The Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Working Paper 223 February 1982

Programming Cliches and Cliche Extraction

D. Scott Cyphers

ABSTRACT

The *programmer's apprentice* (PA) is an automated program development tool. The PA depends upon a library of common algorithms (*cliches*) as the source of its knowledge about programming. The PA can be made more usable if programmers not familiar with its implementation can add programming knowledge to the PA's library. This paper describes cliches and a technique for adding them to the library.

Because cliches often do not correspond to complete code, the library can not simply be a collection of programs. Instead, a *plan* representation is used. The approach taken for adding knowledge to the library is one of *cliche extraction.* A program containing a particular cliche is converted to its plan. The plan is pruned, with the results of the pruned plan being displayed in a code-like form. Eventually, only the cliche remains. The cliche is then added to the library.

This paper is a revision of an earlier Bachelor's thesis.

Acknowledgments

This thesis is dedicated to Richard Amori who provided much of my early programming incentive and the bases of many of the cliches which were used in writing the cliche extractor program.

I would like to thank Richard Waters, my thesis advisor and proof reader, Maurice Hendon, my grammatical proof reader, Charles Rich, who provided a second point of view on many things, David Chapman, who was of great help in learning how to use the Lisp Machine, Paul Detwiler, without whom the boxes would have been impossible, and the Lisp Machine group for developing the Lisp Machines. I thank my parents for their encouragement.

For helping me have a great summer and providing encouragement while working on the thesis, I would like to thank Shao-li, Peter, Patrick, Yam, Yuvadee, Chris, Howard, Ace, and Bruce, among many others.

CONTENTS

1. Introduction

1.1 Developing automated programming tools

Better ways are needed to help programmers develop programs. Automated programming tools provide one way of doing this. For example, programming languages, editors, and debugging systems use the computer to aid the programmer. However, these systems all have the disadvantage of treating the program as a set of discrete steps, leaving the overall picture to the programmer. A better way is to have the computer understand what the program does and how it does it, rather than just the individual steps involved in doing it.

Automatic programming was one attempt to use the computer in programming. The user of an automatic programming system describes a program and lets the computer write it. Work in this area has shown that programming is far more difficult than had been imagined, and that its automation is well beyond the reach of existing technology.

Although it is not currently feasible for the computer to perform the complete programming process, current technology is believed to be capable of providing significantly better programming tools than are now provided. The *programmer's apprentice* group at MIT is working on the programmer's apprentice (PA), an advanced programming tool. The purpose of the PA is to provide new ways to improve the programming process without taking away any benefits of the old ways.

1.2 Cliches and cliche libraries

The PA group has given the name *cliche* to commonly used programming constructs. Some examples of cliches are filtering, aggregation, and enumeration. A program may be viewed as a system of cliches, each of which is responsible for some aspect of the program's behavior.

Inadequate methods for using cliches in conventional programming are a cause of many problems. Often, programming errors are caused by programmers having to recode cliches whenever they want to use them. Though programmers know what cliches are needed in a particular program, they often forget the details of the cliches' implementation. Furthermore, recognition of the cliches in a program becomes more difficult once they have been coded. This makes the program harder to understand, leading to additional errors.

Programming languages provide some mechanisms to allow for more convenient use of certain types of cliches, but these methods are inadequate. Subroutines are the most common of these mechanisms. They allow some cliches to be used without being recoded. Unfortunately, many cliches can not be expressed as subroutines because they do not correspond to syntactically complete pieces of code.

Using cliches from a *cliche library* to construct programs can greatly simplify programming. Such a library is used by the PA. In addition to allowing the programmer to easily use existing cliches, the library also serves as a programming knowledge base for other parts of the PA.

The cliche library is the heart of the PA, and must be accessible to both the PA and users of the PA. The cliches are stored in an internal *plan* representation because conventional programming languages do not provide facilities for expressing them in a way useful to the PA. It is important that the cliche library be easily maintained by programmers, since its contents are influenced both by the domain and by the programmer. Unfortunately, the plan representation does not make this an easy task. It is difficult, tedious, and error prone for a person to describe a cliche as a plan. Such a technique also requires that the programmer be intimately familiar with the plan representation.

In this paper, cliches and a method which may be used to add cliches to the cliche library are described. The approach taken is one of *cliche extraction.* In this approach, code containing a cliche is written and converted to a plan. Then, the parts of the program which are not parts of the cliche are pruned from the plan, and the remaining parts of the program are displayed in a form similar to the language in which it was written. The cliche is all that remains, and is then put into the library.

1.3 Related Work

There are other systems whose goals are similar to those of the PA, although they do not provide mechanisms which allow users of the system to define new cliches. Instead, someone familiar with the implementation of the system must define the cliches.

The SETL [5] group is defining a language which performs the functions of the PA. In such a system the language is the set of cliches, and new cliches must be defined either using subroutines and macros, or by modifying the compiler. No mechanism is specifically provided to allow cliches to be defined by a user of the system.

In other groups, [1,4] programs are constructed through transformations. In these systems, the transformations are defined using programming languages or internal representations, but they still require an expert on the system to add new cliches.

2. Programming with Cliches

2.1 The Programming Process

Programming may be viewed as a series of transformations from high level abstractions to low level abstractions. When writing a program in this way, a programmer begins with a description of what the program is supposed to do. Using this information and information about other programs to be used with the program, the programmer transforms this abstraction into successively lower levels of abstraction. Each level is a complete description of the program. although the higher levels suppress many details, while the detail of the lower levels tend to hide the general ideas in the higher levels. As the program is transformed, the abstractions progress from a simple description built from complex ideas to a complex description built from simple ideas.

