A CONTROL STRUCTURE FOR A QUESTION-ANSWERING SYSTEM

P. Medema

Philips Research Laboratories
Eindhoven, The Netherlands

Abstract

The control structure of a question-answering system is derived from a set of basic assumptions. For its design and implementation, a normal algorithmic language is used. The design leads to "symmetric" procedures, the resulting programs use the stack mechanism for recursion to give the correct storage allocation. Two strategies are discussed in more detail: a trivial well-known depth-first strategy and a new one, called flexible decision tree strategy. No linguistic aspects (either syntactic or semantic) *of* the system are discussed in section 6, no formally defined programming language is used. Most of the symbols are borrowed from ALGOL68, the remaining ones are supposed to be self-explanatory. For design and publi-cation purposes, algorithmic descriptions that are easy to read for human readers are preferred to algorithmic descriptions that can be processed by a compiler.

Introduc tion

The author and his colleagues have developed a prototype of a question-answering system. A document describing the whole system and the considerations that have lead to the design has not yet been published, but wi11 appear before long. As this paper discusses the algorithmic aspects on a fairly abstract level, only a rough sketch of the system's properties is needed. This sketch is given in section 1. In order to be able to give some examples, section 1 also contains a superficial description of some properties of the inverse-of-discourse of our systom. The reader should however realize, that this paper is not - and is not intended to be - a description of the system.

Some of the charurteristies that seem to be common for all A.I. systems, like the existence of local ambiguities, are also present in our system. In contrast to what seems to be customary for the construction of A.I. programs, no list-processing language (LISp or one of its improvements PLANNER or CONNIVER) was used. In the opinion of the author, matters such as automatic backtracking and garbage collection should kept in sight. They may be present only impli-citly - as opposed to being programmed explicitly - but the programmer should control them, as he controls some stora-ge allocation mechanism via invocation and termination of procedures, thereby implicitly controlling the size of the stack.

One of the targets of the project was to reach a clean, well-struetured program by means of a "top-down" design method. This paper claims to show how the control structuro was derived (in an informal way) from a small number of basic assumptions.

For the notation of the algorithms

1• A rough sketch of the properties of the system

The system (called PHLIQA for PHiLIps Question Answering) is able to answer questions formulated in a natural language (English), where the answers to these questions are facts, either to be found directly in a data base inside the system or to be derived from information stored in the data base.

As in most present, day A•1• systerns, the ciaim is not that PHLIQA can under-stand the full natural language; it is only capable of processing the questions relating to the restricted universe of discourse, as dictated by the contents of the data base. However, the system is designed in such a way, that a great part of the program is independent of the actual universe of discourse; this may become clear at the end of this section.

The process that answers a question, is divided into three subprocesses: the interpretation of the question (resulting in an evaluable expression), the evalua-tion of that expression (resulting in the value of the answer) and the formula-tion of the answer. *In* the current paper, only the interpretation process is discussed. In order to get a transparent and well-structured program, the inter-pretation process is too complicated to be performed in one giant step. Between the natural language and the level of the evaluable expressions, a number of inter-mediate language levels has been designed The conversion from an expression at a certain language level to its equivalent at the next lower level is performed by program modules called converters. The general and common aspects of the control structure of those converters form the topic of this paper.

The concrete definitions of the intermediate languages are not relevant to this discussion, and are not given.All these languages have one aspect in common: except for the uppermost level (the natural language) all expressions in these languages are represented as trees. All converters (with the trivial exception of the uppermost one) will have a tree as input and will produce a

(converted) tree as output.

A few words on the universe of discourse: the prototype of PHLIQA contains a small data base on computer-configurations and their users. A few properties of both kinds of objects are stored in this data base. Kor the confi-purations there are, among others, the name of its central processing unit model, the number and type of its peripherals and the name of the manufac-turer. For the users, the data base contains, among others, the company name, the street address and country of the si te.

