Mitsubishi Electric Research Laboratories

Technical Report 91-01 **March 12, 1991**

Cliché-Based Program Editing

by

Richard C. Waters

Abstract

Programs can be constructed and modified more rapidly and reliably if they are built out of standard fragments (*clichés*) than if they are written from scratch. Three experimental cliché-based program editors have been implemented, exploring the tradeoff between power and simplicity.

The Knowledge-Based Editor in Emacs (KBEmacs) is the most powerful of the three editors. It supports a wide range of editing operations and can represent a wide range of clichés, because it uses an internal representation called *plan diagrams*, which combines features of flowcharts and data-flow schemas. Unfortunately, the need to convert back and forth between program text and plan diagrams causes KBEmacs to be complex.

The Tempest editor is the simplest and fastest of the three editors. Because it uses text as its internal representation, it does not have to do any conversions. Unfortunately, Tempest is not useful for editing programs, because the manipulations required when combining programming clichés are too complex to be performed directly on program text. However, Tempest's capabilities are useful in other, simpler contexts.

The Ace editor retains much of the simplicity of Tempest while supporting much of the power of KBEmacs. Ace is relatively simple, because it uses parse-trees as its internal representation. It achieves high power by using a specially modified programming-language grammar, which facilitates the representation and easy combination of clichés. This approach could be used to add powerful and efficient cliché-based editing capabilities to any programming environment.

Submitted to *ACM TOPLAS*, March 1991.

201 Broadway
Cambridge Massachusetts 02139

Publication History:-

1. First printing, TR 91-01, March 1991

Contents

1 Overview

Programmers seldom think about programs directly in terms of primitive elements, such as `+`, `<`, and assignment. Rather, like engineers in other disciplines, they think in terms of clichéd combinations of elements corresponding to familiar concepts, such as searching, summing a sequence of values, and computing a result by successive approximation. In addition to improving productivity, this reuse of ideas improves reliability by taking advantage of past experience.

The importance of clichés in programming is evident in a number of ways. The names of clichés form a large part of the vocabulary of programmers. Descriptions of clichés take up much of every programming textbook. Psychological experiments have demonstrated that programmers think in terms of clichés [19].

An important trend in software engineering has been providing increased support for clichés. For instance, an important contribution of high-level programming languages is the introduction of support for certain clichéd data structures and clichéd patterns of control flow.

Cliché-based editors. Many program editors are merely text editors and do not capture any knowledge of programming *per se*. However, some program editors incorporate an understanding of the grammar of a programming language [2, 4, 6, 11, 20, 18, 25]. These *syntax editors* support the manipulation of the few clichés (such as conditional control flow and looping control flow) that correspond to parts of programming language grammars. In particular, standard control constructs can be inserted in a program, and the nested structure of these constructs can be used as the basis for navigating in a program and moving and deleting chunks of the program. However, syntax editors do not support clichés in any general way.

The central feature of a *cliché-based editor* is a large user-extendable library of clichés. Instances of these clichés can be inserted in a program and the resulting nested structure can be used as the basis for navigation and modification. The goal is to allow the representation and manipulation of as wide a range of clichés as possible.

As illustrated in Section 3, the ability to construct and modify programs based on their structure in terms of clichés, rather than on their syntactic or textual structure, leads to significant improvements in programmer productivity and program reliability. However, while clichés account for the lion's share of what goes on in typical programs, they do not account for everything. It is important that cliché-based editors include the capabilities of syntax editors and text editors, just as it is important for syntax editors to include the capabilities of text editors [12, 27].

Cliché-based editors can be compared with other approaches to software reuse (see [3]) along two primary dimensions: *expressiveness* (the range of things that can be reused) and *tailorability* (the extent to which the reused items can be altered to fit the circumstances at hand). Cliché-based editors take an extreme position on each of these dimensions, supporting very high expressiveness and tailorability.

At the low end of the expressiveness dimension, the earliest approaches to software reuse only supported the reuse of subroutines. Current approaches support a wider range of components. However, the primary emphasis is still on the reuse of components that can be combined solely by passing data values between them.

Cliché-based editors support the reuse of traditional components in traditional ways. However, as illustrated in Section 3, they also seek to support skeletal clichés that have to be combined in more complex ways.

At the low end of the tailorability dimension, alteration is forbidden altogether. This has advantages with regard to reliability and is the focus of much of the work on software reuse. However, greater reuse can always be obtained by allowing tailorability.

Cliché-based editors encourage reuse without alteration. However, they allow unlimited tailoring on the theory that when no cliché is exactly applicable, it is better to start with a cliché that is almost applicable than with a blank screen. Each time a cliché is brought into play, the programmer is free to use the full range of editing operations to adjust it.

Representing clichés. The basic feasibility of cliché-based editing was demonstrated some time ago by the Knowledge-Based Editor in Emacs (KBEmacs) [16, 30]. However, KBEmacs is only a demonstration system and would have to be reimplemented with an eye toward efficiency and robustness before it could be used in a commercial setting. Given that KBEmacs is comparable in size and complexity to an optimizing compiler, it is reasonable to consider whether some intermediate steps are possible—i.e., whether simpler approaches could still provide significant support for cliché-based editing.

The central issue in implementing a cliché-based editor is how clichés are represented. This representation must simultaneously satisfy a number of goals [15]. In the current context, three of these goals are particularly important:

1. It must be possible to express a wide variety of clichés.
2. It must be practical to combine clichés as directed by the user. (Clichés are seldom of much use in isolation.)
3. It must be possible to efficiently convert back and forth between the representation for clichés and program text. (This last goal is unfortunately at odds with the first two.)

To explore the tradeoffs between the goals above, two additional cliché-based editors (named Tempest and Ace) were constructed. As a group, KBEmacs, Tempest, and Ace test the efficacy of three different representations for clichés. Figure 1 shows the relative power

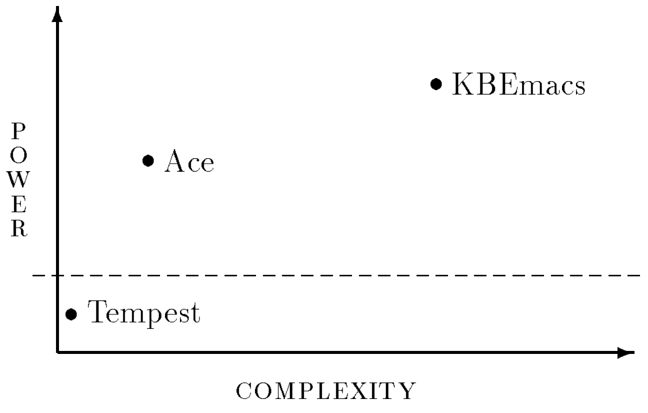


Figure 1: The relative power and complexity of three experimental cliché-based editors. The dashed line indicates the minimum power required for cliché-based editing of programs.

and complexity of the three systems. The dashed horizontal line indicates the minimum power needed to support cliché-based editing of programs.

KBEmacs represents clichés as *plan diagrams*. As discussed in Section 3, this representation is like a flowchart except that data flow as well as control flow is represented using explicit arrows. In addition, the representation is hierarchical in that plan diagrams can be nested within each other and schematic in that a plan diagram can contain boxes that are not yet filled with any specific computation. Plan diagrams are capable of expressing a large variety of clichés and have a number of features that make cliché combination particularly easy. However, conversion between plan diagrams and program text is complex and time consuming.

An obvious way to make conversion between program text and the internal representation used by a cliché-based editor extremely simple is to use program text as the internal representation. This approach was explored by the Tempest system [23] described in Section 4. Tempest is much simpler and many orders of magnitude faster than KBEmacs. Unfortunately, the work on Tempest makes it clear that text is not a satisfactory representation for programming clichés, because it is too restrictive in what can be expressed and because it makes it too hard to combine clichés. Nevertheless, while not useful for program construction, Tempest demonstrates a set of capabilities that could be profitably added to any text editor.

The Ace system [24] described in Section 5 shows that it is possible to make a compromise between KBEmacs and Tempest that comes close to capturing the best features of the two systems. In Ace, clichés are represented as schematic fragments of parse trees. As in a syntax editor, this allows straightforward conversion to and from program text via parsing and unparsing (pretty printing). In addition, if the underlying grammar is designed with care, parse-tree schemas can represent a reasonably wide variety of clichés and can be combined reasonably easily. All in all, Ace manages to support a large part of the functionality of KBEmacs while being much simpler and at least two orders of magnitude faster.

The techniques pioneered by Ace should make it significantly easier for cliché-based editing to become a commercial reality.

2 Clichés

Suppose that a programmer has been given the task of writing a program to print a report of the results of an experiment. Suppose further that the programmer decides to delegate this task to an assistant. The programmer might describe what needs to be done by saying something like the following.

> Write a simple report program `REPORT-TIMINGS` with a parameter `TIMINGS`. Print the title `"Report of Reaction Timings (in msec.)"`. Enumerate the elements in the vector `TIMINGS`, creating a tabular print out of each one. Do not print column headings. Print a summary showing the mean of the elements.

The most important parts of this description are the phrases "simple report", "enumerate ... vector", "tabular print out", and "mean". To understand the directions, one has to know the clichéd algorithms these phrases refer to. Moreover, most of the code which needs to be written follows directly from an understanding of these clichés.

Before considering how clichés like these can be represented and manipulated, it is useful to look in detail at their content. A Cliché has three fundamental parts: a *body* that remains the same in every use of the cliché, *roles* whose contents vary from one use of the cliché to the next, and *constraints* on what can fill the roles.

As a simple example, consider the cliché *tabular print out* described in Figure 2. The top half of the figure describes the algorithm that forms the body of the cliché. An English-language description is used in order to present the essential features of the body independent of any particular programming language that might be used to implement the cliché and independent of any particular representation that might be used for machine manipulation of the cliché.

The notation {*role-name*} is used to refer to roles. The tabular print out cliché has three roles, each of which is referred to several times. The *item* is the object printed. The *format* specifies how the item should be printed. The *width limit* controls the insertion of line breaks.

The bottom half of Figure 2 describes constraints that are part of the cliché. The first constraint specifies a default value for the format role—i.e., a value that should be used if no other value is provided.

The second constraint specifies a fixed relationship between the width limit and format roles. It can be used to derive the value of the width limit based on the format. The derived

```
Body:
  If the number of characters already printed on the current line is greater than {width limit}:
    Start a new line.
  Print {item} as specified by {format}.
Constraints:
  The default value of {format} specifies that {item} should be printed using the standard
      method appropriate for its type.
  {Width limit} is the maximum number of characters that can be printed on a single line minus
      the maximum number of characters that can be output by {format}.
```

Figure 2: English-language description of the tabular print out cliché.

```
Body:
    Open the output file {file name}.
    Print a line containing {title}.
    Print a line containing the current date and time.
    For each element generated by {enumerator}:
        Before the first element is printed and whenever the number of lines already printed on the
                current page is greater than {line limit}:
            Start a new page.
            Print a line containing "page:", the page number, and {title}, along with the current
                    date and time. (The page number of the first page after the title page is 1.)
            {Print headings} appropriate for the body of the report.
        {Print entry} presenting the element on the report.
    {Print summary} at the end of the report.
    Close the output file.
Constraints:
    {Line limit} is the maximum number of lines that can be printed on a single page minus the
        maximum number of lines that can be output when performing the {print entry} and then
        the {print summary}.
```

Figure 3: English-language description of the simple report cliché.

value guarantees that a new line will be started whenever the printed representation of an item might extend beyond the end of the current line.

The simple report cliché. A more complicated cliché is described in Figure 3. This cliché, which specifies a simple method for printing a multi-page report, has seven roles. The *file name* is the name of the file that will contain the report being produced. The *title* and current date are printed on a title page. The *enumerator* enumerates the elements of some aggregate data structure. The *print entry* prints information about each enumerated element. The *print summary* prints some summary information at the end of the report.

Much of the complexity in Figure 3 centers on the difficulties involved with printing multi-page reports. The title and date, along with the page number, are printed at the top of each page of the report. The *print headings* prints column headings on each page of the report that explain the output produced by the print entry. The *line limit* controls the insertion of page breaks. Its value is constrained in such a way that the print entry and summary will never attempt to print off the end of a page, and at least one entry will appear on the same page with the summary.

Suites of clichés. Clichés do not exist in isolation; rather, they have evolved in groups that make sense together. For example, the tabular print out and simple report clichés are linked by the concepts of the number of lines printed on a page and the number of characters printed on a line. Further, the tabular print out cliché fits into the print entry role of simple report and other similar roles.

Figure 4 shows an example of a cliché that fits into the enumerator role of simple report. The body of the cliché specifies how to generate the elements of a vector one at a time, in order. The algorithm is described abstractly in terms of a test that determines when the enumeration is complete, a method for accessing individual elements, and a method for stepping from one element to the next. In most programming languages, these actions would

```
body:
  Initialize a counter to the lower bound of {$vector$}.
  Each time a new element is desired:
    Test whether the counter is greater than the upper bound of {$vector$}.
      If it is, the enumeration has been completed.
    Access the element by fetching the element of {$vector$} indexed by the counter.
    Step to the next element by adding 1 to the counter.
```

Figure 4: English-language description of the vector enumeration cliché.

be implemented as parts of a loop.

A particular programmer might well disagree with some of the details in Figures 2–4. However, most programmers have a basic knowledge of these clichés (and how to compute a mean) and would therefore have no trouble following the directions shown at the beginning of this section. Sections 3–5 show how this knowledge can be codified in machine-manipulable representations that allow a program editor to follow these directions.

A feature that is shared by the editors discussed below is a reliance on *schematic* representations. In each approach, a representation for programs is extended into a representation for clichés by introducing schematic gaps representing roles (e.g., holes that do not yet contain any computation) and annotation specifying constraints. (In this paper, the word cliché is reserved to mean a standard algorithm that programmers know, while the word schema is used to mean a representation of a cliché in a machine manipulable form.) The three editors use three different schematic representations internally; however, they all use schematic program text externally to display information to the user.

3 KBEmacs

This section illustrates the power of cliché-based editing, using the Knowledge-Based Editor in Emacs (KBEmacs) as an example. Since KBEmacs has already been presented elsewhere (see [15, 16, 30]), discussion of the details of KBEmacs is kept to a minimum. This section focuses on the pros and cons of representing clichés as plan diagrams and lays the groundwork for the comparisons in Sections 4 and 5.

Figure 5 shows the architecture of KBEmacs. KBEmacs maintains two representations for the program being worked on: program text and plan diagrams. At any moment, the programmer can either modify the text or the plan diagram. If the text is modified, the *analyzer* module creates a new plan diagram. If the plan diagram is modified, the *coder* module creates new program text.

Three editing modes are supported. To modify the program text, the programmer can use either the *text editor* or the *syntax editor*. To modify the plan diagram, the programmer uses the *schema editor* and *schema library*. The schema library contains a collection of programming clichés represented as plan-diagram schemas. The schema editor can combine schemas chosen from the schema library and manipulate plan diagrams based on their structure in terms of schemas.

Since KBEmacs is implemented on the Symbolics Lisp Machine [37], the standard Emacs-style [20] Lisp Machine program editor is used as the text editor and syntax editor. The schema editor is integrated with the standard editor so that a common interface is presented to the user.

The results of commands to the schema editor are communicated to the programmer by altering the program text seen in the standard editor buffer. The effect is as if a human assistant were sitting at the standard editor modifying the program text under the direction of the programmer.

The examples in this section show KBEmacs being used to construct a Lisp program. As illustrated in [30], KBEmacs can also be used to construct Ada programs. Lisp examples are used here, because they are briefer than the Ada examples in [30]; they are a good basis for comparison with the examples in Sections 4 and 5; and taken together with the examples in [30], they demonstrate the language independence of KBEmacs.