As an example, consider an aircraft controlling program. At a high level of abstraction, the program may be viewed as:

```
Take off.
Fly to destination.
Land.
```

The program may now be transformed to a lower level. Unfortunately, the three actions do not remain isolated; instead they interact with each other. Taking off will depend upon the destination, and landing will depend upon where the plane is coming from. The flying will depend upon both. At this level, the program may be:

```
Find destination.
Compute flight route.
Compute take off based on flight route.
Take off.
Follow the flight route.
Compute landing based on flight route.
Land.
```

The relationship between the actions of various levels of abstraction is usually not straightforward, but may be made explicit through the explicit use of the transforms. Many interrelated abstractions are used to convert the program from one level of abstraction to another. By recording the transformations used, the reasons for the particular choice of implementation used for the program are saved. With the exception of comments, subroutines, and other limited facilities provided by the language, such information is usually lost, so that with each step of the implementation, the program becomes harder to understand, debug, and

modify.

Programming languages provide a subset of the set of actions corresponding to some level of abstraction for a particular application domain. Programming may be viewed as the process of translating an easily understood description such as "Fly" into a sequence of actions in a language. In some cases, the language is well matched to the problem, and there is not a great deal of translation which needs to be done. In other cases, the match is not as good, and greater amounts of translation are required.

2.2 Programming without explicitly using cliches

In this section, a program is developed without using the PA. The program is to print a list of all employees of a company who live in Boston and have paid more than $50 in state taxes during a specified month. There is a file containing the employees, their cities, and their taxes.

The program may be implemented with three different cliches acting on a stream of records. Firstly, there is a generator which generates the employee records. Secondly, there is a filter which passes only the records to be printed. Thirdly, the action of printing the remaining records is performed. The following illustrates this abstraction:

```
INPUTS: MONTH
GENERATOR: (READ PAYROLL-FILE)
FILTER: (AND (EQ EMPLOYEE-CITY "Boston")
             (> EMPLOYEE-STATE-TAX 50.)
             (EQ EMPLOYEE-MONTH MONTH))
ACTION: (PRINT EMPLOYEE)
```

Report generators could use this as a program, but they are not the solution to the problems of programming. Report generators are closely tuned to this type of program since the cliches in this type of program are built into the language. However, report generators quickly lose their appeal when they are used for writing other types of programs. They tend to be too specialized for general programming. If the above program were transformed into Lisp, several changes would take place:

```
(DEFUN REPORT (MONTH &AUX EMPLOYEE)
  (WITH-OPEN-FILE (FILE PAYROLL-FILE ':READ)
      (PROG ()
         LP (SETQ EMPLOYEE (READ FILE NIL))
            (COND ((NULL EMPLOYEE) (RETURN NIL)))
            (COND ((AND (EQUAL (EMPLOYEE-CITY EMPLOYEE) "Boston")
                        (> (EMPLOYEE-STATE-TAX EMPLOYEE) 50.)
                        (EQUAL (EMPLOYEE-MONTH EMPLOYEE) MONTH))
                   (PRINT EMPLOYEE)))
            (GO LP))))
```

The cliches of the higher level abstraction have been scattered in the process of being translated into Lisp, obscuring the actions of the different parts of the program. The employee record generator now occupies code on almost every line of the program (through the control structure for the looping, and the explicit iteration variable). When programmers do this translation, they must explicitly code the cliches. This often results in low level programming errors which distract them from more important, higher level programming problems.

If a programmer has to determine what a program is doing, the transformations must be undone by recognizing the cliches. Many languages provide mechanisms which allow some of the abstractions to remain explicit in a program, the most common of these being subroutines. Subroutines solve some of the problems of ineffective cliche use. They may be stored in libraries, saving the programmer the trouble of recoding them with each use, and the explicit call to a subroutine prevents its code from being scattered.

Unfortunately, subroutines have many deficiencies. The syntax of the language forces them to have specific forms and to be used in specific ways which may or may not be desirable for a particular application. There are very few languages which allow for the definition of more complex constructs, such as new ways to perform iterations. When collected together as a library, subroutines must be made general enough to be useful in a variety of applications, which leads to inefficiency. In many cases, subroutines must be written in the same language as the rest of the program, and, even when other languages may be used, it is often inefficient to do so.

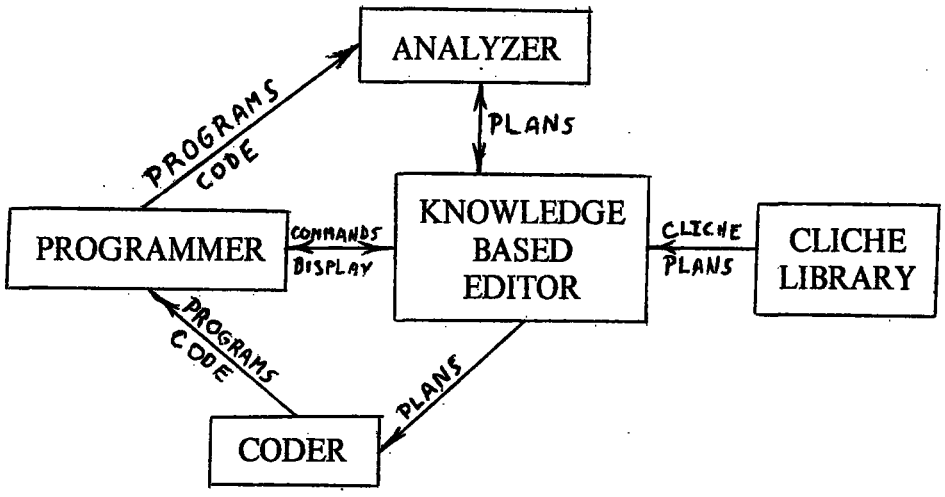
2.3 Explicit use of cliches in programming

The PA provides an alternative to the conventional programming process by allowing the computer to play a greater role. When used to construct a program, the PA allows the programmer to specify the cliches to be used. The PA constructs a plan for the program by combining the plans for the cliches.

