

Stereotyped program debugging: an aid for novice programmers

HARALD WERTZ

C.N.R.S. LA 248 L.I.T.P., 2 place Jussieu, 75005 Paris and Département d'Informatique, Université Paris VIII, Vincennes, 75571 Paris Cédex 12, France

(Received 5 July 1980)

This paper presents a system (PHENARETE) which understands and improves incompletely defined LISP programs, such as those written by students beginning to program in LISP. This system takes, as input, the program without any additional information. In order to understand the program, the system meta-evaluates it, using a library of *pragmatic rules*, describing the construction and correction of general program constructs, and a set of *specialists*, describing the syntax and semantics of the standard LISP functions. The system can use its understanding of the program to detect errors in it, to eliminate them and, eventually, to justify its proposed modifications. This paper gives a brief survey of the working of the system, emphasizing some commented examples.

Introduction

Much effort is spent on the development of tools to help programmers in constructing, debugging and verifying programs. From simple editors and trace-packages, the trend is towards more and more sophisticated automatic programmers (Balzer, 1972, Green & Barstow, 1978), automatic debuggers (Adam & Laurent, 1977; Ruth, 1974; Teitelman, 1974), automatic assistants (Rich & Shrobe, 1978), automatic verifiers (Deutsch, 1973; Igarashi, London & Luckham, 1975) or even the construction of new—semantically more firmly based—programming languages (Pascal, Alphonse).

Most of these tools exhibit some weaknesses such as:

they impose too many constraints on the intuitions of the programmer (cf. Dijkstra, 1976),

they work only on a very limited subset of possible programs (cf. Adam & Laurent, 1977; Ruth, 1974),

they work only on correct programs (cf. Arzac, 1977; Igarashi *et al.*, 1975).

Our aim in the design of our program understanding system was fourfold:

1. we wanted to have a system which makes explicit the knowledge involved in constructing and debugging programs;
2. we wanted our system not to verify the correctness of programs but their *consistency*;
3. we wanted the system to provide *hints* for improving and correcting programs and
4. we wanted a practical, useful and running system.

Thereupon we have built a program understanding system able to automatically correct and improve programs. This system PHENARETE, *assists* beginning programmers during the writing and debugging of their programs.

Programming errors

The construction of a computer program can be divided into different steps: first, the programmer has to have a somewhat precise conceptualization of the activity he desires the computer to perform. As quoted by Model (1979), this conceptualization does not just restrain at the level of overt input/output behaviour, but includes the projected programs manipulation of its internal data structures. Second, the programmer has to conceive a method by which the intended activity may be performed. Third, he has to express this method in terms of a programming language and, fourth, he has to communicate the program to the computer.

This process induces five major types of programming errors, related to these different steps: lexical, syntactic, semantic, teleological and conceptual errors.

Lexical errors are probably the most frequent—and more easily debuggable—errors. They are mainly misspellings or typographical mistakes, and are detectable at the word-level. They refer—in classical compilers—to the lexical analyser or scanner.

Every language has rules governing the allowed forms of program statements. We call deviations of these rules syntactic errors.

These first two kinds of errors, normally detected during the reading of the program (compilation phase), are—in this paper—invariably called surface errors or informalities.

Another kind of errors is that—even when the syntactic form is correct—its meaning may be unclear, contradictory or invalid. Examples of this kind of errors are “division by zero”, an attempt to multiply a number by a string or a call of a function which has no definition. We call such errors semantic errors. They are normally detected during the run-time of the program.

New algorithm languages, such as PASCAL, MESA or ADA, try to detect semantic errors at the semantic level through “strong compile-time checking of data types and program interfaces”, but even when they are detected by syntactic checking, the sources of such errors lie at the semantic level.

Let us note that such compile-time checking is rather limited, since even such a simple error as division by zero escapes the checking if the divisor is not just a constant or a variable, but an expression whose value is computed at run-time and the expression equals zero only sometimes.

