and expansions occur, one for each device. Trying to commute these commuted devices results in duplicating devices already in the network. The system detects this and merges the duplicated devices.

The real action begins when partial-matches for **REDUCE** transforms are propagated. The system finds 72 matches for the five associativity transforms in the system's library. Some examples are:

A: Funny-assoc q->z, r->y, s->c, e->x ((r s) (e) (q))
 Funny-assoc q->x, r->y, s->a, e->z ((r s) (e) (q))

B. Funny-assoc q->yz, r->c, s->y, e->xy ((r s) (e) (q))
 Funny-assoc q->xy, r->a, s->y, e->yz ((r s) (e) (q))

C: Funny-assoc q->z, r->x, s->b, e->y ((r s) (q) (e))

Of course, whenever Funny-assoc matches, +assoc will also.

In problem LE3, no type b applications are possible. The first four matches above all meet the type a criteria. In all four cases, the outline node that could intersect +assume-unknowns is Y (already in that list). The circuit (inputs to outputs of the enablement) is indicated in diagram "Three Linear Equations" and globally contains X, XY, Z, and YZ. Which of these four transforms (or their +assoc counterparts) is actually used depends on "the luck of the draw" (actually it *is* deterministic). Let's follow two different solution paths: LE3A applies

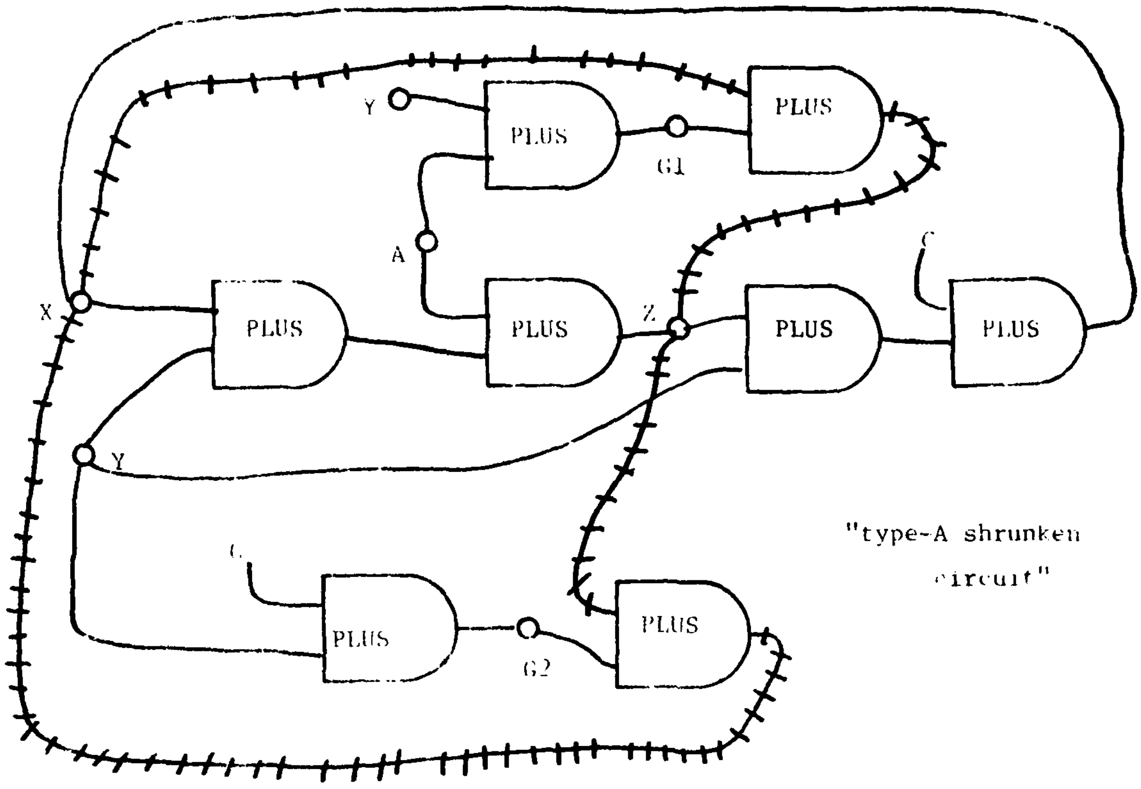
A: Funny-assoc q->z, r->y, s->c, e->x

adding X and Z to +reduce-group, and LE3B applies

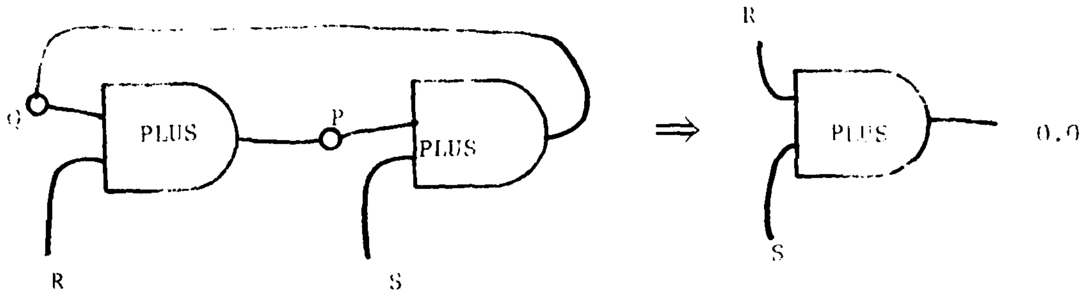
B: Funny-assoc q->yz, r->c, s->y, e->xy

adding XY and YZ to +reduce-group.

In problem LE2 (remember X is known in this example), the last match (C) in the list does meet the type b criteria with enablement ((R S)(Q)(E)):



Transformation rule NEG-ASSOC, type REMOVE:



enablements: (NIL (R) (S)) (NIL (S) (R))

matches with R ➔ G2, Q ➔ Z, P ➔ X, S ➔ G1

DIAGRAM "THREE LINEAR EQUATIONS LE3A"

outline: X, B; input: Z; output: Y.

By the way, this fails type-A criteria number 1. Another enablement for this transform uses nodes Z, Y for the outline, and type-a criteria number 2 fails. The reason Y rather than Z was chosen as the output is that minimizes the number of additions to *assume-unknown.

Continuing with the three solutions LE3A, LE3B, and LE2, now all the devices added must be commuted -- two in each case. No REMOVE type transforms apply, so another round of matching REDUCE type transforms occurs. The number of new matches is as shown below.

LE3A: 56; LE3B: 48; LE2: 56.

Why find more matches before using the ones already known? The system does not follow either a depth-first or a breadth-first strategy. Rather, it collects *all* the transforms (partial matches, actually) it *could* apply, and then decides *which* to apply.

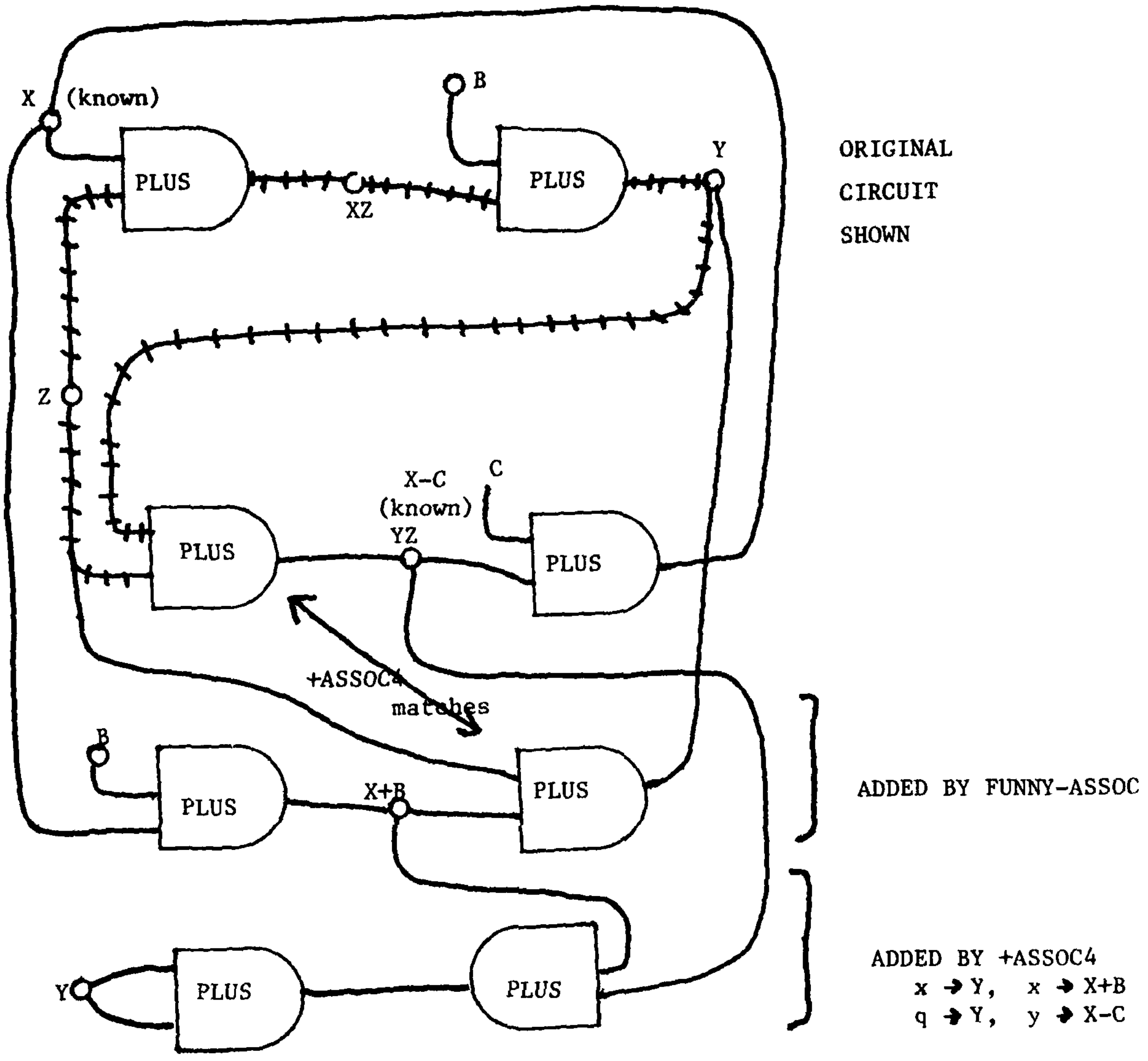
In the LE3 problem, none of these new matches are used. In LE3A, the old Funny-assoc match

Funny-assoc q->x, r->y, s->a, e->z

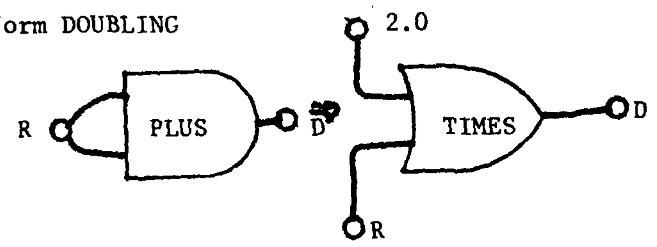
doesn't result in any additions to assumption lists under type-a criteria. Similarly, in LE3B the match

Funny-assoc q->xy, r->a, s->y, e->yz

can be used without additions. Notice that "mixing" these would require additions to the *reduce group. See diagram "Three Linear Equations LE3A" for the situation after these transforms have been applied.



transform DOUBLING



$$Y = \frac{(X+B)+(X-C)}{2.0}$$

diagram "LE2"

Diagram "LE2" shows how one of the newly matched transforms, the +ASSOC4 transform, matches and how later the DOUBLING transform is applied to finally solve the problem.

Another round of matching takes place. In both LE3A and LE3B, a REMOVE type transform named NEG-ASSOC matches (see diagram "Three Linear Equations LE3A"). These have *enablements that satisfy type-b requirements*. Using this transform in LE3A causes one of G1, G2 to be added to *assume-known, and the other to *assume-unknown. Similarly for LE3B. Notice that G1 is a wishful-thinking addition (so is G2).

Another round of propagating REMOVE type transforms occurs in both LE3A and LE3B. Again, matches are found, this time for the DOUBLE-SUM transform (different matches in LE3A and LE3B, of course). This transform is shown in diagram "Double-sum." The enablement is satisfied. When this transform is applied to LE3A and LE3B, the problem is solved, and Y=(A+C)/-2.0.

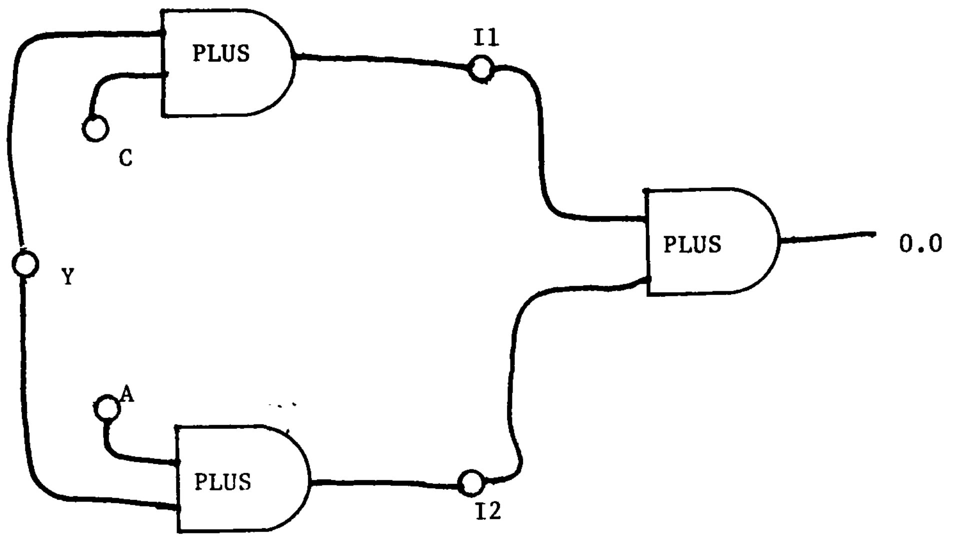
The solution to LE2 was

$$Y = [(X + B) + (X - C)] / 2.0$$

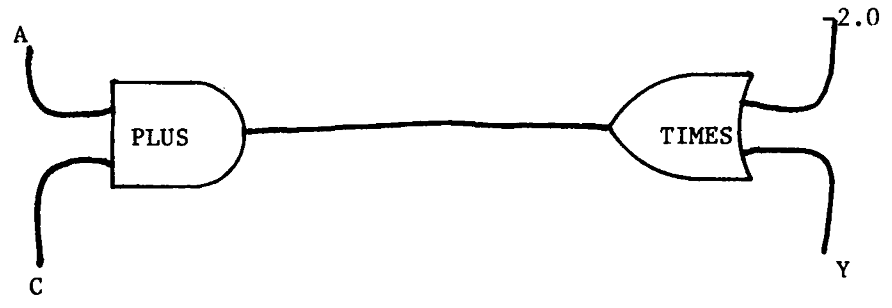
The code for this solution is actually *slower* than the code for LE3. A later section will show how the solution can be improved.