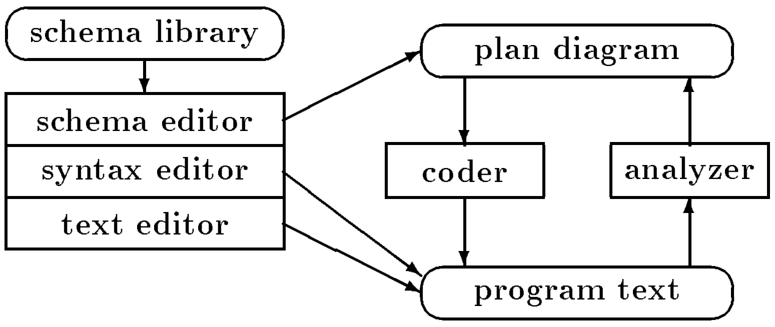
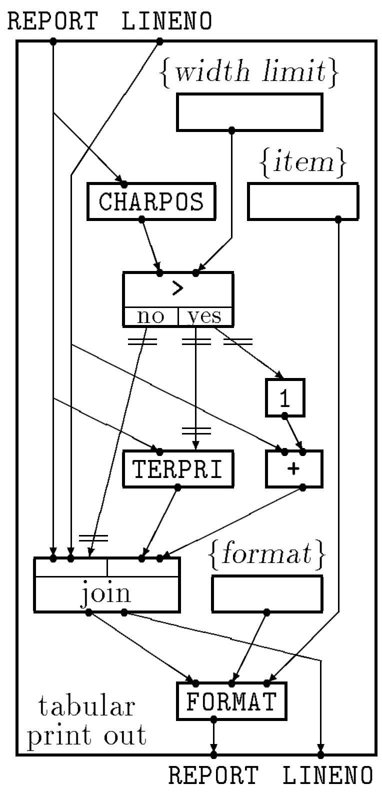


Figure 5: Architecture of KBEmacs.



Default Constraints: {*format*} = `"~15A"`.
Derived Constraints: {*width limit*} = `70-Width(`{*format*}`)`.
Comment: `"prints out` {*item*} `in columns"`.
Described roles: {*format*}, {*item*}.
Input roles: {*item*}.
Primary roles: {*format*}, {*item*}.

Figure 6: The KBEmacs schema for the tabular print out cliché.

3.1 Representing Clichés as Plan diagrams

Figure 6 depicts the schema used by KBEmacs to represent the tabular print out cliché (c.f. Figure 2). The top part of Figure 6 is a drawing of a plan diagram (see [16]). The bottom part of the figure shows constraints and other auxiliary information that is recorded as annotation on the plan diagram.

A plan diagram is a graph in which computation nodes are connected by arcs representing data flow and control flow. In the depiction in Figure 6, computations and roles are represented by boxes. If a box corresponds to a primitive computation, the name of the corresponding Lisp function is written in the box. If a box corresponds to a role, the name of the role (enclosed in braces) is written above the box. If the contents of a box corresponds to a schema, the name of the schema is written in the lower-left corner of the box.

Hierarchical relationships are represented by nesting boxes within boxes. Input and output ports are represented by dots on the edges of the boxes. At the top and bottom of the plan as a whole, ports corresponding to global variables are labeled with the names of the variables.

Data flow is represented by arrows between ports. Control flow is represented by arrows

```
(DEFINE-PLAN tabular-print-out ({item})
    (CONSTRAINTS
      ((DEFAULT {format} "~15A")
       (DERIVED {width-limit} (- 70 (WIDTH {format}))))
     COMMENT "prints {item} in columns"
     DESCRIBED-ROLES ({format} {item})
     PRIMARY-ROLES ({format} {item}))
  (WHEN (> (CHARPOS REPORT) {width-limit})
    (TERPRI REPORT)
    (SETQ LINENO (+ LINENO 1)))
  (FORMAT REPORT {format} {item}))
```

Figure 7: Printed representation of Figure 6.

with cross hatch marks on them. Often the flow of control is implied by the flow of data (e.g., from the box labeled `CHARPOS` to the box labeled `>`). Explicit control-flow arrows are included only when they are not redundant with data-flow arrows.

The box labeled `>` is a test box. It is divided into two labeled regions at the bottom corresponding to the situations where the test succeeds and fails. The control-flow arrows emanating from these regions show the effect that the test has on the overall computation.

The box labeled `join` represents a place where divergent control-flow paths rejoin. The data-flow connections to the join box specify how the choice of control path affects the data flow. For instance, if the `>` test fails, the `LINENO` value that is returned by the plan as a whole is the same as the one provided as an input. If the test succeeds, the returned value of `LINENO` comes from the output of the `+` box.

The most obvious difference between Figure 6 and Figure 2 is that Figure 6 shows a plan diagram rather than an English-language description. However, it differs in two other ways as well.

The last four lines of Figure 6 contain auxiliary annotation that is used by KBEmacs in its internal operation. The *comment* and *described roles* annotations are used for generating descriptions of instances of the schema. Their use is discussed in Section 3.3. Some roles are designated as *input* roles. As discussed in Section 3.3, this information is used to assist in the process of combining schemas together.

When using KBEmacs' command language, a schema is instantiated by using a phrase such as "`tabular-print-out of X and (+ Y Z)`". These phrases specify the name of the schema and may specify computations that fill roles of the schema. The *primary roles* annotation defines which roles can be specified this way and the order in which they must be specified.

The final difference between Figure 6 and Figure 2 is that the algorithm in Figure 6 has been specialized to fit into Lisp. This can be appreciated most easily by looking at Figure 7. (Like most of the figures in this section, Figure 7 is based on a figure from [16]; however, a few cosmetic changes have been made to facilitate comparison with the figures in Sections 4 and 5.)

The Lisp definition in Figure 7 is the external printed form used by KBEmacs for the schema in Figure 6. The definition specifies the name of the schema, a list of of the input roles, the various pieces of annotation, and the algorithm that constitutes the body. The notation {*role-name*} is used to represent a role.

The definition in Figure 7 (which is created from Figure 6 by the coder module) is used when showing the schema to a user, because it is more compact than Figure 6 and more familiar looking to a Lisp programmer. For the same reasons, it is supported as an input notation. Given a definition like the one in Figure 7, KBEmacs uses the analyzer module to generate a schema like the one in Figure 6, which it then adds to the schema library.

The differences between Figures 6 and 7 dramatize an important advantage of plan diagrams as a representation for clichés—they abstract away from the various mechanisms used by programming languages for representing data flow and control flow. In particular, if the algorithm in Figure 7 were written using different control constructs, or in Fortran, it would still correspond to exactly the same plan diagram. This fact is the foundation for the programming-language independence of KBEmacs.

Comparing Figure 6 or 7 with Figure 2 reveals that the algorithm has been specialized in a number of ways. To start with, specific Lisp functions are used to perform the required output. Like the Fortran statement it is named after, the Lisp function `FORMAT` prints values under the direction of an inscrutable but concise control string. The global variable `REPORT` holds the output file. The function `TERPRI` starts a new line.

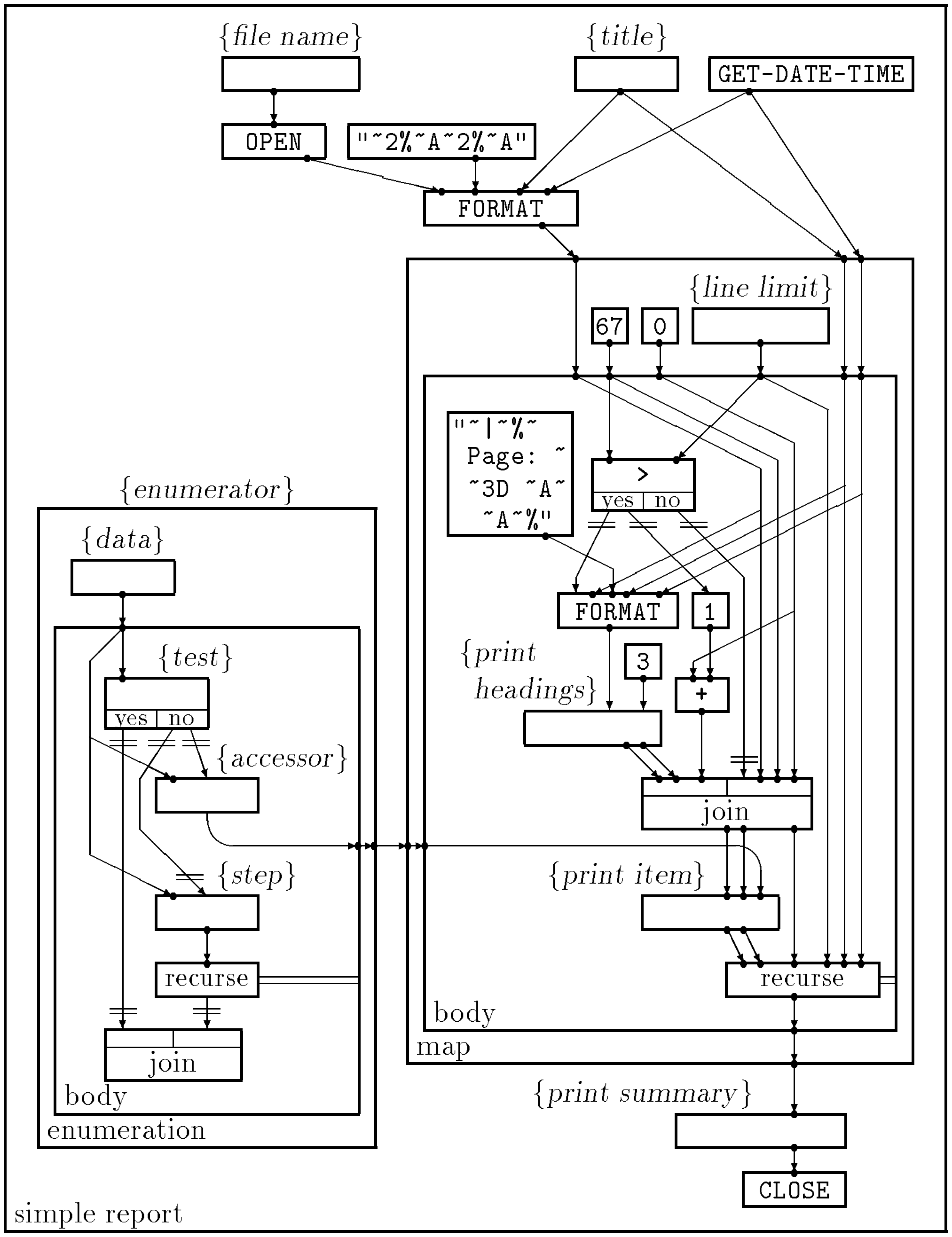
An implicit feature of Figure 2 and the other clichés in Section 2 is the need to keep track of the number of characters printed on each line and the number of lines printed on each page. The function `CHARPOS` accesses information internal to Lisp about the number of characters printed on the current line. Since Lisp provides no internal support for keeping track of the number of lines printed, an explicitly maintained global value `LINENO` is used for this purpose.

The constraints in Figure 7 are defined using a combination of ordinary Lisp code and {...} references to roles. The format role is given a specific `FORMAT` control string as its default value. The constraint on the width limit role reflects the assumption that only 70 characters can be printed on a line. (The function `WIDTH` analyzes the contents of a role and determines the maximum number of characters that can be printed. A number of utility functions of this nature are provided along with KBEmacs for use in constraints.)

The simple report schema. Figure 8 shows the KBEmacs schema for the simple report cliché. Except for the enumeration and map boxes, which will be discussed shortly, it is larger, but not fundamentally more complex than the schema in Figure 6.

Figure 9 shows Lisp code corresponding to the plan diagram in Figure 8. As in the tabular print out schema, a number of Lisp-specific specializations have been introduced in comparison with the general cliché description in Figure 3. In particular, it has been assumed that only 66 lines can be printed on a page, the file name role has been given a default value, and specific output formats have been chosen. The form `WITH-OPEN-FILE` takes care of opening and closing the output file. It is assumed that the function `GET-DATE-TIME` returns a string containing the current date and time.

A significant portion of the computation in the simple report schema is concerned with features that are implicit in the cliché description in Figure 3. In particular, the schema keeps track of the page number and the number of lines printed on the current page. (The relevant computations are associated with the variables `I` and `LINENO` respectively in Figure 9. The use of these variable names is suggested, but not required, by annotation associated with the plan in Figure 8.) The count of lines is initially set to 67 as a trick to simplify the test



Default Constraints: {*file name*} = `"report.txt"`.
Derived Constraints: {*line limit*} = `66-Lines(`{*print entry*}`)-Lines(`{*print summary*}`)`.
Comment: `"prints a report of` {*data* of *enumerator*}`"`.
Described roles: {*file name*}, {*title*}, {*enumerator*}, {*print headings*},
{*print entry*}, {*print summary*}.
Primary roles: {*enumerator*}, {*print entry*}, {*print summary*}.

Figure 8: The KBEmacs schema for the simple report cliché.

```
(WITH-OPEN-FILE (REPORT {file-name} :DIRECTION :OUTPUT)
  (LET* ((DATE (GET-DATE-TIME))
         (I 0)
         (LINENO 67)
         (TITLE {title})
         (DATA {data of enumerator}))
    (FORMAT REPORT "~2%~A~2%~A" TITLE DATE)
    (LOOP DO
      (IF ({test of enumerator} DATA) (RETURN NIL))
      (WHEN (> LINENO {line-limit})
        (SETQ I (+ I 1))
        (FORMAT REPORT "~|~%Page: ~3D  ~A  ~A~%" I TITLE DATE)
        (SETQ LINENO 3)
        ({print-headings} {REPORT, modified} {LINENO, modified}))
      ({print-entry} {REPORT, modified} {LINENO, modified}
                     ({accessor of enumerator} DATA))
      (SETQ DATA ({step of enumerator} DATA)))
    ({print-summary} {REPORT, modified})))
```

Figure 9: Lisp code corresponding to the plan diagram in Figure 8.

controlling the initiation of new pages.

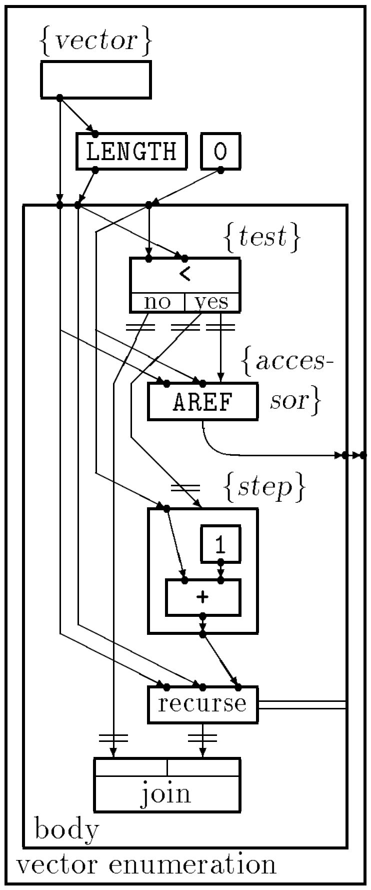
The computations that eventually fill the print headings and print entry roles are expected to correctly update the value recording the number of lines printed. This is signified in Figure 9 by routing the variable `LINENO` through these roles. The notation `{..., modified}` in Figure 9 specifies that the indicated variable is an output as well as an input.

Temporal abstraction. Through the process of *temporal abstraction* [16, 26] plan diagrams represent loops as "compositions" of operations on sequences of values. For example, the main computation in the simple report schema in Figure 8 is represented as the composition of the enumerator and a map. The enumerator creates a sequence of values and the map applies the print entry (and the computation involved with printing page headings) to each element of this sequence.

The fundamental insight behind temporal abstraction is the observation that the only interaction between fragments like the enumerator and the map is that one uses data computed by the other. Temporal abstraction views the sequence of values transmitted between the two fragments as a compound data object (called a *temporal sequence*) and the fragments as functions operating on this sequence. The advantage of temporal abstraction is that it makes the entire computation in Figure 8 as easy to reason about and manipulate as the expression that prints the title and date on the first page of the report.

In Figure 8, the enumerator role is filled with a generic enumeration schema, which contains four roles. The *data* role is the structure to be enumerated. The *test* determines when the enumeration should stop. The *accessor* retrieves individual elements of the structure. The *step* moves from one element to the next.

The plan diagram for the generic enumeration schema represents the computation recursively, rather than using a loop in data and control flow. The data role is only performed once before the enumeration begins. The test, accessor, and step are performed many times. The double line connecting the box labeled `recurse` and the box labeled `body` indicates that the `recurse` box is identical to the `body`. An execution of the `recurse` box is a recursive execution



Comment: "enumerates the elements of {*vector*}".
Described roles: {vector}.
Input roles: {vector}.
Primary roles: {vector}.

