MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

Working Paper 233 May 1982

Code Generation in the Programmer's Apprentice

by

Robert E. Handsaker

ABSTRACT

The Programmer's Apprentice is a highly interactive program development tool. The user interface to the system relies on program text which is generated from an internal plan representation. The programs generated need to be easy for a programmer to read and understand. This paper describes a design for a code generation module which can be tailored to produce code which reflects the stylistic preferences of individual programmers.

A.I. Laboratory Working Papers are produced for internal circulation, and may contain information that is, for example, too preliminary or too detailed for formal publication. It is not intended that they should be considered papers to which reference can be made in the literature.

I. Introduction

The *Programmer's Apprentice* (PA) is a highly interactive program development tool. It is unique among automatic programming systems in that it uses an internal *plan representation* which expresses the functional behavior of a program in a language independent fashion. The PA communicates with programmers and other programs by generating program text which implements the behavior of the underlying plan in a conventional programming language. The module which performs the code generation for the PA is called the *Coder.*

A Coder for the PA faces some unusual problems in code generation because the user interface for the PA centers around the program text which is produced. The Coder must be able to code not only plans representing complete programs, but also partial plans representing programs which are being developed interactively. More importantly, the Coder must produce *aesthetic* programs, i.e. programs which are easy for a programmer to read and understand.

This thesis explores one possible design strategy for a Coder which is able to support the user interface of the Programmer's Apprentice. The approach taken is to develop a Coder which can be tailored to produce code which is sensitive to the stylistic preferences of individual users.

I.1 Related Work

Several source-to-source programming language translators make an effort to generate aesthetic target code, but the techniques they employ often exploit restrictions imposed by their particular domain. Our approach has been more general and aimed at a larger class of applications.

A Fortran to Lisp translator written by Pitman [2], for example, produces target code which can be maintained by Lisp programmers. The target code is written in a highly stylized version of Lisp designed to capture the semantics of the Fortran source in a form which is syntactically similar to the source program. This allows the structure of the source program to be preserved during the translation and its readability to be maintained.

Faust's COBOL to HIBOL translator [1] also produces maintainable target programs, but for only a small class of COBOL programs which are well suited to the semantics of HIBOL. Code generation in this translator is guided by a number of heuristics, many of which do not seem general enough to be useful outside this narrow domain.

The work most influential to that reported here is the coder currently in use by the PA. This coder produces correct Lisp code for both partial and complete plans, but it uses a rigid coding style and very few language constructs. It was never intended to reflect individual coding styles or to accept guidance from the programmer.

II. The Programmer's Apprentice

The PA [3,6] is a unique program development tool. It provides a mechanism for program understanding which allows the computer to play a larger role in the programming process. Because the PA is able to produce program text written in conventional programming languages, it interfaces easily with existing programming tools.

Figure 1 shows a functional diagram of the PA. Within the PA, all information is communicated in terms of the language independent plan representation, although the programmer never interacts with the plans directly. Through the Analyzer, the programmer can create a plan from an existing program and through the Coder, the programmer can view an implementation of a plan in a conventional programming language. The Knowledge Based Editor gives the programmer powerful commands with which to manipulate existing plans and to create new ones by combining plan fragments from a library describing standard algorithms and data structures. Modifications to a plan are communicated to the programmer through changes in the program text corresponding to the plan.

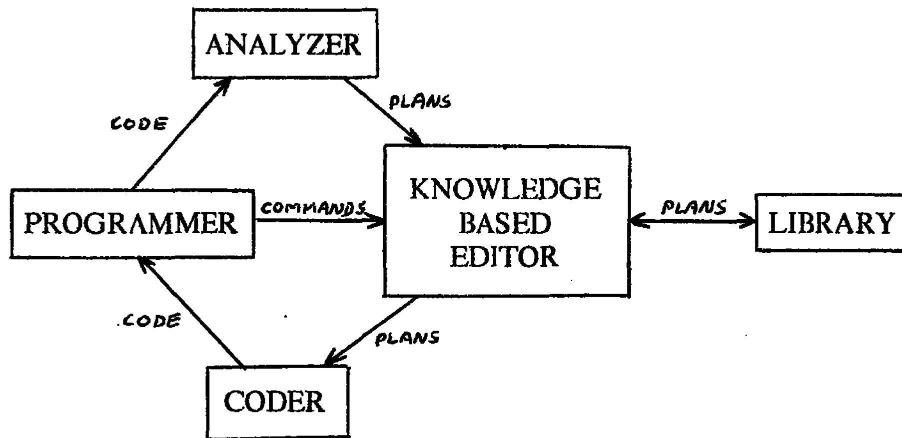


Figure 1. Block Diagram of the PA.