Modification of the Selection Algorithm for Improving Code

In the preceeding sections the emphasis has been on initially finding a computation path from



transformation DOUBLE-SUM ⇓



type REMOVE, enablement: ((A C) NIL (Y))

DIAGRAM "DOUBLE-SUM"

the input nodes to the output nodes. Much use has been made of the simple observation that any node whose value is unknown must depend on those nodes whose values are known. As a very general strategy the system tried to reduce the number of unknown nodes.

If a computation path already exists from input to output, however, the distinction between known and unknown nodes is no longer particularly useful. In its place, the system uses an even more revealing source of information about computational dependence: the computation path already discovered.

The initialization procedure used for a network for which code has already been written includes

1. *All nodes actually used are added to a list *nodes-definitely-used.*

2. All complex devices actually used either directly or in macro-devices are expanded.

3. Nodes used more than once in the computation path indicate the presence of a circuit-like structure in the computation path. These nodes are added to *assume-unknown, and also to *reduce-group.

4. The output is added to *assume-unknown.

5. The *assume-known list is initialized by the inputs minus the multiply-used nodes (4 above) and minus the output.

6. Any macro-devices used have their input nodes added to *assume-known, and their output nodes added to *assume-unknown.

When this initialization has been completed, matching and propagation take place much as in the initial-writing case. Instead of looking for a value in the desired output node, the system looks for an improvement in the time-cost. As mentioned before, the SPEEDUP transform type is used earlier on when improving code.

While propagating values is a network to be "improved," the system maintains a list *IMPROVED-NODES. This list is used in the enablement criteria.

The tests to see whether a transform should be applied must be adapted. As before, two types of application, TYPE-A, and TYPE-B, are used. The restrictions and effects are shown in table "Enablement tests for Improving Code."

Example: Solving systems of linear equations continued

In an earlier section, a problem involving three linear equations was presented and solved for two cases. One (LE2) involved redundant information in the problem statement, and it happened that the system found a less efficient solution in this case than for the LE3 problem

When the system is asked to improve this inefficient code, the system determines that node X is used more than once in finding the solution. The initial assumption lists are

*assume-known: A, B, C; *assume-unknown: X, Y;

*reduce-group: X.

The system starts with a list of all the old matches. Of this initial collection, several can be applied without adding any nodes to the assumption lists, but these create only duplicates of already existing devices.

The actual work begins when an old *ASSOC transform satisfying type-a criteria is selected. This matches the problem network with

D -> Z, C -> A, B -> Y, A -> X.

The enablement is ((B C)(A)(D)).

In fact, this is the "same" transform as the Funny-assoc application that started the LE3A problem variant. The type-a circuit is the same for improving the LE2 code as for writing the LE3 code originally. The LE3B variant solution is not used because it would require a larger number of additions to *reduce-group.

The remainder of the "improvement" example follows exactly the LE3A solution shown previously, except that in addition to the matches found in initially writing the code, 44 more REDUCE type matches were found (and more of other types as well).

FINDING MATCHES

The mechanism that matches transformation rule patterns against the datum network (which is the original problem network with new devices and nodes added) has three distinct parts: a preprocessor, a partial match propagation routine, and routines for finding nodes for U-variable and K-variables. When the matcher has finished, a psuedo-device can be created (as described earlier). The mechanisms for expanding (or instantiating) psuedo-devices will be discussed in the next section.

The preprocessor takes the network that constitutes the transform's pattern, and produces a *match sequence*. The details of this preprocessor are not very interesting. The match sequence starts with the seed device, followed by node/device pairs such that the node has already been matched and the device has the node on one of its terminals. The Add-Break devices are the last devices on the match sequence.

The partial match propagation mechanism is also quite simple. First, the system compares a transformation rule's seed against a datum device. If they match, then the devices' terminals are compared. If they are compatible (constants in pattern only match constants in datum), a partial match is created and stored in the datum node corresponding to the first node in the match sequence.

Partial matches are propagated from one node to another according to the match sequence, subject to the restriction that a datum device cannot be matched by *more than one* device in any pattern (this is critical!) A datum device can of course be involved in several pattern matches. Each propagated partial match is stored in the appropriate "next" node in the match sequence. If propagation fails because a device of the proper type is not on the datum node, and that pattern-device has been flagged as being an add-break device, then a new device is added to the datum network, a round of value propagation takes place, and then partial-match propagation continues. Propagation ends when the match sequence has been exhausted.

The third and final part of the match mechanism finds nodes to match U-variable and K-variable specifications. The algorithms for finding matches in these two cases will be described separately below. The semantics of these specifications were carefully engineered so that for any match of the pattern network to the datum network, if the specification could be satisfied, only one datum node per variable-node needs to be considered.

A lower bound on the system's time performance is provided by the matching mechanism. For any particular transformation rule, the worst-case number of possible matches is computed as follows. Suppose the pattern has M devices, and the datum network has N devices. Then the seed device could match as many as N devices, and then the next device in the sequence could match as many as (N-1) devices, etc. The number of possible matches, assuming all devices in the pattern and in the datum are of the same type, is given by

$$N*(N-1)*(N-2)*...*(N-M+1)$$

This expression is strictly less than N^M for $M > 1$. Of course this worst case never occurs because firstly, the devices are (usually) of different types, and secondly because the terminals of a device are distinguished. The worst observed case was *quadratic* (an associativity pattern).

Finding nodes for K-variable specifications

A K-variable specification is essentially a list of nodes (specified by transform terminal) to be considered as "inputs" for the K-variable. The values of these nodes are currently unknown (else the search for a K-variable node fails), but if they were known would allow some node (the node one wants to find) whose value is currently known (and *not* a mode value) to be computed from the "inputs."

The algorithm for finding these nodes (if they can be found) uses several operations that are also used for finding macro devices (explained in the next section). These are explained in table "Packaging Primitives."

To find a node to match a K-variable specification, the following procedure is used:

```
        If any of the "inputs" are known, fail.
        Push-value-structures.
        Initialize K-variable "inputs."
LOOP
        Propagate values. If a node receives a value and that node was previously
      computable, then go to WIN.
        Suggest external nodes.
        If no nodes can be suggested, then fail (pop value structures and return).
        Go to LOOP.
WIN
        Pop value structures.
        Return node that was previously computable.
```

A part of the algorithm above is testing to see that the external nodes found (if any) are in fact free with respect to the computation path from the k-variable specification "inputs" to the node eventually found (if the search was successful). Since no output can be specified for the computation path, this test takes a very simple form (see discussion following).

Currently, if the search for a K-variable (or a U-variable) fails, it is never attempted again. This may not really be correct.

How much time does this search take? Pushing and popping value structures takes no more than time proportional to the number of nodes. Suggesting external nodes (to be discussed later) also takes time proportional to the number of nodes in the network (it involves analyzing the entire network, and then using the results of this analysis in judging each node to see if it can be suggested). The loop in the algorithm above cannot be executed more than the number of nodes in the net, because at least one node must be suggested as external each time around (and no duplicate suggestions can occur). Therefore, the algorithm (in the worst case) takes time proportional to the square of the number of nodes in the network, and by previous arguments, this is polynomial in the number of transforms applied.

Finding nodes for U-variable specifications

Unlike the algorithm for finding K-variable nodes, value propagation cannot be used in finding U-variable nodes. Recall a U-variable specification is written as:

```
(DEFINE-U-VARIABLE-NODE transform-name
  (U-variable-terminal-name (<input-terminal>) (<output-terminal>)))
```

The idea is that there is a computation path from the input terminals (of the pattern network) to the output-terminals specified. The inputs from the specification may be known, but the outputs must not be currently known. However, if all the inputs and the node matching the specification were known, then the outputs should be computable. The node found should not be an output node.

The problem with satisfying this kind of specification is to avoid examining all computation paths from input to output, because the number of such paths is (in the worst case) exponential in the number of devices. In all other situations where the system must find a computation path, *any* computation path will suffice. But here a special computation path is required. it must involve a node currently unknown as a so-called external node (i.e., it must be free with respect to the computation path specified).

The algorithm makes use of a list *IN-EXT, initialized to contain the also-known extension of the input nodes from the U-variable specification. If any output node is in *IN-EXT, then the search fails.

The system then constructs a list of nodes it will "back-up": *NBKUP. Initially this list contains the output nodes from the specification. The overall algorithm looks like:

```
    Initialize *IN-EXT and *NBKUP.
    If *IN-EXT and *NBKUP intersect, fail.
LOOP
    Form new *NBKUP by backing up from old *NBKUP. This step will be
   explained in detail below.
    If backup step discovered a U-node, then return it.
    If *NBKUP empty, then fail.
    Go to LOOP.
```

The backup step is, for each node in *NBKUP, examine the devices attached to the node ignoring devices that have already been "backed up through." Each device is then examined for a rule that computes the node being backed up. Rules that compute a mode, and rules that use macro-devices are excluded. If such a rule can be found, a list of nodes is formed such that

1. The node is not in *IN-EXT.
2. The node is not known.
3. The node is not in the output node list.
4. The node is needed by the rule.

There are evidently three cases for the length of this list of nodes:

The list could be empty. This is actually impossible, because then the node being backed up could not have been in *NBKUP.

The list contains one node, and all the outputs are in that nodes also-known extension (including *IN-EXT as known). This is the U-variable node. Return it.

Otherwise. Pick one of the nodes in the list (at random) and add it to the new *NBKUP. If any of the nodes are already in *NBKUP, then none need to be added.

This completes the description of the backup step

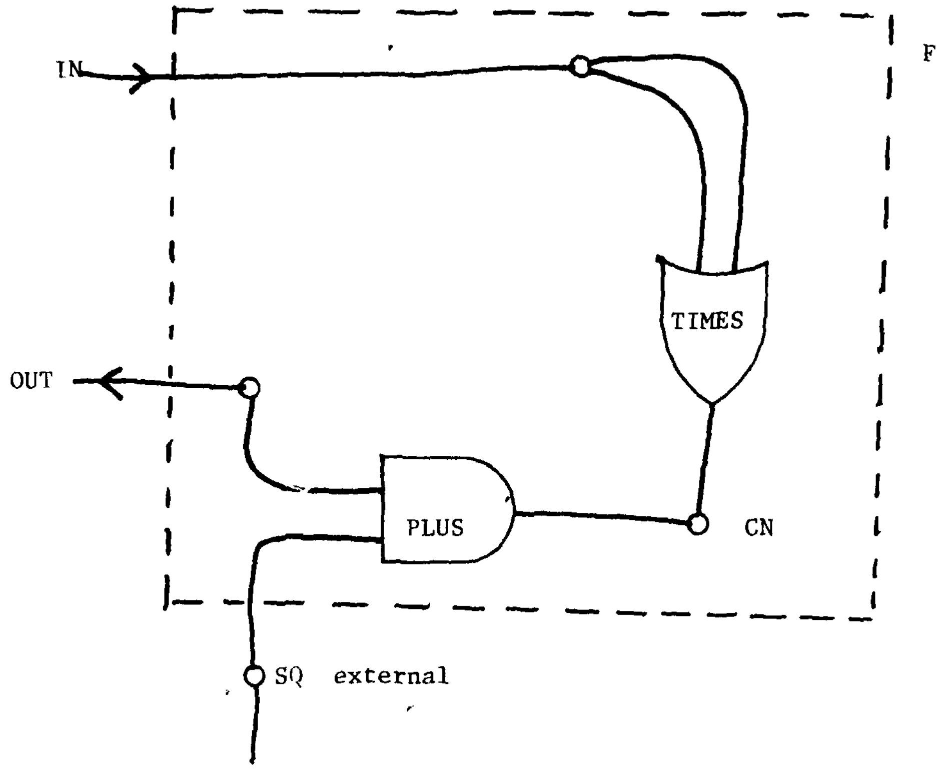
That the node found by this procedure satisfies the specification is evident by testing to see that the outputs are in the also-known extension. Although in the worst case the list *NBKUP can grow exponentially with the number of times around the loop, no duplicate entries are permitted, so the length of *NBKUP is bounded by the number of nodes in the network. Similarly the number of times around the loop is bounded by the number of devices in the network, so the overall time cost is proportional to the number of devices times the number of nodes, and therefore polynomial in the number of transformation rules applied

APPLYING TRANSFORMATION RULES

The result of applying a transformation rule is to add a number of devices to the datum (or "problem") network. The number of devices added is equal to the number of devices in the instantiation network plus the number of macro-device instantiation specifications.

In addition to creating and adding copies of the instantiation network devices and nodes, the critical steps of applying a transformation rule are applying "Free with respect to" tests and finding macro-devices. These two steps are closely related. In fact, testing for nodes being free with respect to a computation path is part of the process of finding a macro-device

The principle problem in finding and packaging macro-devices centers on the so-called



F(IN) = OUT = (IN*IN)-SQ

If F(IN)=0 then IN is the square-root of SQ.

DIAGRAM "EXTERNAL NODE IN SQUARE-ROOT"

"external nodes." For example, in solving the square-root problem the system found a macro device F (see diagram "External node in square root"). This device was specified as computing a node OUT from a node IN. The macro-device found included a third terminal that was connected to the node SQ, and is an "external node." The following sections describe how these macro-devices, and in particular these external nodes are found.

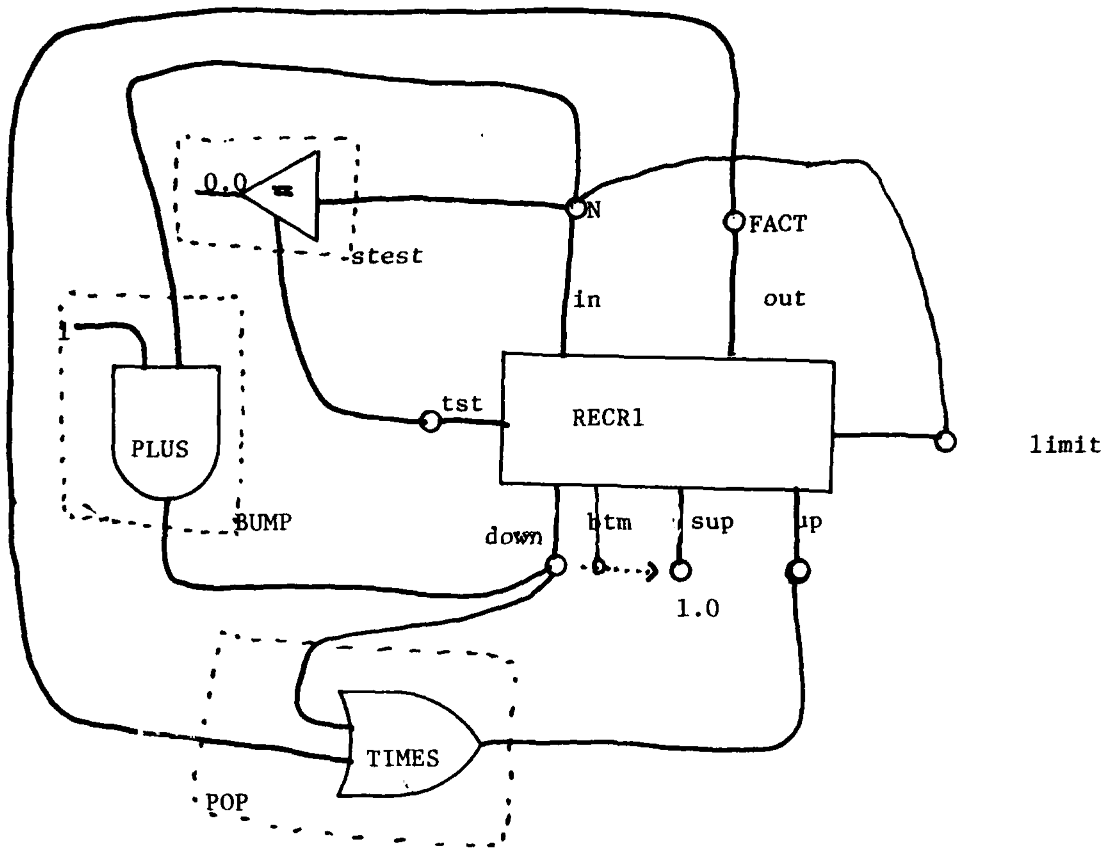
Finding Macro Devices

A macro-device is specified by selecting a set of nodes as inputs, and another set as outputs. It may be that some outputs are also inputs. These "identity function" outputs are handled specially by the system.