The fourth class of errors we can distinguish are teleological errors, which occur when the program does something, but not what was intended. These errors refer to the problems in the scope of the programmers’ precise specification of the computational scheme or algorithm.

In our paper we call errors of the last two kinds deep errors or inconsistencies.

A last kind of error is met if the source of the problem is in the method or approach devised by the programmer to achieve the desired computational activity. These errors are called conceptual errors.

Our system, PHENARETE, analyses the text of a program and, when it detects informalities or inconsistencies, it constructs and proposes possible corrections or improvements. These modified versions of the program, constitute for the novice programmer, a crucial component of his apprenticeship: basing on *examples* the process of improving and correcting imperfect programs, and basing on *models* the process of inventing future programs.

The originality of the system resides (1) in its ability not only to detect low-order errors—a standard in today's compilers—but also middle- and even higher order errors (i.e. semantic and some teleological errors) and (2) in its attempt to use the knowledge necessary to detect errors, also to construct possible corrections.

Overview of the system

The system takes as input the draft version of a LISP program and delivers as result of its treatment one or more versions of the same program, corrected and improved. PHENARETE proceeds in several steps (cf. Fig. 1): first a preliminary analysis of the text of the program is performed in order to detect and correct surface errors.

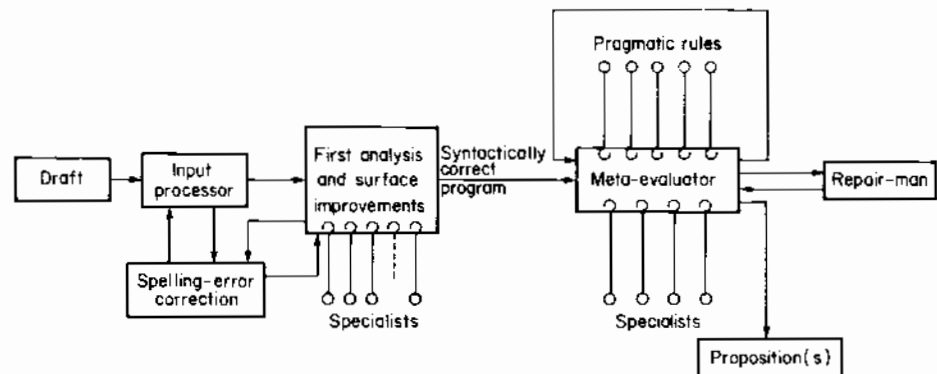


FIG. 1. General flowchart for PHENARETE.

By surface errors we mean errors detectable by local analysis: simple lexical errors such as misspellings or parentheses errors as well as syntactic ones such as errors concerning the wrong number of arguments in function calls. During this first analysis PHENARETE collects information for the following steps. Whenever it meets the name of a function (standard or user defined LISP function) it automatically activates a *specialist*, associated with this function, by the process of data driven function invocation (Sandewall, 1975). These specialists represent its knowledge about the use and the effects of the associated functions.

The result of the first analysis—a syntactically correct program—is then meta-evaluated to verify the programs *well-formedness*. We say that a program is well-formed if it does not contain obvious infinite loops, does not have useless statements (i.e. statements never executed) and if it does not contradict the set of rules incorporated in PHENARETE. Every final version of the program our system delivers is a well-formed program.

When PHENARETE detects some informalities or inconsistencies, it annotates the corresponding part of the program and sends the code and the annotation to the repair-module, a module designed to eliminate errors (with the help of stereotyped programming knowledge). The system stops the analysis process when not finding any more possible improvements.

Organization of PHENARETE

The system is based on four main concepts:

1. During the analysis of a program PHENARETE constructs a description—an internal representation—of the program in terms of *cognitive atoms*. These may be considered as the nodes of a network-like representation of the program.

Each cognitive atom is composed of a set of facets which represent its different aspects, about which questions can be asked, such as:

what type is it?

what is its definition?

(if it is a function) what is its domain?

The facets are filled in during the analysis process.

