MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Working Paper No. 294 May 1987

ACE: A Cliché-based Program Structure Editor

by

Yang Meng Tan

Abstract: ACE extends the syntax-directed paradigm of program editing by adding support for programming clichés. A programming cliché is a standard algorithmic fragment. ACE supports the rapid construction of programs through the combination of clichés selected from a cliché library.

ACE is also innovative in the way it supports the basic structure editor operations. Instead of being based directly on the grammar for a programming language, ACE is based on a modified grammar which is designed to facilitate editing. Uniformity of the user interface is achieved by encoding the modified grammar as a set of clichés.

Contents

1 Introduction 1

1.1 Organization of Thesis 3

2 Related Work 4

2.1 The Cliché Approach to Programming 4

2.2 The Syntax-Directed Paradigm of Program Editing 5

3 Clichés 7

4 Syntax Editing 9

4.1 Syntactic Clichés 9

4.2 Syntactic Cliché Scenario 11

5 Cliché Combination 14

5.1 The Simple_Report Cliché 14

5.2 The Vector_Enumeration Cliché 16

5.3 The Print_Out Cliché 18

5.4 Cliché Combination Scenario 19

6 Implementation 28

6.1 Data Structures 28

6.1.1 Clichés and Roles 28

6.1.2 Constraints 31

6.2 Pretty-Printing 32

6.3 Editor Commands 33

6.3.1 Role Filling 33

6.3.2 Cursor Movement and Tree Traversal 34
6.4 Beyond ACE 35
6.4.1 Features Left Out of the Current ACE System 35
6.4.2 Unsatisfactory Aspects 35
6.4.3 Other Useful Features 36
6.4.4 Future Directions 37

7 Summary **38**

Chapter 1

Introduction

ACE is a Pascal program editor which combines the syntax-directed paradigm of program editing and the cliché approach to programming. A programming *cliché* is a standard algorithmic fragment. The cliché approach to programming is aimed at speeding up the program construction process by reusing clichés stored in a cliché library.

There are two key issues in the cliché approach to program editing: designing clichés and combining them. In order to be useful, clichés must be designed with reusability in mind. This design process is time-consuming and expensive. However, once done, the cost can be amortized over many reuses of the clichés.

In order to reuse clichés, they must be combined together. Unfortunately, when clichés are combined, they usually cannot be merely placed side by side. They must be adjusted so that they are compatible; and their parts have to be intermixed.

ACE takes a step toward addressing the combination problem. ACE supports the instantiation of program clichés from a cliché library. It has some understanding of how clichés must be modified when they are combined together.

ACE adopts the syntax-directed editing paradigm [6,7,24] as the basic framework in which the combination of clichés is studied. In syntax-directed program editing, constructing a program involves building a syntax tree by inserting pre-defined templates into placeholders of previously entered templates. The templates are defined by the syntax of the programming language the editor supports. They have placeholders in them denoting places where additional code is needed to complete the template. Templates are analogous to the right hand side of grammar productions; whereas the placehoders in the templates correspond to nonterminal symbols. An insertion corresponds to a derivation step of the grammar. Creating a program corresponds to deriving a sentence with respect to the grammar for the given programming language [24].

Since the editor automatically inserts keywords and punctuations in a program, syntax-directed program editing spares the user from worrying about these syntactic details. Thus it allows the user to concentrate on the algorithmic aspects of programming. It also helps to prevent and detect some common syntactic errors.

ACE supports syntax-directed program editing for Pascal by using a modified grammar of the Pascal language. The modified grammar is designed to facilitate user editing. For example, expressions in the grammar are simplified so that multiple levels of intermediate nodes in the

grammar are compressed into one by leaving out the precedence information. This makes it easier for the user to construct expressions. (The precedence information is encoded in the printing routines used to pretty-print the parse tree.) In addition, the usual recursive "list-of" construct in context free grammars has been modified so that statements can be direct siblings of one another.

ACE is implemented in Common Lisp on the Symbolics Lisp Machine. The architecture of ACE is shown in Figure 1.1. A program is maintained in the editor in two forms: as text and as a cliché parse tree. A cliché parse tree is similar to an operator-operand tree [1,6,7] where the non-terminals are clichés.

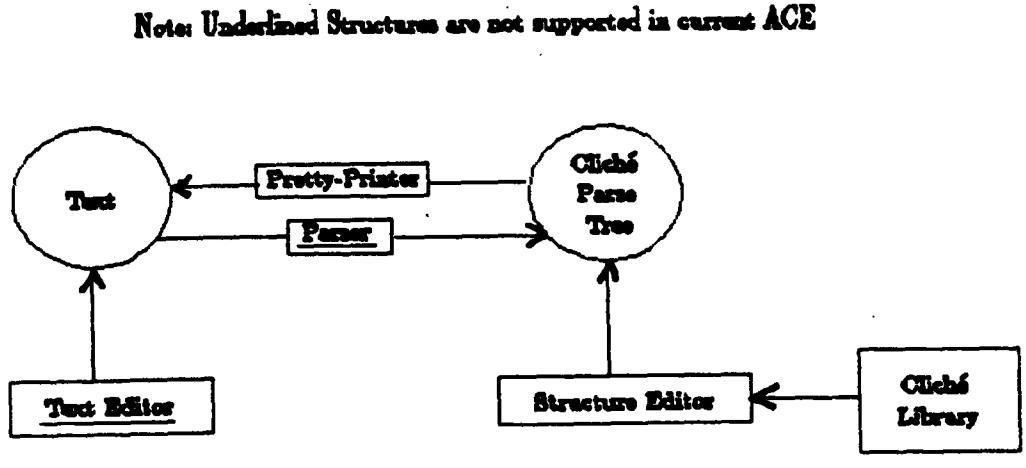

Figure 1.1: The Architecture of ACE.

ACE is intended to have two editing modes: text mode and structure mode. The text mode supports the usual text editing commands. The structure mode supports commands based on the structure of the cliché parse tree (e.g., tree-based navigation and structure-modification commands). Most interestingly, ACE supports commands which instantiate clichés from the cliché library and combine them with a program.

The user can modify the text of the program directly in the text mode. The changes are updated by a parser which parses the text into cliché parse trees. A pretty-printer [26] is used to create program text when the program cliché parse tree is modified.

Since support for clichés is the most novel feature of ACE, this aspect is the exclusive focus of the current thesis. In particular, support is not provided for text editing. Hence, the parser and the text editor are left out in the current system. Also, in the interest of rapid prototyping, the user interface and the syntax of Pascal have both been simplified. Further, ACE does not yet provide any convenient way to define new clichés.

1.1 Organization of Thesis

The above gives an overview of ACE and some relevant background information about the system. Chapter 2 is a brief discussion of the previous work that is related to ACE. Chapter 3 explains the concept of a cliché and gives a simple example of a cliché. Syntactic clichés which encode the syntax of Pascal are discussed in Chapter 4 and a short scenario is shown, illustrating the use of ACE as a syntax-directed program editor. Composite clichés which are used to represent algorithms are discussed in Chapter 5 and a demonstration scenario is also shown. Implementation details are given in Chapter 6 and a summary can be found in Chapter 7.

Chapter 2

Related Work

ACE draws on work from two main fronts: the cliché approach to programming and the syntax-directed paradigm of program editing.

2.1 The Cliché Approach to Programming

ACE is inspired by Waters' KBEmacs system [27]. KBEmacs has demonstrated the usefulness of program clichés. It is able to combine program clichés through the use of an abstract formalism: the plan formalism. This language-independent formalism represents the control and data flow of a library of program clichés. This makes it easy to combine clichés together. However, this power is achieved with some complexity and performance penalties.

Sterpe's TEMPEST system [22] makes an attempt to address the performance issue: it makes use of textual templates to represent clichés. TEMPEST is efficient but it lacks semantic understanding of the templates it operates on. As a result, it is very limited in its ability to combine templates together. Hence it is not really useful as a program editor.

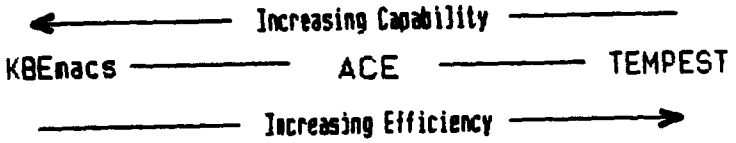

Figure 2.1: Spectrum of cliché editors

ACE presents an intermediate approach: it makes use of syntax-tree-like structures to represent program clichés (see Figure 2.1). This approach is intermediate both power and efficiency between KBEmacs and TEMPEST. The concept of program clichés in ACE comes from KBEmacs. However, unlike KBEmacs, ACE does not understand control and data flow beyond what is implicit in the tree representation. KBEmacs is a semantic editor; ACE is a syntax-directed editor that has some knowledge about the combination of program clichés. Whereas TEMPEST provides the user with a uniform view of both the textual and the structural paradigms

in text editing, ACE presents the user with a uniform view of program editing in terms of clichés in a syntax-directed editor. Unlike either KBEmacs or TEMPEST, ACE explores the use of syntax-directed paradigm as the framework for cliché-based editing.