Another class of outputs the system handles specially are those that are known and free with respect to the specified inputs. This situation arises in the diagram "Alternative Recursive Factorial" in finding the macro-device for START. The node on RECRI's SUP terminal is a constant 1.0, and is free with respect to the node on terminal BTM (compare this with the diagram "single-rec-application"). In this case, the macro-device for START encodes a constant function for SUP.

In the discussion below, it is convenient to ignore this special case of "constant functions" as well as the special case of "identity functions." That is to say, in the following, outputs are not also inputs, and do depend on the inputs for their values.

The process of finding a macro-device closely resembles that of finding a K-variable



```
MACRO DEVICES NEEDED BY RECR1:
      (STEST  (IN) (OUT))
      (BUMP (DOWN) (IN))
      (POP (DOWN OUT) (UP))
      (START (BTM) (SUP)) <======
```

DIAGRAM "ALTERNATIVE RECURSIVE FACTORIAL"

The algorithm, outlined below, uses primitives in the table "Packaging Primitives".

<u>Finding a Macro Device</u>

```
          Push value structures.
          Initialize input nodes.
     LOOP
          Propagate value structures.
          If all output nodes have values, go to WIN.
          Suggest external nodes. If no nodes can be suggested, the macro device cannot be
        found at this time. Pop value structures. Fail.
          Suggestions are recorded as possibly-external.
          Go to LOOP.
     WIN
          Find nodes used in computing output nodes. These nodes are to be divided into two
        classes: internal and external.
          The list of external nodes is initialized to contain the inputs and outputs.
          Divide nodes used. This step is described in detail below.
          Create a complex device with a terminal for each external node.
          Create a defining network by copying all nodes used and all devices with all
        terminals attached to nodes used.
          Write device rules for new complex device by encoding functions for the defining
        network.
          Pop value structures.
          Return complex device.
```

The step that divides the nodes used requires further explanation. If a node is neither external nor possibly-external (i.e., neither an input nor an output nor suggested as an external), then it is an internal node. Each possibly-external node in the list of nodes used is further examined by seeing if it could be computed using other possibly-external nodes (possibly not in the list of nodes used). If it can, then it is made internal, and the possibly-external node(s) used in computing it are added to the list of nodes used.

This complex post-analysis is required because the precise computation path cannot be known when externals are suggested, and so it can happen that mutually computable external nodes are suggested. The post-analysis resolves this uncertainty.

Suggesting Externals

In order to be suggested as an external node, either for the purpose of finding a macro device or for finding a K-variable node, the node in question must both be previously computable (easily decided by using the set of pushed value structures), and be free with respect to the computation path from inputs to outputs.

The process for suggesting externals involves examining each device in the network, and each rule in a device. The system is looking for a (non-special case) device rule whose node computed does not have a value, and whose list of needed nodes contains only nodes that are either known or both previously-computable and free with respect to the input-output computation path. If such a rule can be found, all nodes in the second catagory are suggested as externals.

The notion of "Free With Respect To"

Three processes involve testing to see if a node is free with respect to a specified computation path from input nodes to output nodes: finding K-variable nodes, finding macro-devices, and satisfying FREE-WRT specifications associated with transformation rules. Since the way the system represents iteration and recursion dependins on finding macro-devices, this algorithm is critical to the operation of the entire system.

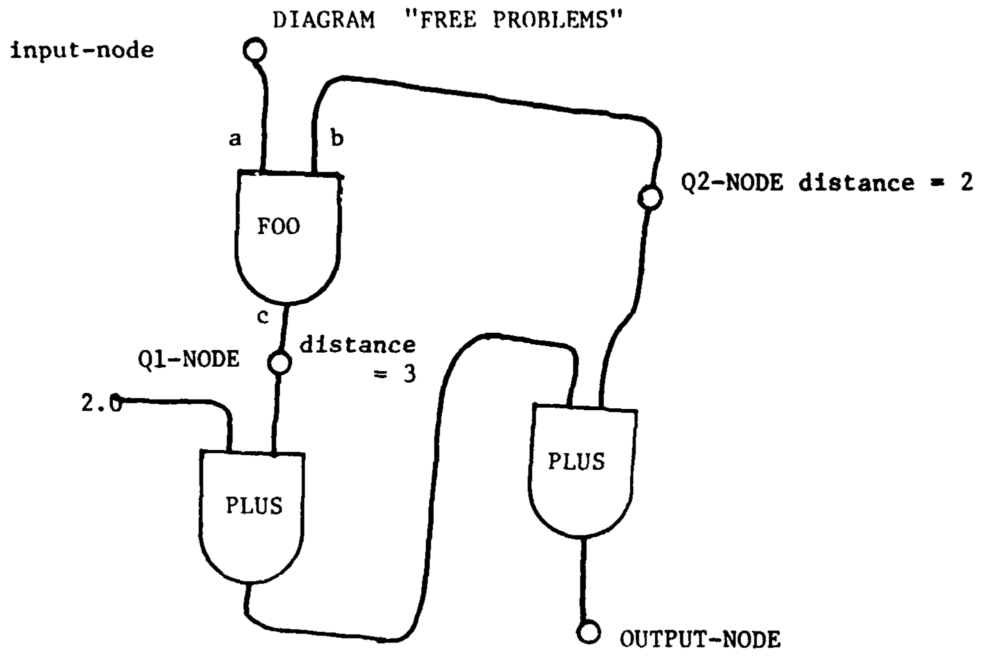
To simplify the following exposition, the term "free" will be used as a shorthand for "free with respect to the computation path from the specified inputs to the specified outputs (if any)".

The notion of a node being "free" has not been defined well enough to allow a proof that the system's test is correct. Consider the situation in diagram "Free Problems." The diagram shows how a node Q2-NODE shifts from being judged free (by the algorithm used by the system) or not free as the situation is slightly changed.

The notion of a node being "free" is similar to the notion of a quantity being a parameter in a mathematical formula. A quantity *can* be a parameter if it can be held constant as the input(s) of the formula are changed. Any node judged free has this property.

Problems arise when more than one computationally related quantity could be thought of as the parameter. In these cases, the system prefers to consider the node furthest from the output as the independent parameter. In most cases this agrees with intuition, but the "right" answer to these problems is sometimes not at all obvious.

The question should not be "is the free test correct?" so much as "does the system do the 'right' thing in situations where the free test is used?" In seeing if FREE-WRT specifications are satisfied, if the node is free, then it is both used to compute the output and not computable using only the inputs (even assuming I-device circuits can always be eliminated and allowing use of constants).

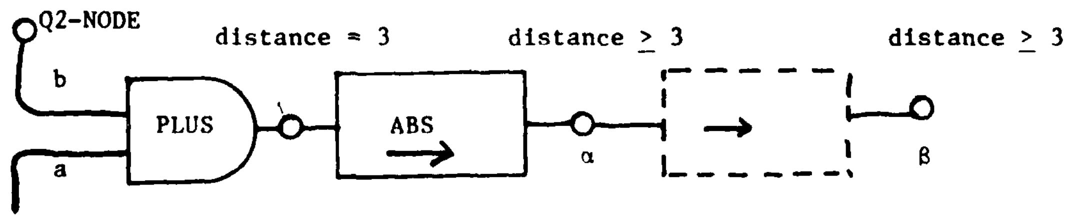


Suppose FOO is a PLUS device. Then Q2-NODE is <u>not</u> free.

Suppose FOO has only the rule c = ABS(a + b). Then both
Q1-NODE and Q2-NODE are free.

Suppose device FOO is replaced as shown below. Then Q2-NODE is <u>not</u> free,

but α = Q1-NODE is free
β = Q1-NODE is free



In finding K-variables the free test is crippled by not having an output specified. In this circumstance the test amounts to judging nodes computable from the input assuming 1-device circuits can be removed as *not* free, and the rest as being free. Considering the nature of finding K-variables, this is appropriate.

In finding macro-devices, the question of which independent parameter depends on which is solved after a value for the output is found.

In conclusion, using the free test below lets the system do the "right" thing in all cases.

"Free with respect to" Testing

The algorithm below tries to determine if a node QNODE (for the node in question) is free. The algorithm may make an error and claim that QNODE is not free, even though it is. The idea is to see if QNODE could possibly be computed on the basis of the input nodes and other nodes "further" from the output nodes than QNODE.

The first step in the free test is to order the nodes in the network according to their distance from the output nodes (if any). The ordering is easy to obtain. The outputs are given distance = 1. Then for each node N newly assigned a distance D the devices connected to it are examined. If any of these devices has a rule computing this node N, then the nodes needed by this rule that have not already been assigned a distance, and are not in the also-known extension of the input nodes are now assigned a distance D+1. This process is repeated until no more nodes can be assigned distances. The distances thus assigned are the minimal number of devices a computation could pass through.

To determine if QNODE is free, its distance is determined using the distances computed above. If there were output nodes specified, then the QNODE must have been assigned a distance (else fail). A list of "addable" nodes is formed containing those nodes strictly further away than QNODE from the outputs (the list is empty if no outputs were specified). A property of this list is that any constants that might be combined with the input to give QNODE a value will be in the addable list.

A list of nodes currently having values is formed; call it NLST. (In the case of checking FREE-WRT specs, NLST is initialized to the nodes specified as inputs instead of nodes having values). Naturally this list cannot contain QNODE (if QNODE can be determined on the basis of inputs, fail). The free test works by adding new nodes to NLST, and seeing if QNODE ever gets added. If it does, then the test returns "no, QNODE is not free." If no more nodes can be added to NLST (whose elements are those nodes *potentially* computable), then QNODE must be free, so the test succeeds.

Nodes are added to NLST by finding device rules meeting certain requirements. If these can be met, then the node determined by the rule is added to NLST. These restrictions should be considered a relaxation of the normal interpretation of the relation of needed-node to node determined by a rule. Even if a needed is not known, if it is further from the output than the node-determined (and QNODE) it will be considered "known," since it has the potential of being known and having the output depend on it. Furthermore, if there is a 1-device long circuit, as evidenced by some of the terminals needed being connected to the terminal computed, then the system is willing to assume that the circuit can be eliminated.

A slightly different set of conditions is used for complex devices because the rules for a complex device generally do not reflect the entire "computational dependency" story.

These requirements are:

FOR SIMPLE DEVICES

1. Node computed is not already in NLST.
2. Rule does not return a mode.
3. Each node-needed is either in NLST or in the addable list, or equal to the node-determined.
4. Rule does not use any macro-devices (this should probably be moderated!).
5. If node determined is addable, then it must have a smaller distance than any of the nodes needed.

FOR COMPLEX DEVICES

1. Form set of nodes attached to complex device terminals not marked as being either constant-terminals or mode-terminals.
2. If QNODE and a member of NLST are in this set, then QNODE is not free. Test fails.
3. If all nodes in this set are in NLST or addable except one, then add that one to NLST.
4. If all nodes in the set are either in NLST or addable, then select the node closest to the output (using distances previously computable) and add it to NLST.

This completes the description of the free test.

Concluding Remarks

All the basic algorithms have now been described. The next (and last) chapter will "walk through" an example that exercises much of the system's capabilities. Hopefully this will put everything into perspective.

Many of the activities of the system have not been described. For example, detailed explanations have not been given for how device rules are interpreted, how doublets are detected, how nodes and devices are merged, and how time-costs are summarized so they can be compared.

Other back streams and side waters of the system will be mentioned in passing during the next chapter. These include topics like how time-costs are propagated through macro-devices and how code is generated.

Similarly, many of the procedures that *have* been described would be horribly time consuming if implemented as described. The system uses many interlocks, tables of previous results, update lists, etc. to actually end up doing as little as possible.

No apology is offered for these sins of ommission and commission; this document is not an implementation manual but the presentation of a theory of problem solving. Hopefully the reader can imagine solutions to all of the problems mentioned above. Some, no doubt, would be superior to the ones actually used.

TABLE "TRANSFORM TYPES"

REMOVE -- removes a circuit, or more generally creates a new computation path from input to output when the pattern doesn't contain such a computation path. The inputs and outputs refer to the newly created computation path.

SUMMARIZE -- this is a transform that doesn't result in any new nodes. Commutation is a good example of this type.

REDUCE -- reduces the size of a circuit by extablishing a new computation path containing fewer devices. If the new path uses a macro-device, the "count" should include the devices packaged into this macro-device.

RANGE -- This is like a REMOVE transform except that only a computation path involving weak device rules is established between inputs and outputs. MULT-SIGN (used in the square-root example) is a good example of this type.

RESTATE -- None of the above. SINGLE-REC-GEN is an example of this (it doesn't result in a *qualitative* speedup).

SPEEDUP -- This is only used for improving code. No new computation paths are created, except those between nodes already having a computation path. The time-cost of the path from input to output is (qualitatively) reduced.

For writing code, partial matches (except from SPEEDUP transforms) are propagated according to the order above. In general, the types propagated are those down to and including the type responsible for the most recent successful rule application.

TABLE "Enablement Tests for Writing Code"

FOR TYPE-A

1. Outline nodes *could* intersect *assume-unknown set. One node in outline is neither known nor in *assume-known. If no outline, TYPE-A fails.
2. One outline is actually known. (dropped if only one outline node is specified)
3. The outputs are unknown and not in *assume-known. This restriction makes sure that the assumptions about the state of the deduction are consistent.
4. All inputs and outputs are in fact unknown. This guarantees that the computation path could be part of a legitamate circuit.
5. No duplications in outline. This was discussed above.
6. If *reduce-group is non-empty, then computation path (inputs and outputs) intersects *reduce-group. This focuses the attention of the system on one circuit at a time for TYPE-A purposes.