Figure 10: The KBEmacs schema for the vector enumeration cliché.

```
(LET* ((I 0)
       (V {vector})
       (SIZE (LENGTH V)))
  (LOOP DO
    (IF {(NOT (< I SIZE)), test} (RETURN NIL))
    {(AREF V I), accessor}
    {(SETQ I (+ I 1)), step}))
```

Figure 11: Lisp code corresponding to the plan diagram Figure 10.

of the `body`. When the test succeeds, the `recurse` box is bypassed and the enumeration is terminated.

The most interesting feature of the plan diagram for the enumeration schema is the output port on the right side. The value at this port is a sequence of the values returned by the accessor, in the order that they are created. The input port on the left side of the map is similar in nature. It represents a sequence of the values required by the third input of the print entry.

```
Command: Define program REPORT-TIMINGS with parameter TIMINGS.
Command: Insert simple-report.
```

```
(DEFUN REPORT-TIMINGS (TIMINGS)
  (WITH-OPEN-FILE (REPORT "report.txt" :DIRECTION :OUTPUT)
    (LET* ((DATE (GET-DATE-TIME))
           (I 0)
           (LINENO 67)
           (TITLE {title})
           (DATA {data of enumerator}))
      (FORMAT REPORT "~2%~A~2%~A" TITLE DATE)
      (LOOP DO
        (IF ({test of enumerator} DATA) (RETURN NIL))
        (WHEN (> LINENO 64)
          (SETQ I (+ I 1))
          (FORMAT REPORT "~|~%Page: ~3D  ~A  ~A~%" I TITLE DATE)
          (SETQ LINENO 3)
          ({print-headings} REPORT LINENO))
        ({print-entry} REPORT LINENO ({accessor of enumerator} DATA))
        (SETQ DATA ({step of enumerator} DATA)))
      ({print-summary} REPORT))))
```

Figure 12: KBEmacs output 1, instantiating a schema.

Because the recursion in the enumeration is a tail recursion, the coder renders it as a loop in Figure 9. The various roles of the enumerator are dispersed in the code, because they have to be mixed with the computation corresponding to the map. Role designators of the form `{test of enumerator}` are used to indicate that the roles of the enumerator are subroles of a larger role called the enumerator.

The subroles of the enumerator can be filled individually, or they can be filled together as a unit by replacing the generic enumeration schema with a more specific enumeration schema such as vector enumeration (see Figures 10 and 11).

Comparing Figure 10 with the enumeration schema in the enumerator role in Figure 8 reveals that the vector enumeration schema is essentially just an instance of an enumeration with some additional setup computation and particular operations filled in for the test, accessor, and step. (The lower bound of every Lisp vector is 0, while the upper bound can be determined by calling the function `LENGTH`. The function `AREF` accesses an element of a vector. In Figure 11, the notation {*code*, *name*} is used to indicate a role in a schema definition that is filled with something.)

3.2 Transcript of KBEmacs in Action

To give a feeling for what cliché-based editing is like in KBEmacs, Figures 12–15 show a transcript of the system in action. The transcript shows the construction and later modification of the program `REPORT-TIMINGS` used as an example in Section 2. By using the schemas discussed in the preceding subsection, KBEmacs is able to respond to commands at a similar level of detail to the directions shown at the beginning of Section 2.

In Figure 12, the programmer starts the construction of `REPORT-TIMINGS` by specifying the name of the program, the parameter list, and the fact that the body of the program is an instance of the simple report schema. The figure is divided into two parts. The upper part

Command: `Fill enumerator with vector-enumeration of TIMINGS.`

```
  (DEFUN REPORT-TIMINGS (TIMINGS)
    (WITH-OPEN-FILE (REPORT "report.txt" :DIRECTION :OUTPUT)
      (LET* ((DATE (GET-DATE-TIME))
             (I 0)
|            (J 0)
             (LINENO 67)
|            (SIZE (LENGTH TIMINGS))
             (TITLE {title}))
        (FORMAT REPORT "~2%~A~2%~A" TITLE DATE)
        (LOOP DO
|         (IF □(NOT (< J SIZE)) (RETURN NIL))
          (WHEN (> LINENO 64)
            (SETQ I (+ I 1))
            (FORMAT REPORT "~|~%Page: ~3D  ~A  ~A~%" I TITLE DATE)
            (SETQ LINENO 3)
            ({print-headings} REPORT LINENO))
|         ({print-entry} REPORT LINENO (AREF TIMINGS J))
|         (SETQ J (+ J 1)))
        ({print-summary} REPORT))))
```

Figure 13: KBEmacs output 2, non-local changes.

shows the commands given to KBEmacs. The lower part shows the program that results.

In the interest of brevity, some of the details of the command interface are omitted. However, the commands have the basic form shown. Each command begins with a verb specifying the action to be performed. Within the command, lower case is used to refer to schemas and various connective words, while upper case is used to refer to literal pieces of code and symbols that appear in the program.

The box that takes up the lower part of the figure shows the state of the editor buffer after the commands are executed. Change bars in the left margin indicate the parts of the buffer that have changed since the previous image. The position of the editing cursor is indicated by the symbol "□". After a set of commands, KBEmacs places the cursor at the location of the first change in the main body of the program.

The program in Figure 12 is an instance of the simple report schema and is very much like Figure 9. However, the constraints and other annotation are not shown in the program produced and `{...}` notation is only used for roles that are empty. The file name and line limit roles are filled as specified by the constraints on them. (The constraint function `LINES` assumes that an empty role will generate a single line of output once it is filled.)

The fact that a standard scheme for dealing with pagination is included in the simple report schema improves the productivity of the programmers using the schema, because in general, they no longer have to worry about pagination. Perhaps more importantly, it improves the reliability of the programs produced using the schema, because these programs always contain a fully general-purpose scheme for dealing with pagination that is at least internally consistent. Further, the fact that the line limit role is derived by a constraint means that the programmer never has to fill it explicitly, and the role is automatically updated whenever the print entry or print summary are changed.

In Figure 13, the programmer tells KBEmacs to fill the enumerator role with a vector

Command: `Fill title with "Report of Reaction Timings (in msec.)".`
Command: `Remove print-headings.`
Command: `Fill print-entry with tabular-print-out of "~9D".`
Command: *direct editing of summary*

```
(DEFUN REPORT-TIMINGS (TIMINGS)
  (WITH-OPEN-FILE (REPORT "report.txt" :DIRECTION :OUTPUT)
    (LET* ((DATE (GET-DATE-TIME))
           (I 0)
           (J 0)
           (LINENO 67)
           (SIZE (LENGTH TIMINGS))
           (TITLE "Report of Reaction Timings (in msec.)"))
      (FORMAT REPORT "~2%~A~2%~A" TITLE DATE)
      (LOOP DO
        (IF (NOT (< J SIZE)) (RETURN NIL))
        (WHEN (> LINENO 63)
          (SETQ I (+ I 1))
          (FORMAT REPORT "~|~%Page: ~3D  ~A  ~A~%" I TITLE DATE)
          (SETQ LINENO 3))
        (WHEN (> (CHARPOS REPORT) 61)
          (TERPRI REPORT)
          (SETQ LINENO (+ LINENO 1)))
        (FORMAT REPORT "~9D" (AREF TIMINGS J))
        (SETQ J (+ J 1)))
      (FORMAT REPORT "~2%Mean: ~8D" (MEAN TIMINGS)))))
```

Figure 14: KBEmacs output 3, text editing and constraint propagation.

enumeration. Note that the changes caused by this command are spread throughout the program. In particular, two new variables are declared and the four subroles of the enumerator role are filled. This illustrates that combining schemas is significantly more complex than just appending chunks of program text or calling subroutines.

Another important feature of Figure 13 is that the variable `I` used by the vector enumeration schema in Figure 11 has been renamed to `J`. This is necessary, because the variable `I` is already being used in the surrounding context (see Figure 12).

In Figure 14, the programmer finishes the program `REPORT-TIMINGS` by filling the title, print entry, and print summary roles and getting rid of the print headings. (To simplify the presentation it is assumed that a function is available for computing the mean of `TIMINGS`.)

The removal of the print headings illustrates the fact that schemas are typically designed with lots of bells and whistles, on the theory that it is easier to get rid of extra things than to think up things that are missing.

The print summary role is filled by using text editing to type in the appropriate code. This illustrates the fact that the programmer is always free to use ordinary editing commands. The goal of KBEmacs is to make a powerful set of additional commands available, without taking away any of the capabilities of standard editors.

A key feature of Figure 14 is that the value of the line limit role is changed from `64` to `63`. The constraint on the line limit role triggered this change, because the print summary specified by the programmer prints two lines of output. This automatic change in the value of the line limit prevents a bug that might well go unnoticed in testing, since the bug has

```
Command: Replace print-entry with simple-print-out of "~%~14D".
Command: Create comment.
```

```
;;; The function REPORT-TIMINGS is a simple-report.
;;;   The file-name is "report.txt".
;;;   The title is "Report of Reaction Timings (in msec.)".
;;;   The enumerator is a vector-enumeration.
;;;      It enumerates the elements of TIMINGS.
;;;   There are no print-headings.
;;;   The print-entry is a simple-print-out.
;;;      It prints out (AREF TIMINGS I).
;;;   The print-summary is an idiosyncratic computation.

(DEFUN REPORT-TIMINGS (TIMINGS)
  (WITH-OPEN-FILE (REPORT "report.txt" :DIRECTION :OUTPUT)
    (LET* ((DATE (GET-DATE-TIME))
           (I 0)
           (J 0)
           (LINENO 67)
           (SIZE (LENGTH TIMINGS))
           (TITLE "Report of Reaction Timings (in msec.)"))
      (FORMAT REPORT "~2%~A~2%~A" TITLE DATE)
      (LOOP DO
        (IF (NOT (< J SIZE)) (RETURN NIL))
        (WHEN (> LINENO 63)
          (SETQ I (+ I 1))
          (FORMAT REPORT "~|~%Page: ~3D  ~A  ~A~%" I TITLE DATE)
          (SETQ LINENO 3))
        (SETQ LINENO (+ LINENO 1))
        (FORMAT REPORT "~%~14D" (AREF TIMINGS J))
        (SETQ J (+ J 1)))
      (FORMAT REPORT "~2%Mean: ~8D" (MEAN TIMINGS)))))
```

Figure 15: KBEmacs output 4, modification and documentation.

an effect only when the print summary is to be printed on the last line of a page.

Program modification and documentation. The transcript in Figures 12–14 shows KBEmacs supporting program construction. Although this is the main focus of KBEmacs, the system provides support for other parts of the programming process as well.

The program in Figure 14 prints out the numbers in the vector TIMINGS several to a line. With the first command in Figure 15, the programmer changes the print entry role so that the numbers are printed in one long column. The replace command used removes the tabular print out filling the print entry role in Figure 14 and replaces it with a simpler schema that prints items one to a line. In general, cliché-based editing is just as useful for program modification as for program creation.

With the second command in Figure 15, the programmer tells KBEmacs to generate a comment describing the program REPORT-TIMINGS. One could easily argue with the details of the comment created. However, the comment has an important property that most comments lack—it is guaranteed to be correct, because it is generated based on the schemas that were actually used to construct the program, and it can be automatically changed if the program changes.

3.3 Manipulating Plan Diagrams

The lion's share of the power of KBEmacs comes from the problem solving technique of *representation shift*—shifting from an obvious representation in which a problem is easy to state, to a less obvious representation in which the problem is easy to solve.

The basic problem of constructing and manipulating programs in terms of clichés is easy to state in terms of program text. In line with this, the interface to KBEmacs is carefully designed to project this image to the programmer. For instance, a textual form is provided for defining schemas representing clichés and the programmer sees a continuous display of the program text resulting from the commands given.

However, actually performing the required manipulations directly on program text would be very difficult. This difficulty is overcome by representing both the schemas and the program being worked on as plan diagrams (implemented as graphs in a simple database). Once this is done, the manipulations required to combine and modify schemas are simple.

In particular, when considering how KBEmacs supports the commands in Figures 12–15, it is important to realize that KBEmacs' internal representation of the program `REPORT-TIMINGS` is very different from the program text shown in Figures 12–15. As an example, Figure 16 shows the internal representation of the version of `SIMPLE-REPORT` shown in Figure 14.

(For the most part, Figure 16 is the same as Figure 8 with Figure 6 inserted in the print entry role and Figure 10 inserted in the enumerator role. However, note that the vector role of the vector enumeration and the item role of the tabular print out have been turned into inputs. In addition, the print headings role of simple report has been removed, while the other roles have been filled. To save space, Figure 16 has been abbreviated by omitting part of the tabular print out and part of the computation that prints headings at the top of each page of the report. The missing sections of the plan diagram in Figure 16 are identical to the corresponding sections of Figures 6 and 8.)

Most of the complexity of KBEmacs is contained in the analyzer and coder modules that convert back and forth between program text and temporally abstracted plan diagrams. The schema editor itself is quite simple due to the following three key advantages that plan diagrams have in comparison with program text.

Every role is a single box. In program text, it is often the case that widely separated parts of the program are part of the same logical role. However, in a plan each role is consolidated into a single box. Because of temporal abstraction, this is even true for loop fragments such as the enumerator in Figure 16. Because every role is a box, adding, removing, and filling roles merely requires adding boxes, deleting boxes, and plugging plan diagrams into boxes.

For instance, the remove command in Figure 14 is supported by deleting a box and the arrows touching it. The insert command in Figure 13 is supported by removing the old contents of the enumerator role (see Figure 8) and replacing it with a copy of the vector enumeration schema (see Figure 10). When an insert command specifies a literal piece of program text, KBEmacs uses the analyzer module to convert the piece of text into a plan and then inserts the plan into the indicated role.

The only operation performed by the schema editor that is at all complex is hooking up data flow. For instance, when the print entry is filled with a tabular print out in Figure 14, a copy of the plan diagram in Figure 6 is placed inside the print entry box in Figure 8

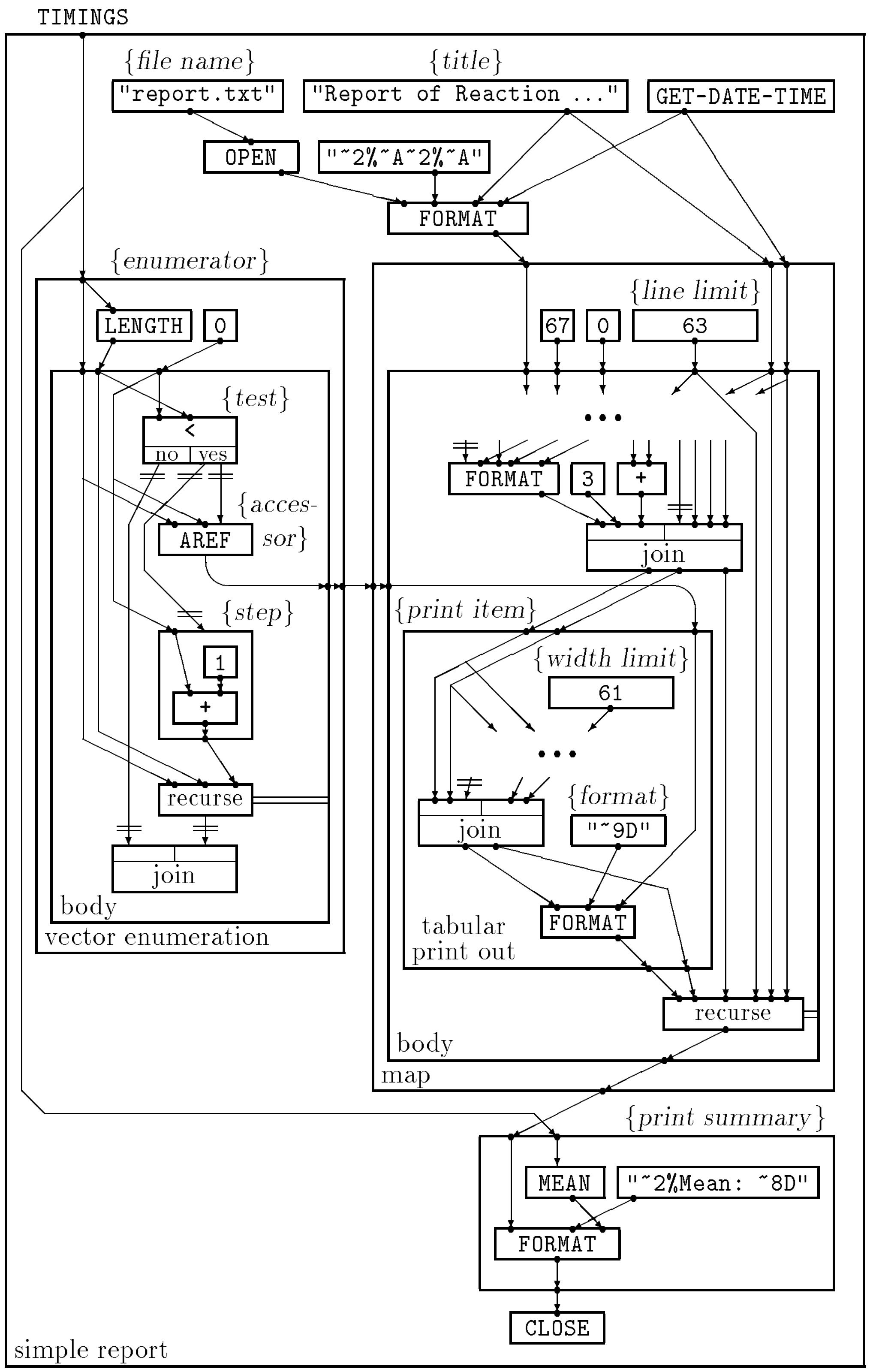


Figure 16: Plan diagram corresponding to Figure 14.

yielding the result in Figure 16. This is trivial to do, except that the outermost inputs of the diagram in Figure 6 have to be matched up correctly with the inputs of the print entry box in Figure 8.