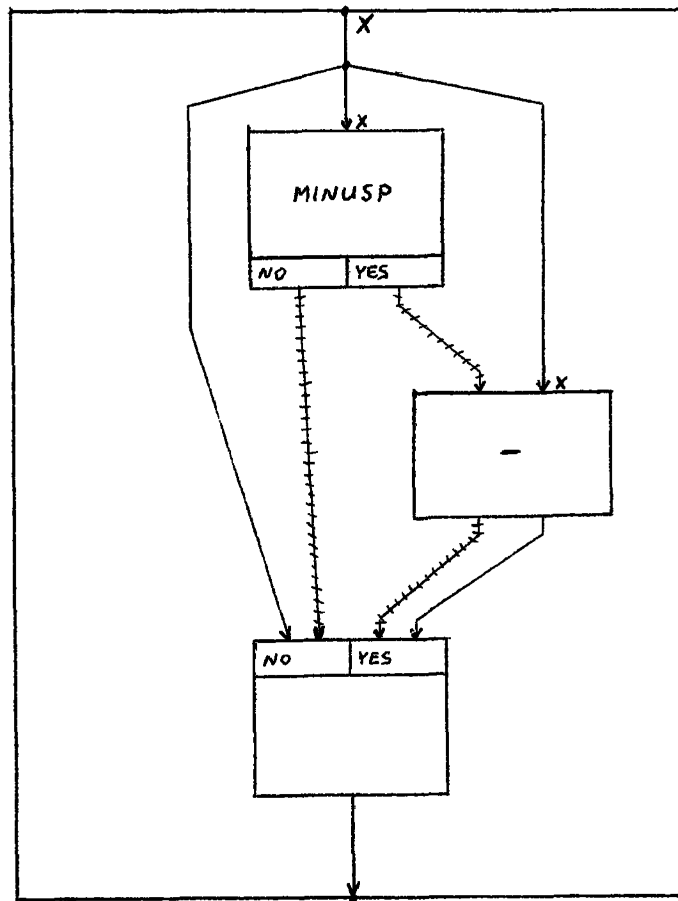


Figure 2. Example plan for `(COND ((MINUSP X) (- X)) (T X))`

The plan representation [5] is the basis for program understanding in the PA. As a first approximation, plans are similar to flow charts, except that data flow as well as control flow is represented by explicit arcs (see Figure 2). The basic unit of a plan is a segment (drawn as a box) which corresponds to a unit of computation. Each segment has a number of input and output ports which specify the input values it receives and the output values it produces. Control flow from one segment to another is represented pictorially as a dashed arc, data flow by a solid arc. The plan for a program is represented by one large segment which is hierarchically decomposed by nesting other segments within it.

III. The Problem

On an abstract level, one function of the PA is to take a behavioral description for a program and transform it, with guidance from the programmer, into program text in some conventional language. We may view this transformation as a series of refinements, with each level becoming more concrete, until finally an implementation is reached. The Coder performs the last step in the chain: the conversion from a detailed plan to program text.

Figure 3 shows a representation of this refinement process. A given behavioral description can be refined into many different plans. These plans differ with respect to which algorithms they use to implement the program. Many of the differences may also reflect language dependent decisions, such as which data structures to use. Other differences may be much less important, such as the way the nesting is done for a particular group of conditional segments. The Coder must supply all of the missing details necessary to implement the plan, mainly an exact evaluation order and mechanisms to implement the data flow.

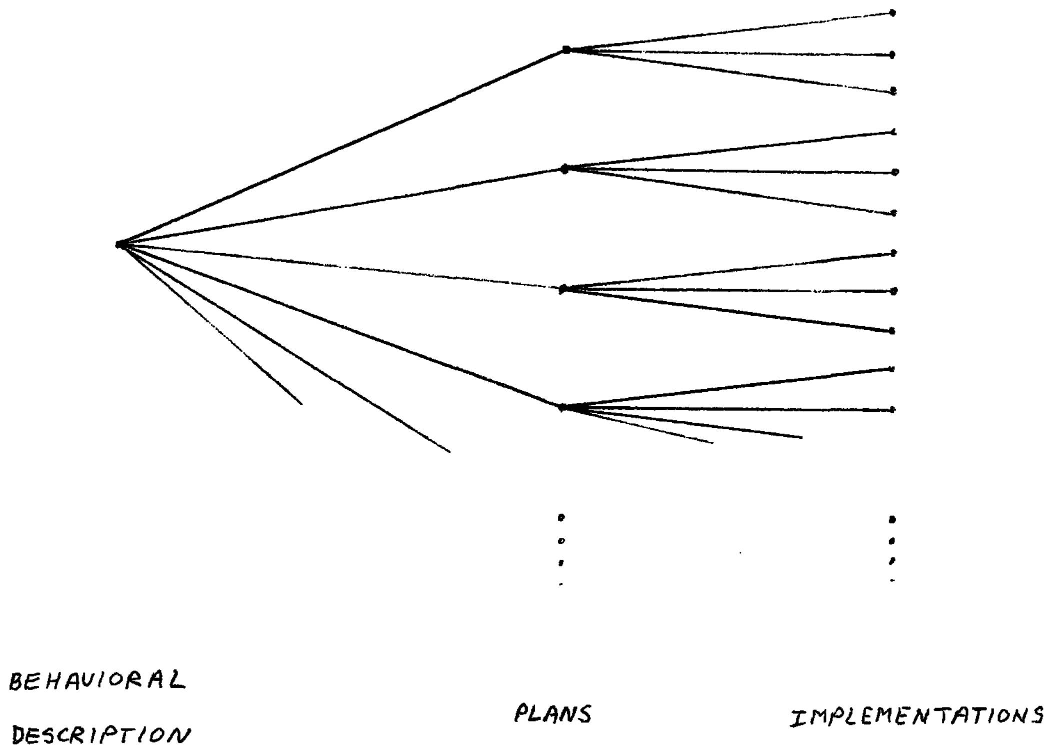


Figure 3.

III.1 Specific Goals

It is easy to produce a program that is a correct implementation of a given plan, but because of the Coder's role as the mainstay of the PA's user interface, it must do much more. To make this task precise, we enumerate some specific goals for the Coder.

First, it is a fundamental requirement that the output of the Coder be a correct implementation of the input plan.

Second, the form of the code produced should reflect as much as possible the basic structure of the underlying plan. The structure of the plan represents the summation of the PA's knowledge about the program and this knowledge is what the Coder wants to communicate to the programmer.

Third, to enhance this communication, it is desirable for the Coder to produce code which can be easily understood and maintained by a programmer. Two types of information are necessary to produce good code: The Coder must have a rich vocabulary of primitive constructs in the target language and, additionally, it must have available explicit aesthetic information about both the target language and the current user. This information will provide criteria for deciding which target language construct is the most appropriate for use in a given situation.