RESPONSE:

Outline not in *assume-known and not actually known added to *assume-unknown. This couples with restriction 1 above.

Inputs and outputs added to *reduce-group. This couples with restriction 6 above.

FOR TYPE-B

1. All outline nodes are known or *assume-known. Compare this to TYPE-A restriction 1.
2. If any outline is in *assume-known, then no inputs or outputs can be in *assume-known. This combines with a side effect below to keep the system from getting distracted by non-productive rule applications.
3. No duplications in outline.
4. All outputs are in fact unknown. This is meant to suggest that the output could lead to computing the desired output of the problem statement.

RESPONSE

Inputs added to *assume-known. This couples with restriction 2 to prevent non-productive rule applications at the same "site" on a circuit. This contrasts with TYPE-A, where multiple applications at the same site are encouraged (A6).

Outputs added to *assume-unknown.

TABLE "Enablement tests for IMPROVING code"

FOR TYPE-A

1. One outline in *asume-known (dropped if only one outline).
2. One outline either not in *assume-known but in nodes-definitely-used, or in *assume-unknown.
3. Output not in nodes-definitely-used.
4. If *reduce-group non-empty, either one of the inputs or outputs is in *reduce-group, or none of the inputs are in nodes-definitely-used.
5. Outline must be non-empty, and no duplicate nodes in it.

RESPONSE:
Inputs and outputs added to *reduce-group.
Outline not already in *assume-known added to *assume-unknown.

FOR TYPE-B

1. All *outline in* nodes-definitely-used or *assume-known.
2. Inputs known or in *assume-known, but not in *assume-unknown.
3. No outline depends on output (or output unknown).
4. Any output not in *assume-unknown but in nodes-definitely-used should depend on inputs This says that a type-b application should shrink previously used paths.
5. Either outline empty, or does NOT intersect *assume-unknown.
6. The inputs intersect either nodes-definitely-used or *improved-nodes.

RESPONSE:
Output added to *assume-unknown Input added to *assume-known

TABLE "PACKAGING PRIMITIVES"

Pushing-value-structures -- at any particular time, nodes in the datum network have value structures. These can be pushed onto an auxiliary list, to be later popped. The node's value structures are replaced with "empty". Note: it turns out that this is never done recursively.

Initialize-a-node -- this sets up a new variable name, and gives the node that variable name for a value. All other facets are similarly initialized.

Previously-computable-node? -- a node can be looked up in the list created by push-value-structures to determine if its value was known when value structures were pushed. If its value was known, then this test succeeds.

Suggest-external-nodes -- This suggests "extra" nodes whose values used to be computable and are guaranteed to be free with respect to the computation paths from the inputs to the outputs (if applicable). Nodes suggested are initialized (see above).

Pop-value-structures -- restores state of world to what it was before pushing.

CHAPTER 4
WALK THROUGH and CONCLUSION

This chapter has two goals. The first goal is to put the various procedures and capabilities of the system into perspective by following its operation while it solves the moderately difficult problem of writing a "Bernoulli number generator." The second goal is to review and speculate about the theory of problem solving presented in the preceeding chapters.

<u>*The problem statement*</u>

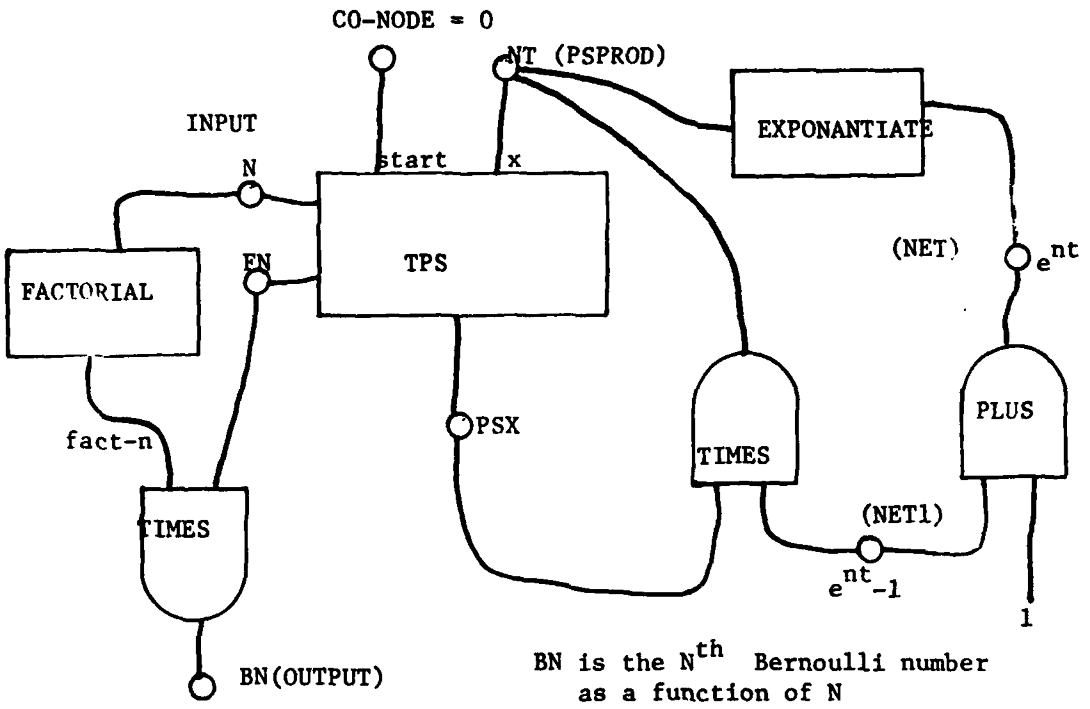
The usual text-book definition of the nth Bernoulli number B(n) is the following claim about the set of Bernoulli numbers:

Sum from n=0 to inf.[B(n)/(n!)]t^n = t/(e^t - 1)

Just to make the problem solution a little shorter, the code-writing problem given to the system will be stated in terms of a power series with the nth coefficient equal to B(n)/(n!). It would be easy to write a transformation rule to make this restatement. Diagram "Bernoulli Problem Statement" shows the original problem network.

On the face of it, the problem statement is asking for the "insides" of a DO-loop to be written, based on the overall result of the DO-loop. The "t" in the definition is universally quantified, but isn't really a parameter: the quantification would be "for all n, exists B(n), for all t" rather than the usual function definition "for all n,t exists B(n)" (where B(n) is a variable, not

DIAGRAM "BERNOULLI PROBLEM STATEMENT"



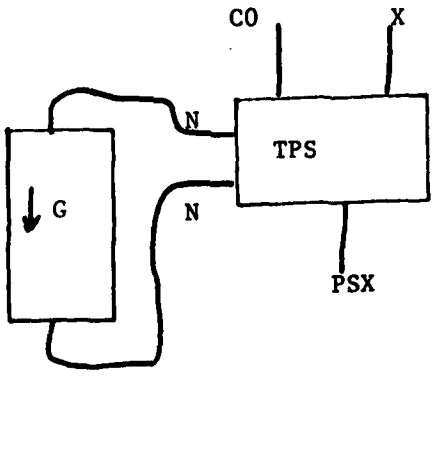
BN is the Nth Bernoulli number as a function of N

$$\sum_{N=0}^{\inf} \frac{B(N)}{N!} t^N = \frac{t}{e^t - 1}$$

This defines B(N). The equation is to hold for all positive t.

- -

Definition of TPS:
convergence is implied



$$PSX = \sum_{N = c0}^{\inf} G(n)\ x^n$$

one rule -- computes PSX from c0 and x.

a Skolem function). The reader is cautioned *against* being concerned with quantification in what follows.

Outline of the solution

The solution to the Bernoulli number generator problem involves applying three transformation rules in the sequence shown:

> 1. Expand the form e^t-1 to a term-wise power series. The transform is named EXP-1-EXPAND.
> 2. Note that "t" is the result of a multiplication of two power series. A new power series is formed for this product. The transform is named TPS-MULT-U.
> 3. Note that if a power series has the property that for all x [P(x)=x], then all coefficients are zero except for the coefficient a_1 (= 1.0) of x.

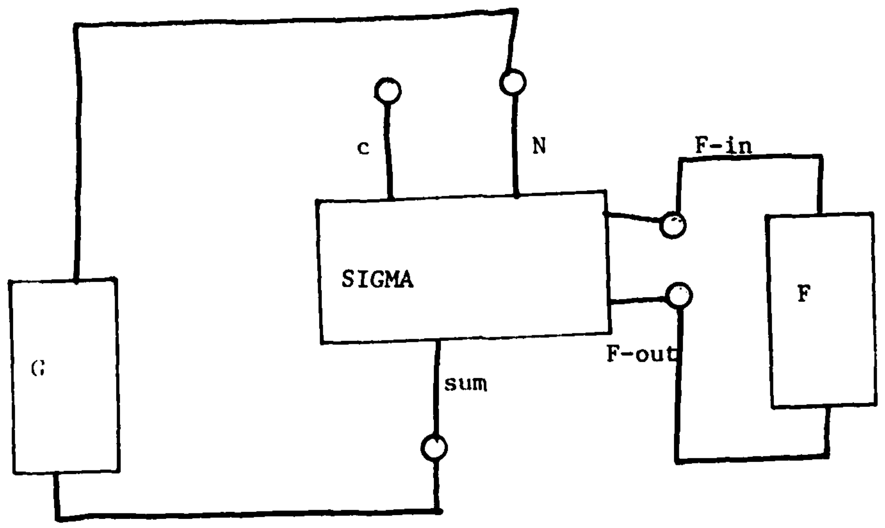
Then a device rule for summation uses the fact that if the result of summing from C0 to N is known for each N, then each term of the summation can be determined.

The summation device used in this problem is closely related to the SIGMA device shown in the diagram "Sigma Device." This device shows the power inherent in the techniques developed within these pages for dynamically configuring looping control structures.

In device SIGMA the decision as to whether the computation from N to SUM is "inside" or "outside" the loop is left until relatively late in the problem-solving effort. While there are other ways to make DO-loops "work" in an EL-like constraint language, no others as yet developed have the capability of defining a device like SIGMA.

The solution to the problem of efficiently implementing the course-of-values recursive function for computing F (in diagram "Sigma Device") will be examined in detail later, when

DIAGRAM "SIGMA DEVICE"



Two rules: Given "c" and "N", compute "SUM"

$$\text{SUM} = \sum_{m=c}^{N} F(m) = G(N)$$

Given "F-in" and "C", compute "F-out" using an equivalent algorithm to the recursive "F":

$$F(c) = G(c)$$

$$F(N+1) = G(N+1) - \sum_{m=c}^{N} F(m)$$

(this only uses F(c),...,F(n) to compute F(n+1)) the solution of the Bernoulli number problem *needs it*. A mathematical treatment of this solution was given in chapter I.

Stating the Problem

The Bernoulli problem is hard to state, and therefore provides a good context in which to compare and contrast this system's approach with the systems built by Burstall and Darlington [Bu77], Barstow [Ba77], and Manna and Waldinger [M79].

Burstall and Darlington's transformation system takes first-order recursion equations as specifications. The Bernoulli problem statement certainly isn't one of these.

Barstow's system takes as its specification the results of McCune's [Mc77] program model builder. These specifications are actually very high level language programs in that they always can be interpreted (however inefficiently). How much effort would be required to convert the Bernoulli problem statement to this form? To start with, the problem would have to be solved, and *then* a high level program would need to be constructed. That is, information gathered while solving the problem may need to be thrown away in order to use Barstow's system.

Manna and Waldinger's system uses a specification technique potentially more powerful than the technique used here, since they can specify side effects. They can "non-constructively" specify a function like GCD:

GCD(x y) <= COMPUTE max{z:z|x and z|y}
where x and y are positive integers

This specification cannot be interpreted as it stands, because without using some other facts there is no obvious upper bound on the values for "z" that must be examined. The braces {..} are set-constructors, and on the face of it the set above is infinite.

Incidentally, this system could state the GCD problem slightly differently:

X positive, Y positive, X|GCD, Y|GCD, X|NE, Y|NE
NE > GCD, and NE has the value (*NOT-EXIST input-spec)

That NE has a value constrains the GCD to be the largest number meeting the other constraints because any value NE larger than GCD also meeting those constraints is known not to exist. Naturally, something like a set-constructor could also be created.

The situation with the Bernoulli problem is that the output of the Bernoulli function is specified in terms of the behavior of the *entire* function, not just by the relation of the output to the input. This distinction is similar to the distinction between first and second-order predicate calculus.

In conclusion, none of the three systems discussed can even state the Bernoulli number problem, and only Manna and Waldinger's system has the potential for a "slight extension" that will let it express the problem.

Convergence and Primitive Recursion

In order to express the Bernoulli number problem and the concepts required for its solution, the system must come to grips with notions like the convergence of power series. Burstall [B69], Floyd [F67], Manna and Waldinger [M78], [M79], and others suggest using *well-founded* sets to prove termination. Even if one can convert a question of a power series converging into a question of a program terminating, finding a well-founded ordering of the real numbers so that a proof can be constructed is as hard as proving convergence in the first place.

The following explains one way to approach the problem of convergence within the "network of constraints" formalism.

The TPS (term-wise power series, see table "TPS Device") device has a rule for computing the value on its PSX terminal using terminals START and X, and uses a macro device from terminal N to terminal FN (called G in diagram "Bernoulli problem statement"). A power series could also be defined by giving a way to compute the "next" coefficient (such an "incremented-term-wise power series" device has not been implemented).

In order to determine time-costs, and in order to guarantee that the programs written by the system terminate, it is necessary to specify an upper bound to the number of times the body of a loop will be executed, or the maximal depth of a recursion. This means that any computation path contained in a network is expressed in a primitive recursive form. There are functions, like the well-known Ackermann's function and the LISP interpreter, that are not expressable in a primitive recursion form.

This limitation isn't as serious as it seems. Most practical non-primitive recursive functions look something like

```
(DO-FOREVER (SETQ X (READ)) (P X))
```

where P is primitive recursive. Language interpreters are not primitive recursive because they must interpret programs that are themselves not primitive recursive.

Another apparent problem concerns the fact that primitive recursive functions written in a primitive recursive language are (in general) much longer than they would be if written in a general recursive language (like LISP). But this isn't really applicable to the synthesis system because the language it writes code in *is* general recursive.

The real difficulty in only working with primitive recursive procedures becomes apparent when trying to write a device rule for TPS. Some power series are uniformly convergent (within the region of interest):

> $S_m(x)$ = (the usual partial sum) = Sum from n=0 to m: $a_n x^n$
> $S(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_n x^n + \dots$
> If for any E, there is an M such that
> $$|S(x) - S_M(x)| < E$$
> then S(x) is *uniformly convergent*.

Even knowing that the power series described by the TPS device is uniformly convergent doesn't tell how fast the series converges.