We distinguish between three classes of cognitive atoms, those concerning variables, those concerning labels and those concerning functions. The facets are the same for every atom inside the same class, but they differ significantly from one class to another.

The representation of a cognitive atom is a set of attribute/value pairs, and the value v found under the attribute P of atom A is simply the answer the expert A can give to question P .

2. A set of *specialists*, i.e. a set of procedural specifications of the syntax and the operational semantics of the standard LISP functions, such as CAR, CDR, EQ, COND etc.

We distinguish between two different types of specialists:

those describing the syntax and

those describing the semantics

of the associated functions.

Example: specialist CAR for the syntax

<pre> [CAR-1 (X) ⇒ v (& atom (CAR X) & type (X) = LISTP) v (& S-expression (CAR X) & type (val (X)) = LISTP) else: modify X until CAR-1 (X) = T] </pre>

paraphrasing:

CAR expects that its argument is

an atom

and the type of the value of the argument is a list

a S-expression

and the type of the value of that S-expression is a list

(e.g. a function call)

else

CAR has to modify the argument until one of these two conditions is true

and the specialist CAR for the semantics

<pre> [CAR-N⇒ arg: (X (meta-eval X)) test: (type (val (X)) = LISTP) → (type (val (X)) = ?) → hypothesize (X, type: LISTP) T → complain (X, type: LISTP) action: if (exist (CAR X)) → (CAR X) else (create (CAR, X)) → (CAR X)] </pre>

or in paraphrasing:

CAR-N

has an argument named X, which must be evaluated; one must verify
 if
 the type of value of the argument is a list, all is ok
 else
 if the type of value of the argument is not known, one has
 to create a hypothetical value of type LIST for X
 else
 one has to call the repair-module to change the text of
 the program in such a way that the value of X becomes a
 list
 the value of CAR is
 if
 there exists already a CAR of X, this CAR
 else
 one has to create a symbolic value for X, the CAR of which
 will be the desired value.

Note that the internal description constructed during the analysis of each user function is used to construct two new such specialists for each user function. The specialists are the agents of the meta-evaluation and they represent the systems knowledge about the programming language used.

3. An algorithm of *meta-evaluation* (Goossens, 1978) which helps the system to analyse each of the possible paths of the program. This algorithm is composed of two parts:

a module to determine the symbolic values, i.e. since the meta-evaluation does not use concrete values in order to evaluate the program, it must deduce from the text of the program the type and the structure of the arguments of the different functions to evaluate. Thus our algorithm is rather different from those proposed in Howden (1977) and King (1975) which receive the symbolic values as input.

an evaluation module, i.e. a module able to evaluate functions on these symbolic arguments.

With this algorithm one *unique* execution is representative for every particular execution (King, 1975).

4. A set of pragmatic rules describing general program constructs and stereotyped methods to repair inconsistencies. These rules formalize and express explicitly the knowledge activated by every programmer when he is reading a program.

We do not use rules concerning the task domain of the programs: our intention was to build a system which does not ask for any additional information, i.e. which works only with the text of the program to be analyzed, so that the programmer is not obliged to accompany the programs by comments, assertions or other specifications. The system should work—without any additional information—in any possible task domain, numeric as well as symbolic.

Our rules are very general and valid for every LISP program obeying to the following restrictions:

- the names of variables, functions and labels are unique,
- each function call must be of the type “call by value” and
- the unique functional arguments permitted are explicit λ -expressions.

We call this subset of LISP: “extended first order LISP”. Examples of pragmatic rules:

rule of the dependence of a loop of the predicate \Rightarrow

$$\begin{array}{l} \text{given: } F \in \{\text{loop}\} \\ \quad T = \{\text{exit-test of } F\} \\ \quad Y = \{\text{variable}\} \text{ so that} \\ \quad \quad \forall y \in Y \quad \exists t \in T \quad y \subset t \\ \text{then: } \forall y \in Y \quad \text{val}(y) [F] \neq \text{val}'(y) \\ \quad \quad \wedge \text{val}(y) = \text{val}'(y) \\ \quad \quad \Rightarrow \text{val}(F) = \text{undefined} \end{array} \quad (1)$$

in paraphrasing:

if no variable of the exit-test of a loop is modified inside the loop-body, then the loop is independent of the exit-test and either its execution is non-terminating or the loop will never be executed.