Fourth, to augment its vocabulary of constructs, the Coder must be extendable with respect to any new control constructs or abbreviations which can be introduced into the target language by the user. A Lisp Coder, for example, should be able to digest and utilize new macro forms defined by the programmer. This may require additional information supplied by the programmer, or a learning phase on the part of the Coder, but the final form of the new knowledge gained by the Coder should be modular and additive so that it may be shared among a community of users.

Fifth, because the PA is designed to be useful in any programming domain, it is desirable to have a Coder which is as language independent as possible. Complete language independence would be an unreasonable goal, but it is desirable for most of the basic theory embodied in the Coder to apply to the majority of conventional programming languages.

III.2 Responsibilities and Limitations

To fully define the task of the Coder, it is necessary to define the division of labor between the Coder and the rest of the PA. The Coder's basic responsibility is to generate a program which uses the constructs available in the target language to reflect as closely as possible the structure of a given plan. It is assumed that the plan for a program represents the best logical description which is available--the Coder does not attempt to transform the plan before generating code from it. This division of labor makes the quality of the Coder's output heavily dependent upon the rest of the PA, but it is effective in constraining the task of the Coder and may also be helpful in detecting bad program analyses.

One natural limit on the effectiveness of the Coder is imposed by the form of the plan representation used by the PA. In order to capture stylistic preferences, it is necessary for the Coder to be able to differentiate between two similar programs which the user prefers to code in different ways. In this design, the Coder differentiates between programs using only information available in the plan. Another possible approach, which was not considered here, also takes into account the visual appearance of the code which is produced. Careful programmers place a great deal of emphasis on the overall appearance of the code they write, indicating that this may be an important area for future research.

The plan representation ordinarily associates a conceptual data type with all objects, although the mechanisms for indicating these types are not present in the current implementation. If this information is not available, the effectiveness of the Coder is severely limited. Data type information is essential when the target language is strongly typed, but it is also an important key to producing meaningful code in other languages. As a particular example, **NOT** and **NULL** are equivalent Lisp functions, but they are not conceptually interchangeable because they operate on different conceptual data types.

The *plan calculus* has been developed by Rich [4] to provide a formal basis for the plan representation used by the PA. The plan calculus provides another type of information which is very useful to the Coder, though it has not yet been integrated into the current implementation of the PA. Among other things, the plan calculus allows more expressive methods for describing the decomposition of programs into conceptual blocks. These decompositions are based on the conceptual roles that different plan fragments play in the program. The plan calculus allows a program to be described at several different levels of abstraction.

As a particular example of how this abstract information can benefit the Coder, consider the Lisp CASEQ construct. CASEQ is typically used as a branching construct which dispatches to one of several clauses based on the value of a variable. The difficulty in modeling situations that are typical use of CASEQ is to differentiate them from situations which are better coded using COND. One heuristic which might be applied to differentiate these two situations is based on the number of clauses present. In situations where both constructs could be used, we might choose, for example, to use CASEQ only when there were three or more branches present. We would soon become dissatisfied with this rule, however, because many situations with only two clauses are also more appropriately implemented using CASEQ. No superficial heuristics of this type will ever be completely satisfactory. The intuitively appealing solution is to distinguish appropriate uses of CASEQ using the conceptual information provided by the plan calculus. CASEQ should be used whenever the conceptual operation being performed is a dispatching function.

IV. Design

This section outlines a design for a Coder which shows potential for meeting most of the goals discussed above. This design was conceived specifically with Lisp in mind as a target language, although consideration was also given to making generalizations to other languages where possible (see Section V). The emphasis of the design is on creating an explicit framework for representing aesthetic criteria. If such a framework is modular, additive, and not overly complex, then it can form a basis for a data-driven coder which will be modifiable and extendable by a community of users.

IV.1 Overview

This design centers around modular packets of aesthetic information referred to as *strategies*, explained in Section IV.2. A collection of strategies forms a data base for the Coder defining the aesthetic preferences of a user with respect to a particular target language.

The Coding process consists of applying strategies to small areas of the plan and then combining these strategies into a complete implementation. Strategies dealing with control flow are applied before strategies dealing with data flow. The combining of strategies is guided by a few simple heuristics, described in Section IV.3. After the strategies are combined, a few specific decisions regarding variable names and scopes for local variables must be made. These decisions are covered in Section IV.4.

Because some of the information relevant for the Coder is not conveniently represented in the plan, an important precursor to the application of strategies is reformulation of certain data. This information restructuring, discussed in Section IV.5, is necessary to insure that the effects of all strategies are local.

A great deal of attention must be paid to the efficiency of the proposed design. In particular, because there are an exponential number of ways to combine strategies into a complete implementation, all available information must be used to constrain this process. Section IV.6 outlines several techniques introduced to make the Coder computationally tractable.

Finally, Section IV.7 mentions some of the mechanisms available for communication with the Coder. These mechanisms provide a base on which to build a user interface which allows the programmer to guide the Coder. They also provide a method for the rest of the PA to communicate directly with the Coder.

IV.2 Strategies

Strategies provide a mechanism for representing pieces of aesthetic information in a modular way. The basic purpose of a strategy is to differentiate between two similar plans which the user wants to treat in dissimilar ways. Each strategy describes one possible way to implement part of a plan. A strategy may serve either as a positive or a negative example. For the purposes of this design, this idea is reduced to simply enabling or disabling the strategy. A collection of strategies serves as a data base for the Coder and defines the aesthetic preferences of a particular user. A new strategy should be added to this data base whenever the user has specific criteria for choosing a particular implementation and these criteria can be expressed in terms of the plan representation.