The decision was made to avoid the convergence problem by *fiat*: the device rule claims that the number of terms required is bounded by the square of the number of bits of accuracy required (compare to Newton's method where the upper bound of the number of terms is the log of the number of bits of accuracy, when it converges). Before meeting this decision with too loud a cry of outrage, consider two points. First, in the Bernoulli problem statement it is impossible to tell how fast the series will converge because the function from N to B(N) hasn't been discovered yet! Second, there isn't anything wrong with using a special construction in cases where a series is known to converge as fast or faster than a certain rate.

Closely related to the rate of convergence is the problem of how the series converges. In the simplest case, one would like the remaining error to be no more than the last correction.

$$|S(x) - S_{M+1}(x)| < |S_M(x) - S_{m+1}(x)|$$

(By the way, this condition alone does *not* guarantee convergence). If this is true, then the general recursive form of the approximation function for S can examine the size of each term to tell if the desired accuracy has been obtained. Unfortunately, this turns out not to be true for the Bernoulli numbers. B(odd-number) is zero except for B(1)=-1/2.

The solution to this problem selected for the TPS device is to refuse to "believe" that the significance of each new term is falling off faster than an exponential decay. Again this is a reasonable, but not universally applicable assumption (if a term's significance falls off very much slower, then a string of 0.0 terms would cause the system to stop summing terms earlier than it should have). The TPS device rule is shown in table "TPS Device".

The problems of rate of convergence and manner of convergence are apparently unavoidable and unsolvable in trying to write a general purpose power series evaluation routine. This illustrates why an automatic program synthesis system is needed. That the system being described *can* handle problems of convergense (at least in some situations) has already been demonstrated in the Newton's method solution to the square-root problem.

Finding an Algorithm

The system breaks the code synthesis task into two distinct phases: finding an algorithm, and implementing the algorithm. The first phase sometimes involves a lot of deduction. The second phase involves little more than propagating CODE-EXPression facets.

Initialization of the Deductive Process

The problem statement is given to the system by first defining the problem network BSPEC (in the diagram "Bernoulli problem statement") and then asking the system to encode a particular computation path from the network terminal N (corresponding to the internal node of the same name) to the terminal BN. The system is further instructed to call this function "BERNOULLI":

```
(define-network bspec (n bn)
                ((tps (*constant 0.0) psprod psx n fn) deftps)
                (factorial n factn)
                ((*c factn fn bn) oprod)
                ((*c net1 psx psprod) dtpsprod)
                (exp psprod net)
                (+c net1 (*constant 1.0) net))
(encode-network 'bspec 'bernoulli '(n) 'bn)
```

After the function ENCODE-NETWORK has completed its effort, in addition to a new LISP function BERNOULLI having been written, there will be a new complex device of type BSPEC with two terminals N and BN available for use in other networks.

After removing value structures (except values noted as being inputs like the constants

0.0 and 1.0), the input node N is given a value

(*VARIABLE N).

The normal assumption list initialization gives

*ASSUME-KNOWN empty
*ASSUME-UNKNOWN BN
*REDUCE-GROUP empty

Value propagation takes place, and only the node FACTN can be given a value. Although chapter 2 showed how the factorial function could be synthesized and a complex device created, here factorial is treated as a primitive.

Value propagation is a two step operation (repeated until no more propagation can be accomplished), *the first propagating* LBOUND, ERROR-BOUND, and UBOUND facets independently, and the second propagating TIME-COST, NODE-USED, and NVALUE in that order. If a TIME-COST facet is propagated, then the resulting rule-closure is used to propagate NODE-USED into the same node. If that succeeds, then NVALUE is propagated, again using the same rule-closure. In what follows, the LBOUND, ERROR-BOUND, and UBOUND facets are never important.

The TPS device has a macro-device specification going from (nodes) N to FN, and N is known while FN is unknown. This configuration satisfies the conditions for the outward macro-device assumption extension rule (in chapter 3, repeated here):

> ***Outward Macro-device Assumption Extension:*** If any of a rule's macro-device input nodes are known, then add the rule's needed nodes to *assume-known. Furthermore, if the macro-device's output node(s) are unknown, add the rule's output to *assume-unknown.

The node PSPROD (on the "X" terminal of the TPS device) is added to *assume-known, and the node PSX is added to *assume-unknown.

Matching (First Rounds)

Since no more value propagation can take place, partial matches are started for REMOVE type transforms. None of these partial matches can succeed at this point. SUMMARIZE type transforms are tried next. Three partial matches are completed, and psuedo-devices are created.

These transforms are all for performing commutations: two for multiplication and one for addition. There are, of course, other ways to tell a system (but not *this* system) about commutative rules. The disadvantages of the approach used here are that the datum network (the "data base") grows, causing the number of partial matches to increase. Another disadvantage is that simply adding these commuted devices takes a while. The advantage is a simpler, more uniform representation of knowledge.

The system performs a round of partial-match propagation for each psuedo-device expansion (transform application) that actually does something (either adding a non-duplicated device or merging nodes). When trying to expand psuedo-devices, all unexpanded devices are examined. In what follows, this subtlety will normally be glossed over.

Again, no REDUCE matches are found, but some SUMMARIZE transforms succeed. These are transforms trying to commute the commuted devices. As soon as each of these twice-commuted devices is added to the datum network, duplicate devices are detected and merged. Part of the merging process involves combining lists of partial matches already tried. This means that since one of the devices being merged (the original) has already had a commutation transform started in it, the result of the merge will be immune to commutation.

There is one more hurdle to go over before the system "gets down to business." The pattern formed by a pair of commuted devices matches two different associativity patterns, each two different ways. Since there are three such commuted pairs, there are twelve REDUCE transforms potentially applicable. The situation is shown in diagram "Why no Duplicates in Outline." All enablements for these situations fail (regardless of the state of the deduction).

Using the power series for exponentiation

A well-known power series expansion for exponentiation is:

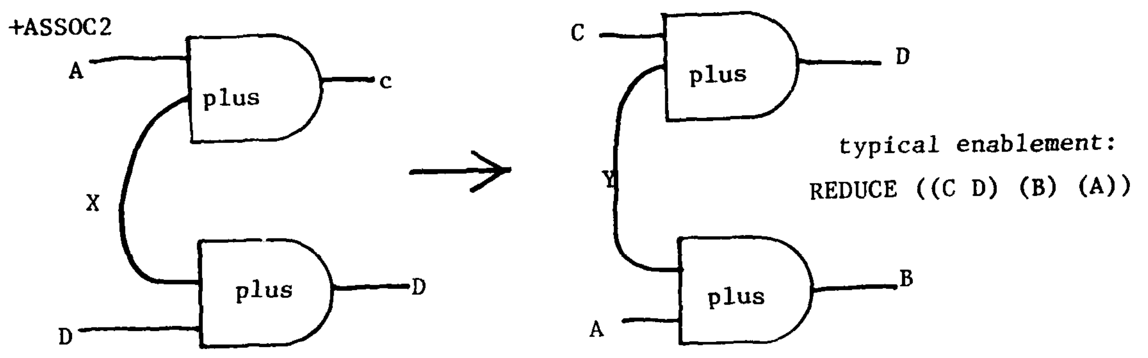
$$e^x = 1 + x + x^2/2! + \ldots + x^n/n! + \ldots$$

In the Bernoulli number problem, there is a form $e^x - 1$ with its own power series expansion derived from the above by ignoring the first term. This power series is encoded into a transform EXP-1-EXPAND with enablement as shown (see diagram "EXP-1-EXPAND"):

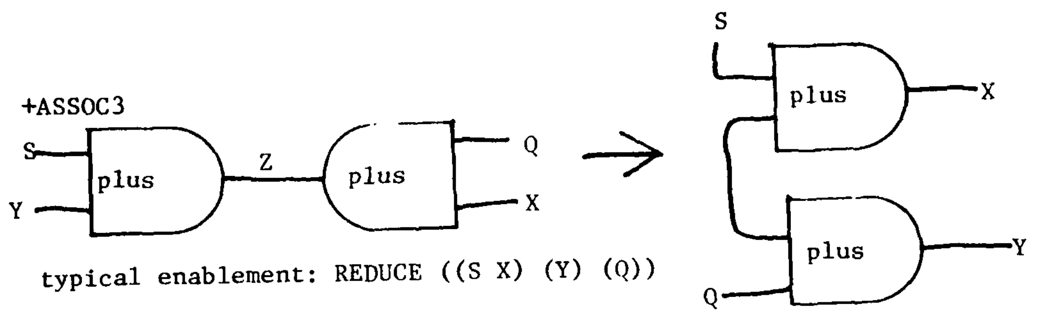
```
(define-transform exp-1-expand (t et-1)
                  ((exp t et)
                   (+c et-1 (*constant 1.0) et))
                  ((tps (*constant 1.0) t et-1 node-n node-fn)
                   (factorial node-n n-fact)
                   (*c node-fn n-fact (*constant 1.0))))

(define-enablement exp-1-expand restate (nil (t) (et-1)))
```

The pattern of this transform matches the problem network, the enablement fails TYPE-A (see

DIAGRAM "WHY NO DUPLICATES IN OUTLINE"

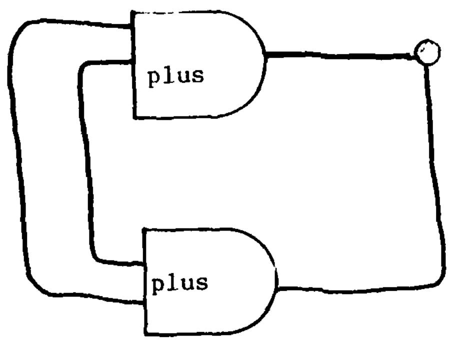


typical enablement:
REDUCE ((C D) (B) (A))



typical enablement: REDUCE ((S X) (Y) (Q))

PAIR OF COMMUTED DEVICES:

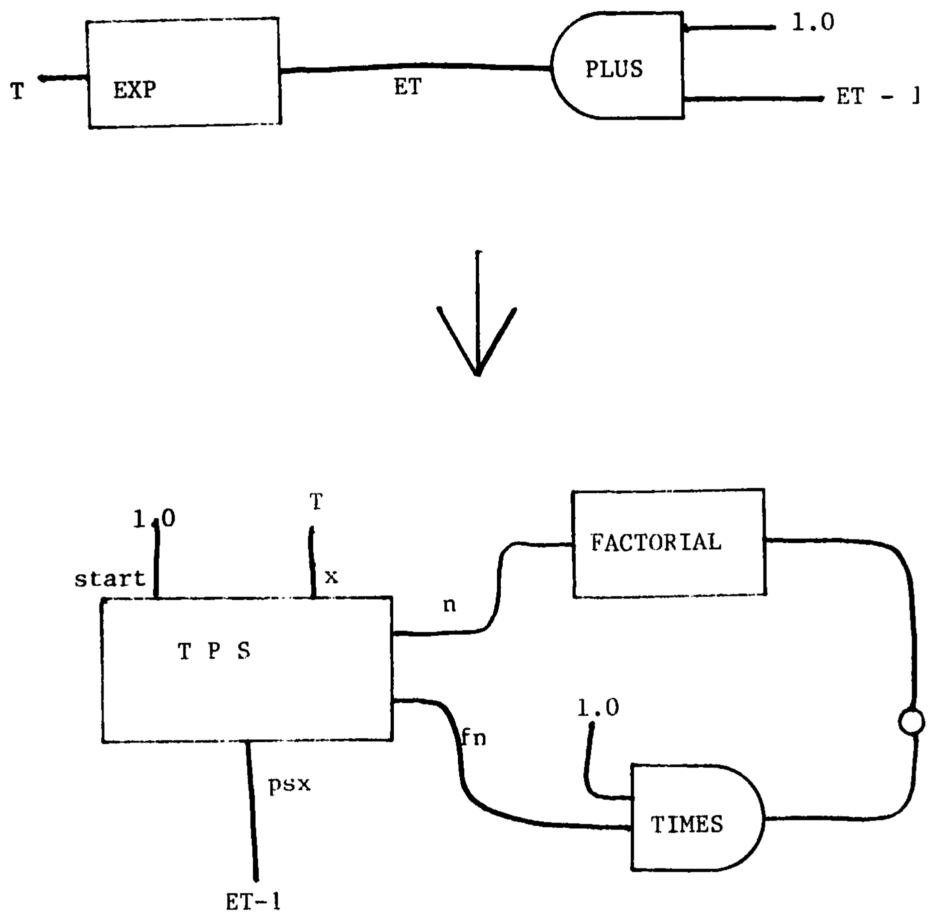


(seed from above can match either device in this pattern)

+ASSOC2
A = B
C = D
fails: duplicate outline

+ASSOC3
S = X
Q = Y
fails : duplicate outline

DIAGRAM "EXP-1-EXPAND"



enablement: RESTATE (NIL (T) (ET-1))

table "Enablement Tests for Writing Code" in chapter 3) but passes the TYPE-B application criteria. In addition to adding three devices to the datum network, node NET1 is added to *assume-unknown (by the way, NET1 was flagged as a "wishful thinking" extension). No other assumption extensions are made. Diagram "Bernoulli after EXP-1-EXPAND" shows the important parts of the datum network after this transform has been applied.

What would happen if a transformation EXP-EXPAND (also a RESTATE type of transform) for expanding the exponentiation device were present in addition to EXP-1-EXPAND?

```
(define-transform EXP-EXPAND (t et)
                  ((exp t et))
                  ((tps (*constant 1.0) t et-1 node-n node-fn)
                   (factorial node-n n-fact)
                   (+c et-1 (*constant 1.0) et)
                   (*c node-fn n-fact (*constant 1.0))))
```

The instantiated +c device from EXP-EXPAND could immediately be merged with the problem device

(+c net1 (*constant 1.0) net)

as a result of doublet processing. Unfortunately if both EXP-EXPAND and EXP-1-EXPAND were applied, the two resulting TPS devices would not currently be merged due to implementation inadequacies involving macro-device indexing.

Using (instead of the above)

(TPS (*constant 0.0) T ET NODE-N NODE-FN)

would be more painful because a later rule pattern (TPS-MULT-U) would no longer match until after a rule pulled the *subtraction into the* TPS "loop."

In the previous chapter a suggestion to measure "amount" of "wishful thinking" was briefly discussed. One reason to believe something like that would be handy is illustrated here. EXP-1-EXPAND involves *less* wishful thinking to get form NET1 to PSX than EXP-EXPAND would in getting from NET to PSX.

Another way to think about expanding the exponentiation device is that the EXP device could be defined as a complex device with both rules (as it has now) and a defining network. The issue involved in expanding user-defined complex devices have not been explored, but it would seem that these expansions should take place *after* matching RESTATE transforms. As mentioned in the previous chapter, any complex devices actually used for writing code are expanded as part of improving code.

This same issue also arises when considering the FACTORIAL device. In chapter 2 there was a demonstration showing how this function could be written from a specification. This device could also be written as a complex device (or system-defined device). The potentials of using complex devices have not been explored.