A slight refinement of this rule is the following one:

rule of the structural well-formedness of loops \Rightarrow (2)

in a loop at least one of the variables used in the exit-test has to change its value inside the body of the loop, in such a way that its structure is simplified and converges towards the satisfaction of the exit-test.

In order to use this rule, we have implemented a small theorem prover which can prove the convergence by induction. As to the proof implied by this rule the theorem-prover takes the symbolic value of the variables at the entry of the loop, and the modified symbolic values of the same variables after *one* meta-evaluation of the loop-body and tries to prove the convergence towards the stop test.

Let us give one last example of a pragmatic rule:

rule of the structure of recursive loops \Rightarrow

$$\begin{array}{l} \text{given: } F \in \{\text{recursive functions}\} \\ \quad A = \{\text{call of } F\} \text{ so that } A \subset F \\ \quad S = \{\text{selection-clause}\} \text{ so that } S \subset F \\ \text{then: } \forall a \in A \quad \exists s \in S \quad a \subset s \\ \quad \quad \wedge \exists s \in S \text{ so that } \forall a \in A \quad a \not\subset s \end{array} \quad (3)$$

paraphrasing:

in a recursive function, the recursive calls have to be inside a selection-clause, and at least one of the clauses must not contain a recursive call.

Presently we have about a hundred rules incorporated in the system, dealing especially with iteration and recursion, the use of variables and list-processing. For almost every rule there exists a dual one indicating how to modify the program in such a way that the first one is satisfied.

Some commented examples

To use PHENARETE, the user has to give the system only the text of the draft version of the program he wants to write, without any additional information like input/output assertions, comments, plans, etc. The system will try to understand what the user wants to do and, if necessary, modify the text of the program.

To give some feeling how the system works, let us examine some examples in detail.

Our first example is a (very) erroneous version of the well known REVERSE function. Here is the actual input to the system:

```
? (P'(DE REV L1 L2 COND ULL L22 A1 T RVE A1 ONS CRA A1 A2))
```

PHENARETE will first perform a preliminary analysis using only its syntactic knowledge, i.e. the specialists for the syntax and the spelling error corrector. The result of this first analysis is a syntactically correct LISP program, a program accepted by any LISP interpreter or compiler.

Here is the actual printout of PHENARETE at this point.

```
ERROR: NAME→(? ULL→NULL)
ERROR: NAME→(? L22→L2)
ERROR: NAME→(? A1→L1)
ERROR: NAME→(? A1→L1)
ERROR: NAME→(? A1→L1)
ERROR: NAME→(? RVE→REV)
ERROR: NAME→(? A1→L1)
ERROR: NAME→(? ONS→CONS)
ERROR: NAME→(? CRA→CAR)
ERROR: NAME→(? A1→L1)
ERROR: NAME→(? A2→L2)
```

SURFACE IMPROVEMENTS:

```
(DE REV (L1 L2)
 (COND
 ((NULL L2)L1)
 (T (REV L1(CONS (CAR L1) L2)))))
```

These first improvements have eliminated all the syntactic errors. However, at least one semantic error remains:

Neither L1 nor L2 are modified in such a way that their values converge towards the stop test; even with a modification of L1 in the recursive call, the recursion

won't stop since the stop-test has as argument L2, a list which grows longer and longer on successive recursive calls (rule 2).

During every analysis, our system collects useful information about the program, represented in the data-base which constitutes the internal description of the analysed program. Figure 2 shows the state of the data-base after the first analysis of the function REV.