Figure 1 shows the interaction of the parts of the PA. An analyzer and a coder provide an interface to the programmer in terms of a specific language. Analyzers, which convert from a source language to the plan representation, exist for subsets of Cobol [3], Fortran, Lisp [6,7], and PL1. An analyzer which uses the cliche library in the analysis is being developed by Brotsky [2]. A coder converts from the plan representation to the source language. Currently, only Lisp [7] and PL1 coders exist.

The PA provides an interface to the user which allows the programmer to interact with the PA in terms of programming languages, whereas the PA itself manipulates plans. A knowledge based editor (KBE) [7] allows the programmer to manipulate the programs. The KBE allows the programmer to add cliches stored in the cliche library to the program being developed.

Fig. 1. Block diagram of the PA



The above report program could be built using the PA as follows:

Note: In this and following scenarios, the programmer types into the top section of the display, and the program responds in the bottom section.

```
>Define a program REPORT with a parameter MONTH.
```

```
(DEFINE REPORT (MONTH)
  ...)
```

In the plan for the program, there is information that part of the program still needs to be written. *Roles*, which will be described in a later section, indicate where these sections are in the plan, and are coded either as "...", representing a role with no name, or "..name..", representing a role with a name.

The programmer implements the "..." portion of the program with a cliche called **REPORT LOOP**. The KBE locates this cliche in the library and adds it to the plan for the program.

```
>Implement the program as a REPORT LOOP.
```

```
(DEFUN REPORT (MONTH &AUX RECORD)
  (PROG ()
     LP (SETQ RECORD (..enumerate..))
        (COND ((..filter.. RECORD)
               (..output.. RECORD)))
        (GO LP)))
```

This cliche has three roles which must be implemented by the programmer: the `enumerate`, the `filter`, and the `output`. The programmer implements the `enumerate` role with a cliche, and fills a role of this cliche during cliche specification:

Note: The PA will not currently handle functions such as "WITH-OPEN-FILE" which contain keywords.

```
>Implement the enumerate as a FILE ENUMERATION of PAYROLL-FILE.
```

```
(DEFUN REPORT (MONTH &AUX RECORD)
  (WITH-OPEN-FILE (FILE PAYROLL-FILE ':READ)
    (PROG ()
       LP (SETQ RECORD (NEXT-RECORD FILE))
          (COND ((NULL RECORD)
                 (RETURN NIL)))
          (COND ((..filter.. RECORD)
                 (..output.. RECORD)))
          (GO LP))))
```

The PA is intended to augment current programming techniques rather than to replace them. To implement the `filter`, the programmer types actual code:

```
>Implement the FILTER as
  '(AND (EQ (EMPLOYEE-CITY RECORD) "Boston")
        (> (EMPLOYEE-STATE-TAX RECORD) 50)
        (EQ (EMPLOYEE-MONTH RECORD) MONTH)).
```

```
(DEFUN REPORT (MONTH &AUX RECORD)
  (WITH-OPEN-FILE (FILE PAYROLL-FILE ':READ)
    (PROG ()
       LP (SETQ RECORD (NEXT-RECORD FILE))
          (COND ((NULL RECORD)
                 (RETURN NIL)))
          (COND ((AND (EQ (EMPLOYEE-CITY RECORD) "Boston")
                      (> (EMPLOYEE-STATE-TAX RECORD) 50)
                      (NEQ (EMPLOYEE-MONTH RECORD) MONTH))
                 (..output.. RECORD)))
          (GO LP))))
```

The program may now be finished:

```
>Implement the output as '(PRINT RECORD).
```

```
(DEFUN REPORT (MONTH &AUX RECORD)
  (WITH-OPEN-FILE (FILE PAYROLL-FILE ':READ)
    (PROG ()
       LP (SETQ RECORD (NEXT-RECORD FILE))
          (COND ((NULL RECORD)
                 (RETURN NIL)))
          (COND ((AND (EQ (EMPLOYEE-CITY RECORD) "Boston")
                      (> (EMPLOYEE-STATE-TAX RECORD) 50)
                      (NEQ (EMPLOYEE-MONTH RECORD) MONTH))
                 (PRINT RECORD)))
          (GO LP))))
```

The PA contains much information in addition to the code displayed. This information can aid the programmer in making changes to the program and in debugging the program. If a programmer wants additional information about the developed code, perhaps at a later time, the PA provides facilities for displaying this information:

```
>Describe the roles of the program REPORT.
```

```
The program is implemented as a report loop with roles:
   Enumerate, implemented as a file enumeration.
   Filter, implemented as
     '(AND (EQ (EMPLOYEE-CITY RECORD) "Boston")
           (> (EMPLOYEE-STATE-TAX RECORD) 50)
           (EQ (EMPLOYEE-MONTH RECORD) MONTH))
   Output, implemented as '(PRINT RECORD)
```

3. The Cliche Extractor

3.1 Introduction

As mentioned earlier, it is desirable for a system to exist which may be used to define new cliches. Because cliches often correspond to incomplete code, code can not be entered by analyzing conventional code for a cliche. A special method is needed to define cliches. This section describes one such method, cliche extraction. Other possible methods are described in the conclusion to this paper.

One approach to solving the problem of defining new cliches is that of *cliche extraction.* To use this method, the programmer writes a program which contains the desired cliche. The programmer then prunes the non-cliche parts of the program using the extractor. This leaves the cliche, which, with additional documentation supplied by the programmer, is added to the cliche library.