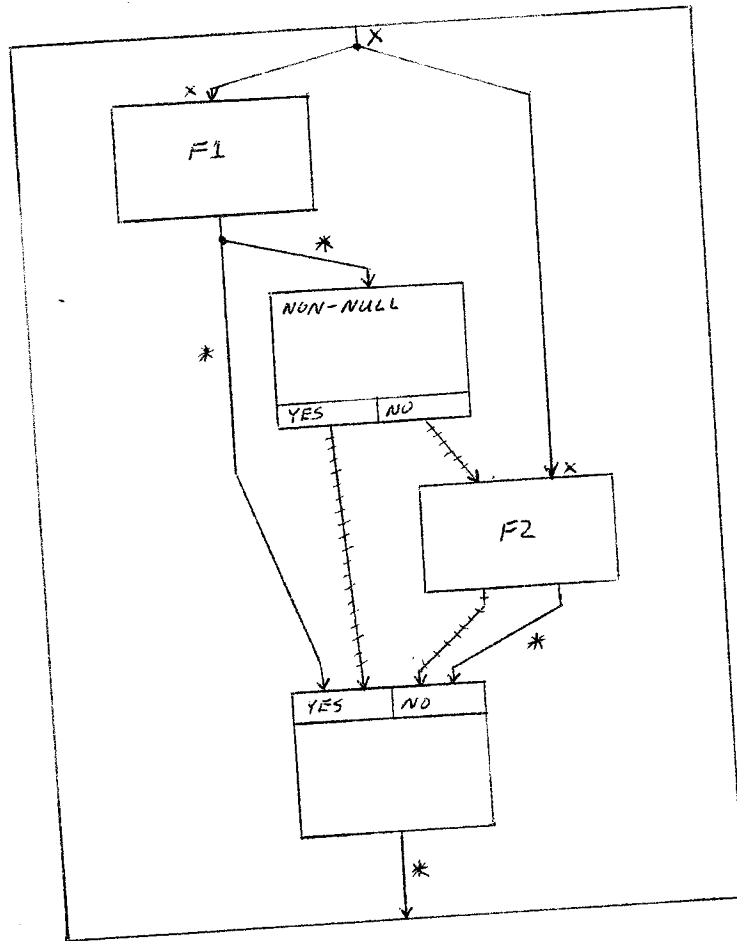
A rough distinction can be made between two types of strategies: *control flow strategies* and *data flow strategies*. This distinction is based on practical differences between the uses and intent of these two types. Data flow strategies generally operate on a more local scale and suggest implementations using constructs which do not affect flow of control.

IV.2.1 Control Flow Strategies

The role of a control flow strategy is to represent one aesthetic use of a control construct from a particular target language. Control flow strategies recognize a narrow class of plan fragments which are typical uses of a language construct and then suggest a specific way of using the construct to implement the fragment. In languages which allow the programmer to define new control constructs (such as Lisp), new control flow strategies for these constructs must be defined before the Coder can use them. Disabling a control flow strategy is equivalent to removing it from the data base, except that the user may still refer to it explicitly (see Section IV.7).

As an example of control flow strategies, consider some typical uses of the **AND** construct in Lisp. **AND** is sometimes used as a predicate, and sometimes as a branching construct. A given usage of **AND** may or may not have side effects, and its return value may or may not be used. Because all of these uses can be differentiated in the plan representation, they can each be represented by different strategies, thus allowing the user to selectively enable each use.

Although control flow strategies are described as encoding information about control constructs, it is important to note that many of these control constructs also have an effect on data flow. When Lisp **OR** is used as a branching construct, for example, it implements not only the control flow arcs shown in Figure 4, but it implicitly implements the four indicated data flow arcs. By contrast, data flow strategies



*DENOTES DATA FLOW IMPLEMENTED IMPLICITLY BY OR CONSTRUCT

Figure 4. Plan for `(OR (F1 X) (F2 X))`

never implement control flow arcs. This is one reason control flow strategies are applied before data flow strategies.

IV.2.2 Data Flow Strategies

The role of a data flow strategy is to represent the mechanisms for achieving data flow which are available in a particular target language. This includes such mechanisms as assignment statements, nesting of expressions, parameter passing, return values and free variables. Data flow strategies are applied after control flow strategies and suggest implementations for any data flow arcs which were not implemented by control flow strategies. Data flow strategies are concerned only with choosing which particular mechanism will be used, and not with the details of its implementation. When a data flow arc is implemented using a variable, for example, decisions regarding the variable name and scope are delayed until just before the code is generated (see Section IV.4).

The intent of data flow strategies is different from that of control flow strategies because there are only a few different data flow mechanisms used in conventional programming languages. A given target language will use some subset of these and, in general, the subset cannot be extended. New data flow strategies, then, never need to be introduced to handle new constructs, but only to differentiate between different uses of old ones.

Although data flow strategies encode mechanisms which never implement explicit control flow arcs, this fact cannot be used to clearly distinguish them. Data flow arcs constrain the flow of control in the same way control flow arcs do, and as a result, control flow arcs function as degenerate data flow arcs. As an example, it is not clear whether the language mechanism of expression nesting should be represented by a control flow strategy, a data flow strategy, or one of each. In the current implementation of plans, control flow arcs within expressions are eliminated, so for convenience, we represent expression nesting with data flow strategies.

IV.2.3 Achieving Modularity

For strategies to be effective as a mechanism for tailoring the Coder to the stylistic preferences of individual users, it is extremely important for them to be modular. The information they use must be available locally in the plan and their effects must also be localized. This is an important precursor to creating strategies which have predictable behavior and are easy to understand and modify.

The meaning of 'local' is different for control flow strategies and data flow strategies. For control flow strategies, a rigorous meaning is difficult to define, but a useful range of effects seems to be one segment within a plan. For data flow arcs, an effective scope seems to be one *data flow cluster.* A data flow cluster is the entire group of data flow arcs which originate on a given port. (If the port where the data flow cluster originates is the output port of a join segment, then the data flow arcs which are joined to create that output port also become part of the cluster).