To come back to tho point of the universe-of-discourse dependence, mentioned earlier: the possible structures of the tree (in other words: the different types of branchings of its nodes) are universe-of-discourse inde-pendent. The only nodes that do have universe-of-discourse dependence, are the terminal ones, which represent constants (e.g. for certain sots, ele-ments of sets, functions on them). A converter is therefore only universe-of-discourse dependent in so far as it manipulates those constants explicitly. An example may clarify this point: a certain sub-tree may represent a expression, the semantics of which are verbally given by"for the set of all configurations, apply the function "name of the C.P.U. model" to the elements of this set, and consitute the set, consis-ting of the results of this function-application". Now, the construet"run through all elements of a set, apply a function, and build up a hew set from the results" is universe-of-discourse independent, where the concrete set "configurations" and the concrete function "name of the C.P.U. model" are univorso-of-discourse dependent,

2. Ha-sic assumptions

2.0. In order to be able to develop the algorithmic structure of the converters, a number ol design decisions have to be made. Some of thorn are axioms: they have to be postulated at the beginni*n*^.Other design decisions only arise due to the actual selection ol" the set of axioms. We will start with our set of axioms, and because we are not going to treat them in a formal way with correctness proofs, we prefer to call them basic assumptions.

2.1. Each converter will perform its task by means of transformations, opera-ting on a sub-tree and producing the lower level equivalent of that sub-tree, (in most cases, only the top nodo of the sub-tree is involved.) When all possible sub-trees have been transformed, the converter is ready.

2.2. A basic difficulty in all A.I. systems is the existence of ambiguities. In our system, wo may have ambiguities inside any converter. A converter may perform more than ono transformation on a certain sub-tree without being ablo to decide which transformation(s) is (are) the correct one(s). These branching points in the program constitute the decision tree. After having selected a certain decision (a branch at a branching point of the decision tree) a converter may meet another ambiguity, leading to a new branching point in the decision tree. The design decision made here is twofold : (l) our system allows for ambiguities in all converters, and (2) in some way or another, all possibilities at a certain branching point are elabo-rated systematically. There might or might not be any preference for the order in which the transformations are "tried", but all of them should have their turn at soon as their predecessors fail.

2.3. The back-tracking mechanism, necessitated by the ambiguities, may be implemented in several ways. We choose for a set of (recursive) normal proce-dures - as opposed to co-routines - where each transformation procedure contains three parts:

(1) the transformation proper
(2) the activation of the next transfor-mation
(3) the cancellation of (l)

2.4. Within a converter, the possible transformations will have no internal static order. In other words, as soon as we require that transformation Tl prece-des transformation T2, Tl and T2 have to belong to different converters.
As a consequence, the program is comple-tely free to perform any transformation of a converter as soon as it sees that this transformation is applicable.
Furthermore, this assumption allows us to have, for each converter, *one* central procedure that looks for the applicabili-ty for all the converter's transforma-tions while it searches the tree.

2.5- Transformation rules are formulated in the form "if existence condition then if applicability condition then transfor-mation fi fi". As one can see, those two conditions do not differ with respect to the system's reaction if they hold for a certain sub-tree; their difference (which one could not see from the simple state-ment given above) is the reaction when they do not hold. When for no sub-tree an existence condition is true, the corres-ponding converter is ready; the complete expression is given to the next conver-ter, if any. When for a certain sub-tree the applicability condition is false (and necessarily the existence condition is true) this means that the corresponding sub-tree cannot be transformed*i* the process is in a dead end of the decision tree. The system should then back up to the previous converter. Some converter may use special cases of this general construct: the topmost converter, which will be called the parser below (although it does more than just a syntactic parse)

does not have an existence condition for separate sub-trees, or rather, the existance condition is always true as long as the complete input question has not yet been reduced to one syntactic symbol.

2.6. Transformations are done in a bottom-up order with respect to the position of their operands in the tree. A node will be transformed if all its descendants are either transformed already or do not need a transformation,

3- Basic assumptions clarified by examples.

A certain convertor inside the system has to replace certain sets by other sets. E.g. the set "computers" does not exist at the data base level; it has to be replaced by either the set of "configurations" or the set of "C.P.U.s". The transformation, mentioned in 2.1, is simple in this case: a sub-troo representing "computers" is replaced by another set. The ambiguity *of* assumption 2.2 is also present hore: the replacement can be done in two ways. The general strueture of the transformation procedure, given in 2.3, will for example look like

1) replace "computers" by "configuratime"
{ activate next transformation
replace "configurations" by "coimputEE**

The net result of the execution of this procedure is nil, but succeeding transformations will process the correct tree, because they are activated from the central point of their predecessors, and not at the end.