3.2 A simple example to illustrate extraction

In the following scenario, the programmer extracts a cliche for averaging two arguments. First, a program is written which contains the cliche.

```
(DEFINE FOO (X Y)
  (/ (+ X Y) 2))
```

The programmer now begins extraction. For use as documentation, the system asks for a description of the cliche.

```
>Extract the program FOO as the cliche AVERAGE.
CLICHE DESCRIPTION: Computes an average of its args.
```

```
(DEFCLICHE AVERAGE
  "Computes an average of its args."
  (/ (+ ... ...) 2))
```

The cliche extractor automatically converts the inputs of the program into roles. At this stage of extraction, there are no descriptions for these roles. They must be supplied by the programmer:

```
>Make the inputs of the + action of the cliche into the role ARG.
ROLE DESCRIPTION: A thing to be averaged.
```

```
(DEFCLICHE AVERAGE
  "Computes an average of its args."
  (/ (+ ..arg.. ..arg..) 2))
```

Since both roles perform the same function, the programmer is able to describe them both at once. It is important to realize that roles are not like variables; they are only places to put things. The `args` will probably be filled in different ways.

The extraction is completed, and the cliche is added to the library:

```
>Add the cliche AVERAGE to the library.
```

During the extraction process, the extractor records the additional information obtained from the programmer about the function of the roles. This information, though not easily shown in the produced code for the cliche, is available to the programmer.

```
>Describe the cliche AVERAGE.
```

```
(DEFCLICHE AVERAGE
 "Computes an average of its args."
 (/ (+ ..arg.. ..arg..) 2))

The cliche AVERAGE has one role type, ..arg.., with two occurrences.
The role arg is a thing to be averaged.
```

Following is an example of the use of this cliche in a session with the PA:

```
>Display the program POSITION-PLANE
```

```
(DEFINE POSITION-PLANE
   . . .
       ..position..
   . . .)
```

The programmer uses the cliche:

```
>Implement the POSITION as an average of 'RUNWAY-RIGHT-EDGE
and 'RUNWAY-LEFT-EDGE.
```

```
(DEFINE POSITION-PLANE
   . . .
       (/ (+ RUNWAY-RIGHT-EDGE RUNWAY-LEFT-EDGE) 2)
   . . .)
```

3.3 Plans and roles

Before discussing cliches in greater detail, it is necessary to describe what plans and roles are. A plan is a hierarchical representation for a program which categorizes and describes various actions of the program and the flow of information between the parts performing the actions.

A plan provides a complete description of a particular action used in programming. It may either represent a primitive operation in the language, such as addition or returning the first element of a list, or it may represent a combination of other plans used to implement the action. Each plan has associated with it information about the control and data flow within the plan. Also, descriptions of the inputs and outputs (ports) are supplied.

In addition to the structural information in a plan, information about what certain parts of the plan are doing, or are supposed to be doing, is provided. In many ways, this information is analogous to comments in a program. These comments are called roles. They consist of a name and a description. Additional information may also be indicated by a role. Roles may exist for inputs, outputs, and sub-plans of the plan being described.

Roles may also describe parts of the plan which are not implemented. Often, a plan represents a class of actions. In this case, the plan does not completely specify an action, but instead has parts which must be filled in when the plan is used in a program. A role is used to indicate what type of thing is to be added to make the plan complete in these situations. When the coder produces code for a role describing a non-implemented portion of the plan, it displays the role name, if one exists, surrounded by dots, for example "`..role-name..`" or "`...`" where the code would be produced, were that section of the plan implemented.

Input roles are coded in a form analogous to function arguments. In the **AVERAGE** example, the **arg** roles were input roles. Sometimes the value of the input is needed in more than one place in the plan. In this case, an assignment to a variable is coded and the variable is used where the value of the input is to be used:

```
(DEFCLICHE SQUARE
  "Computes a square of its arg."
  (PROG (V)
        (SETQ V ..arg..)
        (RETURN (* V V))))
```

The role may be filled either as an input to the **SQUARE** program, or with a plan, as would be done if a cliche were used to fill it.

An output is coded as the variable in a SETQ, unless it describes the returned value of the program, in which case no code is produced. As an example of this, consider the AVERAGE cliche. It may be desirable to have it produce the sum of the args as a second usable value. This would then appear as:

```
(DEFCLICHE AVERAGE
  "Computes an average of its args."
  (/ (SETQ ..sum.. (+ ..arg.. ..arg..)) 2))
```

Plan roles are coded as though they were function calls. All of the inputs to the unimplemented plan are displayed as arguments to the "function". In the following example, AVERAGE has been modified to use arbitrary aggregations and divisions, so that geometric and other types of averages may also be performed.

```
(DEFCLICHE AVERAGE
  "Computes an average of its args."
  (..divide.. (..aggregate.. ..arg.. ..arg..) 2))
```

3.4 Cliches

Cliches are commonly used programming techniques. In general, they do not correspond to syntactic code, since they include unfilled roles where data flow or other cliches are to be put. Though functions are a type of cliche, it is important to realize that cliches are not functions. For example, when the AVERAGE cliche was used in the above program, it was substituted for the role, whereas if it had been a function, it would have been called. Though Lisp allows most cliches to be represented as functions or macros if enough work is done, the flexibility, readability, and ease of modification which explicit use of cliches provides is usually lost in the process.

3.5 Scenarios with the Cliche Extractor

Cliche extraction is a process of pruning the non-cliche portions of a program's plan from the use of a cliche in the program. In the program, the roles of the cliche are already filled. To extract the cliche, the code filling the roles must be removed. The cliche extractor provides several special editing commands for removing this code. In this section, examples of these commands are given.

Cliche extraction begins with a program containing the desired cliche. An existing program may be used, or one may be defined explicitly for the purposes of extraction. If several cliches

are to be added to the library, it may be possible to write one program containing all of them, though, in general, the program should contain as little code in addition to the cliche as is practical, since this code will need to be pruned away during the extraction process.