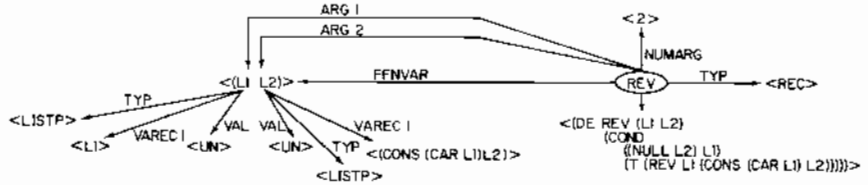


FIG. 2. Data-base after the first analysis.

The knowledge represented in Fig. 2 can be paraphrased as follows:

After the first analysis PHENARETE knows that REV is a recursive function, with two arguments. The first argument is called L1, which the system does not know the value of, but knows it has to be a list. At the recursive call of REV, the first argument is not modified.

The second argument is called L2. PHENARETE does not know its value either. It too has to be a list. At the recursive call L2 takes the value of the expression (CONS (CAR L1) L2).

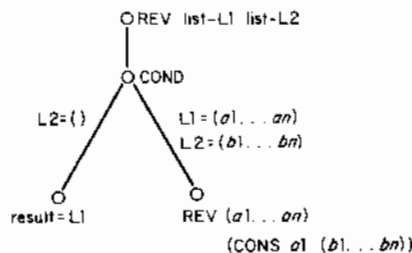
As for the errors detected up to that point, most standard scanners detect this kind of errors, and some compilers even handle this kind of corrections (PL/C or CORC).

For the further improvements, the system proceeds to a meta-evaluation of the current version, using as symbolic value of the arguments just the knowledge that they have to be lists. If further specification about the values is needed, PHENARETE constructs them dynamically.

In this example, the system has—corresponding to the two clauses of the selection (COND)—to develop two paths, one supposing the predicate (NULL L2) satisfied, one supposing it falsified.

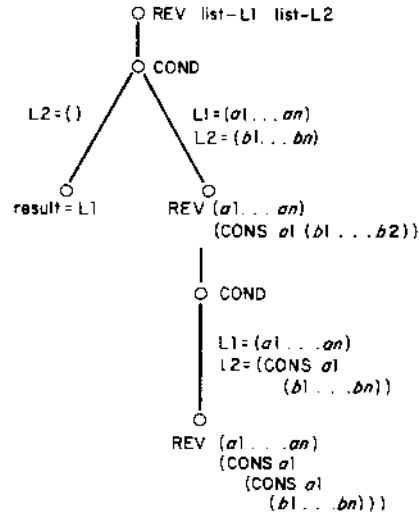
Note that one of the main differences between standard evaluation and meta-evaluation is that the knowledge derived during the meta-evaluation of one of the possible paths can be used during the meta-evaluation of every other path.

After one meta-evaluation of the function REV, PHENARETE has constructed an evaluation tree of the form:



As noted above, the meta-evaluation stops, when it reaches a point where no further continuation is possible (the case of the left branch) or when it reaches a point where control has already passed by (which will be the case later, during the meta-evaluation of the right branch of the tree).

When the system meets a recursive call or an iteration, it carries on the evaluation just as in normal execution. Here the continuation of the right branch of the tree forces the system to falsify the first predicate of REV, since the symbolic value of L2 is a CONS-expression which cannot equal NIL (the empty list). So we find:



At this point PHENARETE has to stop the meta-evaluation: all terminal leaves are either end-points of the program or points already visited during the meta-evaluation.

Application of the pragmatic rules, especially those concerning recursion, indicates that the program won't stop, since the modification of the arguments is not in such a way that they converge towards satisfaction of the exit-test (rule 2): the exit-test checks if L2 is a empty list, but L2 grows long during the successive recursive calls. The demonstration here is straightforward, but we easily realize that the power of the system depends crucially on the power of the incorporated theorem prover.

Each rule is combined with an ordered set of *advices* as to "how to correct" the non-satisfaction of the rule. The advices of rule 2 say that in order to correct the code, we have to suppose that

1. there is an error in the stop test or
2. there is a missing stop test or
3. there is an error in the recursive call.