The inputs of the tabular print out schema (i.e., the free variables `REPORT` and `LINENO`) are specified in the schema definition in Figure 6. The inputs of the print entry role are specified in the plan for the simple report schema in Figure 8. Comparing these two specifications, KBEmacs matches the two `REPORT` inputs and the two `LINENO` inputs together since they refer to variables of the same name.

The schema editor then notes that the print entry role has one more input than the tabular print out schema, but that the item role of the schema is tagged as being an input role. To establish a connection between the elements created by the enumerator and the schema, KBEmacs converts the item role into an input and matches it with the third input of the role. The key thing to note here is that the tabular print out schema is correctly connected into the data flow in the program without the programmer having to say anything.

Data flow and control flow represented explicitly and locally. All information in a plan diagram about inputs, outputs, operations, and the data flow between them is explicitly represented using direct connections between inputs and outputs. One virtue of this is that the information can be rapidly retrieved. For example, if the schema editor wishes to know where the data flow to the print entry comes from, it can find this out by tracing back the relevant arrows without having to perform any global analysis.

Another virtue of this representation is that each arc represents a local fact that is independent of anything else in a plan diagram. As a result, if the schema editor wants to add a new data flow, it just adds a new arrow without having to worry about variable name clashes or anything else. For instance, when the enumerator role is filled with a vector enumeration in Figure 13, the schema editor does not have to do anything to prevent variable clashes, because this problem cannot occur in a plan diagram. Of course, when the coder module converts a plan diagram into program text, it has to make sure that the data flow arcs are accurately rendered using variables and other data-flow mechanisms.

To create aesthetic code, the coder makes use of suggestions about variable names that are stored as annotation on plans. However, it makes up new variables names if the suggestions lead to conflicts. For instance, the plan diagram in Figure 16 has annotation suggesting that both the variable holding the page number and the variable holding the vector index should be called `I`. However, since this is not possible, the coder changes one of the names.

Complete information about the schemas is retained. A plan diagram like the one in Figure 16 contains complete information about the schemas used to create it. In particular, all of the roles are shown (except the print headings, which was removed) and the schemas used to fill the roles are recorded. (The constraints on the roles are recorded along with of the schemas in the schema library.)

The replace command in Figure 15 is supported by discarding the contents of the print entry box in Figure 16, inserting a different plan diagram, and then rechecking the relevant constraints.

The comment shown in Figure 15 is produced by simply reading out the top two levels of the hierarchical structure of the underlying plan diagram in terms of schemas and using some of the auxiliary information associated with the schemas. In particular, the comment gives a brief description of each of the *described roles* in the top-level schema (see Figure 8). When one of these roles is filled with a schema, the *comment* annotation associated with that schema is used to create a one sentence description.

Constraints are supported procedurally. Constraints are represented as procedures that take the appropriate actions. For instance, the `DERIVED` constraint on the simple report schema (see Figure 8) is represented as a procedure that checks the value of the print entry and print summary roles and sets the value of the line limit role. This procedure is rerun whenever the value of either the print entry role or the print summary role changes.

3.4 Evaluation

KBEmacs supports cliché-based editing—namely the creation and modification of programs through the combination of clichés selected by the user from an extensible library—more or less completely. Nevertheless, its capabilities are limited in several crucial ways.

The greatest single weakness of KBEmacs is that it does not have any general-purpose reasoning capabilities. This prevents KBEmacs from handling constraints in a general way—it can only handle constraints that can be expressed as procedures that derive roles from other roles. This also prevents KBEmacs from assessing the reasonableness of the actions taken by the programmer—it merely does whatever the programmer says. Finally, the lack of general-purpose reasoning capabilities prevents KBEmacs from making any interesting decisions by itself—the programmer must explicitly pick every schema to be used.

Another weakness of KBEmacs is that it is somewhat limited in the clichés that it can represent. For the most part, KBEmacs only supports clichéd algorithms like those presented above as opposed to clichéd data structures or clichéd system organizations. This weakness stems from the fact that plan diagrams are the only method KBEmacs has for representing clichés. A more general logical representation is required for representing non-algorithmic clichés.

Work is currently underway on the construction of a *Design Apprentice* that will address the problems above and go beyond KBEmacs by automating low-level program design [14, 34]. With the Design Apprentice, a programmer/designer will outline the basic high-level design, while the system selects the detailed algorithms to use and puts them together into a program. Unlike KBEmacs, the Design Apprentice will be able to reason extensively about the design of the program and critique the statements made by the programmer/designer.

A subtle weakness of KBEmacs concerns the analyzer module. When a plan diagram for a program is constructed in KBEmacs by combining schemas, it contains complete information about the schemas used. Ideally, the analyzer should be able to take a program represented as text and create a plan that contains complete information about how the program could have been constructed from schemas. However, this reverse-engineering task is beyond the capabilities of the current analyzer. In lieu of a fully detailed plan, the analyzer creates a minimal plan that accurately represents the computation, but does not contain any information about schemas.

The weakness of the current analyzer means that while the programmer can apply arbitrary textual editing at any time, this is likely to cause KBEmacs to lose track of the schemas that were previously used to construct the program in question. (KBEmacs is able to carry on with complete schema information after the textual editing in Figure 14 due to a special feature—whenever a textual change is confined solely to the contents of a single role, KBEmacs recognizes that the change is equivalent to filling the role with the indicated text.) Work is currently underway on methods for automatically supporting complete reverse engineering in the general case, see [17, 35].

The weaknesses above notwithstanding, KBEmacs demonstrates valuable capabilities. However, the evolution of the system has reached the point where the creation of a full-scale prototype requires complete reimplementation. Unfortunately, KBEmacs consists of some 40,000 lines of Lisp code and is comparable in complexity to an optimizing compiler. Before attempting such a large reimplementation effort, it is reasonable to consider whether some of the key ideas behind KBEmacs can be extracted and hosted in simpler systems.

One of the most important ideas behind KBEmacs is temporal abstraction—representing loops as compositions of functions on sequences of values. By making the manipulation of loops as simple as the manipulation of expressions, it simplifies ever aspect of KBEmacs' operation.

Conveniently, the concept of temporal abstraction is quite separate from the idea of cliché-based editing and can be supported separately. In particular, a Common Lisp macro package called *Series* [32, 33] has been implemented that makes it possible to represent loops in a functional style, without any loss of efficiency. In addition, a prototype language extension has been implemented for Pascal [33]. In both cases, the language extensions have proven to be extremely valuable, primarily because they allow a wide range of loop clichés such as vector enumeration to be represented as functions.

The other key idea behind KBEmacs is the construction of programs by combining schemas representing clichés. The following sections describe two systems that support the manipulation of schemas and yet are much simpler than KBEmacs.

4 Tempest

From the beginning of work on KBEmacs, it was assumed that a representation shift to something like plan diagrams was essential for supporting cliché-based editing. However, detailed study of KBEmacs in action revealed that KBEmacs did not make use of the power of plan diagrams quite as much as had been presupposed. In particular, it became apparent that it should be possible to reduce the magnitude of the representation shift (and therefore the system's complexity) significantly, while still retaining most of the system's power.

As a radical first experiment in simplifying KBEmacs, a system called Tempest [23] was constructed that eliminates the representation shift altogether. Tempest succeeds in removing almost all of KBEmacs' complexity, but at the price of removing most of KBEmacs' power.

Tempest is a small program implemented in C [9] on an IBM PC rather than a large program on a Symbolics Lisp Machine. It executes commands in fractions of seconds rather than minutes. However, Tempest is not powerful enough to provide useful support for program editing. Nevertheless, it is valuable to discuss Tempest for two reasons.

First, it is illuminating to see exactly why Tempest fails as a program editor. The Ace editor described in Section 5 is a good compromise system, because it solves these specific problems without taking on the full complexity of KBEmacs.

Second, although Tempest is rather weak, its power-to-complexity ratio is much higher than KBEmacs'. Tempest is very useful in simple editing situations such as document preparation and its benefits can be obtained very cheaply.

The architecture of Tempest is shown in Figure 17. Unlike KBEmacs (see Figure 5) Tempest maintains only one representation and therefore does not need anything equivalent to KBEmacs' coder and analyzer modules. The schema library contains a collection of clichés represented as textual schemas. The schema editor performs the same kind of simple operations as KBEmacs' schema editor—i.e., it can insert and replace instances of schemas in the text being edited and supports simple constraints.

Tempest is written as an extension to the commercially available Emacs-style Mince editor [38]. The standard facilities of Mince are used to support both text editing and syntax editing. The commands supported by the schema editor are provided as extensions to the Mince command set and the modifications performed by the schema editor are applied directly to the Mince editing buffer.

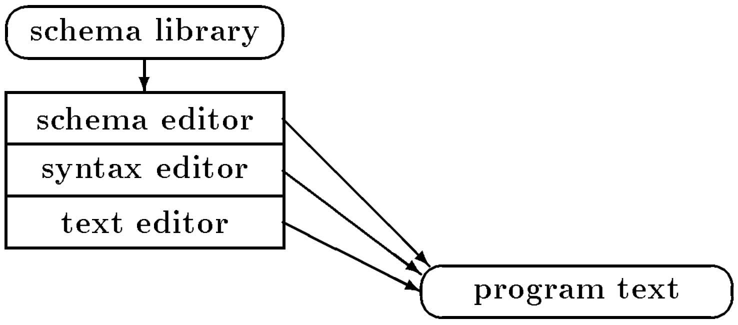


Figure 17: Architecture of Tempest.

```
(PROGN (WHEN (> (CHARPOS REPORT) {width-limit})
         (TERPRI REPORT)
         (SETQ LINENO (+ LINENO 1)))
       (FORMAT REPORT {format} {item}))
```

Figure 18: The Tempest schema for the tabular print out cliché.

```
(WITH-OPEN-FILE (REPORT {"report.txt", file-name} :DIRECTION :OUTPUT)
  (LET* ((DATE (GET-DATE-TIME))
         (I 0)
         (LINENO 67)
         (TITLE {title})
         {prolog of enumerator})
    (FORMAT REPORT "~2%~A~2%~A" TITLE DATE)
    (LOOP DO
      (IF {test of enumerator} (RETURN NIL))
      (WHEN (> LINENO {line-limit})
        (SETQ I (+ I 1))
        (FORMAT REPORT "~|~%Page: ~3D  ~A  ~A~%" I TITLE DATE)
        (SETQ LINENO 3)
        {print-headings})
      {{{accessor of enumerator}, item}, print-entry}
      {step of enumerator})
    {print-summary}))
```

Figure 19: The Tempest schema for the simple report cliché.

```
{(I 0)
 (SIZE (LENGTH {vector})), prolog}
{(NOT (< I SIZE)), test}
{(AREF {vector} I), accessor}
{(SETQ I (+ I 1)), step}
```

Figure 20: The Tempest schema for the vector enumeration cliché.

4.1 Representing Clichés as Textual Schemas

In Tempest, clichés are represented as textual schemas like the ones shown in Figures 18–20. The text shown represents the body of the cliché. Default constraints are specified by providing a non-empty initial *filler* for the role. Simple equality constraints between roles are represented as annotations on the schema (see Section 4.4). Tempest does not support defaults which are computed based on the values of other roles or derived constraints as complicated as the one on the width limit role in Figure 2. In addition, Tempest does not support the various kinds of special annotation supported by KBEmacs.

A new schema can be created by typing text like that shown in Figures 18–20. This is then stored in a file, whose name serves as the name of the cliché. Constraints are created using a special Tempest command and are stored in an auxiliary file.

Comparing Figures 18 and 19 with Figures 7 and 9 (respectively) reveals that the Tempest schemas are very much like the printed representations of the KBEmacs schemas.

Somewhat greater differences are revealed by comparing Figure 20 with Figure 10. The

Command: *direct editing of function heading*
Command: `Insert simple-report.`

```
(DEFUN SIMPLE-REPORT (TIMINGS)
(WITH-OPEN-FILE (REPORT "report.txt" :DIRECTION :OUTPUT)
  (LET* ((DATE (GET-DATE-TIME))
         (I 0)
         (LINENO 67)
         (TITLE {title})
         {prolog of enumerator})
    (FORMAT REPORT "~2%~A~2%~A" TITLE DATE)
    (LOOP DO
      (IF {test of enumerator} (RETURN NIL))
      (WHEN (> LINENO {line-limit})
        (SETQ I (+ I 1))
        (FORMAT REPORT "~|~%Page: ~3D  ~A  ~A~%" I TITLE DATE)
        (SETQ LINENO 3)
        {print-headings})
      {{{accessor of enumerator}, item}, print-entry}
      {step of enumerator})
    {print-summary})))
```

Figure 21: Tempest output 1, instantiating a schema.

roles in Figure 20 have been altered so that they contain all of the computation uniquely associated with the schema (some of the key computation is outside of the roles in Figure 10). In addition, the standard constructs that connect the computation together (e.g., the loop itself) have been omitted. The schema in Figure 20 only makes sense when it is inserted in a role, such as the enumerator in Figure 19, that places the four chunks of text in an appropriate context. The fact that the vector role in Figure 20 appears twice means that whatever is inserted in the vector role will be inserted in both places.

4.2 Transcript of Tempest Being Used To Edit a Program

Figures 21–23 show a transcript of Tempest being used to construct the program `REPORT-TIMINGS`. Like the transcript figures in Section 3, these figures are divided into two parts: a depiction of the editor buffer below a summary of commands. As in the figures in Section 3, the command interface has been somewhat simplified.

The command language supported by Tempest is simpler than the one supported by KBEmacs. There is no special command for defining a program. As a result, ordinary text editing is used to type in the function header at the top of Figure 21. However, the simple report schema is instantiated with an insert command just as in KBEmacs. In Tempest, this is done by merely copying the text in Figure 19 into the editor buffer. As in KBEmacs, filled roles are displayed by showing only their contents—i.e., `{"report.txt", file-name}` is displayed as `"report.txt"`. (The user can override this and force the complete display of information about all roles. This is useful when defining a new schema.)

In Figure 22, the programmer fills the enumerator role. Tempest does this by simply inserting the four parts of Figure 20 into the appropriate four subroles in Figure 21. A separate command has to be used to fill the vector role of the vector enumeration, because Tempest does not support commands containing phrases of the form "`vector-enumeration`

Command: `Fill enumerator with vector-enumeration.`
Command: `Fill vector with TIMINGS.`

```
(DEFUN SIMPLE-REPORT (TIMINGS)
(WITH-OPEN-FILE (REPORT "report.txt" :DIRECTION :OUTPUT)
  (LET* ((DATE (GET-DATE-TIME))
         (I 0)
         (LINENO 67)
         (TITLE {title})
         (I 0)
 (SIZE (LENGTH TIMINGS)))
    (FORMAT REPORT "~2%~A~2%~A" TITLE DATE)
    (LOOP DO
      (IF (NOT (< I SIZE)) (RETURN NIL))
      (WHEN (> LINENO {line-limit})
        (SETQ I (+ I 1))
        (FORMAT REPORT "~|~%Page: ~3D  ~A  ~A~%" I TITLE DATE)
        (SETQ LINENO 3)
        {print-headings})
      {{(AREF TIMINGS I), item}, print-entry}
      (SETQ I (+ I 1)))
    {print-summary})))
```

Figure 22: Tempest output 2, errors creep in.