These guidelines for the scope of a strategy are useful because they correspond to logical parts of the plan, which makes the effect of the strategy easier to understand. For many control constructs whose effects span several nested segments, however, some effort must be expended to achieve this modularity. Usually, single segment modularity can be maintained by defining auxiliary strategies to implement the nested segments. The Lisp **DO** construct, for example, can use auxiliary strategies to detect and implement the iteration forms for local variables or a suitable loop termination to be implemented as the normal exit form.

Some effort must also be expended to limit the scope of data flow strategies to a single data flow cluster. Data flow strategies which attempt to utilize return values from assignment statements, for example, require information regarding the order of evaluation of the destination ports of all data flow arcs originating anywhere on a segment. We need to pre-compute this information and store it locally (see Section IV.5) so that we can determine for a given data flow cluster whether or not a strategy is applicable without inspecting other clusters originating on the same segment.

IV.2.4 Strategies as Labelings

As a unifying concept, both control flow strategies and data flow strategies can be thought of as specifying a pattern of labels to be attached to parts of a plan. Each label specifies which mechanism is used to implement that piece. The pattern of labels on a completely labeled plan uniquely specifies an implementation. There should be a one-to-one correspondence between possible labelings and essentially all legal implementations of the plan. (All implementations which perform exactly those operations required by the plan should have a corresponding labeling. Certainly no labeling would correspond to a legal implementation which had extraneous code that didn't effect the computation).

Labelings are easy to specify for control flow strategies. Each control flow arc or data flow arc would be labeled with the name of the construct, which is the mechanism being used. Labelings are more difficult to specify for data flow strategies, but this is where they are most useful. If one general set of labelings could be devised for all data flow mechanisms in current use, a universal set of data flow strategies could be defined. Adopting these data flow strategies to a particular language, then, would require removing any strategies whose labelings violated the semantics of the language.

As a first step toward creating a complete set of data flow labels, we may differentiate between *implicit* and *explicit* data flow mechanisms. Explicit data flow mechanisms are any mechanisms for implementing data flow which use a *named* location or structure to store the data, such as local variables or a named array. Implicit mechanisms do not have associated names, such as the use of return values or an internal machine stack used for expression evaluation.

In Lisp, there are basically two different mechanisms which implement implicit data flow. The first is data flow which is implemented implicitly be a control construct. The second is Lisp's mechanism for returning values from all expressions, including control constructs and assignment statements. Clearly, labels for implementing the first type of implicit data flow need to be generated on a construct by construct basis. To provide generality, labelings representing return values should probably also be

generated on a construct by construct basis, since languages vary widely with respect to what type of constructs may return values.

In the current implementation of the PA, selection of appropriate data structures from the target language is done in a separate analysis before the Coder receives the plan. Therefore, the only explicit data flow mechanism which the Coder should introduce into programs is the use of variables, which is discussed in greater detail in Section IV.4.

```
(DEFINE PRINT-TAX (EMPLOYEE)
  (LET ((REC (LOOKUP-RECORD EMPLOYEE)))
     (PRINT-LINE
       (GET-NAME REC)
       (GET-TAX REC))))
```

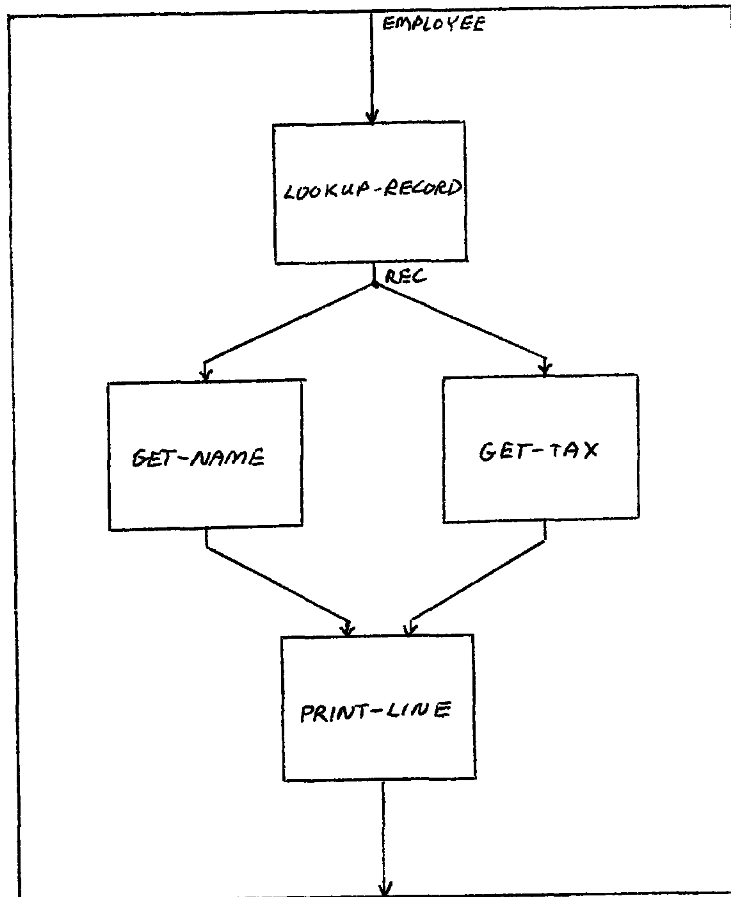


Figure 5. Plan for PRINT-TAX

It is occasionally awkward to apply the data flow labels which have been mentioned to particular implementations. Often part of a plan will seem to require two labels. In Figure 5, for instance, the output port of the segment LOOKUP-RECORD has two different interpretations: In an internal description of the segment, it is implemented as a return value, but viewed from the outside, the data flow is carried in a variable. This is a general problem which arises whenever we try to use a labeling to describe part of an implementation which transfers a data flow between two different data flow mechanisms. This is probably due to the fact that such transfers are abstracted away in the plan representation and as a result, constructs or mechanisms whose purpose is to effect these transfers (such as SETQ or LET in Lisp) have no corresponding entity in the plan that labels can be attached to. Some method for describing these transfers between mechanisms needs to be incorporated into the Coder's labeling scheme, although no effective way to this has been thought out.