2.2 The Syntax-Directed Paradigm of Program Editing

The syntax-directed paradigm of program editing has been used in different directions of research. Early research recognized the benefit of giving the editor some knowledge of the syntax of programming languages in order to help programmers avoid and detect syntactic errors during program editing. More recent work has focussed on efficient algorithms for doing incremental syntactic and semantic analysis on partially complete programs. Others have worked on generating language-based editors and complete language-based programming environments.

Most syntax-directed editors prohibit the user from making textual changes to the program [24]. However, syntax-directed commands are not always more convenient to use in program editing. For example, it is easier to enter an expression as a string of text than to build up a hierarchy of intermediate nodes. The consequences of outlawing textual editing are more thoroughly discussed in [25]. This issue has been addressed by Budinsky et al. [5] who developed a *Syntax Recognizing Editor* (SRE). SRE makes use of incremental parsing techniques to support text editing in addition to syntax-directed editing. Others have also followed suit producing similar editors; some examples are PED [13], ED3 [23], and TOSSED [4].

Another direction of research has been toward the generation of programming aids from formal descriptions of programming languages. Some static semantic checking is also enforced by the systems generated from such descriptions. The *Synthesizer Generator* [17] generates syntax-directed editors that incrementally check the static semantics of the partial program in addition to enforcing syntactic correctness. The PSG system [3] and the *Gandalf* project [9] study the generation of complete programming environments from formal descriptions of programming languages.

ACE takes a different direction: it incorporates the syntax-directed paradigm of program editing and the cliché approach to programming. Besides this novel feature, ACE is innovative in the way it supports the syntax-directed paradigm of program editing.

Early syntax-directed editors had knowledge of the syntax of the language they supported, but they were unduly restrictive. They required the user to build the program tree strictly according to the grammar [5,10]. There is some evidence to suggest that human programmers do not think in the way compilers construct parse trees; humans tend to *flatten* program parse trees [19]. Context free grammars used in parsing are motivated by several factors. They are simple to use, computationally efficient, and can be used to encode the precedence relations of arithmetic operators. However, there is no particular reason to expect that context free grammars are useful in capturing the way human programmers view program trees, neither is there any reason to expect context free grammars to be useful in building program editors. Operator-operand trees do much better in this regard.

An early paper by Donzeau-Gouge and his associates [6] proposed an abstract operator-operand tree representation of program which, among other things, distinguishes operators with fixed arity and those with variable arity. The recursive structure of a statement list in context free grammars is replaced by a node with a variable number of children representing the statements.

The Mentor system is a Pascal programming environment centered around a syntax-directed editor based on operator-operand trees [7].

Unlike Mentor and other more recent syntax-directed editors, ACE offers the user great flexibility in constructing program trees. This is achieved by a modified grammar which is designed to support user editing. For example, instead of a tree of statements, ACE allows the user to build a *bush* of statements, i.e., the user can have subtrees of statements as children of other statements.

In contrast to the program editors generated by most of the systems discussed above, ACE does no static semantic checking beyond what is implicit in the grammar. Nevertheless, through the use of constraints in the definition of clichés, it is possible to specify much of the semantics of a programming language; and hence semantic support could be built into ACE. Since ACE is grammar-driven, the ideas presented here can be extended to other languages.

Chapter 3

Clichés

Clichés are schematic fragments used to express commonly used algorithms. A cliché consists of a set of *roles* and a *matrix*. The roles represent those portions of the cliché which vary from one use of the cliché to another. The matrix specifies the unchanging part of the cliché. It specifies how the roles interact to achieve the goal of the cliché. Constraints can be used to specify relationships between roles.

Clichés can be thought of as templates and roles as placeholders. However, unlike templates used in most syntax-directed editors, clichés are *user-definable* templates and constraints can specify the relationships between the placeholders. As an example of a cliché, consider the *equality_within_epsilon* cliché shown in Figure 3.1.

```
Cliche equality_within_epsilon;
  Primary_Roles: x, y;
  Cliche_Types: boolean_expression, expression;
  Role_Types: {x}, {y}, {epsilon}: number, expression;
  Constraints:
    Default: {epsilon} = 0.00001;
  End;
  Matrix: ABS({x} - {y}) < {epsilon}
End;
```

Figure 3.1: The equality_within_epsilon Cliché.

The *equality_within_epsilon* cliché compares two numbers and returns a boolean value which specifies whether or not the numbers differ by less than a given epsilon. It has three roles: x , y, and *epsilon*. Roles are indicated in the matrix by putting braces around them. Requests to create instances of a cliché can be rendered as indefinite noun phrases, e.g., "an equality_within_epsilon of A and B". Such a noun phrase specifies the name of the cliché and may specify values to fill the roles of the cliché. The *primary roles* declaration specifies which roles can be given in such a noun phrase and the order in which they must appear.

The *cliche_types* declaration specifies a list of types to which the cliché belongs. The *role_types* declaration is a list of type declarations for the roles used in the cliché. The type of a

role consists of two parts: the first specifies the most specific type of the role, and the second, the least specific type of the role. In the example, both x and y have the most specific type *number* and the least specific type *expression.* These types are used to support type checking when roles are filled in.

There are two types of constraints: default constraints are computed only when a cliché is first instantiated; derived constraints are re-computed after every change which occurs in a cliché instance. In the *equality_within_epsilon* cliché, only default constraints are illustrated. The constraint shown specifies that the default value of the *epsilon* role is 0.00001.

Chapter 4

Syntax Editing

The concept of syntax editing is derived from the fact that programs are not plain text but are structured according to a grammar. However, it is difficult to build a flexible user interface in syntax-directed editors from the context free grammars of programming languages directly. ACE takes an approach different from many syntax-directed editors. Instead of using the context free grammar of the programming language directly, ACE uses a modified grammar which is designed to facilitate user editing. This modified grammar is encoded as a set of *syntactic* clichés.

4.1 Syntactic Clichés

Syntactic clichés are a special class of clichés: they are templates which mirror the production rules of the grammar for a programming language. Syntactic clichés are motivated by the desire to present a uniform interface to the user. Ordinary user-definable clichés are built out of syntactic ones in the same way that programs are built out of the grammar for a programming language. As an example of a syntactic cliché, consider the *if_statement* cliché in Figure 4.1.

```
Cliche if_statement;
  Primary_Roles: if_test, then_clause, else_clause;
  Cliche_Types: statement;
  Role_Types:
    {if_test}: expression, expression;
    {then_clause}, {else_clause}: statement, statement;
  Format_Routine: if_statement_format;
  Matrix:
    IF {if_test} THEN
      {then_clause}
    ELSE
      {else_clause}
End;
```

Figure 4.1: The if_statement Cliché.

The format of the if_statement cliché is identical to that of the equality_within_epsilon cliché

except for the format_routine declaration. Syntactic clichés use the format_routine declaration to specify how the construct is to be printed.

In ACE, the grammar for Pascal is modified in several ways to better support user editing:

1. Simplified Expressions: The grammar for expressions has been simplified to allow the user more flexible ways of expressing computations. In particular, precedence relations are not expressed in the grammar. Rather, they are encoded in the print routines for the syntactic clichés. This encoding of the precedence information side-steps the need for multiple levels of intermediate nodes in the grammar. Instead of expanding an *expression* into a *simple_expression*, then a *simple_expression* into a *term*, and then to a *factor*, and finally to an *identifier*, ACE allows the user to fill an *expression* directly with an *identifier*. See Figure 4.2 for the modified expression grammar.

```
expression ::=          binary_operation | unary_operation | subprogram_call |
                        reference | number | identifier
binary_operation ::= expression binary_op expression
unary_operation ::=  unary_op expression
```

Figure 4.2: A Section of the Modified Expression Grammar Used by ACE.

2. Types: Each cliché is labeled with a type, as is each role in a cliché. A role may be filled by a cliché if and only if the type of the cliché matches with the type of the role. Some productions in the original grammar of Pascal are encoded in the type information of the syntactic clichés. For example, an expression can be a binary_operation, a unary_operation, a subprogram_call, a reference, a number, or an identifier. This grammar production is encoded in the cliche_types declaration in the definitions of the respective clichés (e.g., the cliché *binary_op* has type expression). The type constraint is enforced by the editor. Thus from the definition of the if_statement cliché shown eariler, the editor will allow any of the clichés of type *expression* to fill the *if_test* role of the if_statement cliché.

3. Generalized Lists: Many important syntactic constructs are "list-of" constructs (e.g, *statement list*, *expression list*, *argument list* and *formal parameter list*). In context free grammars, these are represented as right or left recursive trees. However, this structure is not a natural way to think about a list of things. It is more natural to think of the statements in a "list of statements" as direct siblings of one another. ACE allows such a "list-of" structure.