`of ...`".

In Figure 23, the programmer fills the title, print entry, and print summary while getting rid of the print headings. As discussed in Section 4.5, the operation of Tempest is extremely simple. However, this does not count for very much in situations such as Figure 23 where incorrect results are obtained.

4.3 Evaluation

In Figures 21–23 the programmer uses commands very much like the ones used in Figures 12–14 and combines the same three clichés together. However, comparison of Figure 23 with Figure 14 reveals that a rather different result is obtained.

The biggest problem concerns the variable `I`. When inserting the vector enumeration schema in Figure 22, Tempest blindly combines the text in Figures 20 and 21. This leads to conflicting bindings and uses of the variable `I`. To fix this problem, Tempest would have to analyze the data flow in Figures 20 and 21 well enough to detect the conflict and rename one of the variables.

A second problem concerns the `PROGN` on the sixth to last line of Figure 23. This is needed in Figure 18 to group the schema into a single expression, which can be inserted in roles that can only hold a single expression. However, it is unnecessary and therefore misleading in Figure 23. To fix this problem, Tempest would have to parse the text in Figure 23 and apply Lisp-specific simplification rules.

A third problem with Figure 23 is that the indentation is wrong in a number of places, which makes the program significantly harder to read than Figure 14. To fix this problem, Tempest would have to parse the text in figure 23 and then pretty print it back into the editor buffer.

Command: `Fill title with "Report of Reaction Timings (in msec.)".`
Command: `Remove print-headings.`
Command: `Fill print-entry with tabular-print-out.`
Command: `Fill format with "~9D".`
Command: *direct editing of summary*

```
(DEFUN SIMPLE-REPORT (TIMINGS)
(WITH-OPEN-FILE (REPORT "report.txt" :DIRECTION :OUTPUT)
  (LET* ((DATE (GET-DATE-TIME))
         (I 0)
         (LINENO 67)
         (TITLE "Report of Reaction Timings (in msec.)")
         (I 0)
 (SIZE (LENGTH TIMINGS)))
    (FORMAT REPORT "~2%~A~2%~A" TITLE DATE)
    (LOOP DO
      (IF (NOT (< I SIZE)) (RETURN NIL))
      (WHEN (> LINENO {line-limit})
        (SETQ I (+ I 1))
        (FORMAT REPORT "~|~%Page: ~3D  ~A  ~A~%" I TITLE DATE)
        (SETQ LINENO 3)
        )
(PROGN (WHEN (> (CHARPOS REPORT) {width-limit})
         (TERPRI REPORT)
         (SETQ LINENO (+ LINENO 1)))
       (FORMAT REPORT "~9D" (AREF TIMINGS I)))
      (SETQ I (+ I 1)))
    (FORMAT REPORT "~2%Mean: ~8D" (MEAN TIMINGS)))))
```

Figure 23: Tempest output 3, things deteriorate further.

A final problem is that Tempest does not support the constraints on the line limit and width limit roles. As a result, these roles are still unfilled in Figure 23. Tempest could support the required constraints using the same kind of constraint procedures supported by KBEmacs.

There is a common thread that goes through most of the problems above. Basically all Tempest does is paste textual schemas together without doing anything to alter them so that they fit together nicely. This is satisfactory only in situations where it is appropriate to render schemas character-for-character exactly the same way independent of the context of their use.

Whenever a schema has to be modified based on global considerations, Tempest's approach breaks down. Because this is often necessary when editing programs, Tempest is not adequate as a program editor. However, it is worthy of note that there are two program editing situations where the no-representation-shift approach of Tempest may prove adequate.

One situation is suggested by Figure 21. Simple textual insertion works pretty well if the only thing you want to do is insert a single schema into an empty (or almost empty) buffer, because there is no (or very little) context to worry about. However, you have to restrict yourself to operate in a predominantly non-clichéd manner after the initial insertion of a schema. This approach is taken by the Paris system [8], where the nesting of schemas is mentioned as a theoretical possibility, but not pursued.

```
@make(letter)@style(LeftMargin=1.5in, RightMargin=1.5in)
{@Value(Date), date}
@begin(address)
{recipient}
@end(address)
@begin(body)
@greeting(Dear {greeting-title} {greeting-name}:)
{message}
@end(body)
{Sincerely Yours, salutation},
@blankspace(2 lines)
{sender}
```

Derived Constraints: {*greeting title*} = {*title* of *recipient*}.
{*greeting name*} = {*last name* of *recipient*}.

Figure 24: The Tempest schema for the simple letter cliché.

```
{Dr., title} {John, first-name} {J., middle-name} {Smith, last-name}
{77 Elm St., street}
{Evansville, city}, {IN, state} {47712, zip}
```

Figure 25: The Tempest schema for the Doctor Smith cliché.

Another situation where you might be able to dispense with representation shift is one where the user-visible representation of programs is similar to plan diagrams. For instance, a graphical programming tool such as Prograph [39] could be extended to support cliché-based editing with no more difficulty than Mince was extended to become Tempest. However, the result would be much more powerful than Tempest, because the data-flow-graph-based representation used by Prograph reduces the need to consider the context when combining fragments.

4.4 A Situation Where Tempest Works Well

The rather negative assessment of Tempest in the previous subsection might give the impression that Tempest was not useful for anything. However, while Tempest is not particularly useful for the cliché-based editing of programs, it is quite useful in a number of other situations. For example, Figures 24–28 show Tempest being used for document preparation.

Figure 24 shows a Tempest schema for constructing a simple letter using the Scribe [40] document preparation system. The schema includes two equality constraints, which are shown at the bottom.

As with all non-interactive text formatters, Scribe processes files containing a mixture of commands (beginning with `@`) and text. The `@make` command specifies the *type* of document. Each type of document has various associated style parameters that can be changed and *environments* that can be filled with text. (In the case of the letter document shown in Figure 24 the environments are the *address*, *greeting*, *body*, etc.). To specify that the text *text* is to appear in the environment *env* you can either type `@`*env*`(`*text*`)` or `@begin(`*env*`)`*text*`@end(`*env*`)`. The user must specify the environments in the proper order. Scribe takes care of the rest—i.e., justification, pagination, etc.

Command: `Insert simple-letter.`

```
@make(letter)@style(LeftMargin=1.5in, RightMargin=1.5in)
@Value(Date)
@begin(address)
{recipient}
@end(address)
@begin(body)
@greeting(Dear {greeting-title} {greeting-name}:)
{message}
@end(body)
Sincerely Yours,
@blankspace(2 lines)
{sender}
```

Figure 26: Tempest output 4, instantiating a schema.

Command: `Remove date.`
Command: `Fill recipient with doctor-smith.`
Command: *direct editing of message and salutation*

```
@make(letter)@style(LeftMargin=1.5in, RightMargin=1.5in)
@begin(address)
Dr. John J. Smith
77 Elm St.
Evansville, IN 47712
@end(address)
@begin(body)
@greeting(Dear Dr. Smith:)
My wife and I very much enjoyed seeing you and your wife at the annual
conference.  We would particularly like to thank you for your generous
hospitality.  We look forward to your visit this summer.
@end(body)
See you soon,
@blankspace(2 lines)
{sender}
```

Figure 27: Tempest output 5, filling roles and constraint propagation

Command: `Replace top-level with informal-letter.`

```
@make(letter)@style(LeftMargin=1.5in, RightMargin=1.5in)
@begin(address)
Dr. John J. Smith
77 Elm St.
Evansville, IN 47712
@end(address)
@begin(body)
@greeting(John,)
My wife and I very much enjoyed seeing you and your wife at the annual
conference.  We would particularly like to thank you for your generous
hospitality.  We look forward to your visit this summer.
@end(body)
See you soon,
```

Figure 28: Tempest output 6, modification.

The schema in Figure 24 specifies the standard text that must be included when writing a simple letter using Scribe. The seven roles specify the places where variable text has to be entered. The *date* has as its default value a Scribe command that generates the current date. The *recipient* is the address of the recipient of the letter. The *greeting-title* and *greeting-name* are the title and name of the recipient. The *message* is the main body of the letter. The *salutation* has a default value of "`Sincerely Yours`". The *sender* is the name of the person writing the letter. It is printed below the salutation leaving space for the sender's signature.

The constraints specify that the greeting-title and greeting-name should be the title and last-name (respectively) of the recipient. These constraints are designed to work in conjunction with clichéd addresses such as the schema *Doctor Smith* shown in Figure 25, which can be used to fill the recipient role. A non-clichéd address is typed using the general schema *address*, which has the same roles as Doctor Smith, but no default values.

Figures 26–28 show a transcript of Tempest being used to compose a letter. (A much larger transcript appears in [23].) In Figure 26, the user begins by inserting an instance of the simple letter schema.

In Figure 27, the user removes the date role, fills the recipient role with an instance of the schema Doctor Smith, and uses ordinary text editing to fill the body of the letter and alter the salutation. The two greeting roles are automatically filled by the constraints on the simple letter schema.

When typing the new, less-formal salutation, the user realizes that the letter as a whole is too formal. In Figure 28, the user rectifies this by switching to an informal letter schema for the letter as a whole. This schema is much the same as simple letter except that it does not have a typed sender line and the constraint on it specifies a less formal greeting.

Tempest's replace command is interesting in that it is more powerful than KBEmacs' replace command. In particular, when replacing one schema instance with another, KBEmacs' replace command simply throws out the old instance and replaces it with the new one. Tempest's replace command goes beyond this by copying over the contents of the roles of the old schema. Whenever an empty role in the new schema has the same name as a filled role in the old schema, the new role is filled with the old contents. This is a capability that could and should be added to KBEmacs. In Figure 28, the recipient, text, and salutation are copied over from Figure 27. The new greeting line is constructed by a constraint.

Tempest works much better in Figures 26–28 than in Figures 21–23, because Scribe clichés are much more local in nature than programming clichés. This is partly due to the fact that cutting and pasting a document together is fundamentally simpler than cutting and pasting a program together. However, it is predominantly due to the intervention of Scribe itself. Literal cutting and pasting of textual schemas inevitably leads to an ugly Scribe input file. However, Scribe is specifically designed to turn this ugly file into an attractive document. It takes care of modifying the textual schemas (e.g., by indentation) to adapt them to their contexts. Essentially, Scribe provides the support for adaption that Tempest does not provide.

4.5 Manipulating Textual Schemas

Tempest's schema editor is extremely simple. For the most part, the only thing it does is insert, delete, and move named chunks of text. As in KBEmacs, constraints are supported

```
@make(letter)@style(LeftMargin=1.5in, RightMargin=1.5in)
@begin(address)
{{Dr., title} {John, first-name} {J., middle-name} {Smith, last-name}
{77 Elm St., street}
{Evansville, city}, {IN, state} {47712, zip}, recipient}
@end(address)
@begin(body)
@greeting(Dear {Dr., greeting-title} {Smith, greeting-name}:)
{My wife and I very much enjoyed seeing you and your wife at the annual
conference.  We would particularly like to thank you for your generous
hospitality.  We look forward to your visit this summer., message}
@end(body)
{See you soon, salutation},
@blankspace(2 lines)
{sender}
```

Schemas used: top-level is simple-letter.
{*recipient*} is doctor-smith.
Derived Constraints: {*greeting title*} = {*title* of *recipient*}.
{*greeting name*} = {*last name* of *recipient*}.

Figure 29: Internal representation corresponding to Figure 27.

by procedures that take the appropriate action.

There are only two minor points of complexity in all of Tempest. First, a moderate amount of effort has to be expended on maintaining background information about the positions of filled roles, the schemas used, and constraints. For instance, each time a character is added or deleted, the role positions have to be updated. Second, Mince is altered by adding guards on its various commands to ensure that role delimiters are only created and deleted in matching pairs. This is important to ensure that the nested structure of roles always makes sense.

Tempest's internal representation for Figure 27 is illustrated by Figure 29. It consists of the simple letter schema (Figure 24) with the Doctor Smith schema (Figure 25) inserted in the recipient role and some other roles filled with literal text. As indicated by the `{...}` notation in the upper part of Figure 29, Tempest maintains complete information about the position of roles. As indicated by the lines at the bottom, Tempest also maintains complete information about the schemas used and the constraints on these schemas.

(When the contents of the editor buffer are written to a file, the information about roles, schemas, and constraints is written into a separate auxiliary file so that the main file can be passed directly to other systems without having to be converted in any way.)

4.6 Reevaluation

In summary, Tempest is only useful in some situations, and in general, these situations do not include program editing. However, Tempest has the advantage that it is simple, flexible, and blindingly fast. It would be a valuable extension to any text editor. An example of an extension of this kind is the GNU Emacs [21] template mode implemented by Ardis [1].

5 Ace

The experiment with Tempest revealed that some amount of representation shift is essential for supporting cliché-based program editing. However, the feeling remained that a full shift to plan diagrams was not required. In an attempt to strike a balance between KBEmacs and Tempest, a system called Ace [24] was constructed that uses specially designed parse trees as its internal representation. Ace has proved to be a very beneficial balance, providing much of the power of KBEmacs at only a fraction of the cost.

The architecture of Ace is shown in Figure 30. It has the same basic form as the architecture of KBEmacs (see Figure 5). However, the internal representation is parse trees rather than plan diagrams. The value of using parse trees is that it is much easier to shift between program text and parse trees than between program text and plan diagrams. Rather than a complex analyzer module, one merely needs a parser. Similarly, rather than a complex coder module, one merely needs an unparser. Ace's schema library contains a collection of clichés represented as parse-tree schemas. The schema editor performs the same kind of simple operations as the schema editors in KBEmacs and Tempest—it can insert and replace instances of schemas in the parse tree being edited and supports simple constraints.

Like KBEmacs, Ace is implemented on the Symbolics Lisp Machine. Also like KBEmacs, Ace is a demonstration system rather than a full prototype. In the interest of saving time, the work on Ace has focussed on the areas where the most innovation is required—devising an appropriate parse-tree representation and implementing the schema editor. In particular, the implementation of the three dashed boxes in Figure 30 was largely omitted.

Support for text editing and syntax editing was omitted on the grounds that it could be straightforwardly leveraged off of an existing editor as in KBEmacs and Tempest. Similarly, implementation of the parser was omitted on the grounds that parsing is a well understood problem. However, it is worthy of note that incremental parsing [5] is required to provide efficient combined support for text editing and syntax editing.

While unparsing is just as well understood as parsing, an unparser had to be implemented to enable the ready demonstration of Ace's capabilities. By using the PP Lisp pretty printer [29] (which is an intermediate version of the pretty printers described in [28] and [31]) it was possible to implement the unparser in only a few pages of code.

As with KBEmacs, the concepts underlying Ace are programming-language independent. Nevertheless, while it would have been very convenient to use Lisp as the target language of

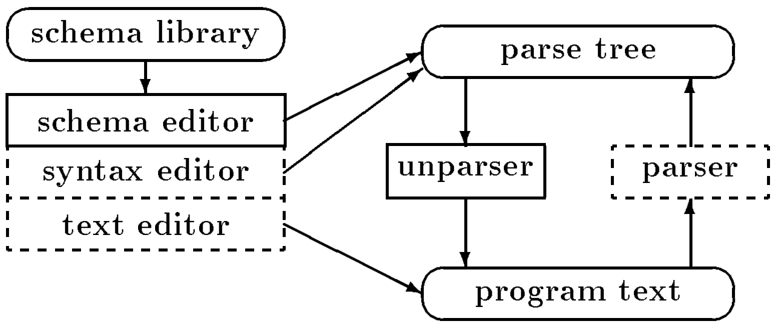
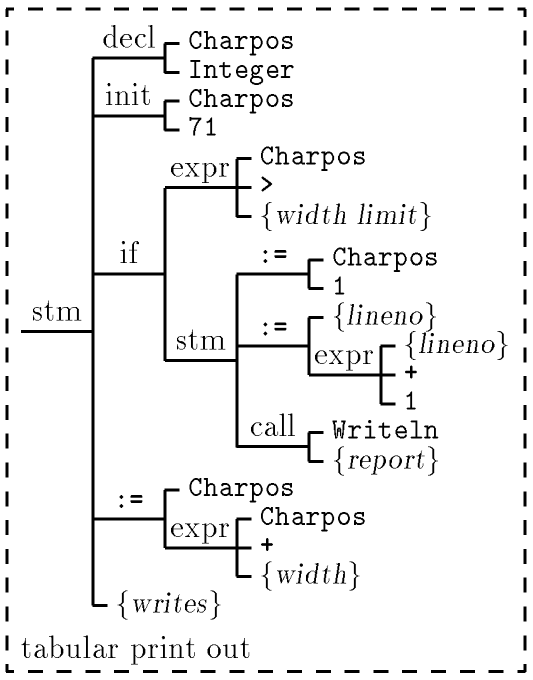


Figure 30: Architecture of Ace.