Multiplication of Power Series

After worrying about commutation a while, the system again looks at the partial matches of REDUCE type transforms. There are two ways *the pattern network* of the TPS-MULT-U transform (see diagram "Transform TPS-MULT-U" and table "TPS-MULT-U") can match the datum network. The two resulting partial matches are both instantiated, creating two psuedo-devices. The psuedo-device from the partial match with the seed TPS device matching the original TPS device fails the U-variable specification because the constant 1.0 is *not* unknown.

If the TPS-MULT-U seed device matches with the datum TPS created by EXP-1-EXPAND, then the U-var[illegible]le search succeeds with UFY -> BN. The enablement for this transform meets the TYPE-B criteria, so expansion of the psuedo-device takes place. A new complex device type is created for computing (l/x!). A device of this new type is added to the datum network according to the transform's macro-device specification along with copies of the devices in the transforms instantiation network.

The assumptions are not modified as a result of applying this transform.

Although the TPS-MULT-U transform looks like a handy transform for multiplying power series together, using it this way is made difficult by the presence of the U-variable specification. In order to write a power series multiplication rule (call it TPS-MULT), about all that would be needed is to drop the U-variable specification and change the SIGMA2 device (defined below) to the SIGMA device discussed earlier. Unfortunately, it turns out that the suggested rule TPS-MULT does not lead to a solution of the Bernoulli number problem.

The SIGMA2 device is defined as shown in table "SIGMA2". Note the use of embedded *DO constructions and the multi-directional macro-device specification. This device is a close relative to the SIGMA device discussed earlier. It is used here because the terms

DIAGRAM "BERNOULLI AFTER EXP-1-EXPAND"

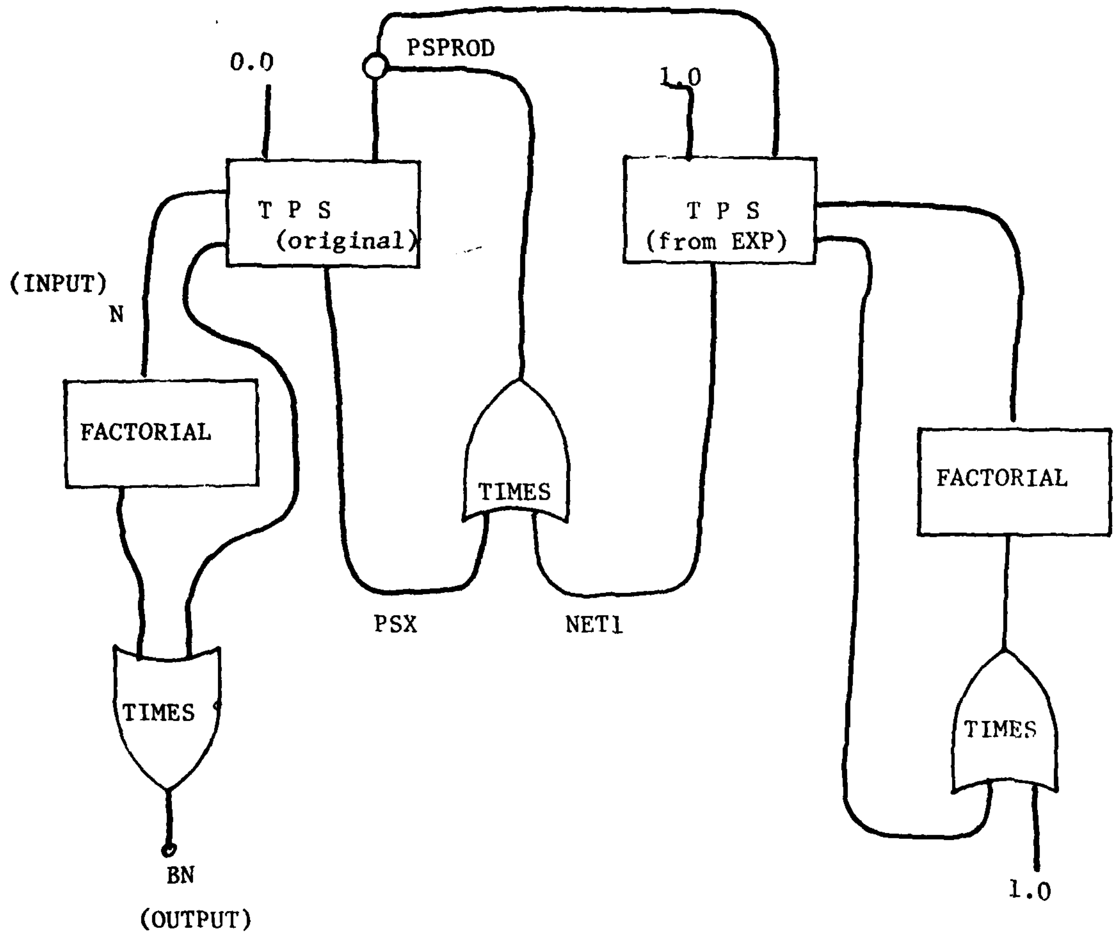
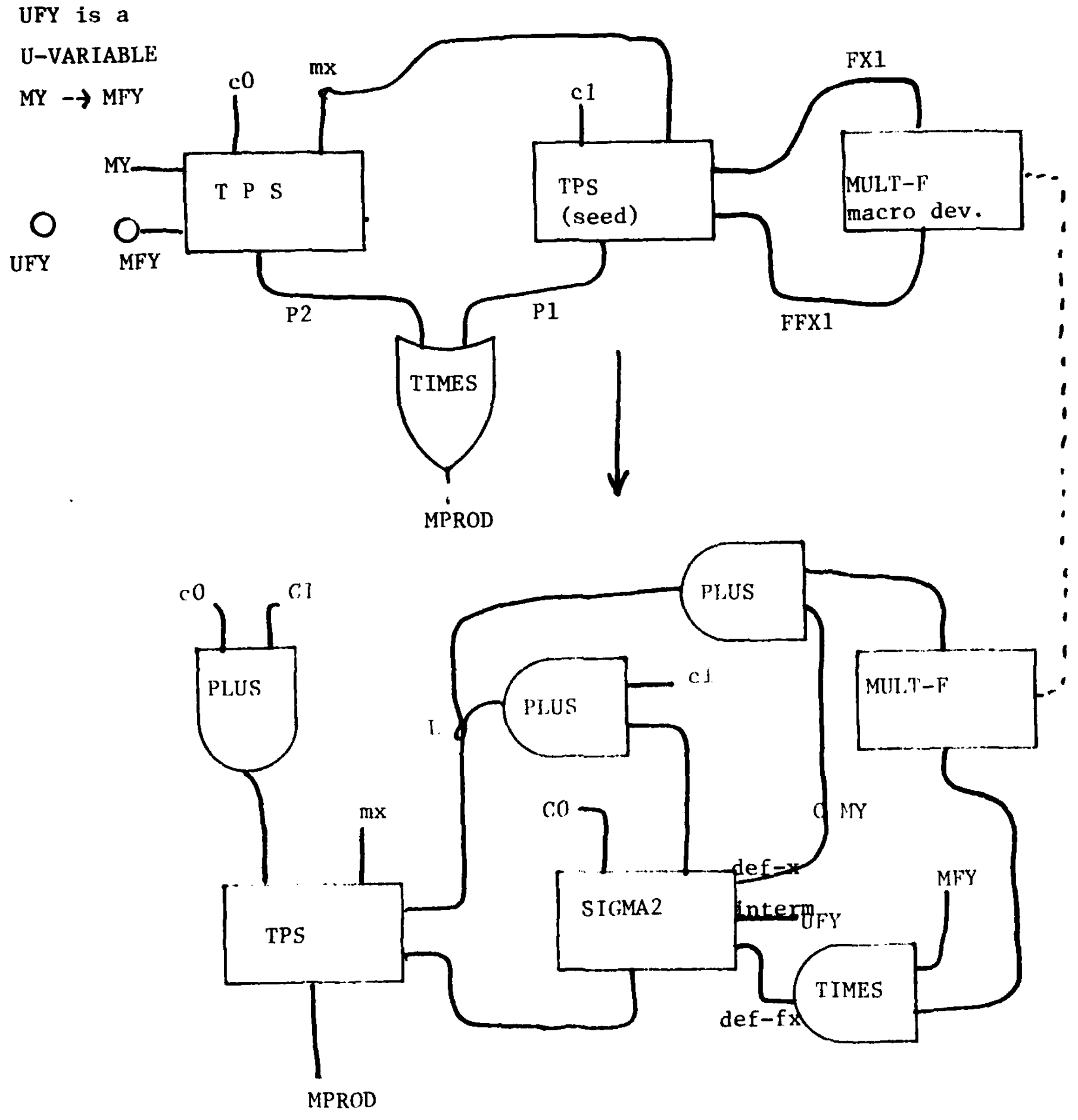


DIAGRAM "TRANSFORM TPS-MULT-U"



enablement: REDUCE((CO C1) (MY) (UFY))

being summed depend not only on the summation index (in MY, terminal DEF-X) but the value of K (or L) on terminal TO. The explicit claim about SIGMA2 is that the value on terminal INTERM depends only on the value on the DEF-X terminal, and *not* on terminal TO.

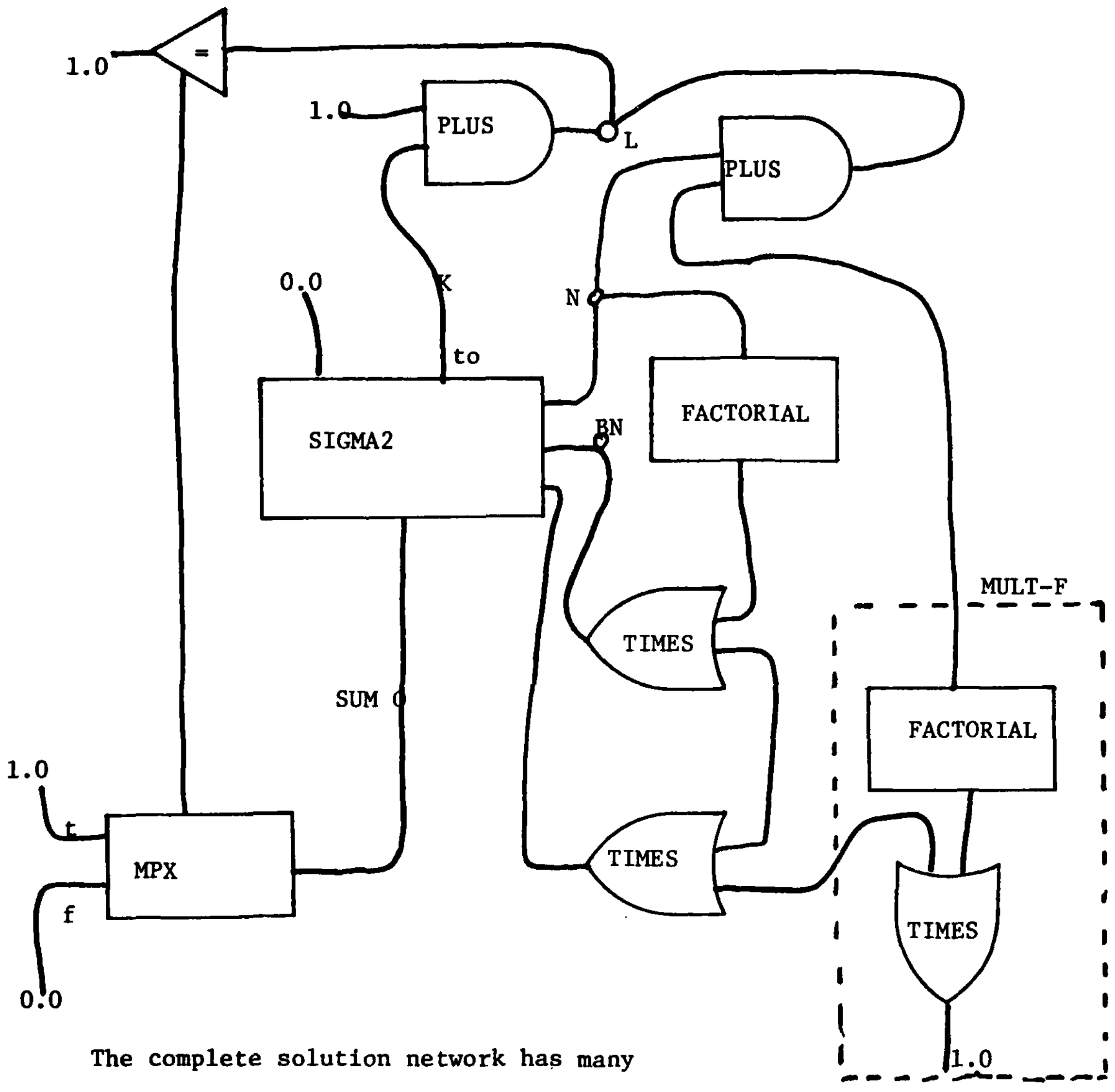
The SIGMA2 rule for computing INTERM uses data structures. Discussion of this topic will be postponed.

Completing the Deduction

Only one more transform application is required to obtain a solution network for the Bernoulli number problem. The TPS-CONSTANT-COLLAPSE, of type REMOVE, is shown in table "TPS-CONSTANT-COLLAPSE." To see how the pattern configuration arises, notice that when the TPS-MULT-U transform was applied, MX -> PSPROD, MPROD -> PSPROD so a TPS device was created with its "input" and "output" tied together.

The significant parts of the solution network, after applying TPS-CONSTANT-COLLAPSE, are shown in diagram "Bernoulli Solution." Remember that the solution network has many more devices than shown in this diagram.

As a result of applying this transform, *assume-known is extended to include node I, and *assume-unknown is extended to include node SUM. But it hardly matters, because the problem is solved by virtue of the rules in SIGMA2: the rule claims to compute a value for the node on terminal INTERM, and as shown in diagram "Bernoulli Solution," this terminal is



The complete solution network has many more devices than shown above.

connected to node BN; BN is the node for the solution.

DATA STRUCTURES AND CODE GENERATION

The SIGMA2 device has a pre-packaged solution to a typical data-structure design problem: speed up an implementation of a function defined by course-of-values recursion. There are several possibilities including association-lists, stacks, and (used here) arrays.

How does SIGMA2's rule work? A variable P is initialized as an empty vector (but the synthesizer "thinks" P is a floating point number) containing some number of elements. The vector runs from the value on the terminal FROM (in this case 0.0) to the value on the terminal DEF-X (in this case the input N). The first element in the vector is also assigned

Vectors (in this "toy" implementation) are manipulated by three functions:

```
(CREATE-VECTOR from to initial-value) => vector
(STORE-VECTOR vector index value) => vector
(ACCESS-VECTOR vector index) => value (or error!)
```

The deductive system only has two "types" of values: floating point numbers (of unspecified precision) and mode values (currently only *TRUE and *FALSE, though all code is completely general). This type-less approach is inadequate for any domain except the essentially untyped domain of numerical programs investigated here. At the very least one needs to distinguish among integers, floating point numbers, character strings, and complex data structures. A possible way to represent this "type" information is as a separate facet in the value structure. The proposed way to design data structures is an elaboration on this idea

Both data-structures and control-structures are in a sense prepackaged in the device rules. Two methods of combining these prepackaged control-structures are used: concatenation and packaging via macro-devices (and the associated multi-return facility to be discussed shortly). What would it mean to combine prepackaged data structures?