   As mentioned in the chapter on related work, ACE also allows the user to represent "list-of" construct as a *bush*. As a result, statements can be grouped together according to their origin, (i.e., the clichés they come from). This flexibility is important in supporting cliché combination because it allows for the logical grouping of the sub-parts of clichés. (The pretty-printer is used to unparse such bushes of statements.)

4. Generalized Declarations: Pascal syntax requires a strict ordering of the various types of declarations, namely, TYPE, VAR, FUNCTION, and PROCEDURE declarations. This is rather inflexible for program entry. In the modified grammar, the various types of declarations in Pascal are grouped into one. This makes for a more pleasant user interface. More

importantly, this allows for the logical grouping of declarations which facilitates cliché combination. The pretty-printer is used to sort the declarations into their respective types when they are displayed.

5. Hidden Terminals: Terminals in the grammar are encoded by the printing routines in syntactic clichés. The use of pretty-printing also means that different printing formats could be defined for different users.

4.2 Syntactic Cliché Scenario

The following scenario shows ACE in action as a syntax-directed editor. For the sake of brevity, only part of an editing session is shown. The scenario begins after the user has instantiated an if_statement cliché and filled in the if_test role. Note that when clichés are used, only the matrix is visible to the user through the editor. The rest of the information is maintained in the background by the editor.

```
:
IF x = 1 THEN
  [{then_clause}]
ELSE
  {else_clause};
:
```

Each of the frames below has a Command section which shows the command typed in by the user. The rest of the frame shows the section of the buffer that has been changed by the commands. The cursor is indicated by a box, and the braces {} enclose roles not yet filled in. A description follows the frame which briefly explains the effect of the command on the buffer. (The box is used here for the reader's aesthetic pleasure. To achieve rapid prototyping, a special character is used to indicate the position of the cursor with the help of a status line that indicates the name of the node on which the cursor sits.)

Command: FILL_WITH assignment_statement

```
⋮
IF x = 1 THEN
   {destination} := {source}
ELSE
   {else_clause};
⋮
```

The cursor was initially at the empty role *then_clause*. The FILL_WITH command tells the editor to fill the current role with an instantiation of the *assignment_statement* cliché. The cursor automatically moves to the next unfilled role in the buffer in anticipation that the next command will fill this role.

Command: FILL_WITH "y"

```
⋮
IF x = 1 THEN
   y := {source}
ELSE
   {else_clause};
⋮
```

The *destination* role is filled with the identifier "y". Double quotes are used to indicate input which is program text Single quotes indicate strings. Cliché names are given without any quotes. The cursor moves to the next unfilled role.

Command: UP

```
  ⋮
IF x = 1 THEN
  [y := {source}]
ELSE
  {else_clause}
  ⋮
```

This moves the cursor up the parse tree one level, from the *source* role to the *then_clause.* ACE supports some other navigational commands, i.e., DOWN, NEXT, and PREVIOUS. DOWN moves the cursor down the parse tree by one level. NEXT moves the cursor to the next role. PREVIOUS does the opposite of NEXT.

Command: INSERT while_statement

```
  ⋮
IF x = 1 THEN
  BEGIN
    y := {source}
    WHILE [{while_test}] DO
      {while_body}
  END
ELSE
  {else_clause}
  ⋮
```

The INSERT command shown causes the editor to insert a while statement at the current cursor position. In keeping with Pascal syntax, a compound statement is generated in order to accommodate the new statement.

In The frames above, no complex action has to be taken when clichés were combined. Here, ACE demonstrates more understanding when combining clichés together. ACE has potential for making such minor but yet pleasant niceties by taking care of some details, thus relieving the user from mundane tasks.

Chapter 5

Cliché Combination

The equality_within_epsilon cliché shown in Figure 3.1 is an example of a non-syntactic cliché. It is a very simple example to illustrate some of the features of a cliché. The algorithm expressed in the equality_within_epsilon cliché can also be expressed in a subroutine. Clichés would not be very useful if all clichés were like the equality_within_epsilon cliché. This chapter presents three more complicated clichés and a demonstration scenario of how these clichés are combined together in ACE. The scenario is modeled after a scenario in [27]; it illustrates that ACE can duplicate many of the features of KBEmacs.

5.1 The Simple_Report Cliché

The cliché *simple_report* defines a high-level algorithm for printing a simple report. The simple_report cliché is shown in Figure 5.1. (Note that some lines contain more than one statement in Figure 5.1. This is done so that the figure can fit in one page. In the actual editor each statement is printed on a separate line.) The most specific type of the simple_report cliché is *report*, which is a class of reporting programs; and its least specific type is *List of statements*. As a result, this cliché can be inserted in any role that is of type *List of statements*.

The cliché simple_report has eight roles. The *enumerator* enumerates the elements of some aggregate data structure. The *print_item* is used to print out information about the enumerated elements. The *summary* allows the program to print out some summary information at the end. The *file_name* specifies the name of the file which contains the report. The *line_limit* is used to determine when page breaks should occur. The title of the report is given by the *title* role. The *title_length* role specifies the length of the string containing the title. The *column_headings* is used to print out the column headings on each page.

The enumerator is a compound role with five sub-roles: the *prolog*, the *output_element*, the *step*, the *end_test* and the *epilog*. These can be filled in individually. Alternatively, they can be filled in all at once by a cliché of type *enumerator* which gives the values for each of the sub-roles.

Two types of constraints are illustrated in this cliché. The *file_name* role defaults to the string `'report.txt'`. There are two derived constraints that determine the values of the *line_limit* role and the *title_length* role. When a role is determined by a derived constraint in

```
Cliche simple_report;
  Primary_Roles: enumerator, print_item, summary;
  Cliche_Types: report, List of statements;
  Role_Types:
    {enumerator}: enumeration, enumeration;
    {print_item}, {summary}: printing, List of statements;
    {file_name}, {title}: string, string;
    {line_limit}: expression, expression;
    {title_length}: number, number;
  Constraints
    Default : {file_name} = 'report.txt';
    Derived :
      {line_limit} = (65 − (SIZE_IN_LINES({the print_item})
                          + SIZE_IN_LINES({the summary})));
      {title_length} = SIZE_OF_STRING({title});
  End;
  Declarations:
    VAR
      dated: PACKED ARRAY [1..9] OF CHAR;
      outfile: text; pageno, lineno: INTEGER;
      title: PACKED ARRAY [1..{title_length}] OF CHAR;
  Matrix:
    rewrite(outfile, {file_name}); pageno := 0; lineno := 66; title := {title};
    dated := date; writeln(outfile); writeln(outfile); writeln(outfile);
    writeln(outfile, title); writeln(outfile); writeln(outfile, dated);
    {prolog of the enumerator};
    WHILE TRUE DO
      BEGIN
        {end_test of the enumerator};
        IF lineno > {line_limit} THEN
          BEGIN
            page(outfile); pageno := pageno + 1; writeln(outfile);
            writeln(outfile, 'Page: ', pageno:3, ' ', title, ' ', dated);
            writeln(outfile); lineno := 3;
            {column_headings}(outfile, lineno)
          END;
        {print_item}(outfile, lineno, {output_element of the enumerator});
        {step of the enumerator}
      END;
    {epilog of the enumerator};
    {summary}
End;
```

Figure 5.1: The simple_report Cliché.

terms of other roles, its value is automatically re-computed every time the roles it depends on change; the user never has to fill such roles explicitly. SIZE_IN_LINES computes the number of lines printed by the argument role it is passed; in this case, it is the *print_item* role.

The simple_report cliché is complicated by the desire to keep track of the line number and page number, and to determine when a page break should occur. The cliché assumes 66 lines

to a page, and the line number is initially set to 66 to force a page break after the title page has been printed. After a page break occurs, the line number is reset and the page number is incremented. The *line_limit* and the *column_headings* are expected to update the line number appropriately. The first derived constraint specifies that the line limit should be 65 minus the number of lines printed by the print item and the summary. This ensures that, whenever the line number is less than or equal to the line limit, there will be room for both the print item and the summary to be printed on the current page.

Another derived constraint uses the function SIZE_OF_STRING to compute the length of the title string. This is used to fill in the *title_length* role which is used in the type declaration of the variable `title`. Using such constraints, ACE can take care of some of the type declaration details.

The declarations introduced by the simple_report cliché are gathered in one place, under the part labeled *Declarations*; this could have been part of the *Matrix* but for the sake of clarity, it is kept under a separate label. Note that the types of the sub-roles of the enumerator are not shown in the definition, they are implicit in the structure of an enumerator.