The pragmatic rules as well as the advices (which are just pragmatic rules of correction) are hierarchically organized, from the general to the concrete, i.e. the application of one rule may result in the application of a lot of other rules.

In this case the first advice PHENARETE finds applicable is the third one. Here is another point of the system which deserves some criticism: PHENARETE supposes

that at least one of the arguments has to simplify its structure in the successive calls. If this is not the case, and the system does not find an error, everything is O.K., but otherwise it forces a simplification. Here the simplification is forced by the use of the CAR of L1: PHENARETE assumes the user wants to work with successive elements of L1, which makes it introduce the modified recursive call

```
(REV (CDR L)(CONS (CAR L1) L2)).
```

No doubt, if the programmer did not want that, eventually he would not understand any more the correction of PHENARETE, but in constructing the system we wanted the programmer not to be obliged to give assertions or comments, so we have no means to know anything about the intentions of the programmer (except what is explicitly stated in the code), and the proposed corrections may differ significantly of those used by an experienced programmer knowing the intentions.

To justify our approach, before beginning to write the system, we have systematically studied, for one year, the programs written by our students. The corrections PHENARETE proposes, are those corresponding to the statistically most current errors. On the other hand, we think a program which stops and delivers a result different from the intended one is easier to debug than a program entering in an infinite loop. PHENARETE just corrects programs in such a way that they don't abort and stop delivering a result, when executed.

After each modification, the system re-meta-evaluates the program, to check if any error remains.

In our example, PHENARETE cannot disambiguate this function—it does not know anything of the intentions of the programmer—so it gives two different propositions:

PROPOSITION 1:

```
(DE REV (L1 L2)
  (COND
    ((NULL L2) L1)
    ((NULL L1) L2)
    (T (REV (CDR L1)(CONS (CAR L1) L2)))))
```

AT LEAST YOUR FUNCTION SEEMS OK.

In this first proposition, PHENARETE supposed the given stop-test correct, but assumed that the user omitted a second stop-test in case the second argument is not NULL at the initial call of REV.

PROPOSITION 2:

```
(DE REV (L1 L2)
  (COND
    (NULL L1) L2)
    (T (REV (CDR 1)(CONS (CAR L1) L2)))))
```

AT LEAST YOUR FUNCTION SEEMS OK.

In this second proposition, PHENARETE supposed that the user inadvertently inverted the arguments of the stop-test, so it inverts the two arguments L1 and L2.

PHENARETE secures that the two corrected versions of the initial draft-program (which are not identical) will stop and deliver a result when executed.

The data-base after the completion of the analysis of REV is represented in Fig. 3 (only for the second version).

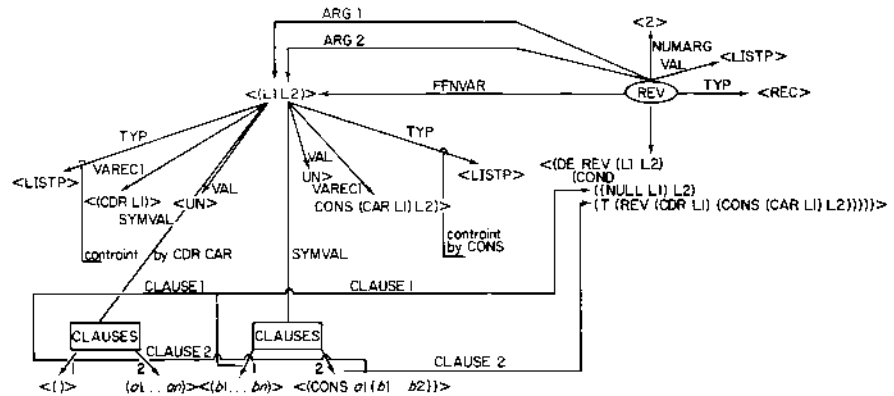


FIG. 3. Data-base after the analysis.

Two sources of information are available after the complete analysis of a program:
 the program as an executable algorithm and
 the description of the program in the data-base.

This data-base does not describe "how" or "what" the program computes, but gives valuable indications on the conditions which have to be verified if the program is activated and on those which are verified after its execution.