Once the program has been written, the extraction process begins. Using **EXTRACT**, the extractor is told what portion of the code contains the cliche to be extracted. The programmer is then asked to supply a comment describing the entire cliche. A copy is made of the indicated portion of the program so that the program is not affected, and data flow to the program fragment is removed. The resulting code is displayed as a **DEFCLICHE** rather than a **DEFINE**, indicating that it is to be a cliche rather than a program.

3.5.1 Average revisited

A second look at the extraction of the **AVERAGE** cliche should illustrate the above points. The **AVERAGE** cliche is embedded in a larger program which performs a statistical analysis of family earnings. The original program is:

```
(DEFINE AVERAGE-FAMILY-INCOME (PERSON &AUX SPOUSE)
  (SETQ SPOUSE (SPOUSE PERSON))
  (COND (SPOUSE (/ (+ (* (HOURS SPOUSE) (RATE SPOUSE))
                      (* (HOURS PERSON) (RATE PERSON)))
                   2))
        (T (* (HOURS PERSON) (RATE PERSON)))))
```

In this program, pruning will be required since the cliche is not the entire program. Part of the program may be pruned with **EXTRACT**:

```
>Extract the first action of the program AVERAGE-FAMILY-INCOME as the
cliche AVERAGE.
CLICHE DESCRIPTION:  Averages its args.
```

```
(DEFCLICHE AVERAGE
  "Averages its args."
  (LET (SPOUSE PERSON)
    (SETQ SPOUSE ...)
    (SETQ PERSON ...)
    (/ (+ (* (HOURS SPOUSE) (RATE SPOUSE))
          (* (HOURS PERSON) (RATE PERSON)))
       2)))
```

The inputs to the extracted section of the program were SPOUSE and PERSON. These inputs were converted into potential roles, although, in this case, they will not be roles. The programmer now makes the inputs of the + into roles:

```
>Make the inputs of the + action into the role ARG.
ROLE DESCRIPTION:  An arg to be averaged
```

```
(DEFCLICHE AVERAGE
  "Averages its args."
  (/ (+ ..arg.. ..arg..) 2))
```

Much of the code has been deleted by the extractor because it is no longer being used in the cliche. The cliche may be added to the library.

3.5.2 Loop cliches

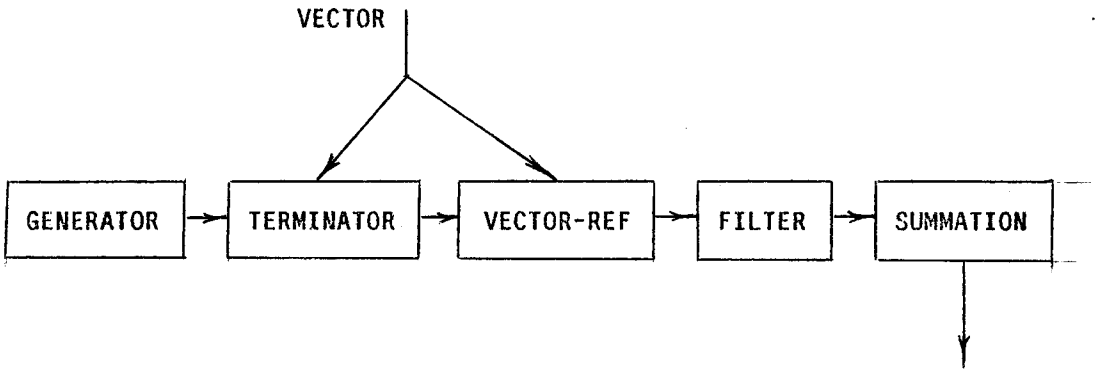
In the following examples, several cliches are extracted from a program. The original code sums the positive elements of a vector.

```
(DEFINE VECTOR-POSITIVE-SUM (VECTOR &AUX SIZE SUM INDEX ELEMENT)
  (PROG ()
        (SETQ SUM 0)
        (SETQ SIZE (VECTOR-SIZE VECTOR))
        (SETQ INDEX 0)
    LP  (COND ((= INDEX SIZE)
               (RETURN SUM)))
        (SETQ INDEX (+ INDEX 1))
        (SETQ ELEMENT (VECTOR-REF VECTOR INDEX))
        (COND ((> ELEMENT 0)
               (SETQ SUM (+ SUM ELEMENT))))
        (GO LP)))
```

In this case, it is necessary for the programmer to know the way in which the plan for the program is grouped. The PA allows the programmer to determine this by highlighting portions of the code which correspond to different sub-plans in the plan for the code. Figure 2 shows this grouping.

In the plan for the above program, there are five things: A **generator** produces a sequence of integers from one to infinity. A **terminator** limits the sequence to **SIZE**. A **vector-ref** uses this limited sequence as input and produces the values in the array as output. A **filter** removes the non-positive values from the sequence of values. Finally, the **summation** adds the

Fig. 2. Block diagram of Vector-Positive-Sum



selected values to produce the output.