5.2 The Vector_Enumeration Cliché

A complimentary cliché to the simple_report cliché is the *vector_enumeration* cliché shown in Figure 5.2. The vector_enumeration cliché illustrates a few additional features of the syntax of clichés. The notation {*code, name*} means that *name* role of the cliché has been filled with *code*. Thus the *prolog* role of the vector_enumeration cliché consists of the assignment statement, "`index := {start_count}`".

The vector_enumeration cliché belongs to a class of clichés referred to as *enumerators*. An enumerator has eight roles, five of which are already filled in. Two roles handle the input/output of the cliché: the *input* role gives the aggregate structure to be enumerated, and the *output_element* role specifies the output element. The *end_test* role determines when the enumeration is to be terminated. The *step* role steps from one element of the structure to the next. The *prolog* role specifies the computation to be done before entering the loop, and the *epilog* role specifies the computation to be completed after exit from the loop.

In the vector_enumeration cliché, the input role is the *input* role, a primary role. The *output_element* role specifies the output element to be "`{input}[index]`". The *end_test* role is the if statement, which specifies when the loop should end. The *step* role increments the variable `index` by one. The *prolog* role specifies the initial value of `index`. The *epilog* role consists of a label. Note that this cliché introduces a new variable, `index`, and this is shown in the declaration field of the cliché.

The definition of the vector_enumeration cliché is written in a way that facilitates its combination with other clichés. The various parts of a loop are made explicit in the definition. An infinite WHILE loop is used to indicate the position of a loop in a cliché where combination with other clichés may occur; it can be thought of as *syntactic glue* to allow loops to be combined together. In this way, the combination of loops is straightforward: match the various parts of the loops together and fill in the roles with the respective parts of the loop. This method can be extended to handle combinations of multi-role clichés in general. As long as the roles in the

```
Cliche vector_enumeration;
  Primary_Roles: input;
  Cliche_Types: enumeration, loop_fragment;
  Role_Types:
    {start_count}, {end_count}: expression, expression;
    {input}: array_type, identifier;
  Constraints:
    Default:
      {start_count} = LOWER_BOUND({input});
      {end_count} = UPPER_BOUND({input});
  End;
  Declarations:
    LABEL 0;
    VAR index: INTEGER;
  Matrix:
    {index := {start_count}, prolog};
    WHILE TRUE DO
      BEGIN
        {IF index > {end_count} THEN GOTO 0, end_test};
        {{input}[index], output_element};
        {index := index + 1, step}
      END;
    {0:, epilog}
End;
```

Figure 5.2: The vector_enumeration Cliché.

clichés are explicit, no ambiguity can arise in the combination of such clichés.

The goto statement with its corresponding label is printed by a special format routine as a FOR loop. This routine checks to make sure that the transformation from the WHILE loop to the FOR loop is valid and prints out the equivalent FOR loop. (The labels are used to allow a more general loop cliché to be represented.) The equivalent FOR loop for the matrix of the *vector_enumeration* cliché is shown in Figure 5.3. This is the way the cliché will be printed out when it is used.

```
FOR index := {start-count} TO {end-count} DO
  {{input}[index], output_element};
```

Figure 5.3: The Equivalent FOR loop.

5.3 The Print_Out Cliché

The *print_out* cliché prints out an item with the help of a non-Pascal procedure named FORMAT. It is shown in Figure 5.4. The FORMAT procedure takes an output stream variable, a format

string, and an item to be written in the specified output stream. It returns a list of WRITELN or WRITE procedure calls (i.e., Pascal code) that implements the required output. This procedure is inspired by the FORMAT procedure in LISP [21].

Besides increasing the flexibility of input/output capabilities of Pascal, the print_out cliché performs another important function: it updates the line number. This is carried out by the same SIZE_IN_LINES function that is used in the simple_report cliché.

```
Cliche Print_Out(VAR outfile, lineno; items);
  Primary_Roles: format_string, items;
  Cliche_Types: printing, List of statements;
  Role_Types:
    {format_string}: string, string;
    {outfile}: identifier, identifier;
    {items}: expression, expression
  Constraints:
    Derived:
      {writeln_calls} = FORMAT({outfile}, {format_string}, {items});
      {increment} = SIZE_IN_LINES({format_string});
  End;
  Matrix:
    {writeln_calls};
    {lineno} := {lineno} + {increment}
End;
```

Figure 5.4: The Print_Out Cliché.

5.4 Cliché Combination Scenario

The following scenario shows how ACE can be used to construct a simple reporting program. The scenario begins with an empty buffer.

Command: DEFINE_PROCEDURE "report_timings" WITH '(VAR timings: vector)'

```
PROCEDURE report_timings (VAR timings : vector) ;
{procedure_declarations};
BEGIN
  {procedure_body}
END;
```

This defines a procedure with the given name and formal parameter declarations. This is a convenient command that enables a procedure to be easily defined without explicitly building a parse tree. The cursor is positioned at the first unfilled role. (This scenario assumes that VECTOR is declared as an array of integers with bounds from 1 to 100.)

Command: FILL_ROLE procedure_body WITH simple_report

```
PROCEDURE report_timings (VAR timings : vector) ;
VAR
  dated: PACKED ARRAY [1..9] OF CHAR;
  outfile: text;
  pageno, lineno: INTEGER;
  title: PACKED ARRAY [1..{title-length}] OF CHAR;
BEGIN
  rewrite(outfile, 'report.txt');
  pageno:= 0; lineno := 66;
  title := {title};
  dated := date; writeln(outfile); writeln(outfile); writeln(outfile);
  writeln(outfile, title); writeln(outfile); writeln(outfile, dated);
  {prolog of the enumerator};
  WHILE TRUE DO
    BEGIN
      {end_test of the enumerator};
      IF lineno > 63 THEN
        BEGIN
          page(outfile); pageno := pageno + 1; writeln(outfile);
          writeln(outfile, 'Page: ', pageno:3, '  ', title, '  ', dated);
          writeln(outfile); lineno := 3;
          {column_headings}(outfile, lineno)
        END;
      {print_item}(outfile, lineno, {output_element of the enumerator});
      {step of the enumerator}
    END;
  {epilog of the enumerator};
  {summary}
END;
```

ACE searches the buffer looking for the role named procedure_body. The most specific type of the simple_report cliché is *report* which does not match with that of the procedure_body role. ACE then uses the least specific type of the simple_report cliché, which is *List of statements*, as the type of the cliché. This matches the type of the procedure_body role; and hence the cliché is instantiated and inserted. ACE understands that the declarations from the cliché are to be inserted into the declaration part of the enclosing procedure. The default constraint in simple_report assigns the file_name role the value of `'report.txt'`. The line_limit role is filled by the derived constraint which assumes that the summary role and the print_item role will be filled by clichés that output only one newline each.

The cursor moves to the next unfilled role in the buffer. ACE knows that the title_length role, which is the next unfilled role, is supposed to be filled in automatically by constraints, not by the user. Thus it skips the title_length role and moves on to the title role.

Command: FILL_WITH 'Report of Reaction Timings (in msec).'

```
PROCEDURE report_timings (VAR timings : vector) ;
VAR
  dated: PACKED ARRAY [1..9] OF CHAR;
  outfile: text;
  pageno, lineno: INTEGER;
  title: PACKED ARRAY [1..37] OF CHAR;
BEGIN
  rewrite(outfile, 'report.txt');
  pageno:= 0; lineno := 66;
  title := 'Report of Reaction Timings (in msec.)';
  dated := date; writeln(outfile); writeln(outfile); writeln(outfile);
  writeln(outfile, title); writeln(outfile); writeln(outfile, dated);
  {prolog of the enumerator};
  WHILE TRUE DO
    BEGIN
      {end_test of the enumerator};
      IF lineno > 63 THEN
        BEGIN
          page(outfile); pageno := pageno + 1; writeln(outfile);
          writeln(outfile, 'Page: ', pageno:3, ' ', title, ' ', dated);
          writeln(outfile); lineno := 3;
          {column_headings}(outfile, lineno)
        END;
      {print_item}(outfile, lineno, {output_element of the enumerator});
      {step of the enumerator}
    END;
  {epilog of the enumerator};
  {summary}
END;
```

The title role in the program is filled with a string. The title string is 37 characters long. The title_length role is filled in by a constraint derived from the input string for the title role. The changes are highlighted in bold. The cursor moves to the next unfilled role.