This description is used to construct new specialists, in order not to reanalyse the function if it is called by another one.

Here are the two specialists:
 specialist REV for the syntax

```

[REV-1 (X) =>
  & (v (& atom (CAR X)
        & type (CAR X) = LISTP)
     v (& S-expression (CAR X)
        & type (val (CAR X)) = LISTP))
  & (v (& atom (CADR X)
        (& type (CADR X) = LISTP)
        v (& S-expression (CADR X)
           & type (val (CADR X)) = LISTP)
     else:
       modify X until REV-1 (X) = T
    
```

specialist REV for the semantics

```
[REV-N ⇒
  arg: (X Y(meta-eval (X) meta-eval (Y)))
  test: (type (X) = LISTP)
        → (type (Y) = LISTP) →
           (type (Y) = ?)
           → hypothesize (Y, type: LISTP)
           T → complain (Y, type: LISTP)
        (type (X) = ?)
        → hypothesize (X, type: LISTP)
        go test
        T → complain (X, type: LISTP)
        go test
  action: ⇒ {LISTP, execute-symb (REV X Y)}]
```

As can be easily verified, the specialists are just algorithmic transcriptions of the data-base.

Now, when PHENARETE meets, during the analysis of another program, a call to the REV-function, it need not reanalyse the body of the function, but it can use the specialists of REV in the same manner it uses the specialists of the standard function.

Our second example is an extremely "simplified" version of the equally well-known FACTORIAL function. Here it is:

```
? (DE FACT N TIMES N FACT N)
```

As in the previous example, PHENARETE will first translate this unparenthesized expression into an well parenthesized one:

```
SURFACE IMPROVEMENTS:
```

```
(DE FACT (N) (TIMES N (FACT N)))
```

This first proposition is a syntactically correct program, but semantically it presents some difficulties:

1. at the recursive call, N is not modified (rule 1) and
2. there is no stop-test at all (rule 3), so there are two (!) reasons to make the recursion infinite.

Remember that PHENARETE does not know the intentions of the programmer, so it must detect these errors without any additional information: all that can be used in the further analysis are the semantic specialists and the pragmatic rules. So let us look at its versions:

```
PROPOSITION:
```

```
(DE FACT (N) (COND
  ((LE N 0) 1)
  (T (TIMES N(FACT (SUB1 N))))))
```

```
AT LEAST YOUR FUNCTION SEEMS OK.
```

This corrected version is actually a correct version of the factorial-program. The performance illustrates well the pertinence of the pragmatic rules, knowing that the system works completely automatically without asking any questions to the user and without any information about the supposed intention.

One last example:

```
? (DE ADDIT M N (((ZEROP N) M))
      (T (ADDIT SUB1 M ADD1 N)))
```

SURFACE IMPROVEMENTS:

```
(DE ADDIT (M N)
  (COND
    ((ZEROP N) M)
    (T (ADDIT (SUB1 M) (ADD1 N)))))
```

PROPOSITION:

```
(DE ADDIT (M N)
  (COND
    ((ZEROP N) M)
    (LE M 0) N)
  (T (ADDIT (SUB1 M) (ADD1 N)))))
```

AT LEAST YOUR FUNCTION SEEMS OK.

The difference between this example and the previous one consists mainly in the constructed return value of the function. In the FACT example, the value returned was just the neutral element of the operation applied to the result of FACT (e.g. TIMES), in this example a value is computed during the recursion, thus this newly computed value will be the result of the function.

Conclusion

We have presented a running system which corrects incorrect student programs in some stereotyped way.

There are two major shortcomings we would like to mention:

1. Our system relies heavily upon a (still very small) theorem-prover and—related to it—the internal representation of the abstract data-structures. These two modules are currently much too limited to apply the system to real user programs (and not only to small student programs). But looking at most other systems of meta-evaluation (for example, Howden, 1977; King, 1975) it is one of the first systems carrying on the meta-evolution not only on numeric programs but on symbolic ones, where no algebraic theory yet exists.