Assumption 2.4 says that there is no problem whatsoever to replace in the same converter the set "users" by "corporations". However, it is dangerous to combine the replacement of "computers" by "C.P.U.'s" and the replacement of "C.P.U.'s" by "model names" in the same converter. The search process would then have the task to scan the whole tree, as soon as one of the converter's transformations has boon performed. The existence condition, mentioned j.n 2.5, is very simple in this case: if there is an occurrence of the set "computers" in the tree, the existence condition for this converter holds. The applicability condition could in this case 1ook for the context of the sub-tree to be replaced. It is conceivable that the set "C.P.U.'s" does not fit in the context, where "configurations" would fit. We then say that the applicabi1ity condition for the first transformation is false, while it is true for the second transformation.

4. Further design considerations

4.1. When designing a general control strueturo, one of the fundamental decasions to bo made is how the system should react when a sub-1roe has more than one applicable transformation. We will elaborate on two alternative approaches:

(1) select one of the transformations and perform it; the process has to be backed up to this selection point to perform *one of the* remaining alternatives;
(2) defer the selection decision and see if there are other sub-trees for which only one transformation is applicable.

Although this paper mainly deals with the second alternative (sections 5 and 6) we will make a few remarks here that are either valid for both approaches ox* only apply to the first approach.

The first alternative is well knwon. It can be characterized by saying that the program uses a depth-first strategy.Its use in the current system is justified by the additional observation that for some converters, on almost all occasions, the number of applicable transformations is 1. This suggests that the performance of the depth-first stratogy is not much worse than the performance of other strategies, while it avoides the bookkepping overhead of those strategied.

4.2. The central procedure mentioned in 2.4, to bo called "condition procedure", will havo to search the complete tree, Therefore, this condition procedure wi11 use recursion; at any moment, there will be as many instances of the procedure as the current node has ancestors in the tree. As soon as an applicable transformation is found, this transformation is performed (4.1-1).

4.3. The activation of the new search process from a point inside a transformation procedure may be done by a recursive call of the whole converter, which always starts with an invocation of tho central condition procedure. There are as many instances of the convertor active as there are successful transformations.

4.4. As soon as the condition procedure finds a dead end (which means that there is a sub-tree for which the existence condition holds and the applicability condition does not hold, see 2*5), all existing instances of the current converter have to be terminated.

5- The flexible decision tree strategy

5.1. As stated before, a decision point or a branching point in a program is a point where a decision has to be made with respect to the next action to be performed, and more than one action turns out to be feasible (in our Q.A.-system: when a sub*tree has more than one applicable transformation). So far, all well-known strategies have one property in common: the order in which the declaim points are passed is dictated by the structure of the algorithm. Given this structure and a particular input and a decision at a certain point, the next decision point is completely determined.

Moreover, when the algorithm arrives at this point, it has to make a decision before it is able to continue. The points in the decision tree have a fixed order.

The alternative strategy we wish to elaborate in this section is called the method of the flexible decision tree. This means that a new docision point is riot necessarily added to the decision tree when the algorithm meets it. A decision point may remain "dangling" for a while, and be added to the decision tree at a lutei, more appropriato momont, preferably at a moment that sufficient information is available to make the correct decision.

The main argument for this method is to have an elegant way to improve the overal 1 efficiency. Take the very simple example of a decision tree with two branching points A and B, where A happens to have (for some actual input) *2* alternatives and lor B only one decision is possible. We will prefer a tree where B precedes A. In that case we perform the actions following point B only one, whereas in the case that A precedes B, the actions following B are performed twice, viz. once for each branch of A. Thus, the main phi1osophy behind the method is to defer decisions as long as one possibly can afford. It wi1 1 be cleer, that such a deferment only takes place if the number of possible decisions is greater than 1. In all cases where only one decision can be taken, this should be done immediately. Hopefully, such decisions will *due*reaso the number o1 alternatives at other branching points.

We should realize that the name "flexible decision tree" is appropriate only with respect to the original level of considered decisions. The defermont of a certain decision is the consequence of a (non-deferred) decision, existing one meta-level higher. These "meta derisions" are ordered in a fixed decision tree, just as each algorithm has its fixed decision tree, dictated by the strueture of the algorithm. At this meta-levol, it is again true that the algorithm has to make a decision (concerning whether or not to defer a decision of the lower level) before it is able to continue.

3.2. The flexible decision tree strategy implies the necessity of an explicit (data) representation foar decisions, because the program has to manipulate them. Other strategies do not need such a representation; their docision points are implicitly present in the program.