IV.3 Applying Strategies Successfully

Enabled strategies serve as generators in suggesting to the Coder possible implementations for small parts of the plan. To create a complete implementation, the Coder needs to choose a particular combination of strategies from the many available. If the rules guiding this decision process are complex and full of special cases, predicting the effect a particular strategy will have on the code produced becomes increasingly difficult. It is therefore desirable that the Coder use only a few simple rules to choose appropriate combinations of strategies, and also that these rules be generally applicable to many target languages to make the Coder more easily transportable.

As part of this design, two general heuristics for guiding the Coder in combining strategies are suggested: the *fewest constructs rule* and the *most specific construct rule.* The fewest constructs rule gives precedence to implementations which use the least number of constructs in the target language. If two implementations use the same number of constructs, it is often the case that the major construct used by one will be less general than the major construct used by the other, in the sense that the functionality of the first is a proper subset of the functionality of the second. In these situations, the most specific construct rule gives precedence to the less general construct.

The fewest constructs rule is motivated by the assumption that implementations generated by enabled strategies will be aesthetically acceptable on a local scale. Thus, the key to creating an aesthetic implementation is to reduce the verbosity inherent in poorly chosen combinations of strategies. Because of the way plans are structured, it is sometimes the case that a construct will be redundantly nested inside itself, as is the OR shown in Figure 6. Unless the Coder can determine a conceptual reason not to, it is

generally desirable to treat a situation like **-a-** as a syntactic abbreviation for **-b-** (see Section V). In this case, the fewest constructs rule should count this as only *one* use of the **OR** construct.

Figure 6.

In certain instances, however, the form of **-a-** may be conceptually clearer than **-b-**. This situation could be recognized by the Coder as follows: If the conceptual information which distinguished this situation was available to the Coder, then a more specific strategy could be defined which would handle the implementation of the nested **OR** in **-a-**. The two **OR** constructs in **-a-** would then originate from two different strategies. Depending on the particular strategies involved, it might or might not be appropriate to treat them as a syntactic abbreviation for **-b-**.

The most specific construct rule is based on the premise that constructs are introduced into a language with specific uses in mind. General constructs exist to give the language maximum expressive power and more specific constructs are included to more conveniently handle frequently occurring situations. Lisp's **MAPCAR**, for example, implements a useful specialized iteration. The basic idea is that if it is aesthetically acceptable to use the more specific construct, that construct is likely to be more appropriate.

For the fewest constructs rule and the most specific construct rule to be effective, any two strategies which suggest a use of the same construct should be applicable to disjoint domains of plan fragments. If this is not the case, these heuristics will not be able to choose between certain implementations. Having disjoint domains for strategies is a desirable property for the Coder's data base of strategies anyway, since it makes it easier to determine the effect of modifying or adding a strategy. An alternative solution might be to preferentially order all of the strategies for a given construct, but it may be difficult to make this modular if different users share libraries of strategies.

It may be necessary to include a few other heuristics to guide the combination of strategies, but this should be done with care. A numerical limit on the depth of expression nesting, for example, seems like a plausible addition, but it has been purposely omitted. This rule is not as general as the two above because it applies to only one type of construct. Further, the benefits of such an arbitrary limit on nesting depth are unlikely to outweigh the additional complexity required to analyze the effects of strategies. In practice, the motivations involved when a programmer breaks up an expression may be closely related to the topographic features of the program. However, once it has been decided to break up an expression, higher level information seems necessary to find conceptually appropriate points at which to do so.

IV.4 Implementing Variables

During the coding process, several critical decisions regarding variables are delayed until after all strategies have been applied and combined into a single implementation. These are decisions about specific names for variables and parameters and about the scopes given to local variables. The goal in delaying these decisions is both to simplify the process of applying and combining strategies and also to simplify the decisions themselves--it is much easier to analyze variable name and scoping conflicts when the control structure of the implementation is certain. After these decisions are made, a simple code generator familiar with the syntax of the target language can produce the final output of the Coder.

The decisions which are delayed by the Coder deal with explicit data flow mechanisms. In particular, they are decisions about the names and scopes which will be associated with the structures or locations implementing these explicit mechanisms. For simplicity, this discussion is limited to the implementation of local variables. The decisions made about local variables may be roughly divided into four areas: choosing names, re-using names, choosing scopes, and implementing initializations.

Choosing mnemonic variable names is a difficult task. Plans contain commentary which suggests names for some variables, but often no good suggestions are available. Many different heuristics could be used to generate names, based for instance on the functional role of the object being transported or on its data type. The parameter names from any procedures using the data object can also be inspected to yield suggestions. Higher level knowledge can also be useful. In particular, if several procedures are passing one data object around as a parameter, it is often customary to use the same name within each procedure.

```
(LET ((X))                  (LET ((X))                  (LET ((X (INIT))
   (SETQ X (INIT))             (F (SETQ X (INIT)))         (F X)
   (F X)                       (G X))                      (G X))
   (G X))

     -a-                          -b-                        -c-
```

Figure 7.