3.5.2.1 Filtering positive values

First, the programmer extracts the filter for positive values from the program. The analyzer has grouped this as a unit, so it may be directly extracted:

```
>Extract the filter of the program as the cliche FILTER-POSITIVES.
CLICHE DESCRIPTION:  Filters positive values of a temporal flow.

(DEFCLICHE FILTER-NON-POSITIVES
  "Filters positive values of a temporal flow."
  (PROG (ELEMENT)
     LP (SETQ ELEMENT ...)
        (COND ((> ELEMENT 0)
               (SETQ ... ELEMENT)))
        (GO LP)))
```

In this case, the input and output roles are *temporal.* This means that they use or produce a sequence of values instead of a single value. These are made into roles:

```
>Make the temporal input into the role FLOW-IN.
ROLE DESCRIPTION:  The temporal flow into the filter.

>Make the temporal output into the role POSITIVE-OUT.
ROLE DESCRIPTION:  The positive values from the input flow.
```

```
(DEFCLICHE FILTER-POSITIVES
  "Filters positive values of a temporal flow."
  (PROG (ELEMENT)
     LP (SETQ ELEMENT ..flow-in..)
        (COND ((> ELEMENT 0)
               (SETQ ..positive-out.. ELEMENT)))
        (GO LP)))
```

3.5.2.2 Vector enumeration

In this example, the programmer extracts a cliche for enumerating the elements of a vector. In this case, the cliche consists of several groups of code, the **generator**, **terminator**, and the **vector-ref**, so the entire program must be extracted. Because the extractor modifies a copy of the original program, the original program remains intact.

```
>Display the program VECTOR-POSITIVE-SUM.
```

```
(DEFINE VECTOR-POSITIVE-SUM (VECTOR &AUX SIZE SUM INDEX ELEMENT)
  (PROG ()
        (SETQ SUM 0)
        (SETQ SIZE (VECTOR-SIZE VECTOR))
        (SETQ INDEX 0)
     LP (COND ((= INDEX SIZE)
               (RETURN SUM)))
        (SETQ INDEX (+ INDEX 1))
        (SETQ ELEMENT (VECTOR-REF VECTOR INDEX))
        (COND ((> ELEMENT 0)
               (SETQ SUM (+ SUM ELEMENT))))
        (GO LP)))
```

Extraction begins:

```
>Extract this as the cliche VECTOR-ENUMERATE.
CLICHE DESCRIPTION:  Enumerates the elements of a vector.
```

```
(DEFCLICHE VECTOR-ENUMERATE
  "Enumerates the elements of a vector."
  (PROG (VECTOR SUM SIZE INDEX ELEMENT)
        (SETQ VECTOR ...)
        (SETQ SUM 0)
        (SETQ SIZE (VECTOR-SIZE VECTOR))
        (SETQ INDEX 0)
     LP (COND ((= INDEX SIZE)
               (RETURN SUM)))
        (SETQ INDEX (+ INDEX 1))
        (SETQ ELEMENT (VECTOR-REF VECTOR INDEX))
        (COND ((> ELEMENT 0)
               (SETQ SUM (+ SUM ELEMENT))))
        (GO LP)))
```

The summation must be deleted from the cliche. To do this, the command **DELETE** is used. **DELETE** deletes only the thing which it is told to delete, and does not have extensive side effects, as does **KILL**, described below.

```
>Delete the summation of the cliche.
```

```
(DEFCLICHE VECTOR-ENUMERATE
  "Enumerates the elements of a vector."
  (PROG (VECTOR SIZE INDEX ELEMENT)
        (SETQ VECTOR ...)
        (SETQ SIZE (VECTOR-SIZE VECTOR))
        (SETQ INDEX 0)
     LP (COND ((= INDEX SIZE)
               (RETURN NIL)))
        (SETQ INDEX (+ INDEX 1))
        (SETQ ELEMENT (VECTOR-REF VECTOR INDEX))
        (COND ((> ELEMENT 0)
               (SETQ ... ELEMENT)))
        (GO LP)))
```

Next, the filter for positive values is eliminated.

```
>Delete the filter of the cliche.
```

```
(DEFCLICHE VECTOR-ENUMERATE
  "Enumerates the elements of a vector."
  (PROG (VECTOR SIZE INDEX)
        (SETQ VECTOR ...)
        (SETQ SIZE (VECTOR-SIZE VECTOR))
        (SETQ INDEX 0)
     LP (COND ((= INDEX SIZE)
               (RETURN NIL)))
        (SETQ INDEX (+ INDEX 1))
        (SETQ ... (VECTOR-REF VECTOR INDEX))
        (GO LP)))
```

All that remains to be done is role description. Note that the output is temporal in this case, while the input is not.

```
>Make the input of this into the role VECTOR.
ROLE DESCRIPTION:  A vector to be enumerated.

>Make the temporal output of this into the role ELEMENTS.
ROLE DESCRIPTION:  The elements of the vector.
```

```
(DEFCLICHE VECTOR-ENUMERATE
  "Enumerates the elements of a vector."
  (PROG (VECTOR SIZE INDEX)
        (SETQ VECTOR ..vector..)
        (SETQ SIZE (VECTOR-SIZE VECTOR))
        (SETQ INDEX 0)
     LP (COND ((= INDEX SIZE)
               (RETURN NIL)))
        (SETQ INDEX (+ INDEX 1))
        (SETQ ..elements.. (VECTOR-REF VECTOR INDEX))
        (GO LP)))
```

3.5.2.3 A generator of integers

Another cliche in the main program is the generator of a sequence of integers from one to an upper value. This cliche is also in the `vector-enumerate` cliche which was just extracted. The cliche extractor allows cliches to be extracted from other cliches, and, in this case, this is the easier way to proceed.

```
>Extract the cliche VECTOR-ENUMERATE as the cliche GENERATE-INTEGERS
CLICHE DESCRIPTION:  Generates a sequence of integers from one to an
upper value.
```

```
(DEFCLICHE GENERATE-INTEGERS
  "Generates a sequence of integers from one to an upper value."
  (PROG (VECTOR SIZE INDEX)
        (SETQ VECTOR ..vector..)
        (SETQ SIZE (VECTOR-SIZE VECTOR))
        (SETQ INDEX 0)
     LP (COND ((= INDEX SIZE)
               (RETURN NIL)))
        (SETQ INDEX (+ INDEX 1))
        (SETQ ..elements.. (VECTOR-REF VECTOR INDEX))
        (GO LP)))
```

In this case, the upper bound should be a role rather than a computed value.

```
>Make the second input of the = into the role UPPER-VALUE.
ROLE DESCRIPTION:  The upper limit of the sequence.
```

```
(DEFCLICHE GENERATE-INTEGERS
  "Generates a sequence of integers from one to an upper value."
  (PROG (INDEX)
        (SETQ INDEX 0)
     LP (COND ((= INDEX ..upper-value..)
               (RETURN NIL)))
        (SETQ INDEX (+ INDEX 1))
        (SETQ ..elements.. (VECTOR-REF ..vector.. INDEX))
        (GO LP)))
```

The program changed significantly with the previous step because the uses of several parts of it were removed when the role was added. These changes were made in the plan and reflected in the code by the coder.