The accessing pattern packaged in SIGMA2 might be described as follows

1. Size of *indexed* structure known when it is created.
2. Elements of structure are floating point numbers.
3. Stores occur in sequence, starting at the bottom index.
4. Accesses ("fetches") occur in sequence, starting at the bottom and going all the way to the most recently stored.

A data structure with these properties can be implemented in various ways including a LISP list (do a RPLACD at the end to add a new element), or an array A more interesting and efficient implementation uses a block of contiguous memory and two pointers. The first pointer is the "next fetch" pointer, and the second is the "next store" pointer. The beauty of this implementation is that the only arithmetic operation needed for manipulating the pointers is "increment by 1" (usually a very cheap machine instruction). General vectors and arrays do not have this nice property Such a structure might be called a file.

How could a "data structure designer" be added to the synthesis system? As a speculation, suppose descriptions similar to the one given above for the structure used by SIGMA2 could be propagated much like values are currently propagated. As this description was propagated through the network, and used in computing "cost", it could be modified by relaxing various restrictions (for example, the "fetches become non-sequential but always in increasing index). Perhaps at the end of the deductive process, after the output has received a value but *before* code-expressions are propagated, the data structure specifications could be examined. There are probably only a few fundamentally distinct data structures, so it might be possible to do the design almost by table lookup.

This approach should be contrasted with Barstow's [Ba77] method of refining along (essentially) an "implements a kind of" hierarchy. Rosenschein and Katz [R77] discuss notions similar to the proposal here in a user-interactive context.

The view suggested here is that the synthesizer should worry about the influence of control-structure decisions on data-structure design, rather than data-structure decisions on control-structure design. Several investigators have advocated concentrating on data over procedures. Hewitt [H79] shows how control structures can be viewed in terms of the interaction among datum. Dennis [D75] discusses a similar view of computation in terms of data flow. Liskov and Ziles [L75] discuss language design issues inherent in this "data first" view.

I believe something along the lines sketched out above could be made to work despite the large number of open problems involved. I also believe this is the most important direction in which to extend this research.

Generating Code

At this point, node BN (the desired output) has received a value structure with expressions for its TIME-COST (see table "Bernoulli Time-cost"), NODE-USED, and NVALUE (see table "Bernoulli Nvalue") facets. Some peculiar things were done to obtain its time-cost; these will be discussed shortly. The system is now ready to write code. Of course, this is not the first time the system has generated code while solving the problem: it wrote code for the MULT-F macro-device when the TPS-MULT-U transform was applied, and it wrote three subroutines when finding macro-devices for the sigma2 device. The code for these functions (it isn't very pretty) is shown in table "Bernoulli other code." The resulting code for computing Bernoulli numbers is shown in table "Bernoulli code."

When code fragments are being propagated, the system knows how many times each node was used. If a node was used more than once, the first time its value is used, a SETQ is constructed and the code fragment in the node is modified to be the (gensym) variable SETQed. Unfortunately this problem has no examples of this.

SIGMA2 does not use a macro-device with more than one output specified, so the multi-return facility was not employed. When this is used, the code written for the first of the outputs includes SETQs to temporary gensymed variables for all of the other outputs. If the macro-device is used and the first output is specified, then the nearest PROG variable list has the temporary variable names added to it. Code is also generated for all the other outputs specified when the macro-device is created. However, before "plugging in" these code fragments, a check is made to see if the most recent use of the macro-device computing the first

output involved the same input forms, and if so the proper temporary variable is used instead. The code that eventually emerges is correct but (according to some) ugly because it uses a non-local variable.

Lambda applications result from using macro-devices (in a round-about way). To make the code look a little more attractive, a form involving a lambda application is massaged by checking the arguments to the ' mbda, and noting any that are *atomic* (either a variable name or a number). These atomic arguments (if any) are used to rewrite the lambda form to reduce the number of arguments taken by the lambda. One can see this by close inspection of the tables of code.

The above operation is an example of the kind of improvement found to be more easily accomplished on the resulting code than during the propagation process. Another improvement performed as a post processing step is so-called "constant folding." If all of the arguments to a LISP function are numeric, then the function is evaluated and the form replaced by the result. COND forms are also improved by constant folding and dead-code elimination. Earnest [E74] gives a good summary of code optimization (sic) techniques.

Another kind of improvement *not* performed involves repeated terminal references in a device rule. The code has several instances of the following:

```
(COND ((= (RFACTORIAL gensym) 0.0) (ERROR))
      (T (QUOTIENT 1.0 (RFACTORIAL gensym)))).
```

This code fragment results from the rule for the *C device:

```
(RULE-OF *C A (B C)
(*CASES ((PRIM= B (*CONSTANT 0.0))
         (*TRUE (*CASES ((PRIM= C (*CONSTANT 0.0))
                         (*TRUE (*ANOMALOUS Z-DIV-Z))
                         (*FALSE (*NOT-EXIST DIV-BY-ZERO)))))
         (*FALSE (PRIM-DIV C B)))))
```

Notice that the device rule uses the terminal more than once. It would be easy enough to detect this situation and issue a SETQ or build a LAMBDA expression, but the current implementation doesn't handle this in the best way.

Why does the test to see if the factorial of a number is equal to 0.0 occur? If there were a weak rule for the factorial device noting that the lower bound of the output is 1.0, then this test would have been eliminated.

Packaging Code for a Complex Device

In order to be able to use complex devices for value propagation, a minimum of four facets must be handled: TIME-COST, NODE-USED, NVALUE, and CODE-EXP. Of these, three are quite easy, given that the code for the defining network has been written and given a name. The methods used are outlined below.

NODE-USED -- combine node-used expressions of the input terminals
NVALUE -- return a form applying the name of the function to the NVALUE interpretations of the arguments.
CODE-EXP -- return a form applying the name of the function to the CODE-EXP interpretations of the arguments. Actually, the lambda expression for the function definition is used, instead of the name so that the various improvement techniques can be applied.

It would not have been too difficult to be able to package knowledge about bounds (LBOUND and UBOUND) but the current implementation does not have this capability.

Information relating the name of the LISP function written for a complex device, the defining network, the device type, and how the various facets are to be propagated is put together in a structure indexed by a atomic encoded function name (usually gensymed). Encoded functions can be used, for example, in NVALUE-like expressions.

The Truth about TIME-COST Propagation

Propagating TIME-COST through a complex device turns out to be difficult. Consider the macro-device formed for MULT-F (see diagram "Bernoulli Solution"). The time cost of computing RFACTORIAL turns out (in this system) to be proportional to the size of the input. But the size of the input is not known when the complex device is created.

Instead of a TIME-COST expression, a summary is recorded (along with other information) when an *encoded function* is defined for a network. This summary (type and one or two numeric parameters) gives the "order" of the time cost according to the following classification scheme ("Lg" is Knuth's suggested abbreviation for "Log base 2"):

CONSTANT	A
POLY-LOG	$Bx^{A}Lg(x)$
POLY	Bx^{A}
POLY-EXP	$Bx^{A}2^{x}$

In the above, A and B are numeric parameters, and "x" is used to represent the most significant input terminal(s). For example, if the time cost depended on the product of two input terminals, then the summary would be like x^2. The system summarizes Lg(Lg(x)) as a constant, and a similar *faux pas* is used to close the summaries under each operation.

The system's treatment of complex devices with regards to TIME-COST analysis is similar to the style of analysis humans perform. In order to propagate a TIME-COST through a complex device (or, equivalently, the "device" found for a macro-device specification), an expression is built using the complex device's summary and the NVALUE of the "principle terminal." The principle terminal of a complex device found for a macro-device specification is (arbitrarily) that terminal corresponding to the first argument of the specification

The summary algorithm is one of the two techniques used for comparing TIME-COST expressions. The other uses "typical values" for inputs and does a numeric comparison of evaluated expressions. These two analyses will not always agree!

Another peculiarity about TIME-COST propagation concerns cases when the body of a loop (either iterative or recursive) has a time cost depending on the value of a loop variable. This problem arises in the Bernoulli number example (see table "SIGMA2"). There is no way to produce an accurate time cost in these situations without possibly introducing iterative constructions into time-cost expressions. While there is no theoretical reason why this could not be done, in practice these expressions would be extremely clumsy to deal with. As an engineering judgemental design choice (read "kludge") any *DO-VARIABLE form in a time-cost expression is arbitrarily replaced by the expression for "maximum number of times the loop body will be executed." The final time-cost for the solution to the Bernoulli number problem (see diagram "Bernoulli Solution") is in table "Bernoulli Time Cost." The system discovered that the code ran like N^3. The final code is in table "Bernoulli Code." This code has defects similar to those noted for the Newton's method SQRT code.

The classical algorithm for finding Bernoulli numbers is the N^3 algorithm found by the system. There are much better algorithms. The Bernoulli problem is an instance of a more general class of problems involving the reversion of formal power series: Given a power series S(x) in terms of its coefficients, find the coefficients for another power series P(x) so that S(P(x))=x. Brent and Kung [B78] show that using Newton's method yields an $O(N^{5/2})$ algorithm, and also using Fast Fourier techniques gives an $O((N \lg N)^{3/2})$ algorithm for solving power series reversion problems.

CONCLUSION

This report has described a system for writing and improving code described by (essentially) input/output specifications. Although the system has solved a variety of coding problems, it should *not* be called an expert at this task in the domain of numerical programs. That was not the goal of the research.

The goal was to explore a different way to build an automatic problem solving system. The goal was to construct a system robust enough to solve several kinds of problems in a complex domain, and still have the property that the solution effort was "coherent." This goal has been obtained.

Coherent behavior was obtained from local, axiom-like rules by first restricting the expressive power of these rules, and then making use of two consequences of this restriction. The first consequence was that "backtracking", "guessing", splitting the data base to handle disjunctions, and other operations to accomplish the same task were made unnecessary because disjunctions were eliminated. For the same reason tautologies in the restricted rule language could be recognized in polynomial time. The second consequence was that the rules were simple enough to automatically predict their effect on the state of the solution effort.

By deriving an abstract characterization of the rule's effect on a problem network in terms of its effect on circuits and computation paths, possible rule application sites could be ordered in terms of a selection criteria. The classification scheme, acceptence criteria for the two types of application, and the methods of modifying the problem state descriptions are basic results of this research

Another basic result concerned a way to represent recursive and iterative constructions within the network of constraints formalism. This method involved dynamically finding portions of the problem network to "fill in blanks" in a pre-packaged schema involving both control structures and data structures. The advantages of this approach are two: complex time-cost projections can be easily encoded in the system's rules, and manipulation of things like power series became surprisingly easy (the Bernoulli example in this chapter used only a few short rules).

The system's computational mechanisms heavily use a propagation process in one guise or another. Propagation processes have the important property that they easily lend themselves to a parallel-processor implementation (provided, as in this system, no "race conditions" occur because of backup). More traditional goal/subgoal deductive systems do not have this property. As VLSI technology makes multi-processor processors financially more attractive, this consideration will become increasingly important.

Although primarily concerned with problem-solving in a particular domain, the deductive procedures, control mechanisms, and representational scheme should be adaptable to other domains that share the following properties with the programming task:

1. The "answer" is an arrangement of steps.
2. The steps generate and/or modify objects of some sort.
3. Questions can be asked of some objects.

Even in domains that do not have the properties above, use can be made of the key observation that if the knowledge representation scheme is *weak* and *inexpressive*, then the deductive system can use the fact that certain things can*not* happen to be more efficient.

Table "TPS Device"

```
(make-device-type tps start x psx n fn)
(rule-of tps psx (start x)
         (*do ((prim+ (*constant 36.0) (prim* x x))
               (prim-less (*do-variable mva)
                          ;;Insist on six decimal degits of accuracy
                          (prim* x (*constant 0.000001)))
               (*do-variable sum)
               ;;Note well: first pass is setup only
               ;;indx goes from start to whatever
               ;;nt is the new term. it is added to sum, and averaged into mva
               ;;mva is a 4-wide moving average. it starts out at x (not the first term!)
               ;;xprod starts at x^start, and gets multiplied by x at each step
               (nt (*constant 0.0)
                   (prim* ((*macro-device ftps fn)
                           (*do-variable indx))
                          (*do-variable xprod)))
               (xprod (expt x start) (prim* (*do-variable xprod) x))
               (indx start (prim+ (*do-variable indx) (*constant 1.0)))
               (sum (*constant 0.0)
                    (prim+ (*do-variable sum) (*do-variable nt)))
               (mva (prim* (*constant 4.0) (prim-abs x))
                    (prim+ (prim* (*constant 0.75)
                                  (*do-variable mva))
                           (prim-abs nt))) ))
         ;;Macro-device specification:
         ((ftps (n) (fn))) )
```

TABLE "TPS-MULT-U"

```
(define-transform tps-mult-u
                  (c0 c1 mx mprod fx1 ffx1 my ufy mfy)
                  ((tps c1 mx p1 fx1 ffx1)
                   (tps c0 mx p2 my mfy)
                   (*c p1 p2 mprod))
                  ((tps c12 mx mprod 1 y2)
                   (+c c1 c0 c12)
                   (+c k c1 1)
                   (+c my y1 1)
                   (*c mfy fy1 sig)
                   (sigma2 c0 k y2 my ufy sig)) )
 (define-transform-macro tps-mult-u
                   (mult-f (fx1) (ffx1))
                   (y1 fy1))
 (define-u-variable-node tps-mult-u
                         (ufy (my) (mfy)))
 (define-enablement tps-mult-u reduce
                   ((c0 c1) (mx) (mprod))
                   ((c0 c1) (my) (ufy)) )
```

TABLE "SIGMA2"

```
(make-device-type sigma2 from to result def-x interm def-fx)
(rule-of sigma2 interm (def-x from)
          (*do (def-x
                (*mode-expression (prim= (*do-variable count) def-x))
                ;;return
                (access-vector (*do-variable p) def-x)
                ;;iteration variables. Remember that "bumps" occur sequentially
                (count from (prim+ (*constant 1.0) (*do-variable count)))
                (p (create-vector from def-x
                                  ((*macro-device f2 interm)
                                   from from
                                   ((*macro-device k result) from)))
                   (store-vector (*do-variable p)
                                 (*do-variable count)
                                 ((*macro-device f2 interm)
                                  (*do-variable count) (*do-variable count)
                                  (prim- ((*macro-device k result)
                                          (*do-variable count))
                                         (*do (;;The number of iterations is really
                                               ;;the (*DO-VARIABLE COUNT)
                                               DEF-X
                                               (*mode-expression
                                                (prim= (*do-variable m) (*do-variable count)))
                                               (*do-variable sum)
                                               (sum (*constant 0.0)
                                                    (prim+ (*do-variable sum)
                                                           ((*macro-device f def-fx)
                                                            (*do-variable count)
                                                            (*do-variable m)
                                                            (access-vector (*do-variable p)
                                                                           (*do-variable m)))))
                                               (m from (prim+ (*constant 1.0)
                                                              (*do-variable m))))))))))
          ((f (to def-x interm) (def-fx))
           ((f2 f) (to def-x def-fx) (interm))
           (k (to) (result))) )
```

```
(rule-of sigma2 result (from to)
        (*do ((prim- from to)
             (prim= (*do-variable count) to)
             (*do-variable sum)
             (count from (prim+ (*do-variable count) (*constant 1.0)))
             (sum ((*macro-device sigbod def-fx) from from)
                 (prim+ sum
                        ((*macro-device sigbod def-fx)
                         from
                         (*do-variable count))))))
        ((sigbod (from def-x) (def-fx))))
```