The Coder does not want to maximally re-use local storage, but neither does it want to create a new variable for each named data flow. To create readable code, the key is selective re-use of variable names to implement data flow in different clusters. The approach suggested here is to create new local variables for each data flow cluster by default, but recognize special situations in which it is appropriate to re-use variables. Situations such as 'incrementing operations' are typical places where re-use is appropriate.

There are several methods by which the Coder can choose scopes for local variables. There is a tradeoff between wanting to make variable scopes as small as possible, which makes all variable references textually local in the program, and wanting larger variable scopes which make modifications easier and tend to use less variable declaration constructs. Many language constructs, especially looping constructs, also function as variable declaration constructs and should be utilized by the Coder. The problem of choosing variable scopes was not studied in detail for this design.

Many programming languages have special constructs to allow variable initialization at the point where they are declared. Because variable scopes are not coded until after the strategies have been applied, the code for variable initializations may have to be moved into the declaration constructs. Generally, this movement is desirable, and constraints on this movement are easy to compute from the plan. A problem concerning appropriate choices of data flow labels is illustrated by this movement, however, as shown in Figure 7. One would want to produce `-c-` from either `-a-` or from `-b-` by moving the variable initialization. However, the data flow labels used in `-b-` are different from those used in `-c-` although the `-a-` and `-c-` are labeled the same.

IV.5 Restructuring Global Information

Often, information useful to the Coder is represented within a plan in a form which is not directly useful. Sometimes parts of the relevant information are scattered throughout the plan, and sometimes part of a plan may be critically effected by a small bit of information in a very different part of the plan. In order to construct simple, modular strategies with only local effects, it is necessary to re-organize this information into constraints which appear locally before any strategies are applied to the plan.

As an example, information regarding order of evaluation is essential to the Coder, but appears distributed throughout the plan. Necessary constraints on evaluation order are represented both by control flow arcs and data flow arcs. Determining the order of evaluation of two points in the plan requires computing the transitive closure of the relation expressed by these arcs. This information is needed, for instance, when attempting to nest expressions. Order of evaluation constrains which data flow arcs originating on a given segment may be implemented as return values. Usually only one data flow arc is eligible to be implemented as the return value for segment, but to determine this, all order of evaluation constraints need to be considered. It is to the Coder's advantage to pre-compute the needed order of evaluation information for two reasons: It preserves the modularity of data flow strategies and it reduces the number of possible implementations the Coder has to consider.

As another example, a convenient mechanism is to give the user control over the output of the Coder in specific cases by allowing him to specify particular implementation labels on parts of the plan. This kind of information is located in only one place in the plan but could have wide ranging effects. The user may want to assist the Coder, for instance, by constraining certain ports or data flow arcs to be implemented with variables using particular names. This introduces widely scattered constraints which complicate processes such as detecting name conflicts and efficiently re-using variables. Again, an important aid in these processes is a proper organization of the available constraints.

IV.6 Implementation Issues

A major concern in implementing this design is achieving an acceptable level of efficiency. The Coder must perform quickly enough to be effective as a user interface. There are two processes in this design which tend to reduce its efficiency: The pattern matching used by each strategy to locate applicable plan fragments and the process of combining applicable strategies to create a complete implementation. Among these two, the latter problem seems to be the most difficult for the Coder to overcome.

If implemented carefully, the pattern matching required for a strategy to recognize an applicable plan fragment can be made reasonably efficient. Control flow strategies are the most expensive because they are matched against whole segments, which are essentially graphs. For each segment, all applicable control flow strategies must be located. Most segments, however, can be easily differentiated on the basis of large features, such as number of cases or subsegments, so many strategies can be quickly eliminated from consideration. Those which remain will have a high probability of being applicable and so are worth more detailed consideration.

If the process of combining strategies into a complete implementation is viewed as happening after all of the applicable strategies for each segment have been located, then it is a formidable task. A better approach seems to be to combine strategies incrementally as applicable strategies are located for each new plan fragment. Care must be taken, however, as the heuristics which guide the combination process (described in Section IV.3) are meant to apply to the complete implementation and not necessarily any parts of it.

A control mechanism which seems appropriate is to allow several different viewpoints to exist during the application and combination of strategies. Each viewpoint would correspond to one particular combination of strategies, which could be represented as a set of labels for part of the plan. Any viewpoint which cannot possibly lead to the an implementation which would eventually be selected according the guiding heuristics can be removed from further consideration. Pruning these viewpoints helps keep the number of ways of combining strategies manageable. Because of the simplicity of the heuristics being used, detecting these useless viewpoints is not difficult.

An important aid to efficiency is to have the strategy application/combination process proceed from parts of the plan which are highly constrained, and therefore have fewer applicable strategies, to parts of the plan which are less constrained. A difficulty arises, however, because there are generally two highly constrained areas in a plan: The top level segment, which usually represents a procedure definition and therefore has a well defined interface, and the primitive segments in the plan, which usually represent procedure calls and are similarly well defined. Both top-down and bottom-up methodologies will fail to efficiently utilize all of these constraints. Of these two, the top-down approach may be slightly preferable, though, because it creates a tendency to push certain data flow constructs (notably `SETQ` in Lisp) toward more deeply nested segments, which seems to be a preferable default. Control schemes besides top-down and bottom-up have not been considered in detail here and may be a fruitful area for future research.

The Coder may also benefit from *firebreaks*, an important new concept being developed for use throughout the PA to improve performance on large programs. Firebreaks basically enclose sections of plans which function as conceptual units and have relatively simple behavioral descriptions. A typical location for a firebreak is around a sub-procedure which has been expanded in line, perhaps because it is not generally useful as a separate procedure. Firebreaks are useful to the PA because they partition the plan into small sections with narrow communication channels between them. The PA may treat firebreaks as separate units during analysis and modification. Changes which occur within one firebreak will generally not propagate outside of it.