The INDEX values are made into an output role:

```
>Make the INDEX temporal output of this into the role SEQUENCE.
ROLE DESCRIPTION:  The sequence of integers produced.
```

```
(DEFCLICHE GENERATE-INTEGERS
  "Generates a sequence of integers from one to an upper value."
  (PROG (INDEX)
        (SETQ INDEX 0)
     LP (COND ((= INDEX ..upper-value..)
               (RETURN NIL)))
        (SETQ ..sequence.. (SETQ INDEX (+ INDEX 1)))
        (SETQ ..elements.. (VECTOR-REF ..vector.. INDEX))
        (GO LP)))
```

All that now remains to be done is to delete the part which produces the elements. The DELETE command just deletes the thing it is given, without side effects. In this case, KILL is more useful. KILL deletes the thing it is given and anything else that becomes unused as a result:

```
>Kill the elements role.
```

```
(DEFCLICHE GENERATE-INTEGERS
  "Generates a sequence of integers from one to an upper value."
  (PROG (INDEX)
        (SETQ INDEX 0)
     LP (COND ((= INDEX ..upper-value..)
               (RETURN NIL)))
        (SETQ ..sequence.. (SETQ INDEX (+ INDEX 1)))
        (GO LP)))
```

The cliche is now extracted and ready to be added to the library.

3.5.3 Regrouping plans

Often the analyzer does not group the program in the most desirable way for extracting a particular cliche. The analyzer uses general guidelines for grouping the program, and though a particular grouping may be good for the extraction of one cliche, it may be bad for another.

Consider the following scenario:

```
>Display the program BOX-COST.
```

```
(DEFINE BOX-COST (LENGTH WIDTH HEIGHT)
  (AREA-COST (* 2 (+ (* LENGTH WIDTH)
                     (* LENGTH HEIGHT)
                     (* HEIGHT WIDTH)))))
```

Given a length, a width, and a height for a closed box, the program computes a cost for material to construct the box. The programmer wishes to extract a cliche which calculates a cost using an arbitrary area function based upon the length, width, and height, so that costs may be computed for other types of boxes.

The programmer begins the extraction:

```
>Extract the program BOX-COST as the cliche CONTAINER-COST.
CLICHE DESCRIPTION: Computes a cost for a box.

>Make the first input into the role LENGTH.
ROLE DESCRIPTION: The length of the box.

>Make the second input of this into the rol- WIDTH.
ROLE DESCRIPTION: The width of the box.

>Make the third input of this into the role HEIGHT.
ROLE DESCRIPTION: The height of the box.
```

```
(DEFCLICHE CONTAINER-COST
  "Computes a cost for a box."
  (LET (LENGTH WIDTH HEIGHT)
    (SETQ LENGTH ..length..)
    (SETQ WIDTH ..width..)
    (SETQ HEIGHT ..height..)
    (AREA-COST (* 2 (+ (* LENGTH WIDTH)
                       (* LENGTH HEIGHT)
                       (* HEIGHT WIDTH))))))
```

The programmer wants the cliche:

```
(DEFCLICHE CONTAINER-COST
  "Computes a cost for a box."
  (AREA-COST (..compute-area.. ..length.. ..width.. ..height..)))
```

The problem is that the area is being computed by a collection of explicit functions. To make all of these into one role, the programmer may begin with the outermost part of the area computation, by making it into a role:

```
>Make the first * into the role COMPUTE-AREA.
ROLE DESCRIPTION: Computes an area.
```

```
(DEFCLICHE CONTAINER-COST
  "Computes a cost for a box."
  (LET (LENGTH WIDTH HEIGHT)
    (SETQ LENGTH ..length..)
    (SETQ WIDTH ..width..)
    (SETQ HEIGHT ..height..)
    (AREA-COST (..compute-area.. 2
                (+ (* LENGTH WIDTH)
                   (* LENGTH HEIGHT)
                   (* HEIGHT WIDTH))))))
```

The `2` is subsumed by the area computation, so it is deleted.

```
>Delete the 2.
```

```
(DEFCLICHE CONTAINER-COST
  "Computes a cost for a box."
  (LET (LENGTH WIDTH HEIGHT)
    (SETQ LENGTH ..length..)
    (SETQ WIDTH ..width..)
    (SETQ HEIGHT ..height..)
    (AREA-COST (..compute-area..
                (+ (* LENGTH WIDTH)
                   (* LENGTH HEIGHT)
                   (* HEIGHT WIDTH))))))
```

If the programmer deletes the + , then the (* LENGTH WIDTH) type calculations will no longer be used, and will be deleted too. What is wanted is to combine the action of the + with the actions to be performed by the compute-area role. COMBINE is provided for this purpose.

```
>Combine the + with the compute-area role.

(DEFCLICHE CONTAINER-COST
  "Computes a cost for a box."
  (LET (LENGTH WIDTH HEIGHT)
    (SETQ LENGTH ..length..)
    (SETQ WIDTH ..width..)
    (SETQ HEIGHT ..height..)
    (AREA-COST (..compute-area..
                (* LENGTH WIDTH)
                (* LENGTH HEIGHT)
                (* HEIGHT WIDTH)))))
```