A decision point is characterized by two things: the sub-tree to be transformed and the set of applicable transformations for that expression. When a sub-tree is cneountered fur the first time, a condition procedure will tell which transformations are applicable, if any. Af a later stage, the property of a decision point may change, i.e. the number of still applicable transformations may decrease, or one of the transformations may have been seleeted. Let us therefore decide that the representation of a decision point consists of a reference of the top node of the corresponding sub-tree, a list of references to still applicable transformations and possibly the selected transformation.

5.3. The results of the lloxible decision tree strategy (in terms of the efficiency of the complete program) might be improved if decisions influence each other.Tf a certain transformation produces information that is a help for previously deferred decisions, by means of reducing the number of alternative transformations at that decision point, it is certainly worthwhile to make use of this extra information.

In general, it means that the condition procedure for its evaluation not only maJkes use of the information of the sub-tree concerned, but also uses information of other sub-trees. In our system, where the flexible decision tree strategy is used during the (bottom-up) parsing process, we have a clear example of such helpful extra information, viz. the (syntactic) context of the node to be reduced.

The existence of such helpful information implies, that the conditions may include terms of which the truth value is not known, because the corresponding information is not yet available. This imp!ies that the evaluation of these conditions has to be done in a three-valued logic: "true", "false", and "unknown".

5.*k*. If wo do have dangling decision points, what is the appropriate moment to reconsider them ? One could think of a very general control structure whore all dangling decisions are reconsidered *an* soon as any other decision has been made. This set-up will presumably lead to inefficiencies because derisions arc then reconsidered many times, without new information becoming available. A better approach is to let decisions tell which other decisions they might influence,and to reconsider only the indicated decisions.

Instead of developing a general "theory" for these influences, we will see immediately what this means in the case of a bottom-up parser. It is immediately clear' that two neighbours (two consecutive terms in a partially reduced sentence) might influence each other: one of them might belong to the context that is needed to make a correcf decision for the other. A successful transformation on a certain node (in this case: a syntactic reduction or another parsing decision) is the appropriate occasion to re-evaluate the conditions for its neighbours.

3.5- As long as we cannot prove that for all the sub-trees to be transformed a situation will be reached in which only

one transformation is applicable, the system's control structure should be able to handle those ambiguities. Their number might decrease by an intelligent use of helpful information, but they do not - at least in our system - completely vanish. This means that the program has to make an (arbitrary or "best") choice for one of the dangling decisions. *In* that case, it will use the strategy to select the decision point having the lowerst number of alternative transformations. This strategy is used by human beings when they solve cryptarithmetic puzzles and keeps the decision tree as small as possible. The selection of the next sub-tree to be transformed can now be done by one selection process, both for unambiguous and ambiguous sub-trees. As unambiguous sub-trees only have one applicable transformation, they will automatically be selected prior to any ambiguous sub-tree,

6. The algorithm of a flexible decision tree parser

6.0. In this section, we will show the program lor the flexible decision tree strategy, as applied to the bottom-up parser of our system. The decisions of the parser relate to syntactic nodes. In contrast to the other converters of the system, the parser produces sub-trees on which itself has to operate, as long as the final reduction to the syntactic node "sentence" has not yet been made. Each time a new syntactic node is produced, its corresponding decision point is added to a set of dangling decisions, called "DAD". Obviously, when the parsing process is ready, this set is empty. The existence condition for the parser is true, as long as the set DAD is not yet empty.

6.1. A decision point is represented by a structure, declared by mode decision point = (ref node trafo top,
[1 : T ref proc trafos ,
ref proc condition,
int selected,
bool trafos known);

The variables "selected" and "trafos known" indicate the status of a decision point. If "selected" / 0, the corresponding value indicates the number in the list of trafos that has been selected,In this case, a decision has been made, so the decision point does not belong to DAIX If "selected" = 0, the decision point belongs to DAD. There are however two kinds of dangling decisions: those for which all applicable trafos are known,and those for which that is not the case. (This is the three-valued logic mentioned in 3-3.) The difference is indicated by the variable "trafos known". If "trafos known" is false, not enough information was available to determine which transformation were applicable; in that case, sooner or later the procedure in the field "condition" has to be (re*Jactivated. The re-evaluation of the corresponding condition expression may or may not lead to "trafos known" is true.