Default Constraints: {*format*} = `'~15A'`.
Derived Constraints: {*width limit*} = `70-`{*width*}.
{*width*} = `Width(`{*format*}`)`.
{*writes*} = `GenerateWrites(`{*report*}`,` {*format*}`,` {*item*}`)`.
Input roles: {*report*}, {*lineno*}, {*item*}.
Primary roles: {*format*}, {*item*}.
Schema type: statements.
Role types: {*report*}, {*lineno*}: identifier.
{*item*}: expression.
{*format*}: string.
{*writes*}: statements.
{*width*}, {*width limit*}: number.

Figure 31: The Ace schema for the tabular print out cliché.

Ace, an Algol-like language (Pascal [7]) was used to emphasize the practical applicability of the system.

5.1 Representing Clichés as Parse-Tree Schemas

Figure 31 shows the schema used by Ace to represent the tabular print out cliché. The top of the figure depicts a parse tree with the root at the left. Non-terminals are indicated by the abbreviations *stm* (statement list), *decl* (declaration), *init* (initialization), *if* (if statement), `:=` (assignment statement), *expr* (expression), and *call* (function call). Lines are used to indicate the children of each non-terminal, with the leftmost child at the top. The terminals consist of symbols, numbers, and roles. A dashed box is used to delimit the parse tree corresponding to a schema. The name of the schema is written in the lower left-hand corner of the box. The bottom of the figure shows auxiliary information maintained by Ace as

```
    var Charpos: Integer;
  begin
    Charpos := 71;
    ...
    if Charpos > {width-limit} then
       begin Charpos := 1; {lineno} := {lineno}+1; Writeln({report}) end;
    Charpos := Charpos+{width};
    {writes}
  end
```

Figure 32: Pascal code corresponding to the parse tree in Figure 31.

annotation on the parse tree.

The most important difference between the Ace schema in Figure 31 and the one used by KBEmacs (see Figure 6) is the fact that a parse tree is used as the basic representation. However, before discussing this, it is useful to set the scene by discussing the differences that stem from the fact that Ace's schema corresponds to Pascal code instead of Lisp code. These differences are most easily appreciated by considering Figure 32, which shows the way the parse tree in Figure 31 is displayed by Ace's unparser. (Like most of the figures in this section, Figure 32 is based on a figure from [24]; however, a few cosmetic changes have been made to facilitate comparison with the figures in Sections 3 and 4.)

(Since a parser module has not yet been implemented, Ace does not support text like Figure 32 as an input form for schemas. However, once a parser module exists, this will be easy to do. All that is necessary is to introduce declaration-like syntactic extensions of Pascal that support the specification of the annotation information at the bottom of Figure 31.)

Comparison of Figure 32 with the KBEmacs schema in Figure 7 reveals that, while the same cliché is involved, there are a number of differences due to semantic differences between Lisp and Pascal.

Pascal does not maintain information about the number of characters printed on the current line. As a result, the Ace tabular print out schema maintains an explicit record of the character position in the variable `Charpos`. As part of this, the schema contains an additional role *width* and a constraint that derives its value (see Figure 31).

Pascal has nothing corresponding to the Lisp function `FORMAT`. Rather, output must be done using sequences of simple output operations. To retain the benefits of a compact notation for describing output, a simple program generator `GenerateWrites` was written that can convert format strings like those used by `FORMAT` into an appropriate sequence of Pascal operations. Used in a constraint (see Figure 31), this generator forms a bridge between the format role of the Ace tabular print out schema and a new role called *writes* that gets filled with Pascal statements.

The use of `GenerateWrites` in the tabular print out schema illustrates an important point—clichés represented as program generators can be combined with clichés represented as schemas. This is done by using the same representation for schemas and for the output of the generators. Once this is done, the result of a program generator is indistinguishable from a schema and can be freely combined with other schemas. The program generator can be viewed as representing a class of schemas that are generated when needed.

The Ace tabular print out schema uses two roles *report* and *lineno* in place of the free variables `REPORT` and `LINENO`, which are used in the KBEmacs schema. This makes it easier for Ace to combine the tabular print out schema with other schemas, without analyzing data and control flow. The lineno role appears twice in the Ace schema. As in Tempest, this means that whatever is used to fill the role will be inserted in both places. Multiple insertion is particularly useful in conjunction with roles that are intended to be filled with variables.

Comparison of Figures 6 and 31 reveals that Ace and KBEmacs support much the same annotation. However, while it could easily be added, Ace does not support the generation of comments describing a program in terms of the schemas used to create it. As a result, Ace schemas do not contain information about how to generate comments.

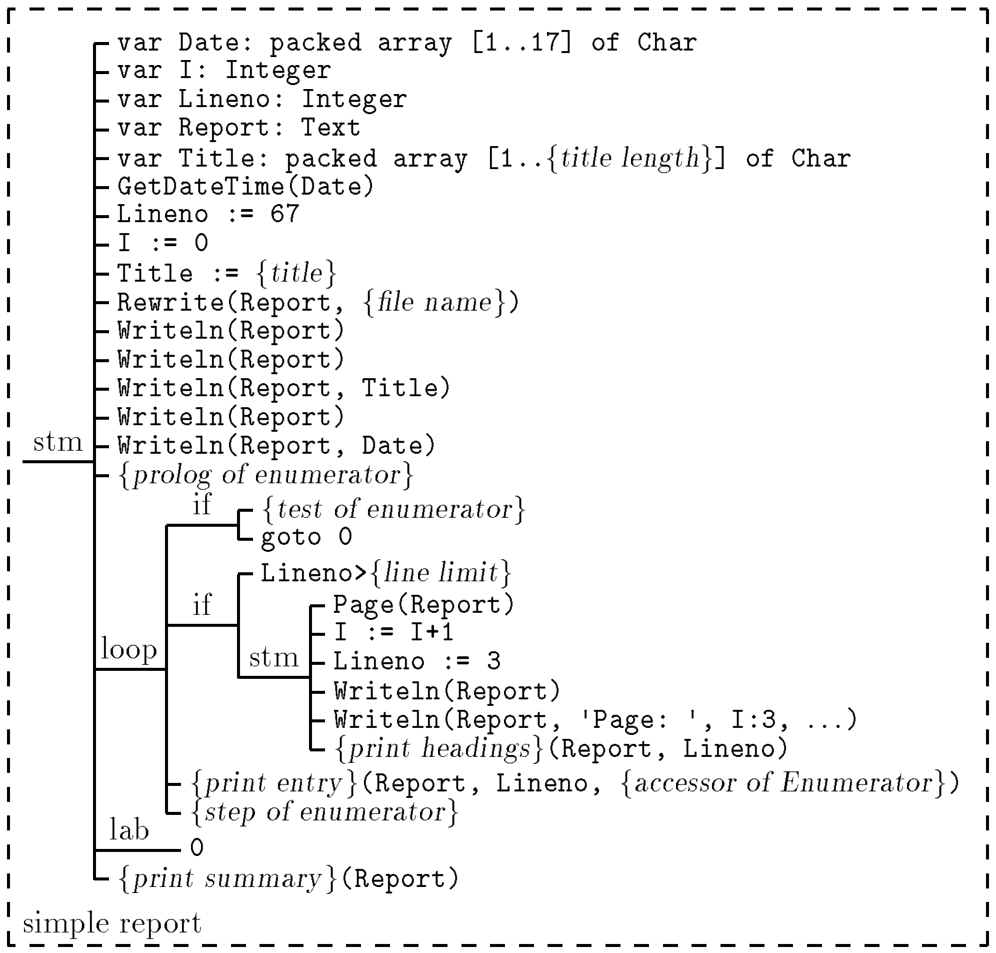
On the other side of the ledger, Ace schemas contain some information that KBEmacs schemas do not—each schema and each role has a syntactic type. These types are used for reasonableness checking when Ace combines schemas together. (As noted at the end of Section 3, KBEmacs simply does whatever the programmer says, whether or not it is reasonable.)

A special programming-language grammar is used. Returning to a consideration of the parse tree in Figure 31, the most important thing to notice is that the grammar underlying the parse-tree is different from the standard grammar for Pascal. The grammar is radically reformulated to simplify the operations performed by Ace's schema editor. In addition, the grammar is extended to reflect the fact that roles can appear anywhere any other syntactic constituent can appear.

Programming-language grammars have traditionally been developed with a number of desiderata in mind, such as unambiguousness, capturing a wide range of information such as operator precedence, and linking up with particular kinds of efficient parsing methods such as LALR. This is convenient for many purposes (e.g., compiler writers), but by no means for all. For instance, these desiderata encourage grammars that lead to deeply nested parse trees. Because the intermediate nodes in these parse trees have little or no significance to human programmers, it has proved useful to use altered grammars that lead to shallow parse trees as the bases for syntax editors [10].

The grammar used by Ace is based on three principal desiderata: minimizing the number of non-terminal nodes in parse trees, minimizing the alterations that must be performed when parse-tree schemas are combined, and moving as many syntactic details as possible out of the parse trees and into the unparser. Several features of the Pascal grammar used by Ace are illustrated in Figure 31.

Rather than using a recursive tower of non-terminals to represent a list of statements, the non-terminal *stm* has a flat structure that can directly contain any number of statements or roles. In addition, no record is maintained in the parse tree of where `begin/end` pairs should appear. Rather, the non-terminal *stm* is allowed to appear directly wherever a single statement can appear (e.g., as a statement in a list of statements or as the `then` clause of an `if`). This approach discards some information about the input and allows ambiguous parses; however, it has three important advantages. The insertion of `begin/end` pairs in the correct places is relegated to the unparser and need not be considered by the schema editor. A schema containing more than one statement (e.g., tabular print out) can be inserted anywhere a statement can appear without having to alter the schema. Since the alteration of schemas is minimized, the structure of a parse tree created by combining several schemas



```
var Date: packed array [1..17] of Char
var I: Integer
var Lineno: Integer
var Report: Text
var Title: packed array [1..{title length}] of Char
GetDateTime(Date)
Lineno := 67
I := 0
Title := {title}
Rewrite(Report, {file name})
Writeln(Report)
Writeln(Report)
Writeln(Report, Title)
Writeln(Report)
Writeln(Report, Date)
{prolog of enumerator}
    {test of enumerator}
    goto 0
    Lineno>{line limit}
        Page(Report)
        I := I+1
        Lineno := 3
        Writeln(Report)
        Writeln(Report, 'Page: ', I:3, ...)
        {print headings}(Report, Lineno)
  {print entry}(Report, Lineno, {accessor of Enumerator})
  {step of enumerator}
0
{print summary}(Report)
```

Default Constraints: {*file name*} = `'report.txt'`.
Derived Constraints: {*line limit*} = `66-Lines(`{*print entry*}`)-Lines(`{*print summary*}`)`.
{*title length*} = `SizeOfString(`{*title*}`)`.
Primary roles: {*enumerator*}, {*print entry*}, {*print summary*}.
Schema type: statements.
Role types: {*file name*}, {*title*}: string.
{*enumerator*}: enumeration.
{*print headings*}, {*print entry*}, {*print summary*}: statements.
{*title length*}, {*line limit*}: number.

Figure 33: The Ace schema for the simple report cliché.

contains a clear record of the schemas used.

The grammar rules for expressions are modified in a similar way. In particular, while the parser has to take cognizance of operator precedence and the placement of parentheses, no explicit record is kept in the parse tree about either one. Rather, the unparser takes care to insert parentheses where needed. This allows the parse-tree representation for expressions to be as simple and straightforward as the Lisp representation for expressions. In particular, when a subexpression is inserted in a role in an expression, it never has to be altered to suit the precedence of the containing operator.

```
    label 0;
    var Date: packed array [1..17] of Char;
        I, Lineno: Integer;
        Report: Text;
        Title: packed array [1..{title-length}] of Char;
  begin
    GetDateTime(Date); Lineno := 67; I := 0; Title := {title};
    Rewrite(Report, {file-name}); Writeln(Report); Writeln(Report);
    Writeln(Report, Title); Writeln(Report); Write(Report, Date);
    {prolog of enumerator};
    while True do
      begin
        if {test of enumerator} then goto 0;
        if Lineno>{line-limit} then
           begin
             Page(Report); I := I+1; Lineno := 3; Writeln(Report);
             Writeln(Report, 'Page: ', I:3, '  ', Title, '  ', Date);
             {print-headings}(Report, Lineno)
           end;
        {print-entry}(Report, Lineno, {accessor of enumerator});
        {step of enumerator}
      end;
  0:{print-summary}(Report)
  end
```

Figure 34: Pascal code corresponding to the parse tree in Figure 33.

To allow a schema to be inserted in a parse tree without having to move the declarations in it to any special place, declarations are allowed to appear anywhere that a statement can appear. (For instance, the type declaration for the variable `Charpos` is included in the statement list in Figure 31.) The unparser takes care of moving declarations to the appropriate place when displaying a program.

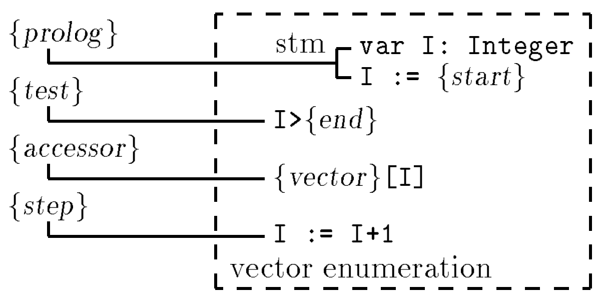
In a similar vein, parse trees are not required to contain any information about label declarations. These declarations are inserted by the unparser as needed.

A new non-terminal *init* is introduced, which is the same as an assignment statement except that it is treated like a declaration specifying an initial value for a variable. It is moved by the unparser to a location where it is only evaluated once.

The simple report schema. Figures 33 and 34 show the Ace schema for the cliché simple report. The top-level structure in the parse tree is a list of nineteen statements, declarations, and labels. To fit this into a reasonable amount of space in Figure 33, subtrees corresponding to simple statements and declarations (e.g., "`var I: Integer`") are abbreviated by simply showing the text itself in place of the subtree.

The non-terminal *loop* corresponds to the construct `while True do`. The non-terminal *lab* corresponds to a label definition. Labels are represented as separate statement-level constituents rather than part of statements because this allows greater flexibility when schemas are combined.

Comparing Figure 34 with Figure 9 reveals that syntax aside, the schema used by Ace is very much like the one used by KBEmacs. However, differences between the semantics of Pascal and Lisp force some notable differences. For instance, the Ace schema uses `Rewrite`



Default Constraints: {*start*} = `LowerBound`({*vector*}).
{*end*} = `UpperBound`({*vector*}).
Input roles: {*vector*}.
Primary roles: {*vector*}.
Schema type: enumeration.
Role types: {*vector*}: identifier.
{*start*}, {*end*}, {*accessor*}: expression.
{*prolog*}, {*test*}, {*step*}: statements.

Figure 35: The Ace schema for the vector enumeration cliché.

```
  var I: Integer;
begin
  {I := {start}, prolog};
  ...
  {I>{end}, test};
  ...
  {{vector}[I], accessor};
  ...
  {I := I+1, step}
end
```

Figure 36: Pascal code corresponding to the parse tree in Figure 35.

instead of `WITH-OPEN-FILE` and multiple calls on `Write`, `Writeln`, and `Page` instead of `FORMAT`.