Command: FILL_ROLE enumerator WITH vector_enumeration OF "timings"

```
PROCEDURE report_timings (VAR timings : vector) ;
VAR
  dated: PACKED ARRAY [1..9] OF CHAR;
  index: INTEGER;
  outfile: text;
  pageno, lineno: INTEGER;
  title: PACKED ARRAY [1..37] OF CHAR;
BEGIN
  rewrite(outfile, 'report.txt');
  pageno:= 0; lineno := 66;
  title := 'Report of Reaction Timings (in msec.)';
  dated := date; writeln(outfile); writeln(outfile); writeln(outfile);
  writeln(outfile, title); writeln(outfile); writeln(outfile, dated);
  FOR index := 1 TO 100 DO
    BEGIN
      IF lineno > 63 THEN
        BEGIN
          page(outfile); pageno := pageno + 1; writeln(outfile);
          writeln(outfile, 'Page: ', pageno:3, ' ', title, ' ', dated);
          writeln(outfile); lineno := 3;
          {column_headings}(outfile, lineno)
        END;
      {print_item}(outfile, lineno, timings[index])
    END;
  {summary}
END;
```

The enumerator role in the program is filled by the vector_enumeration cliché. This changes all the parts of the program which are parts of the enumerator. There are two steps involved in this frame. The first is the combination of the two clichés – the filling in of the enumerator role of the simple_report cliché. As discussed in the previous section, this can be done in a straightforward manner by matching up the various explicitly named parts. The result of combining the two clichés is shown in Figure 5.5 on the next page. (The roles that are affected by the combination are in bold.) The bounds of **timings** are computed by the functions LOWER_BOUND and UPPER_BOUND. ACE looks up the declaration of the variable **vector**, and using the constraints, the bounds of the array are automatically filled in.

The second step involves transforming the WHILE loop into the FOR loop shown in the scenario. This is done by the special routine that prints the WHILE loop in the form of a FOR loop discussed in the earlier section. The cursor moves to the next unfilled role.

```
rewrite(outfile, 'report.txt');
pageno := 0; lineno := 66;
title := 'Report of Reaction Timings (in msec).');
dated := date; writeln(outfile); writeln(outfile); writeln(outfile);
writeln(outfile, title); writeln(outfile); writeln(outfile, dated);
index := 1;
WHILE TRUE DO
  BEGIN
    if index > 100 then goto 0;
    IF lineno > {line_limit} THEN
      BEGIN
        page(outfile); pageno := pageno + 1; writeln(outfile);
        writeln(outfile, 'Page: ', pageno:3, ' ', title, ' ', dated);
        writeln(outfile); lineno := 3;
        {column_headings}(outfile, lineno)
      END;
    {print_item}(outfile, lineno, timings[index]);
    index := index + 1
  END;
0:;
{summary};
```

Figure 5.5: Internal Result of Combining the Vector_Enumeration Cliché and the Simple_Report Cliché.

Command: REMOVE_ROLE column_headings

```
PROCEDURE report_timings (VAR timings : vector) ;
VAR
  dated: PACKED ARRAY [1..9] OF CHAR;
  index: INTEGER;
  outfile: text;
  pageno, lineno: INTEGER;
  title: PACKED ARRAY [1..37] OF CHAR;
BEGIN
  rewrite(outfile, 'report.txt');
  pageno:= 0; lineno := 66;
  title := 'Report of Reaction Timings (in msec.)';
  dated := date; writeln(outfile); writeln(outfile); writeln(outfile);
  writeln(outfile, title); writeln(outfile); writeln(outfile, dated);
  FOR index := 1 TO 100 DO
    BEGIN
      IF lineno > 63 THEN
        BEGIN
          page(outfile); pageno := pageno + 1; writeln(outfile);
          writeln(outfile, 'Page: ', pageno:3, ' ', title, ' ', dated);
          writeln(outfile); lineno := 3
        END;
      {print_item}(outfile, lineno, timings[index])
    END;
  {summary}
END;
```

The column_headings role is removed, and the arguments it takes are also removed. The cursor is moved to the print_item role. Note that the print_item role has three arguments: outfile, lineno and timings[index]. These arguments can be used by the cliché which fills the role. This is demonstrated in the next scene.

Command: FILL_WITH print_out OF '~%~*:10'

```
PROCEDURE report_timings (VAR timings : vector) ;
VAR
  dated: PACKED ARRAY [1..9] OF CHAR;
  index: INTEGER;
  outfile: text;
  pageno, lineno: INTEGER;
  title: PACKED ARRAY [1..37] OF CHAR;
BEGIN
  rewrite(outfile, 'report.txt');
  pageno:= 0; lineno := 66;
  title := 'Report of Reaction Timings (in msec.)';
  dated := date; writeln(outfile); writeln(outfile); writeln(outfile);
  writeln(outfile, title); writeln(outfile); writeln(outfile, dated);
  FOR index := 1 TO 100 DO
    BEGIN
      IF lineno > 63 THEN
        BEGIN
          page(outfile); pageno := pageno + 1; writeln(outfile);
          writeln(outfile, 'Page: ', pageno:3, '  ', title, '  ', dated);
          writeln(outfile); lineno := 3
        END;
      writeln(outfile, timings[index]:10);
      lineno := lineno + 1
    END;
  {summary}
END;
```

The print_item role is filled by an instantiation of the print_out cliché with the first role of print_out taking the string '~%~*:10'. The string specifies that a newline is to be added and then the argument to be printed should be appended with the output format as specified in the string that comes after it. In this case, `timings[index]` is to have a field width of 10.

As noted in the previous scene, the print_item role has three arguments: outfile, lineno, and timings[index]. These provide access to the variables in the program which are side-effected by the print_out cliché. From the given format string, the print_out cliché computes the number of newlines; and creates a statement which updates the *lineno* variable. It also creates `writeln` statements according to the specification given by the format string. The statements are installed by a derived constraint in the cliché definition. The line_limit role does not change since the default value turns out to be correct in this case: the print_out cliché introduces only one newline. The result returned by the FORMAT function is shown in the buffer in bold (see the previous section on the print_out cliché).

Command: FILL_WITH "writeln(outfile, '————————'); writeln(outfile, 'mean: ', mean(timings):10)"

```
PROCEDURE report_timings (VAR timings : vector) ;
VAR
  dated: PACKED ARRAY [1..9] OF CHAR;
  index: INTEGER;
  outfile: text;
  pageno, lineno: INTEGER;
  title: PACKED ARRAY [1..37] OF CHAR;
BEGIN
  rewrite(outfile, 'report.txt');
  pageno:= 0; lineno := 66;
  title := 'Report of Reaction Timings (in msec.)';
  dated := date; writeln(outfile); writeln(outfile); writeln(outfile);
  writeln(outfile, title); writeln(outfile); writeln(outfile, dated);
  FOR index := 1 TO 100 DO
    BEGIN
      IF lineno > 62 THEN
        BEGIN
          page(outfile); pageno := pageno + 1; writeln(outfile);
          writeln(outfile, 'Page: ', pageno:3, ' ', title, ' ', dated);
          writeln(outfile); lineno := 3
        END;
      writeln(outfile, timings[index]:10);
      lineno := lineno + 1
    END;
  writeln(outfile, '————————');
  writeln(outfile, 'mean: ', mean(timings):10)
END;
```

The summary role is filled by the text given. The text is parsed and inserted. The line_limit role is now changed by the constraint in the simple_report cliché to 62 since the summary role now introduces two newlines. The cursor moves to the top of the procedure as the default whenever there are no unfilled roles in the buffer. (Since rapid prototyping is desired and parsing is well-understood, a mock-up parser is used.)

Command: REPLACE_ROLE print_item BY print_out OF '~2%~*:16'

```
PROCEDURE report_timings (VAR timings : vector) ;
VAR
  dated: PACKED ARRAY [1..9] OF CHAR;
  index: INTEGER;
  outfile: text;
  pageno, lineno: INTEGER;
  title: PACKED ARRAY [1..37] OF CHAR;
BEGIN
  rewrite(outfile, 'report.txt');
  pageno:= 0; lineno := 66;
  title := 'Report of Reaction Timings (in msec.)';
  dated := date; writeln(outfile); writeln(outfile); writeln(outfile);
  writeln(outfile, title); writeln(outfile); writeln(outfile, dated);
  FOR index := 1 TO 100 DO
    BEGIN
      IF lineno > 61 THEN
        BEGIN
          page(outfile); pageno := pageno + 1; writeln(outfile);
          writeln(outfile, 'Page: ', pageno:3, '  ', title, '  ', dated);
          writeln(outfile); lineno := 3
        END;
      writeln(outfile);
      writeln(outfile, timings[index]:16);
      lineno := lineno + 2
    END;
  writeln(outfile, '----------');
  writeln(outfile, 'mean: ', mean(timings):10)
END;
```

The role *print_item* is replaced by another instantiation of the print_out cliché to demonstrate the replace_role command. The format string specifies that there should be two newlines and the output item is to have a field width of 16. Now the new line_limit becomes 61 since one more newline has been introduced in the print_item role. The variable `lineno` is incremented by 2 instead of by 1 and the field width specification of `timings[index]` is updated appropriately. Note that a new `writeln` procedure call has been inserted to output one more newline.

Chapter 6

Implementation

It is appropriate to highlight the objective of this thesis in order to better appreciate the task at hand. This thesis attempts to demonstrate the feasibility of cliché-based editing in a syntax-directed paradigm. In the interest of time, no attempt was made to implement proven technologies.