Now, the remaining multiplications are combined with the actions to be performed by the role:

```
>Combine the first * with the compute-area role.

(DEFCLICHE CONTAINER-COST
  "Computes a cost for a box."
  (LET (LENGTH WIDTH HEIGHT)
    (SETQ LENGTH ..length..)
    (SETQ WIDTH ..width..)
    (SETQ HEIGHT ..height..)
    (AREA-COST (..compute-area.. LENGTH WIDTH
                                 (* LENGTH HEIGHT)
                                 (* HEIGHT WIDTH)))))
```

When something is combined with the actions to be performed by a role, all of its arguments are transferred to those of the role. The extraction continues:

```
>Combine the first * with the compute-area role.

(DEFCLICHE CONTAINER-COST
  "Computes a cost for a box."
  (LET (WIDTH HEIGHT)
    (SETQ WIDTH ..width..)
    (SETQ HEIGHT ..height..)
    (AREA-COST (..compute-area.. ..length.. WIDTH HEIGHT
                                 (* HEIGHT WIDTH)))))
```

COMBINE checks the plan when it transfers arguments to the role from the combined thing. Though LENGTH and WIDTH were both transferred, COMBINE knew that the value LENGTH was already an argument to the role and should not be made an argument a second time.

A second change which took place was the removal of LENGTH from the LET. This is because COMBINE knew that LENGTH now had only one use, and made appropriate modifications to the plan to let the coder know this.

```
>Combine the * with the compute-area role.

(DEFCLICHE CONTAINER-COST
  "Computes a cost for a box."
  (AREA-COST (..compute-area.. ..length.. ..width.. ..height..)))
```

The cliche is now ready to be added to the library.

3.6 Summary of extraction commands

The following section summarizes the actions of the commands provided by the cliche extractor.

`Extract <thing> as the cliche NAME`: Begins the extraction process on <thing>. Asks for a description of the cliche.

`Combine <thing1> with the role <thing2>`: The actions of <thing1> and <thing2> are combined into the actions of the role <thing2>. In LISP, this is like a splice out in the code, though in other languages the appearance of the code change is different. This action is not performed if an invalid plan would result.

`Delete <thing>`: Deletes <thing> from the plan.

`Kill <thing>`: Deletes <thing> and all things that were used only by <thing>.

`Make <thing> into the role NAME`: Causes <thing> to be made into a role. It asks for a description of the role. If <thing> is an input, it `KILL`s the input first.

3.7 Implementation

The actions of the cliche extraction commands are dependent upon the thing which they are given to act upon. For example, the actions involved in making something into a role depend upon whether the thing is an input, an output, a plan, temporal or nontemporal, and other things. Many things depend upon the type of the plan of which they are a part.

To allow for changing the actions relatively easily, a message passing approach is used. Extractor commands and internal commands have message handlers for each of the classifications of objects. This was done using flavors with Lisp Machine Lisp. This proved to be a much better development tool than writing generic functions.

4. Conclusion

4.1 Alternatives to extraction

In addition to cliche extraction, there are two other possible ways to define cliches. These are *cliche construction* and *cliche definition.* In this section, the advantages and disadvantages of these methods are discussed. Neither possibility has been implemented.

Cliche construction allows a programmer to define new cliches in terms of simpler, existing cliches using the knowledge based editor. The method is identical to that of constructing a program from cliches, except that roles must be indicated and given names, and the result of the construction is not a complete program. Because this method uses the KBE, the programmer is not required to learn additional editing commands which are used mainly with cliche extraction. However, new cliches which can not be expressed in terms of existing cliches can not be defined in this manner. The set of core cliches would need to be defined using some other technique.

Cliche definition allows the programmer to define a cliche in a way similar to the way the coder displays cliches. This might be a more convenient way to define cliches than extraction. It would require the programmer to learn a new dialect of the source language, though this would need to be learned in any case since it is the dialect being produced by the coder.

4.2 Extraction

Extraction is an intermediate step in the definition of cliches for the library. Cliche extraction has the advantage of allowing the programmer to define new cliches relatively easily, without requiring a detailed knowledge of the plan representation. However, some knowledge of the plan representation is still necessary, since the grouping of the program by the analyzer has a great impact upon how extraction commands are performed. However, the programmer requires some knowledge of the grouping of plans to use the PA anyway, and the editor [7] provides methods for showing the grouping to the programmer.

The cliche extractor provides several commands which are useful in the editing of cliches which are not otherwise included in the PA. Because cliches are building blocks rather than finished products, the things that are done to them are different from the things done in defining programs.

4.3 The future

The most useful addition to the cliche definition process appears to me to be that of allowing cliches to be defined using code. Such a system could translate such code to complete code, which would then be analyzed. The translations would then be used to drive the extractor, allowing the added code to be automatically pruned. Because most of the process uses existing software, the process should not be overly complex. It also allows the programmer to use the extractor in cases where extraction may appear to be more useful.

5. Bibliography

[1] R. Balzer, "Transformational Implementation: An Example", *IEEE Trans. on Software Eng.*, Vol. 7, No. 1, January, 1981.

[2] D. Brotsky, "Program Understanding Through Cliche Recognition", (M.S. Proposal), MIT/AI/WP-224, December, 1981.

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

[4] Z. Manna and R. Waldinger, "Knowledge and Reasoning in Program Synthesis", *Artificial Intelligence, 6*, 1975, pp. 175-208.

[5] J.T. Schwartz, "On Programming", An Interim Report on the SETL Project, Courant Institute of Mathematical Sciences, New York University, June 1975.

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

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