6.2. The parser starts with the construction of decision points for all the words of the input question, and places these decision points in DAD. In algorithmic form, the main program of the parser looks like

```
proc parser =:
    for word thru all words of input sentence
        do add to DAD (word ) od;
    do next action;
end of the parser.
```

In this program, "do next action" is the name of a procedure that is the heart of the flexible decision tree strategy; its algorithm is given in 6.*k*. It is important to realize, that when the program is executed and the semicolon following the call of "do next action" is met, all alternative transformations on all nodes have been performed. We could replace the semicolon by "; output ("aJ1 interpretations of the current question have been tried");".

6.3. We will first proceed with the procedure "add to DAD". We do not only need it for the words of the input sentence, but also for the (top of the) partially reduced subsentence.

```
proc add to DAD = (ref node node):
    decision point dp;
    determine the condition procedure,
    that belongs to this node; apply
    this condition procedure;
    its result may be:
      true:    all transformations that have
               found to be applicable,are
               stored in the list "trafos"
               of dp; trafos known:=true;
      false:   no transformations were appli-
               cable, the parser is in a
               dead end of its decision tree;
               trafos known := true;
      unknown: not enough context of this
               node was available to evaluate
               the condition;
               this very condition procedure
               is stored in "condition" of
               dp;
               trafos known := false;
    selected ojf dp := 0;
    the decision point dp is now
    added to DAD
end of the procedure "add to DAD".
```

We will not give the details of the condition procedure; there is one condition procedure for each syntactic category and they use more than Just syntactic information. We only remark that they only work with applicability conditions; the existence condition for the parser is used on one central place,viz. the procedure "do next action".

6.J4. The procedure "do next action" has the responsibility to select the next decision point that has to be added to the decision tree. Each of the appxicaoie

transformations should have its turn. Together with the test for the existence condition, this leads to something like:

```
proc  do next action =:
      if  DAD is empty
      then  the existence condition is not
            longer true, so activate the
            converter that succeeds the
            parser
      else  among the decision points in
            DAD with trafos known, select
            dec!sion point d with the
            minimum number of applicable
            trafos; remove d from DAD;
            for t thru  applicable trafos
                        of d
                do perform trafo t on node
                   of d od;
            d is added to DAD again
      fi
end of procedure "do next action".
```

The central point is this procedure is the activation of a transformation, in the program abovo stated by "perform trafo t on d". Here again, coming back to the calling program means that all alternatives resulting from transformation t have been tried, either leading or not loading to a complete interpretation of the whole question. This central point is at the same time the most important point of the structure of the whole program. Due to this invocation of a transformation procedure, which in its turn invokes "do next action" (as is shown in 6.5; both procedures are mutually rocursive), there is one instance of "do next action" for each level of the decision tree. In this way, a stack mechanism that deals with recursion enables us to have a clean and correct program that systematically elaborates all alternative transformations. This style of programming, called symmetric programming, was postulated in subsection 2.3- In the procedure "do next action", the symmetry shows up in the fact that addition to DAD is the mirror image of removal from DAD. As the reader may have noticed, no special care is needed for dead ends of the decision tree. Such a dead end occurs, when a decision point has no applicable trarafbrmations at all. Such a decision point will bo selected immediately, while it certainly has the minimum number. The loop through the transformations wil1 be performed zero times, so the procedure immediately backs up; another instance of the same procedure will than proceed by performing the next possible trans* formation related to a decision point higher up in the decision tree.

6.5. The transformation procedures are, in the same way as the condition procedures were, dependent of the syntactic category they have to process. We will show the general structure of such a procedure:

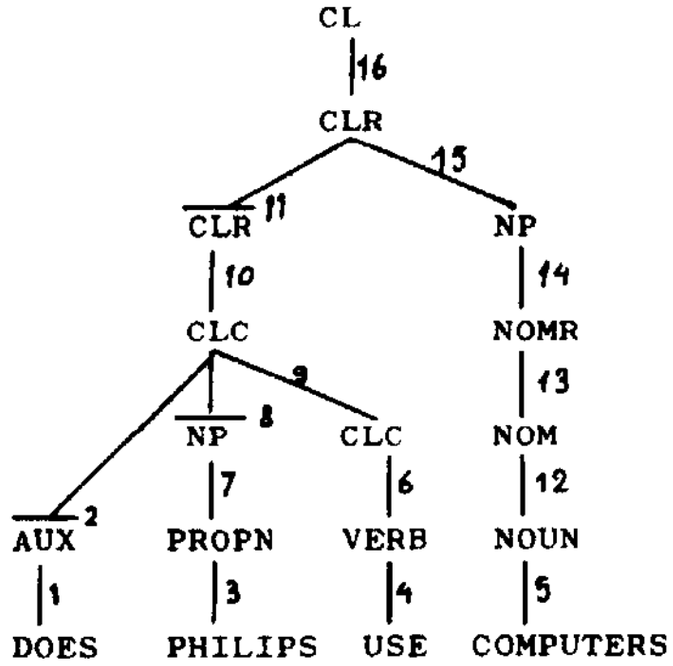
```
proc  trafo = (ref node node):
        perform the syntactic reduction
        from node and zero or more of its
        left-hand neighbours to a new
        node n;
        add to DAD (n), leading to decision
        point d;
        do next action;
        remove d from DAD;
        delete n;
end of a transformation procedure.
```