This chapter is divided into three sections. The first section describes the data structures used in ACE. Pretty-printing is described in the second section. The last section is a discussion of some of the interesting editor commands.

6.1 Data Structures

In this work, clichés are hand-coded but a grammar for them can be formally specified so that they could be easily parsed. There are three basic entities in ACE: roles, clichés and constraints. The first two are discussed together below due to their complementary nature. The next subsection then describes constraints.

6.1.1 Clichés and Roles

A role is represented as a Lisp structure that has the following named components:

- name: gives the name of the role.
- parent: contains a link to the syntactic parent of the role.
- c-parent: contains a link to the composite cliché parent of the role in the case when the role is part of a composite cliché. (The need for this link will become clear in the discussion at the end of this section.)
- typeof: contains the most specific type and the least specific type of the role.
- optionalp: contains a flag to indicate whether the role is optional.
- skippedp: contains a flag to indicate whether this role is to be skipped in printing.

- multirole: contains a list of the component sub-roles if this role is a multi-part role; it is nil otherwise.
- parent-role: contains a link to the parent role in the case when this role is a sub-role of another role.
- filled-by-constraints-p: contains a flag to indicate whether the role is automatically filled in by constraints or not. This helps the editor decide whether to put the cursor on this role.
- content: contains the content that fills the role. The content may be a cliché, a list of clichés or a string representing terminals.
- filledp: contains a flag to indicate whether the role has been filled in. This component is not essential since it depends on the values of the optionalp, skippedp and content components of the role. However, since the test to see if a role is filled is a very common operation in ACE, this component exists for ease of programming and efficiency reasons.

To appreciate how the roles are used, it is necessary to show how clichés are represented in ACE. A cliché, like a role, is a Lisp structure. It has the following components:

- name: contains the name of the cliché.
- parent: contains a link to its parent.
- typeof: contains a list of types it belongs to.
- content: contains a list of roles and clichés that makes up the cliché.
- prole: contains a list of the primary roles the cliché takes.
- formals: contains a list of the formal parameter roles the cliché takes.
- actuals: contains a list of roles that are the actual parameters of the cliché.
- allroles: contains a list of all the roles that appear in the cliché.
- decls: contains a list of declarations introduced by the cliché.
- format: contains the name of the formatting function for this cliché. This is only used for syntactic clichés.
- multipart: contains a list of the roles that are the sub-parts of this cliché. This is complementary to the sub-role feature of roles.
- default-constraints: contains the name of the function that implements the default constraints for this cliché and a list of the roles that this function takes.
- derived-constraints: similar to the default-constraints component, used for derived-constraints.

The simplest clichés in ACE are the syntactic clichés. They are really a degenerate class of clichés. The components of interest here are the NAME, PARENT, TYPEOF, CONTENT, and FORMAT components. The actual implementation makes a distinction between syntactic clichés

and composite ones; this distinction is unnecessary but it is useful for ease of programming. To simplify programming and to increase the ease of modifiability, the actual implementation collapses the three types of entities into one structure, called a *node*, whose components are the union of all the components of the two entities described. The different entities it represents are distinguished by another component named *kind*. Thus a role is a node of the kind *role*.

```
(defun if_statement (parent)
  (let* ((nod (make-node :kind 'syntactic :parent parent
                         :name 'if_statement
                         :typeof 'statement :format 'if_statement_format))
         (label (make-node :kind 'role :parent nod :name 'statement_label
                           :typeof (cons 'number 'number)
                           :optionalp t :skippedp t))
         (if_test (make-node :kind 'role :parent nod :name 'if_test
                             :typeof (cons 'expression 'expression)))
         (then_clause (make-node :kind 'role :parent nod
                                 :name 'then_clause
                                 :typeof (cons 'statement 'statement)))
         (else_clause (make-node :kind 'role :parent nod
                                 :name 'else_clause
                                 :typeof (cons 'statement 'statement))))
    (setf (node-content nod) (list label if_test then_clause else_clause))
    (setf (node-filledp nod) t)))
```

Figure 6.1: Lisp Code for the If_Statement Cliché.

In the actual implementation, macros are used to define procedures like the if_statement clichés. The code for the if_statement cliché is straightforward, as shown in Figure 6.1. The format function is discussed in the section on pretty-printing. Each invocation of the if_statement function returns an instance of the if_statement cliché.

Composite clichés are constructed out of syntactic ones. A section of the cliché parse tree for the vector_enumeration cliché is shown in Figure 6.2. Note the use of the c-parent link of the *start_count* role in the tree. A different parent link is needed for roles of composite clichés since they also exist to fill the roles of the underlying syntactic clichés. To ensure that the editor can access the composite cliché after the role has been filled in by a composite cliché, the roles originating from composite clichés need to have a link to their *c-parent*. The distinction between the parent link and the c-parent link helps the pretty-printer to walk through the tree traversing only the parent links, and forget about the c-parent links altogether. In this experimental system, composite clichés are hand-coded. A true prototype of ACE should have a parser that transforms user-defined clichés into data structures described above.

6.1.2 Constraints

Constraints are kept in clichés as a pair: the first part is a procedure name and the second part is a list of arguments that this procedure takes. Calling a constraint simply runs the procedure with the given argument list.

The function in Figure 6.3 illustrates the simple representation of constraints. The function calls LOWER_BOUND and UPPER_BOUND look up the bounds of the vector node given. Number returns

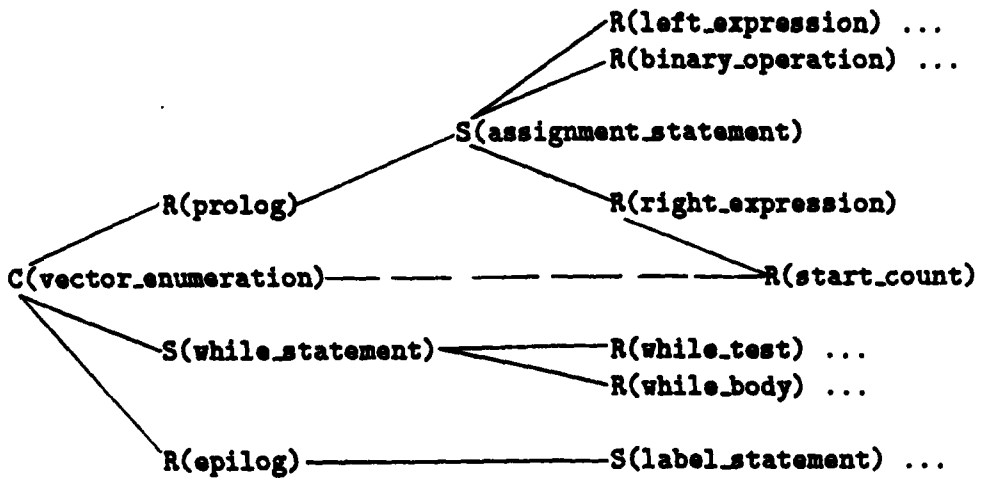

Figure 6.2: A Section of the Cliché Parse Tree for the Vector_Enumeration Cliché.

a syntactic cliché of type *number* with the parent component filled by the given argument.

Since constraint propagation is well-understood, the implementation does not attempt to do full constraint propagation. It has no mechanism for keeping track of which constraints have been run. It keeps a global list of derived constraints that are to be run, and the programmer decides how many times they are to be invoked! Default constraints are run when clichés are instantiated.

```
(defun vector-enumeration-default-constraints (arglist)
  (let* ((scount-role (first arglist))
         (ecount-role (second arglist))
         (vector-role (third arglist))
         (init-count (LOWER_BOUND vector-role))
         (end-count (UPPER_BOUND vector-role))
         ; instantiation of number cliche
         (init-count-cliche (number scount-role))
         (end-count-cliche (number ecount-role)))
    (setf (node-parent init-count-cliche) init-count)
    (setf (node-parent end-count-cliche) end-count)
    (fill-with scount-role init-count-cliche)
    (fill-with ecount-role end-count-cliche)))
```

Figure 6.3: Lisp Code for the Default Constraints in Vector_Enumeration Cliché.

6.2 Pretty-Printing

The pretty-printer used is Waters's PP [26]. PP proves to be a very versatile tool. It is the workhorse in ACE. Every structure in Lisp can be defined so that it is associated with a user-supplied print function. This function specifies how the structure is to be printed. The print function has access to the entire structure when it is called. Hence, it can determine how to print the specific structure based on the information stored in the structure. When a syntactic node is encountered, the print function calls the formatting function stored in the format component of the node with the content component of the node.