The idea of a firebreak may be extended by the Coder to further control the cost of combining strategies. Besides using the large firebreaks described above, the Coder may draw smaller firebreaks around certain parts of the plan whose implementations have little impact upon their neighbors. Predicates, for instance, expressions, and groups of conditional statements behave in this way. The Coder may safely code these groups independently, using the heuristics guiding strategy combination to select a single best implementation for each group. By applying all of the techniques described in this section to these small areas of the plan, the coding process should be made efficient enough to fulfill all of the requirements of the PA.

IV.7 User Interface

The PA is a highly interactive system. As a result, the Coder needs to have an extremely flexible user interface. Besides capturing the general aesthetic preferences of the user, it should also be able to accept extra guidance in specific situations. At any point, the user should be able to control the output of the Coder in as specific a manner as he desires.

A general interface to allow any external entity to provide guidance to the Coder is not difficult to implement. All that is needed is to allow implementation labels to be specified for parts of the plan. These labels would be treated as constraints by the Coder. One way of achieving this is to create for each segment a 'global viewpoint' which would contain these implementation labels. Every viewpoint created for a segment would be required to be consistent with that segment's global viewpoint.

Besides specifying implementation labels, the user interface for the Coder can also selectively enable or disable strategies. Thus the user can instruct the Coder to code part of a plan using a particular strategy which is normally disabled because it is seldom used.

A very useful property for the Coder to exhibit is *hysteresis*, in the sense that, when a programmer is interactively creating or modifying a program using the PA, small conceptual changes to the plan should be reflected in small changes to the code which the user sees. The Coder needs to be influenced not only by the current state of the plan, but also by the way in which it was last implemented. If the Coder exhibits too little hysteresis, the overall control structure of the program may suddenly change, making it difficult for the user to follow the logical development of the program.

The use of firebreaks creates hysteresis in the Coder because the firebreaks limit the distance that the effects of a change can propagate. Because firebreaks enclose conceptual units of programs, the effects of program modification should correspond closely to what the programmer expects. It is interesting to note that firebreaks, which might have been introduced into the Coder purely for efficiency reasons, also improve the performance of the user interface by giving it more human qualities.

V. Other Issues

Since all communication between the PA and the programmer is based on the implementation generated by the Coder, it is essential that the Coder be reliable. The Coder must be able to produce some correct implementation for any given plan, even if none of the existing strategies can be applied. To guarantee reliability, a module which uses a simple, algorithmic approach to coding should always be used to handle cases on which the Coder fails. The coder which is currently being used in the PA is more than adequate for this purpose.

In order to meet the needs of a community of users, the representation of aesthetic criteria has to be modular and additive. Modularity is achieved by limiting the effects of a strategy to a small area of the plan and by consolidating the information necessary to apply a strategy so that it is locally available. In a step toward achieving additivity, the most specific construct heuristic is based on comparison of an intrinsic property of each construct. Other measures, for instance an arbitrary total ordering imposed by the user, would make it difficult to have multiple libraries of strategies that could be shared by a community of users.

A primary concern of this design was that the Coder should be extendable, with regard both to new constructs added to the target language and to new target languages. To adapt the Coder to a new target language one must build a model for the semantics of that language. Viewing data flow strategies as labelings will hopefully make this modeling easier. In addition, control flow strategies can be made to model a language in several ways. In certain cases, a construct may have implementation dependent behavior which should not be relied upon. If no strategies are defined which depend on this behavior, then it has been effectively omitted from the Coder's model of the language. Other constructs which are only rarely used can be modeled by strategies which are disabled by default, since the user can always temporarily enable these strategies in particular situations.

An increasing number of programming languages have mechanisms, such as Lisp macros, which allow the programmer to easily define new language constructs. Strategies are designed to be very simple pattern matching generators to facilitate automatic generation of strategies for newly introduced constructs. The major difficulty in creating strategies from a macro definition is deciding which plan features are relevant in differentiating possible uses of the macro. There is a class of macros, however, which seem to function primarily as simple abbreviations for other constructs, as SETF does, for example, in Lisp. The uses of these macros may be simple enough to allow automatic generation of strategies for them. In the general case, automatically producing strategies for macros seems very difficult, especially

when the macro implements both control and data flow. It may eventually be possible to employ techniques developed for synthesizing natural language sentences, for example, to learn programmer's coding style from examples, but more knowledge is first needed about effective ways to represent aesthetic criteria.

In summary, the overall purpose of the Coder proposed here is to explicitly recognize the problems associated with aesthetic code generation and to investigate methods for representing individual programming styles. Although many aspects still need to be worked out in detail, the basic principles discussed here form what seems to be an effective design for representing some of the stylistic preferences of programmers. It is hoped that this effort will be a first step toward increasingly automated systems for generating aesthetic programs.

VI. Bibliography

[1] G. Faust, "Semiautomatic Translation of COBOL into HIBOL", (M.S. Thesis), MIT/LCS/TR-256, March, 1981.

[2] Kent M. Pitman, "A Fortran to Lisp Translator", *Proc. of the 1979 Macsyma User's Conference*, Washington, D.C., June 1979.

[3] C. Rich, H.E. Shrobe, R.C. Waters, G.J. Sussman, and C.E. Hewitt, "Programming Viewed as an Engineering Activity", (NSF Proposal), MIT/AIM-459, January, 1978.

[4] C. Rich, "Inspection Methods in Programming", MIT/AI/TR-604, (Ph.D. thesis), December, 1980.

[5] R.C. Waters, "Automatic Analysis of the Logical Structure of Programs", MIT/AI/TR-492, (Ph.D. Thesis), December, 1978.

[6] R.C. Waters, "The Programmer's Apprentice: Knowledge Based Program Editing", *IEEE Trans. on Software Eng.*, Vol. SE-8, No. 1, January 1982.