Further, due to the typing requirements of Pascal, the Ace simple report schema contains detailed type declarations not needed in Lisp schemas. The fact that Pascal requires the length of a string to be declared as part of its type necessitates an additional role *title length*. A constraint specifies that this role should be the length of whatever string fills the title role.

The treatment of enumerators. A final difference between the schemas used by Ace and KBEmacs concerns the enumerator role. In the KBEmacs schema (see Figure 9) the variable `DATA` is used to indicate the data flow connections between the various subroles of the enumerator. However, in the Ace schema (see Figure 34) the subroles are treated as procedure calls without regard for what values they return or alter. This simplification is possible (and necessary), because Ace does not do data-flow analysis.

Figures 35 and 36 show the Ace schema for the vector enumeration cliché. In the parse tree at the top of Figure 35, hooked lines (such as the one beginning below the prolog role) are used to indicate a subtree that is contained in a role. The four roles prolog, test, accessor, and step form the *signature* of the vector enumeration schema—they specify the environment

Command: `Define-procedure "ReportTimings" of "(var Timings: DataVector)".`
Command: `Insert simple-report.`

```
procedure ReportTimings (var Timings: DataVector);
  label 0;
  var Date: packed array [1..17] of Char;
      I, Lineno: Integer;
      Report: Text;
      Title: packed array [1..{title-length}] of Char;
begin
  GetDateTime(Date); Lineno := 66; I:= 0; Title := {title};
  Rewrite(Report, 'report.txt'); Writeln(Report); Writeln(Report);
  Writeln(Report, Title); Writeln(Report); Write(Report, Date);
  {prolog of enumerator};
  while True do
    begin
      if {test of enumerator} then goto 0;
      if Lineno>64 then
         begin
           Page(Report); I := I+1; Lineno := 3; Writeln(Report);
           Writeln(Report, 'Page: ', I:3, '  ', Title, '  ', Date);
           {print-headings}(Report, Lineno)
         end;
      {print-entry}(Report, Lineno, {accessor of enumerator});
      {step of enumerator}
    end;
0:{print-summary}(Report)
end
```

Figure 37: Ace output 1, instantiating a schema.

in which it must be placed. As indicated by the dashed box the contents of these roles form the schema itself.

Comparing the Ace schema with the schemas supported by KBEmacs (see Figure 11) and Tempest (see Figure 20) reveals that the Ace schema has basically the same form as the Tempest schema. In particular, all of the computation in the Ace schema is contained in one or the other of the roles. (In the KBEmacs schema, the initialization of `I` and `SIZE` are outside of the roles, see Figure 11.) In addition, the extraneous constructs that connect this computation together have been omitted. (As noted in the schema type annotation, the schema can only be used to fill the subroles of an enumerator.)

Pascal does not provide functions for determining the bounds of a vector at run time. The Ace vector enumeration schema contains two additional roles (*start* and *end*) and two constraints that determine bounds information at program-construction time.

5.2 Transcript of Ace in Action

Figures 37–40 show a transcript of Ace being used to construct the same simple reporting program that is used in the examples in the preceding sections. The key feature of these figures is that even though Ace is much simpler than KBEmacs, the figures are very much the same as Figures 12–15. While there are a number of minor differences, the programmer basically constructs the same program using the same clichés and the same commands.

The first command in Figure 37 begins the definition the procedure `ReportTimings`. The

Command: `Fill enumerator with vector-enumeration of "Timings".`

```
procedure ReportTimings (var Timings: DataVector);
  var Date: packed array [1..17] of Char;
|     I, J, Lineno: Integer;
      Report: Text;
      Title: packed array [1..{title-length}] of Char;
begin
  GetDateTime(Date); Lineno := 67; I:= 0; Title := {title};
  Rewrite(Report, 'report.txt'); Writeln(Report); Writeln(Report);
  Writeln(Report, Title); Writeln(Report); Write(Report, Date);
| for J := 1 to 32 do
    begin
      if Lineno>64 then
         begin
           Page(Report); I := I+1; Lineno := 3; Writeln(Report);
           Writeln(Report, 'Page: ', I:3, '  ', Title, '  ', Date);
           {print-headings}(Report, Lineno)
         end;
|     {print-entry}(Report, Lineno, Timings[J])
    end;
  {print-summary}(Report)
end
```

Figure 38: Ace output 2, non-local changes.

command used is similar to the KBEmacs command in Figure 12. However, Ace commands are somewhat simpler in a number of ways. In particular, the parameter list of a procedure is specified by typing it in literally and everything corresponding to pieces of program text has to be delimited using double quotes. (The command processor contains a parser for simple Pascal expressions and statements.)

The argument list for `ReportTimings` refers to the type `DataVector`. It is assumed that before beginning the definition of `ReportTimings`, the programmer has defined this type as follows.

```
type DataVector = array [1..32] of Integer
```

With the second command in Figure 37, the programmer specifies that `ReportTimings` is a simple report. As in Figure 12, the code shown in Figure 37 is derived directly from the simple report schema (see Figure 34) with a few constraints being run. On the theory that this is the most convenient place for it to be, Ace automatically moves the editing cursor to the first role in the body of the program that needs to be filled. (As in KBEmacs and Tempest, there is no explicit indication of the positions of filled roles. However, commands exist for moving the cursor to these positions.)

In Figure 38, the programmer fills the enumerator of `ReportTimings` with a vector enumeration. As in Figure 13, this results in changes scattered throughout the program. In particular, the subroles of the enumerator are filled. The constraints associated with the vector enumeration schema (see Figure 35) fill the start and end roles with `1` and `32` based on the definition of `DataVector`.

Given the form of the loop in Figure 37, one might expect that the loop in Figure 38 would have the following form.

```
Command: Fill title with "'Report of Reaction Timings (in msec).'".
Command: Remove print-headings.
Command: Fill print-entry with tabular-print-out of "'~9D'".
Command: Fill print-summary with
            "Writeln(Report); Writeln(Report, 'Mean: ', Mean(Timings):8)".
```

```
procedure ReportTimings (var Timings: DataVector);
  var Charpos, I, J, Lineno: Integer;
      Date: packed array [1..17] of Char;
      Report: Text;
      Title: packed array [1..37] of Char;
begin
  Charpos := 71; GetDateTime(Date); Lineno := 67; I:= 0;
  Title := 'Report of Reaction Timings (in msec.)';
  Rewrite(Report, 'report.txt'); Writeln(Report); Writeln(Report);
  Writeln(Report, Title); Writeln(Report); Write(Report, Date);
  for J := 1 to 32 do
    begin
      if Lineno>63 then
         begin
           Page(Report); I := I+1; Lineno := 3; Writeln(Report);
           Writeln(Report, 'Page: ', I:3, '  ', Title, '  ', Date)
         end;
      if Charpos > 61 then
         begin Charpos := 1; Lineno := Lineno+1; Writeln(Report) end;
      Charpos := Charpos+9; Write(Report, Timings[J]:9)
    end;
  Writeln(Report); Writeln(Report, 'Mean: ', Mean(Timings):8)
end
```

Figure 39: Ace output 3, constraint propagation.

```
I := 1;
...
while True do
  begin
    if I>32 then goto 0;
    ...
    I := I+1
  end;
```

However, Ace's unparser recognizes that this loop can be rendered more compactly in Pascal using `for`.

This is another example of moving syntactic details out of the internal grammar used by Ace into the parser and unparser. The parse-tree representation used by Ace only supports loops of the form `while True do`; however, the unparser generates the full range of `while`, `repeat`, and `for` loops as appropriate.

In Figure 39, the programmer completes the implementation of `ReportTimings` by removing the print headings role and filling the title, print entry, and print summary roles. (Because a full parser module has not yet been implemented for Ace, all of the role filling has to be done via explicit commands rather than direct editing.)

The most interesting changes in Figure 39 are the instantiation of the tabular print out schema and the actions of various constraints. When the tabular print out is inserted, the

Command: `Replace print-entry with simple-print-out of "'~%~14D'".`

```
procedure ReportTimings (var Timings: DataVector);
  var Date: packed array [1..17] of Char;
      I, J, Lineno: Integer;
      Report: Text;
      Title: packed array [1..37] of Char;
begin
  GetDateTime(Date); Lineno := 67; I:= 0;
  Title := 'Report of Reaction Timings (in msec.)';
  Rewrite(Report, 'report.txt'); Writeln(Report); Writeln(Report);
  Writeln(Report, Title); Writeln(Report); Writeln(Report, Date);
  for J := 1 to 32 do
    begin
      if Lineno>63 then
         begin
           Page(Report); I := I+1; Lineno := 3; Writeln(Report);
           Writeln(Report, 'Page: ', I:3, '  ', Title, '  ', Date);
         end;
      Lineno := Lineno+1; Writeln(Report, Timings[J]:14)
    end;
  Writeln(Report); Writeln(Report, 'Mean: ', Mean(Timings):8)
end
```

Figure 40: Ace output 4, modification.

`GenerateWrites` constraint on the schema converts the format string `'~9A'` into the `Write` on the fourth to last line of Figure 39. The constraint on the line limit role of simple report, changes the contents of the line limit from `64` to `63`, because the print summary prints out two lines of output. The constraint on the title length fills that role with `37`, which is the length of the title string.

The command in Figure 40 illustrates that Ace supports program modification as well as program construction. Ace maintains a record of the positions of filled roles and like KBEmacs can replace the contents of a role when requested.

Although they are not illustrated in Figures 37–40, Ace supports a number of syntactic-structure-based commands for inserting syntactic schemas (e.g., for conditionals and loops) and moving around the syntactic structure of the program being edited. The full range of syntax-editing commands could easily be supported.

5.3 Manipulating Parse-Tree Schemas

As with KBEmacs, the key to Ace's operation is the representation shift away from program text to a representation that facilitates the manipulations that have to be performed. Figure 41 shows Ace's internal representation for Figure 39.

The most important aspect of Figure 41 is that while the insertion of schemas in Figures 37–39 results in non-local changes in the program text, the schemas are plugged together in a local way in the internal parse tree. (For the most part Figure 41 is the same as Figure 33 with Figure 35 and Figure 31 nested in it. Note that complete information is maintained about roles even after they are filled.)

As in KBEmacs and Tempest, the fundamental operation of Ace's schema editor is the fill command, which inserts a schema into a role. This is done in five steps: (i) check that

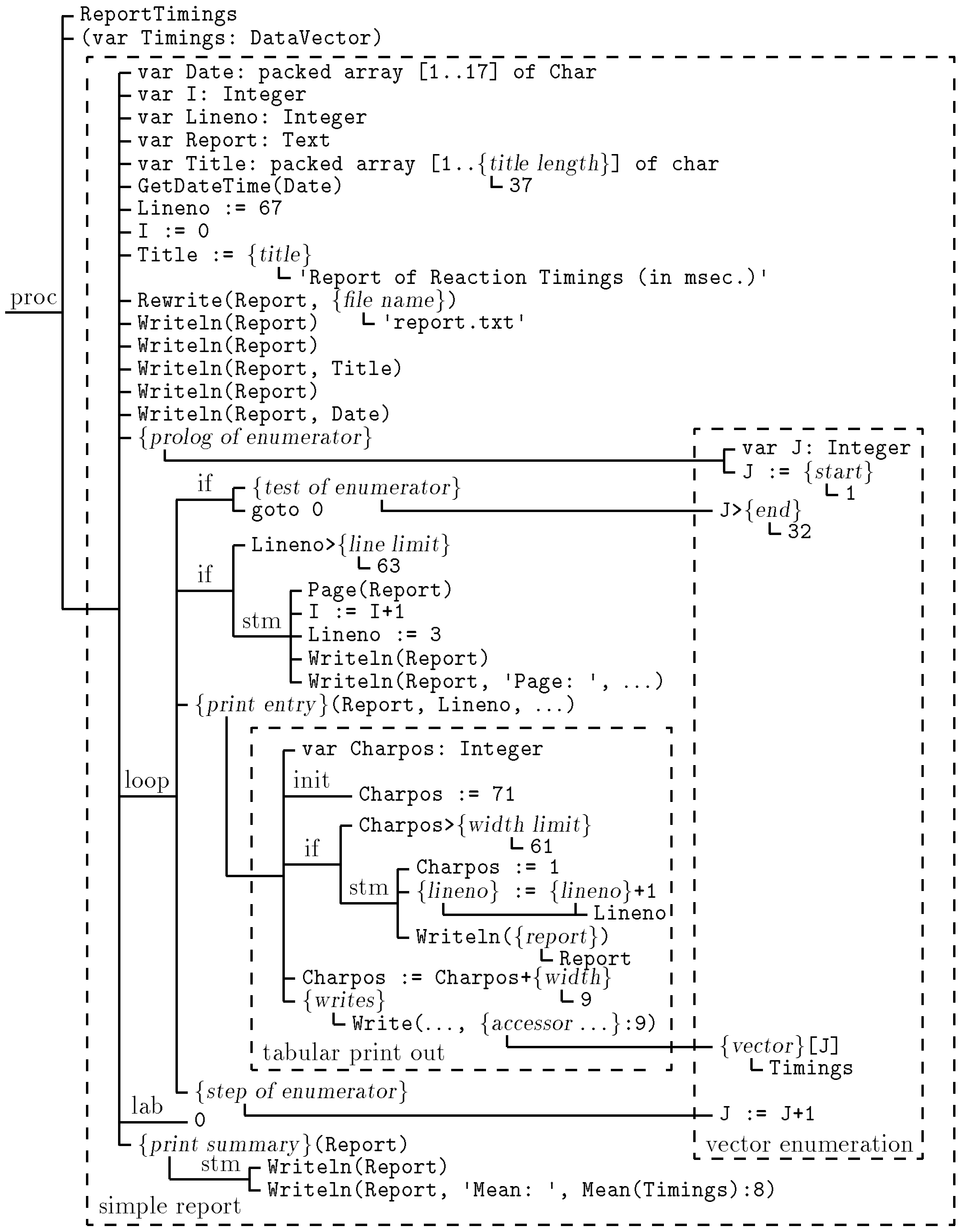


Figure 41: Internal representation corresponding to Figure 39.

the schema will not clash with anything in the pre-existing tree, (ii) fill primary roles in the schema as specified by the command, (iii) fill additional roles in the schema based on the parameter list of the role to be filled, (iv) insert the schema in the parse tree being edited, and (v) run constraints (which may lead to additional roles being filled).

Checking for clashes is one area where Ace's schema editor faces significantly greater challenges than KBEmacs' schema editor. As the first part of this checking, Ace compares the syntactic type of the schema with the syntactic type of the role to be filled, issuing an error message if the schema cannot be made to fit. (KBEmacs should perform this kind of reasonableness checking, but does not.)

Ace then checks to see that inserting the schema in the role will not alter the computation specified by the schema or the surrounding structure. In particular, it checks to see that the schema does not change the definition of any variables or labels that are used in the surrounding context. If a problem is found, it is resolved by renaming the variable or label used in the schema. (KBEmacs does not have to check for this kind of clash, because the plan diagram representation it uses is designed so that the insertion of a schema cannot alter the computation performed by the schema or the surrounding structure.)

Under the mediation of the primary-roles annotation in the definition of the schema, Ace uses any values specified in the fill command to fill roles in the schema. In addition, any default constraints associated with the schema are run. Both of these operations are done in exactly the same way as in KBEmacs.

Ace then uses any values provided in the parameter list of the role to fill additional roles of the schema. For instance, when the tabular print out schema is inserted in the print entry role in Figure 39, the parameter list specified with the role, i.e.,

```
{print-entry}(Report, Lineno, Timings[I])
```

is matched up with the input roles of the schema ({*report*}, {*lineno*}, {*item*}) and the corresponding roles filled with the specified parameter values.

This step is analogous to the work done by KBEmacs to figure out how to connect a schema with the surrounding context. However, it is simpler because all questions of connection are reduced to filling roles of the schema and all matching is done by a simple comparison of the input roles of the schema in order with the parameter list of the role. (KBEmacs sometimes has to convert a role into an input when establishing a connection and has to use heuristics to guess what should be connected to what.)

Once the appropriate roles have been filled in the schema, the schema is inserted in the parse tree. The parse-tree representation makes this step simple. When a schema is used to fill a non-compound role (e.g., when the tabular print out schema is inserted in the print entry role) the only thing Ace has to do is install the schema as the child of the role.