The format component of the node structure may contain a formatting function which specifies how the node is to be printed. This feature is used in syntactic clichés. As an example, the format function for the if_statement cliché is shown in Figure 6.4. By changing the formatting functions of the syntactic clichés, the user can customize the way the editor prints out the content of the buffer.

```
(defun if_statement_format (stream list)
  ; extract the appropriate roles for testing
  (let ((label (first list))
        (else (fourth list)))
    (if (node-filledp label)
        ; labels to be printed
        (print-if-statement-with-labels stream list)
        (if (node-skippedp else) ; ELSE clause missing
            (format stream "~!~*IF ~W THEN~2I~%~W~-2I~*~." list)
            (format stream "~!~*IF ~W THEN~2I~%~W~-2I~%ELSE~2I~%~W~-2I~." list)))))
```

Figure 6.4: Lisp Code for the Format Function of If_Statement Cliché.

As discussed in the scenario above, loops are represented in the most general form possible in Pascal, and this led to the use of GOTO and labels. For reasons of aesthetics, the GOTO statement is transformed into an equivalent FOR loop. However Pascal labels must be declared before use. This meant that the transformation analysis has to be done before the root of the tree is printed. The analysis, in this case, only makes very simple checks to make sure that such a transformation is possible before printing it out as a FOR loop. The data structure still retains the generalized version of the loop. More general and careful checking is desirable but it has not yet been introduced in the interest of time.

6.3 Editor Commands

The various navigational and structure commands in the editor are implemented straightforwardly, based on the structure of the tree. Only the more interesting commands are discussed here.

6.3.1 Role Filling

The most interesting operation in ACE is filling roles. It does most of the real work. It comes in several forms: FILL_WITH, FILL_ROLE, REPLACE_ROLE and INSERT. There are several key aspects to filling roles:

- Type Checking: Each role has a type which describes the set of clichés that can be used to fill it. Type checking has to be done before a role is filled to ensure that the cliché is type compatible with the role. Type compatibility is currently explicitly defined in a table. For example, a role node of type *expression* can accept a cliché of any of the following types: *identifier*, *number*, *binary_operation*, *unary_operation*, *function_call*, *reference*, and *string*.
- Handling Declarations: A cliché may introduce new declarations that must be placed in the appropriate places. Such new declarations are placed in the lexically closest declaration block from the role which receives the instantiated cliché.
- "Cliché Call": The fill command has to check to see if the current role provides any actual parameters to the instantiated cliché. An example from the scenario is the filling of the print_item role with the print_out cliché. The print_out cliché is supplied with the arguments already given in the print_item role. The editor does an implicit *fill* on the formal parameter roles with the actual parameters resident on the print_item role. The pretty-printer prints out the actual parameters of print_item in the same way as a Pascal procedure call. When the print_out cliché is instantiated and filled in, the role node is actually filled by the derived constraints, as explained in the earlier scenario.
- Sub-Roles and Sub-Parts of Clichés: When the instantiated cliché has several sub-parts that are to be connected to different places in the role that is receiving it, the fill command needs to properly connect them together. This is done by convention: if the cliché has multiple sub-parts and the role has multiple sub-roles, they are matched up one by one and the respective sub-roles are filled by the respective sub-parts. Since both the sub-roles and the sub-parts are named, they can be matched up in a simple manner.
- Handling BEGIN and END: When an INSERT command is used to insert a statement into an already filled statement role, ACE introduces the BEGIN and END compound statement automatically and filled it with a list of the two statements.

As indicated in the introduction, a key issue in cliché-based editing is providing an editor which can modify the clichés appropriately when they are combined. Three aspects of this modification are shown in the scenario:

1. Careful Design of Modified Grammar: By allowing generalized declarations to go out of order and relaxing statement lists to become statement trees, components of clichés can be kept together. The pretty-printer is able to print out the tree correctly.
2. Use of Constraints to Propagate Information: In more complex modifications to the program, constraints can be used to propagate some the information. This is illustrated by the scenario involving the print_out cliché.

3. Taking Care of Some Details: The editor can take care of some details imposed by the Pascal language, as illustrated by the insertion of a compound statement when more than one statement is inserted to a role of the statement type.

If more extensive examples were used, there would have been much more modifications needed, e.g., variable renaming. The modifications involved in loop adjustments have already been discussed before.

6.3.2 Cursor Movement and Tree Traversal

Navigational commands moves the cursor which traverses the parse tree. The key requirement of the traversal is that the terminals must appear from left to right since this is how the program looks to the user. Due to the way pretty-printing is done, the parse tree may not correspond exactly to the way the user sees the program. For example, declarations are sorted before they are printed out. Besides, some fill commands take a named role as one of their arguments. The search for the named role entails a walk of the tree starting from the current cursor position. The search order ought to be the same as the order in which the program is printed out.

In ACE, navigational commands are not fully supported. To fully support navigational commands, the order of printing carried out by the pretty-printer has to be explicitly recorded. In searching for a named role, ACE also includes the parent-role of the node being visited to handle the case when only the sub-roles are present in the visible buffer. This was the case when the eumerator role was filled in the scenario.

6.4 Beyond ACE

This section contains some notes that are relevant to further experimentation in the ACE framework. This includes brief discussions on the features that have been intentionally left out of ACE, the unsatisfactory aspects in the current implementation of ACE that ought to be improved upon, other features that will be useful in a prototype, and future directions.

6.4.1 Features Left Out of the Current ACE System

The combination of text and cliché-based editing is an issue left out in this work. When is a cliché still valid after textual changes have taken place is an important question. Its solution will determine how useful a dual approach (text and cliché editing) can be. Much of the usefulness of the current work rests on the premise that this issue can be adequately addressed. Wills's work [29] on automated program recognition has taken a step in that direction but incorporating her work into an editor has not been done.

Currently, there is no way to define a cliché other than to build its parse tree representation manually. A parser is needed for parsing textual input of clichés into the internal tree representation.

A more usable version of ACE must pay careful attention to user interface issues which have also been left out of this study. An input expression parser is needed to suport the input of

expressions without explicitly building parse trees. Perhaps a mouse-sensitive window-oriented display will increase the bandwidth of interaction with the user and thus enhanced its usefulness.

The complete syntax of Pascal should be supported, but extending the current set of Pascal syntax to the complete one should not be difficult.

6.4.2 Unsatisfactory Aspects

The single most unsatisfactory part of ACE is the way the generalized loop is printed as a FOR loop. The vector_enumeration cliché could have been written differently to anticipate this. However, the loop combination in general would no longer be straightforward matching of sub-roles with sub-parts. A much more complicated algorithm is needed. It is not clear if this alternative approach is better than the one chosen here. Another alternative is to sacrifice program aesthetics and modularity by printing out the general representation in the form of GOTO statements and labels.

In retrospect, printing the generalized loop into a FOR loop was not a good idea. In some sense it is pushing the pretty-printing game a step too far. What ought to be done is to perform a genuine transformation at the time when the combination takes place. The internal representation of the generalized loop should have been transformed into a FOR loop if such a transformation is valid when the FILL takes place. To the user, there is no difference, since the appearance is the same in both cases. However, in the printing case, navigational commands based on the structure of the tree will not work correctly. (The cursor cannot get to the printed FOR loop.)

Unfortunately, transformation is no panacea. It is essential to keep the old loop representation around in order to support retraction, or undo-ing previous commands. Maintaining the consistency of both representations is at best difficult, and possibly impossible to achieve in general.

The way composite clichés are represented in a tree-structure is not a foregone question. Composite clichés have their own roles, connected to the parent via the c-parent link. However, composite clichés are made up of syntactic ones, which have their own named roles. To illustrate the problem, take the example of the *title* role of the simple_report cliché. The printed statement looks like: `title:= {title}`. The assignment_statement cliché has two roles, the destination and the source. ACE views it as the source role being filled by the title role. This results in a general case where a chain of role nodes may be present in the tree. Another possible view is to treat the title role and the source role as aliases for the same thing. In the first case, the uniformity of the tree is reduced by the existence of possibly long chains of role nodes being filled in by role nodes. The attendant gain is the simplicity of printing empty role nodes and filling in the role nodes. In the latter case, the uniform nature of the tree structure is preserved at the expense of more complicated checking when an empty role is printed out.

A new syntax should be developed for expressing the idea of a cliché call. Currently, its syntax is the same as that of a function call in Pascal; and this can lead to some confusion.

6.4.3 Other Useful Features

Renaming of variables in the buffer ought to be supported. Some constraints need to be introduced to do some simple static semantic checking like checking for duplicate and missing declarations, and possibly adding automatic declaration of variables. An editor capable of keeping track of minor details can go a long way toward reliable and pleasant programming. With the cliché library, some assistance could be provided in presenting the user with the list of all possible clichés that can fill a particular role based on the type of the role.

6.4.4 Future Directions

There is one aspect of this work that has not been explored adequately. It is the retraction mechanism. The data structures used have the property that the changes made to them can be retracted/undone if desired. This is a very useful feature in a program editor since program development is a not a monotonic activity. However, if the simple solution of replacing the generalized loop with a more specific and prettier loop without keeping the old version around is adopted, this goal of maintaining retraction capability will stand in conflict. How the retraction capability can be preserved and at the same time support flexible and efficient loop combination is one useful aspect to study.