We think we will use for the next version of the system the abstract list-representation proposed by Goossens (1978).

2. The total absence of any mean to communicate to the system the intentions of the programmer constitutes—as soon as one leaves more-or-less toy programs—a severe handicap. Presently we are investigating possibilities to encompass this

shortcoming. Different approaches are possible and we don't know yet which one to choose: either allow comments (but what should we do if there are errors in the comments?) or oblige the programmers to give some input/output examples, which would permit to use real *and* symbolic values (but same question as above arises).

We are presently implementing PHENARETE as a standard VLISP (Chailloux, 1978; Greussay, 1977) error routine, i.e. as soon as during the execution of a program an error occurs, we trap it and apply the system to the program which caused the error. This will eliminate the—always existing—possibility that the program modifies a correct—but for it unintelligible—program.

A word about LISP: the only reason why we have chosen LISP is the isomorphism between external and internal representations, and because our students learn LISP as first programming language. The algorithms and rules used are very general and not LISP-specific. An implementation of PHENARETE for an algorithmic language, say Pascal, for example seems straightforward; the system would even have less work in determining the symbolic values because of the typed declaration of all identifiers.

The system is running on PDP-10, uses about 25k word memory, is implemented in VLISP, and is used by about 600 students in our university.

A more detailed description may be found in WERTZ (1978).

References

- ADAM, A. & LAURENT, J. P. (1977). Transformation de programmes et correction de Programmes, *Colloque sur l'Intelligence Artificielle*, Strasbourg, CNRS, 41–83.
- ARSAC, J. (1977). *La Construction de Programmes Structurés*. Paris: Dunod-Informatique.
- BALZER, R. (1972). *Automatic Programming (Draft)*. ISI, University of Southern California.
- CHAILLOUX, J. (1978). *VLISP-10.3 Manuel de Référence*. Département Informatique, Université Paris 8, RT-17-78.
- DEUTSCH, P. L. (1973). *An Interactive Program Verifier*. Xerox Parc, CSL 73-1.
- DIJKSTRA, E. W. (1976). *A Discipline of Programming*. Englewood Cliffs, New Jersey: Prentice-Hall.
- GOOSSENS, D. (1978). A system for visual-like understanding of LISP programs. *Proceedings AISB/GI Conference*, Hamburg, RFA.
- GREEN, C. & BARSTOW, D. (1978). On program synthesis knowledge. *Artificial Intelligence* 10(3), 241–279.
- GREUSSAY, P. (1977). *Contribution à la définition interprétative et à l'implémentation des lambda-languages*. Thèse, Université Paris 7.
- HOWDEN, W. E. (1977). Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Transactions on Software Engineering*, SE-3(4), 266–278.
- IGARASHI, S., LONDON, R. L. & LUCKHAM, D. C. (1975). Automatic program verification 1: logical basis and its implementation. *Acta Informatica*, 4, 145–182.
- KING, J. C. (1975). A new approach to program testing. *Proceedings ACM International Conference on Reliable Software*, Los Angeles, 228–233.
- MODEL, M. L. (1979). *Monitoring System Behaviour in a Complex Computational Environment*. Xerox, Palo Alto Research Center, CA, CSL-79-1.
- RICH, C. & SHROBE, H. E. (1978). Initial report on a LISP programmer's apprentice. *IEEE Transactions on Software Engineering*, SE-4(6), 456–467.
- RUTH, G. R. (1974). *Analysis of Algorithm Implementations*. M.I.T., MAC-TR-130.
- SANDEWALL, E. (1975). *Ideas About Management of LISP Data Bases*. AI-Memo-332, Cambridge, Massachusetts: M.I.T.
- TEITELMAN, W. (1974). *INTERLISP Reference Manual*. Xerox PARC, Palo Alto.
- WERTZ, H. (1978). *Un système de compréhension, d'amélioration et de correction de programmes incorrects*. Thèse de 3ème cycle, Université Paris 6.