TABLE "TPS-CONSTANT-COLLAPSE"

```
(define-transform tps-constant-collapse
                  (c2 x y)
                  ((tps c2 v v x y))
                  ((eq? x (*constant 1.0) pred)
                   (mpx (*constant 1.0)
                        (*constant 0.0)
                        pred
                        y)))

(define-enablement tps-constant-collapse remove
                   ((c2 ) (x) (y)))
```

TABLE "Other code written during Bernoulli number problem"

Code for macro-device MULTI-F:

```
(DEFUN G0042 (G0039 G0041)
  (COND ((= (RFACTORIAL G0041) 0.0) (COND ((= G0039 0.0) (ERROR))
                                          (T (ERROR))))
        (T (QUOTIENT G0039 (RFACTORIAL G0041)))))
```

Code for macro-device F:

```
(DEFUN G0069 (G0066 G0067 G0068 G0063 G0064)
  (TIMES (COND ((= (RFACTORIAL G0067) 0.0) (COND ((= G0068 0.0) (ERROR))
                                                 (T (ERROR))))
               (T (QUOTIENT G0068 (RFACTORIAL G0067))))
         ((LAMBDA (G0041)
            (COND ((= (RFACTORIAL G0041) 0.0) (COND ((= G0063 0.0) (ERROR))
                                                    (T (ERROR))))
                  (T (QUOTIENT G0063 (RFACTORIAL G0041)))))
          (DIFFERENCE (PLUS G0066 G0064) G0067))))
```

Code for macro-device F2 (note duplicated variable names):

```
(DEFUN G0071 (G0066 G0067 G0065 G0063 G0064)
       (TIMES (COND ((= ((LAMBDA (G0041)
                                 (COND ((= (RFACTORIAL G0041) 0.0)
                                        (COND ((= G0063 0.0) (ERROR)) (T (ERROR))))
                                       (T (QUOTIENT G0063 (RFACTORIAL G0041)))))
                         (DIFFERENCE (PLUS G0066 G0064) G0067))
                        0.0)
                     (COND ((= G0065 0.0) (ERROR)) (T (ERROR))))
                    (T (QUOTIENT G0065
                                 ((LAMBDA (G0041)
                                          (COND ((= (RFACTORIAL G0041) 0.0)
                                                 (COND ((= G0063 0.0) (ERROR)) (T (ERROR))))
                                                (T (QUOTIENT G0063 (RFACTORIAL G0041)))))
                                  (DIFFERENCE (PLUS G0066 G0064) G0067)))))
              (RFACTORIAL G0067)))
```

Code for macro device K:

```
(DEFUN G0087 (G0086 G0081 G0082 G0083 G0084)
       (COND ((= (PLUS G0086 G0082) G0081) G0084) (T G0083)))
```

Table "Bernoulli Code"

```
(DEFUN BERNOULLI (N)
  (DO ((COUNT 0.0) (P (CREATE-VECTOR 0.0 N 1.0)))
      ((= COUNT N) (ACCESS-VECTOR P N))
   (SETQ COUNT (PLUS 1.0 COUNT) P
    (STORE-VECTOR
     P
     COUNT
     ((LAMBDA (G0065)
        (TIMES (COND ((= ((LAMBDA (G0041)
                            (COND ((= (RFACTORIAL G0041) 0.0) (ERROR))
                                  (T (QUOTIENT 1.0 (RFACTORIAL G0041)))))
                          (DIFFERENCE (PLUS COUNT 1.0) COUNT))
                         0.0)
                      (COND ((= G0065 0.0) (ERROR)) (T (ERROR))))
                     (T (QUOTIENT G0065
                                  ((LAMBDA (G0041)
                                     (COND ((= (RFACTORIAL G0041) 0.0) (ERROR))
                                           (T (QUOTIENT 1.0 (RFACTORIAL G0041)))))
                                   (DIFFERENCE (PLUS COUNT 1.0) COUNT)))))
               (RFACTORIAL COUNT)))
      (DIFFERENCE
       (COND ((= (PLUS COUNT 1.0) 1.0) 1.0) (T 0.0))
       (DO ((SUM 0.0) (M 0.0))
           ((= M COUNT) SUM)
        (SETQ SUM
         (PLUS SUM
               ((LAMBDA (G0068)
                  (TIMES (COND ((= (RFACTORIAL M) 0.0)
                                (COND ((= G0068 0.0) (ERROR)) (T (ERROR))))
                               (T (QUOTIENT G0068 (RFACTORIAL M))))
                         ((LAMBDA (G0041)
                            (COND ((= (RFACTORIAL G0041) 0.0) (ERROR))
                                  (T (QUOTIENT 1.0 (RFACTORIAL G0041)))))
                          (DIFFERENCE (PLUS COUNT 1.0) M))))
                (ACCESS-VECTOR P M)))
         M (PLUS 1.0 M))))))))
```

TABLE "Bernoulli Time Cost"

```
 (*EXPRESSION
  (PRIM+
   (*EXPRESSION
    (PRIM*
     (*EXPRESSION
      (PRIM+
       (*CONSTANT 14.0)
       (*EXPRESSION
        (PRIM+
         (*EXPRESSION (PRIM* (*CONSTANT 2.0) (*VARIABLE N)))
         (*EXPRESSION
          (PRIM+
           (*CONSTANT 27.0)
           (*EXPRESSION (PRIM* (*EXPRESSION (PRIM+ (*CONSTANT 23.0)
                                                   (*EXPRESSION (PRIM* (*CONSTANT 2 0)
                                                                       (*VARIABLE N)))))
                                (*VARIABLE N))))))))))
     (*VARIABLE N)))
   (*CONSTANT 14.0))))
This can be summarized to:
(POLY 3.0 2 0)
```

This means the time cost is essentially $2N^3$

TABLE "Bernoulli NVALUE facet"

```
(*DO
 ((*VARIABLE N)
  (*MODE-EXPRESSION (PRIM= (*DO-VARIABLE COUNT) (*VARIABLE N)))
  (*EXPRESSION (ACCESS-VECTOR (*DO-VARIABLE P) (*VARIABLE N)))
  (COUNT (*CONSTANT 0.0)
         (*EXPRESSION (PRIM+ (*CONSTANT 1.0) (*DO-VARIABLE COUNT))))
  (P
   (*EXPRESSION (CREATE-VECTOR (*CONSTANT 0.0) (*VARIABLE N)
                               (*EXPRESSION (G0071 (*CONSTANT 0.0) (*CONSTANT 0.0)
                                                   (*EXPRESSION (G0087 (*CONSTANT 0.0)
                                                                       (*CONSTANT 1.0)
                                                                       (*CONSTANT 1.0)
                                                                       (*CONSTANT 0.0)
                                                                       (*CONSTANT 1.0)))
                                                   (*CONSTANT 1.0) (*CONSTANT 1.0)))))
   (*EXPRESSION
    (STORE-VECTOR (*DO-VARIABLE P) (*DO-VARIABLE COUNT)
     (*EXPRESSION
      (G0071 (*DO-VARIABLE COUNT) (*DO-VARIABLE COUNT)
       (*EXPRESSION
        (PRIM-
         (*EXPRESSION (G0087 (*DO-VARIABLE COUNT) (*CONSTANT 1.0) (*CONSTANT 1.0)
                             (*CONSTANT 0.0) (*CONSTANT 1.0)))
         (*DO
          ((*VARIABLE N)
           (*MODE-EXPRESSION (PRIM= (*DO-VARIABLE M) (*DO-VARIABLE COUNT)))
           (*DO-VARIABLE SUM)
           (SUM (*CONSTANT 0.0)
            (*EXPRESSION
             (PRIM+ (*DO-VARIABLE SUM)
                    (*EXPRESSION (G0069 (*DO-VARIABLE COUNT) (*DO-VARIABLE M)
                                        (*EXPRESSION (ACCESS-VECTOR (*DO-VARIABLE P)
                                                                    (*DO-VARIABLE M)))
                                        (*CONSTANT 1.0) (*CONSTANT 1.0))))))
           (M (*CONSTANT 0.0)
              (*EXPRESSION (PRIM+ (*CONSTANT 1.0) (*DO-VARIABLE M))))))))
      (*CONSTANT 1.0) (*CONSTANT 1.0))))))))
```

Bibliography

[Ba77] Barstow, D. *Automatic Construction of Algorithms and Data Structures Using a Knowledge Base of Programming Rules*, Stanford AI Memo 308, 1977.

[Bi76] Biermann, A. and Krishnaswamy, R. "Constructing programs from example computations," *IEEE Transactions on Software Engineering*, Vol. SE2, Sept. 1976

[B78] Brent, R. and Kung, H. "Fast Algorithm for Manipulating Formal Power Series," *JACM* Vol. 25, No. 4, Oct. 1978.

[Br77] Brown, R. *Use of Analogy to Achieve new Expertise*, AI-TR-403, 1977

[Br76]Bruno, J. and Sethi, R. "Code generation for a one-register machine," *JACM* Vol. 23, No. 3, July 1976.

[B69] Burstall, R. "Proving properties of programs by structural induction" *Computer J.*, Vol 12, Feb. 1969

[Bu77] Burstall, R. and Darlington, J. "A Transformation System for Developing Recursive Programs," *JACM* Vol. 24, No. 1, Jan 1977.

[dK79] de Kleer, J., Doyle, J., Steele, G., Sussman, G. "Explicit Control of Reasoning" in *Artificial Intelligence: An MIT Perspective* Winston and Brown (eds.), MIT Press, 1979.

[D75] Dennis, J. *First Version of a Data Flow Procedure Language*, MIT LCS LM-61, 1975

[Do79] Doyle, J. "A Glimpse of Truth Maintenance" in *Artificial Intelligence: An MIT Perspective* Winston and Brown (eds.), MIT Press, 1979.

[E74] Earnest, C. "Some Topics in Code Optimization," *JACM* Vol. 21, No. 1, Jan. 1974.

[E76] Emden, M. and Kowlaski, R. "The Semantics of Predicate Logic as a Programming Language," *JACM* Vol. 23, No. 4, Oct. 1976.

[F67] Floyd, R. "Assigning meanings to programs" *Proceedings of Symposium on Applied Mathematics*, Vol. 19, Schwartz (Ed), AMS, 1967.

[G76] Green, C. "The Design of the PSI Program Synthesis System," *Proceedings of the*

Second International Conference on Software Engineering, 1976.

[G69] Green, C. "Application of theorem proving to Problem Solving," *IJCAI* 1969.

[H75] Hardy, S. "Synthesis of LISP functions from examples," *IJCAI* 1975.

[H74] Henschen, L. and Wos, L. "Unit Refutations and Horn Sets," *JACM* Vol. 21, No. 4, Oct. 1974.

[H79] Hewitt, C. "Control Structure as Patterns of Passing Messages" in *Artificial Intelligence: An MIT Perspective* Winston and Brown (eds.), MIT Press, 1979.

[K77] Kant, E. "The Selection of Efficient Implementations for a High-Level Language," *Proceedings of the Symposium on Artificial Intelligence and Programming Languages*, 1977.

[L75] Liskov, B. and Ziles, S. "Specification techniques for data abstractions," *IEEE Transactions on Software Engineering*, Vol. SE1, No. 1, March 1975.

[L77] Long, W. *A Program Writer* LCS TR-107, November 1977.

[L78] Low, J. "Automatic Data Structure Selection: An Example and Overview," *CACM* Vol.21, No. 5, May 1978.

[M75] Manna, Z. and Waldinger, R. "Knowledge and Reasoning in Program Synthesis,"*Artificial Intelligence*, No. 6, 1975.

[M78] Manna, Z. and Waldinger, R. "The Logic of Computer Programming," *IEEE Transactions on Software Engineering*, Vol. SE4, No. 3, May 1978.

[M79] Manna, Z. and Waldinger, R. "Synthesis: Dreams => Programs," *IEEE Transactions on Software Engineering*, Vol. SE5, No. 4, July 1979.

[Mc77] McCune, B. "The PSI Program Model Builder," *Proceedings of the Symposium on Artificial Intelligence and Programming Languages*, 1977.

[R69] Rektorys, K. *Survey of Applicable Mathematics* MIT Press, 1969.

[R79] Rich, C. and Shrobe, H. "Design of a Programmers Apprentice," in *Artificial Intelligence: An MIT Perspective* Winston and Brown (eds.), MIT Press, 1979.

[R80] Rich, C. *On the Use of Inspection Methods in Programming*, forthcomming thesis 1980.

[R77] Rosenschein, S. and Katz, S. "Selection of Representations for Data Structures" *AIPL* 1977.

[S75A] Sacerdoti, E. *A Structure for Plans and Beharior*, SRI Technical Note 109, 1975.

[S75B] Sacerdoti, E. *The Non-linear Nature of Plans*, SRI Technical Note 101, 1975.

[Sh79] Shrobe, H. *Dependency Directed Reasoning for Complex Program Understanding*, MIT-AI-TR503, 1979.

[S79] Stallman, R. and Sussman, G. "Problem Solving About Electrical Circuits" in *Artificial Intelligence: An MIT Perspective* Winston and Brown (eds.), MIT Press, 1979.

[S77] Summers, P. "A Methodology for LISP Program Construction from Examples," *JACM* Vol. 24, No. 1, Jan. 1977.

[Su75] Sussman, G. *A Computer Model of Skill Acquisition*, American Esevier, 1975.

[U77] Ulrich, J. and Moll, R., "Program Synthesis by Analogy," *Proceedings of the Symposium on Artificial Intelligence and Programming Languages*, 1977.

[We76] Wegbreit, B. "Goal-Directed Program Transformation," *IEEE Transactions on Software Engineering*, Vol. SE2, No. 2, June 1976.

[W79] Winston, P. "Learning by Creating and Justifying Transfer Frames"in *Artificial Intelligence: An MIT Perspective* Winston and Brown (eds.), MIT Press, 1979.

[Wu76] Wulf, W., London, R., and Shaw, M. "An Introduction to the Construction and Verification of Alphard Programs," *IEEE Transactions on Software Engneering* Vol SE2, No. 4, Dec. 1976.

DATE

ILMED

-8