When a schema is inserted into a compound role, the insertion is slightly more complex, but still straightforward. There is no single node in the parse tree corresponding to the compound role as a whole. Rather, there is a node corresponding to each of the subroles. Each of the subroles is filled by linking it to the part of the schema contained in the role of the same name. However, even in this more complex situation, the schema remains as an intact unit after insertion, as shown in the lower right of Figure 41. (The fact that the type of the schema matches the type of the compound role ensures that the proper roles exist in the schema and that the computational context of the subroles in the parse tree is compatible with the context expected by the schema. The fact that all of the computation in a schema such as vector enumeration is contained in roles ensures that nothing is left out when the schema is inserted in this way.)

Once the schema has been inserted, Ace runs constraints in exactly the same way as KBEmacs. In particular, it runs all the derived constraints associated with the schema inserted and the schema containing the role it was inserted into. This may cause roles to be filled or changed, which may cause more constraints to be run. With regard to constraints, the only difference between Ace and KBEmacs is that the constraint functions used by Ace have to operate by directly inspecting parse trees rather than inspecting plan diagrams. This is more complex in many situations, but often not overly so.

With regard to most of the operations above (e.g., checking for clashes, deciding what should be used to fill which role in a schema, and running constraints) the parse-tree representation used by Ace has very little advantage over program text. However, it has important advantages in the critical step of inserting something in a role. The grammar used allows enough flexibility in the ordering of constituents so that Ace can insert almost anything anywhere by simply putting it there. This is analogous to the situation with KBEmacs' plan diagrams and in marked contrast to the situation with program text (or parse trees based on standard programming-language grammars) where considerable adjustment and mixing has to occur.

In Ace, it is the unparser module that does most of the mixing and adjusting that is required when schemas are combined. Conveniently, this is relatively easy to do. For example, when the unparser creates program text corresponding to Figure 41, it scans the parse tree, collects all the declarations together (including the type declaration for the variable `Charpos` contained in the tabular print out schema and the variable `J` contained in the vector enumeration schema), sorts them by type, and prints them compactly in accordance with the rules of Pascal, as shown in Figure 39.

As a second example of what the unparser does, consider that there is no indication in Figure 41 of where `begin/end` pairs should be placed. When printing out statements, the unparser inserts these where necessary. For instance, it inserts a `begin/end` around the statements in the `then` clause of the conditional in the tabular print out, because there are three statements in the clause. However, it does not insert `begin/end` around the statements forming the top level of this schema, because they can be directly included in the statement list containing the print entry role. In a similar fashion, the unparser inserts parentheses based on the priorities of the operators in an expression.

As noted above, the primarily value of Ace's ability to insert almost anything anywhere by simply putting it there is that it makes the fill command easy to support. However, it has another important value as well—it enables the parse tree to contain a clear record of the schemas used to construct a program. This provides the essential foundation for cliché-based modification and documentation of programs. For example, the replace command operates by removing the old contents of a role and then using the fill command to insert something new.

Like Tempest, Ace goes beyond KBEmacs by using knowledge of what was used to fill roles in the removed schema to help fill roles in the schema to be newly inserted. For instance, in Figure 40, Ace uses the accessor of the enumerator to fill the item role of simple print out, because it was used to fill the item role of tabular print out in Figure 39. (KBEmacs was able to get away without this feature, because in KBEmacs most of the connections between a schema and its surroundings are supported by data flow connected to inputs and outputs rather than by filling roles.)

It is worthy of note that while the grammar used by Ace is ambiguous and leaves out a significant amount of information that is present in the standard grammar for Pascal, this need not present a problem for parsing. The ambiguity is not a problem, because one only needs to arrive at some accurate parse. Discarding information is never hard.

5.4 Evaluation

The value of Ace is that it combines the key capabilities of KBEmacs with the simplicity of Tempest. In particular, Ace is not significantly more complex than standard syntax editors, and yet it supports the construction and modification of programs in terms of clichés. However, Ace is weaker than KBEmacs in a number of ways and shares some problems with it.

Ace has components corresponding to each of the components of KBEmacs, however, these components are individually weaker. In particular, parse-tree schemas are not as flexible a representation for clichés as plan diagrams. As a result, Ace cannot be used to operate on as wide a range of clichés.

The reason Ace supports the combination of schemas corresponding to programming clichés much better than Tempest is that Ace's unparser module provides uniform support for most of the modification required when a schema is placed in a particular context. However, Ace's unparser is not as powerful as KBEmacs' coder module. As a result, Ace is not as flexible in the way it combines schemas as KBEmacs.

The central problem in both of these areas is that parse-tree schemas are syntactically rigid in the way they represent clichés. In particular, a parse-tree schema corresponds to a particular syntactic representation for a computation. The unparser may move some parts of this around, however, except for a few special situations (e.g., the way loops are unparsed) the syntactic parts specified always appear somewhere in the code produced. In contrast, plan diagrams express the bare bones of the computation directly, leaving it to the coder to decide how this can best be realized syntactically. This requires a lot of work, but provides the freedom to do a better job.

A problem shared by Ace and KBEmacs is that neither system provides anything other than very rudimentary support for locating the appropriate clichés to be used in a given situation. In the context of component libraries of various kinds, this issue is being actively pursued by many researchers (see for example [8, 13]). Ace's schema library could and should be extended to support browsing, a hierarchical organization of the schemas, and indexing based on keywords or some other kind of descriptors.

As noted at the end of Section 3, another problem with KBEmacs is that its analyzer module is not as strong as it should be, because it is not capable of recognizing the schemas that could have been used to create the program being analyzed. It is inevitable that Ace's parser module will have the same problem. It is easy to arrive at some parse of a given program, but it is far from easy to arrive at a parse that corresponds to some way of building the program from a given library of parse-tree schemas. Further, while it appears that this problem can be fixed in the case of KBEmacs (see [17, 35]), there is little likelihood that it can be fixed in Ace. As above, the central problem is the syntactic rigidity of parse-tree schemas. A small syntactic difference can block a parse-tree schema from matching a piece of code even though the code embodies the same cliché.

An important area where Ace is a lot weaker than KBEmacs is its support for schemas corresponding to loop clichés. By means of temporal abstraction, KBEmacs can compose loop schemas with each other and with non-loop schemas as easily as it can compose non-loop schemas. In contrast, Ace cannot compose loop schemas and can only insert them in roles specially designed to accept them.

(This limitation could be overcome more or less completely by introducing temporal abstraction directly into the programming language being edited, as discussed in [33]. There would then be very little need for Ace to manipulate loop clichés.)

The various defects of Ace discussed above are best viewed as the price of Ace's simplicity. All in all, while the defects are significant, the price paid is relatively small in comparison with the dramatic reduction in complexity. As a result, while it should be valuable in the long run to construct systems with the full power of KBEmacs, it should be valuable in the short run to construct systems like Ace.

6 Conclusion

KBEmacs shows that cliché-based editing can provide powerful support for the implementation and modification of programs. However, since KBEmacs is a quite complex system, two experiments were pursued in an attempt to find simpler ways of supporting cliché-based editing.

The Tempest experiment shows that it is impractical for an editor that operates solely at the level of text to support cliché-based editing of programs. However, it also shows that an extremely simple tool can support cliché-based editing in simpler situations. It would be both practical and valuable to add Tempest-like capabilities to any text-based editor.

The Ace experiment shows that significant cliché-based editing of programs can be supported using a much simpler representation shift than the one supported by KBEmacs. Further, it appears that Ace is an excellent compromise in the sense that it would be hard to get significantly closer to the full power of KBEmacs without introducing the full complexity of supporting plan diagrams. It would be both practical and valuable to add an editor like Ace to any programming environment.

Given that syntax editors have existed for fifteen years or more [6, 11, 20, 25] and reuse has been lauded as a key goal of software engineering for much longer than that, it is somewhat surprising that there are not already many editors like Ace. Nevertheless, while essentially every syntax editor supports syntactic schemas corresponding to basic programming constructs (conditionals, loops, etc.), it is not clear that any syntax editor supports algorithmic schemas like those supported by Ace.

Any syntax editor could be trivially extended to support algorithmic schemas that correspond to well-formed syntactic chunks. It appears that this has been tried occasionally (for instance see [36]), however, by itself, this is not very useful.

The key impediment that blocks typical syntax editors from usefully supporting algorithmic schemas is the grammars they use as the basis for their operation. It has become generally recognized that syntax editors should not be based on the standard grammar for a programming language [10]. However, they typically try to stay as close to the standard grammar as possible. In particular, they typically assume that the grammar should be unambiguous and that the order of the constituents in the parse-tree should be the same as their order in the program text. As long as this is the case, parse trees are not significantly more supportive of cliché-based editing than program text.

The problem is that most algorithmic clichés do not correspond to contiguous chunks of a program, but rather to pieces of information distributed around a program (e.g., a declaration and some statements). In addition, while the standard unparsing process automatically supports more adjustment of schemas than is provided by Tempest, it is not nearly enough to support cliché-based editing of programs.

To support cliché-based editing of programs in a syntax editor, one must take the crucial step of designing the grammar used by the editor with clichés and cliché combination in mind and allow the unparsing process to become somewhat more complex.

6.1 Acknowledgments

The concept of cliché-based editing evolved out of the work of the Programmer's Apprentice project as a whole [16]. In particular, Charles Rich made important contributions to all of the work described here. Further, under the supervision of the author, Peter J. Sterpe implemented Tempest while Yang Meng Tan implemented Ace.

Support for the research described here was provided in part by the National Science Foundation under grant IRI-8616644, in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-88-K-0487, and in part by the IBM, NYNEX, Siemens, and Microelectronics and Computer Technology corporations.

6.2 References

[1] M.A. Ardis, *Template-Mode for GNU Emacs*, draft documentation, Wang Institute, Tyngsboro MA, July 1987.

[2] R. Bahlke and G. Snelting, "The PSG System: From Formal Language Definitions to Interactive Programming Environments", *ACM Transactions on Programming Languages and Systems*, 8(4):547–576, October 1986.

[3] T.J. Biggerstaff and A.J. Perlis (editors), *Software Reusability*, Volumes 1 and 2, Addison-Wesley, Reading MA, 1989.

[4] R. Bilos, "A Token-Based Syntax Sensitive Editor", Linköping University Technical report LiTH-IDA-R-87-02, Linköping Sweden, February 1987.

[5] P. Degano, S. Mannucci, and B. Mojana, "Efficient Incremental LR Parsing for Syntax-Directed Editors", *ACM Transactions on Programming Languages and Systems*, 10(3):345–373, July 1988.

[6] V. Donzeau-Gouge et.al., "A Structure Oriented Program Editor: a First Step Towards Computer Assisted Programming", *Proc. Inter. Computing Symp.*, Antibes France, 1975.

[7] K. Jensen and N. Wirth, *Pascal User Manual and Report* third edition, revised for the ISO Pascal Standard, Springer-Verlag, New York, 1985.

[8] S. Katz, C.A. Richter, and K.S. The, "Paris: A system for Reusing Partially Interpreted Schemas", in *Software Reusability, Volume 1, Concepts and Models*, 257–273, T.J. Biggerstaff and A.J. Perlis (editors), Addison-Wesley, Reading MA, 1989.

[9] B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice-Hall Inc., Englewood Cliffs NJ, 1978.

[10] A. Lomax, "The Suitability of Language Syntaxes for Program Generation", *ACM SIGPLAN Notices*, 22(3):95–101, March 1987.

[11] R. Medina-Mora and P. Feiler, "An Incremental Programming Environment", *IEEE Transactions on Software Engineering*, 7(5):472-482, September 1981.

[12] B. Melese, "Structured Editing—unstructured editing, cooperation and complementarity", *Proc. Second Software Engineering Conference*, 48–53, Nice France, June 1984.

[13] R. Prieto-díaz, "Classification of Reusable Modules", in *Software Reusability, Volume 1, Concepts and Models*, 99–123, T.J. Biggerstaff and A.J. Perlis (editors), Addison-Wesley, Reading MA, 1989.

[14] C. Rich and R.C. Waters, *The Programmer's Apprentice: A Program Design Scenario*, MIT Artificial Intelligence Laboratory memo MIT/AIM-933a, November 1987.

[15] C. Rich and R.C. Waters, "Formalizing Reusable Software Components in the Programmer's Apprentice", in *Software Reusability, Volume II, Applications and Experience*, 313–343, T. Biggerstaff and A. Perlis (editors), Addison-Wesley, Reading MA, 1989.

[16] C. Rich and R.C. Waters, *The Programmer's Apprentice*, Addison–Wesley, Reading MA, 1990.

[17] C. Rich and L.M. Wills, "Recognizing a Program's Design: A Graph-Parsing Approach", *IEEE Software*, 7(1):82-89, January 1990

[18] J.J. Shilling, "Fred: A Program Development Tool", *Proc. Second Conference on Software Development Tools, Techniques, and Alternatives*, 172–180, IEEE Computer Society Press, Washington DC, December 1985.

[19] E. Soloway and K. Ehrlich, "Empirical Studies of Programming Knowledge", *IEEE Transactions on Software Engineering*, 10(5):595–609, September 1984.

[20] R.E. Stallman, "Emacs: the Extensible, Customizable, Self-Documenting Display Editor", *Proc. ACM SIGPLAN-SIGOA Symposium on Text Manipulation*, *ACM SIGPLAN Notices*, 16(6):108–116, June 1981.

[21] R.E. Stallman, *GNU Emacs Manual*, Free Software Foundation, Cambridge MA, March 1987.

[22] G.L. Steele Jr., *Common Lisp: the Language*, Digital Press, Maynard MA, 1984.

[23] P.J. Sterpe, *Tempest: A Template Editor for Structured Text*, MIT Artificial Intelligence Laboratory technical report MIT/AI/TR-843, June 1985.

[24] Y.M. Tan, *Ace: A Cliché-Based Program Structure Editor*, MIT Artificial Intelligence Laboratory working paper MIT/AI/WP-294, May 1987.

[25] T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", *Communications of the ACM*, 24(9):563-573, September 1981.

[26] R.C. Waters, "A Method for Analyzing Loop Programs", *IEEE Trans. on Software Engineering*, 5(3):237–247, May 1979.

[27] R.C. Waters, "Program Editors Should Not Abandon Text Oriented Commands", *ACM SIGPLAN Notices*, 17(7):39–46, July 1982.

[28] R.C. Waters, "User Format Control in a Lisp Prettyprinter", *ACM Transactions on Programming Languages and Systems*, 5(4):513–531, October 1983.

[29] R.C. Waters, *PP: A Lisp Pretty Printing System*, MIT Artificial Intelligence Laboratory memo MIT/AIM-816, December 1984.

[30] R.C. Waters, "The Programmer's Apprentice: A Session With KBEmacs", *IEEE Transactions on Software Engineering*, 11(11):1296–1320, November 1985.

[31] R.C. Waters, "Pretty Printing", in *Common Lisp: the Language*, Second Edition, 748–769, G.L. Steele Jr., Digital Press, Burlington MA, 1990.

[32] R.C. Waters, "Series", in *Common Lisp: the Language*, Second Edition, 923–955, G.L. Steele Jr., Digital Press, Burlington MA, 1990.

[33] R.C. Waters, "Automatic Transformation of Series Expressions into Loops", *ACM Transactions on Programming Languages and Systems*, 13(1), to appear January 1991.

[34] R.C. Waters and Y.M. Tan, "Toward a Design Apprentice: Supporting Reuse and Evolution in Software Design", to appear in *ACM SIGSOFT Software Engineering Notes*, 16(2), April 1991.

[35] L.M. Wills, "Automated Program Recognition: A Feasibility Demonstration", *Artificial Intelligence*, 45(1–2):113–172, September 1990.

[36] W. Yin, *et al*, "Software Reusability: a Survey and a Reusability Experiment", *Proc. 1987 Fall Joint Computer Conference: Exploring Technology Today and Tomorrow*, 65–72, IEEE Computer Society Press, Washington DC, October 1987.

[37] *Lisp Machine documentation (release 4)*, Symbolics, Cambridge MA, 1984.

[38] *Mince*, Mark of the Unicorn Inc., Cambridge MA, 1981.

[39] *Prograph Technical Specifications*, TGS Systems inc., Halifax NS Canada, 1990.

[40] *Scribe Document Production System, user manual*, Unilogic Ltd., Pittsburgh PA, 1984.