The data structures used in ACE are simple, familiar and versatile. The node structure for roles and clichés can be extended to include some summarized form of data and control flow information. It is possible to do detailed data and control flow analysis on the program tree in the same way as optimizing compilers. However, it is likely that most of the data and control flow information useful for cliché-level "semantic" processing can be abstracted or summarized over the cliché unit. This should result in better efficiency in analyzing the program.

Further exploration may be fruitful in the realm of type-checking and type coercion in the ACE framework. There is scope for smartness in a system which can figure out the right type for a role and does some automatic type coercion to achieve the desired effect.

Chapter 7

Summary

The primary achievement of this work lies in the incorporation of the cliché-based approach to programming with the syntax-directed paradigm in program editing.

ACE departs from the usual practice of remaining blindly faithful to the grammar of the programming langauge a program editor supports. It uses a modified grammar which is designed to facilitate editing. Uniformity of the user interface is achieved by encoding the modified grammar as a set of clichés.

The cliché approach to programming is useful for reliable and rapid construction of programs. This work has also indicated that where programming is tedious and difficult due to language limitations, this approach can significantly mitigate the unpleasantness of the langauge.

The scenario so far demonstrated by ACE are too few and too specific to support a definitive statement about the feasibility of ACE. The cliché approach to programming has many difficulties which need to be addressed before it can be practical. However, this work has helped to demonstrate that some of the mileage this approach provides can be reaped without paying a huge efficiency penalty.

It is pertinent to restate the dictum that *knowledge is power*. The usefulness of a knowledge-based program editor depends on its knowledge base of programming clichés. ACE has helped demonstrate that at least the approach is not limited by the lack of efficient tools that can make use of the knowledge base.

Acknowledgments

I would like to thank Dr Richard C. Waters, my thesis supervisor, for guiding me in my work and for initiating many of the ideas presented here. Chuck Rich and Howard Reubenstein have lent their support and assistance to me in my work in one way or another. I would also like to thank Tze-Yun Leong for her constant support and encouragement.

Appendix A: A Subset of Pascal Grammar in ACE

program ::=	PROGRAM identifier LISTOF(id, 'commas, 'parentheses); LISTOF(declaration, 'semi-colons, 'parentheses); BEGIN LISTOF(statement, ';') END.
declaration ::=	type_declaration | variable_declaration | function_declaration | procedure_declaration
label_declaration ::=	LABEL LISTOF(number, 'commas);
variable_declaration ::=	VAR LISTOF(id, 'commas) : type
function_declaration ::=	FUNCTION identifier LISTOF(formal, 'commas 'parentheses)): type; LISTOF(declaration, 'semi-colons); BEGIN LISTOF(statement, 'semi-colons) END
procedure_declaration ::=	PROCEDURE identifier LISTOF(formal, 'commas, 'parentheses)); LISTOF(declaration, 'semi-colons); BEGIN LISTOF(statement, 'semi-colons) END
type_declaration ::=	TYPE identifier = type
type ::=	identifier | array_type | string_type | INTEGER_TYPE | CHAR_TYPE
array_type ::=	ARRAY [number..number] OF type
string_type ::=	PACKED ARRAY [1..number] OF CHAR
integer_type ::=	INTEGER
char_type ::=	CHAR
formal ::=	value_formal | reference_formal
value_formal ::=	LISTOF(id, 'commas) : type
reference_formal ::=	VAR LISTOF(id, 'commas) : type
statement ::=	assignment_statement | procedure_call | if_statement | label_statement | while_statement | for_statement | compound_statement
compound_statement ::=	BEGIN LISTOF(statement, 'semi-colons) END
assignment_statement ::=	left_reference := expression
if_statement ::=	IF expression THEN statement ELSE statement
while_statement ::=	WHILE expression DO statement
for_statement ::=	FOR identifier := expression TO expression DO statement
label_statement ::=	number
expression ::=	binary_operation | unary_operation | function_call | right_reference | number | identifier
binary_operation ::=	expression binary_op expression
unary_operation ::=	unary_op expression
function_call ::=	id LISTOF(expression, 'commas, 'parentheses)
procedure_call ::=	id LISTOF(expression, 'commas, 'parentheses)
left_reference ::=	id LISTOF(expression, 'commas, 'square-brackets)
right_reference ::=	id LISTOF(expression, 'commas, 'square-brackets)
binary_op ::=	= | < | > | <= | >= | <> | + | - | * |/
unary_op ::=	-| NOT
number ::=	an integer
identifier ::=	an identifier (underscores allowed but no hyphens)

Note: LISTOF(*item, separator, enclosed-by*) is a shorthand to encode the information: a list of *item* is allowed, separated by *separator* and if it is non-empty, it is enclosed by *enclosed-by*. Parentheses refers to (), commas ",", semi-colons ";", and square-brackets "[]". When the *enclosed-by* is missing, it means it is not enclosed by anything.

Bibliography

[1] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley 1986.

[2] V. Ambriola and C. Montagero, "Automatic Generation of Execution Tools in a GANDALF Environment", *Journal of Systems and Software*, 5(2):155-171, May 1985.

[3] R. Bahlke and G. Snelting, "The PSG System: From Formal Language Definitions to Interactive Programming Environments", *ACM TOPLAS*, 8(4):547-576, October 1986.

[4] R. Bilos, "A Token-Based Syntax Sensitive Editor", *Linkoping University, Dept of Computer and Information Science, Sweden*, LiTH-IDA-R-87-02, February 1987.

[5] F. Budinsky, R. Holt and S. Zoky, "SRE – A Syntax Recognizing Editor", *Software Practice and Experience*, 1985.

[6] V. Donzeau-Gouge, G. Huet, G. Kahn, R. Lang, and J.J. Levy, "A structure-oriented program editor: A First Step Towards Computer Assisted Programming", *Tech. Rep. 114, INRIA, Recquencourt, France*, April 1975.

[7] V. Donzeau-Gouge, G. Huet, G. Kahn, and R. Lang, "Programming Environments Based on Structured Editors: The MENTOR Experience", *Tech. Rep. 26, INRIA, Rocquencourt, France*, May 1980.

[8] R.J. Ellison and B.J. Staudt, "The Evolution of the GANDALF System", *Journal of Systems and Software*, 5(2):107-119, May 1985.

[9] A.N. Habermann and D. Notkin, "Gandalf, Software Development Environments", *IEEE Transactions on Software Engineering*, 12(12):1117-1127, December 1986.

[10] W.J. Hansen, *Creation of Hierarchic Text with a Computer Display*, Ph.D. Dissertation, Dep. Comput. Sci., Stanford Univ., June 1971.

[11] S. Horwitz and T. Teitelbaum, "Generating Editing Environments Based on Relations and Attributes", *ACM TOPLAS*, 8(4):577-608, October 1986.

[12] K. Jensen and N. Wirth, *Pascal User Manual and Report 3rd Edition: ISO Pascal Standard*, Springer-Verlag, 1985.

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

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

[15] R. Milner, "A Theory of Polymorphism in Programming", *Journal of Computer and System Sciences*, 17(3):348-375, December 1978.

[16] T. Reps, T. Teitelbaum and A. Demers, "Incremental Context-Dependent Analysis for Language-based Editors", *ACM TOPLAS*, 5(3):449-477, July 1983.

[17] T. Reps, *Generating Language-based Environments*, MIT Press, Cambridge, MA, 1984.

[18] C. Rich and R.C. Waters, "Formalizing Reusable Software Components", *ITT Workshop on Reusability in Programming*, Newport RI, September 7-9, 1983.

[19] W.A. Spitzak, *A Display Generator for Structure Editors*, MIT B.S. Thesis, May 1983.

[20] R.M. Stallman, *EMACS: The Extensible, Customizable Self-Documenting Display Editor*, MIT/AIM-519a, March 1981.

[21] G.L. Steele, *Common Lisp*, Digital Equipment Corporation, 1984.

[22] P. Sterpe, *TEMPEST - A Template Editor for Structured Text*, MIT/AI/TR-843, June 1985.

[23] O. Stromfors, "Editing Large Programs Using a Structure-Oriented Text Editor", *Linkoping University, Dept of Computer and Information Science, Sweden*, LiTH-IDA-R-86-43, December 1986.

[24] T. Teitelbuam and T. Reps, "The Cornell Program Synthesizer: A Syntax-directed Programming Environment", *CACM*, 24(9):563-573, September 1981.

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

[26] R.C. Waters, *PP: A Lisp Pretty Printing System*, MIT/AIM-816, December 1984.

[27] R.C. Waters, *KBEmacs: A Step Toward the Programmer's Apprentice*, MIT/AI/TR-753, May 1985.

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

[29] L.M. Wills, *Automated Program Recognition*, MIT/AI/TR-904, February 1987.