The fact that for the new node a new decision point should by constructed,and that this decision point is added to DAD, should not need any explanation. The symmetry of the procedure, already discussed in 6.4, is more obvious here.

6.6. Not all transformations of the parser produce a new syntactic node: there is one exception. This transformation, called "shift", transforms the syntactic node into a status that means that the node concerned will not be the rightmost and controlling constituent of a syntactic reduction. Only nodes that have this "shifted" status, may act as (left) context of their (right-hand) neighbours. This means, that when a node receives this status, this is an excellent moment to re-evaluate the condition procedure of its right-hand neighbour. This brings us to the following algorithm:

```
proc shift = (ref node node):
      change status of node to "shifted";
      re-evaluate the condition procedure
      of the righ-hand neighbour of node;
      do next action;
      cancel the results of the re-evalu-
      ation;
      change status of node to "not
      shifted";
end of procedure "shift".
```

To illustrate the working of the flexible decision tree parser, the parsing of the question "Does Philips use computers ?" is shown:



In this figure, vortical and diagonal linos denote syntactic reductions, horizontal lines denote the shift-trans-formation. (This paper is not the appropriate place to explain to names of the syntactic categories.) The numbers indicate the order in which the decisions actually were made by the parser. The final reduction has the number 16, which shows that the dec!sion tree has 16 levels. This decision tree is not shown hero; the only decisions having more thai 1 applicable transformation are those with the numbers 6, 7, 9 10 and 11. Worth noting i is that decisi*on* 5 (the reduction from COMPUTERS to NOUN) procodes decision 6; in a conventional lolt-to-right parser, decision 5 would have boon made alter decision 11. The doferment of decision 1 2, although having only 1 applicable transformation for this example, is due to the i'act that it needs some context to reach that conclusion. This context is not available before the shift transforma-tion of decision 11. Actually, about the only gain for the flexible decision tree strategy in this example *in* the fact that the transformation of decision 5 is performed only once instead of some 6 times in the case of a conventional parser. Fortunately, more complicated examples lead to better results.

7. Concluding remarks

By means of "symmetric programming", one is able to construct clean, well structured programs for a question-answering system. The symmetry is possi-blo because ali procedures that perform actions know how to activate the next action; there is a central starting point lor the next action, Procedures are terminated only when their results may be destroyed; this allows the pro-grammer to store the results as local quantities, leaving the responsibility for storage allocation to the.stack mechanism. In the case that an explicit separate stack mechanism is needed, it is easy to program it correctly: duo to the symmetry, the statements that care for cancellation are just a few lines lower on the programming sheet than the statements that care for the creation.

The programming language used lor implementation of these ideas, should have

- recursion
- struetures (records)
- pointers (references)
- mforoneOH to procedures (entry vnrLablos).

l'ot our system, wo used an internal dialect of PL/l, the concrete choise being not too important.

As far as the strategy is concerned, we might observe that intelligent systems should use an intelligent strategy to solve their decision problems; from this point of view, the flexible docision tree strategy is certainly more promising than simple depth-first and breadth-frist strategies. There exists a striking and attractive similarity between this strategy and the way in which human beings solve certain combinatorial problems like cryptarithmetic ones.

It is clear on the other hand, that the flexible decision tree is not the final solution for decision problems. In the author's opinion, intelligent strategies should have at least adequate solutions for the following two problems:

- to know to what extent decisions are dependent of other decisions; one could think of an implementation in which decisions are no longer linearly ordered in one single decision tree, but where there exist as many separate decision trees as there are mutually independent chains of decisions.
- to know, at a dead end, at which decision point (presumably many levels higher in a decision tree) a wrong decision was made, so that the program can immediately back up to the correct point, instead *of* trying useless alternatives lower down in the decision tree.