CMU-CS-85-129

510.7808
C28r
85-129
c.2

Flexible Unparsing in a Structure Editing Environment

David Garlan

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213

April 1985

Abstract

Generators of structure editing-based programming environments require some form of *unparse specification language* with which an implementor can describe mappings between objects in the programming environment and concrete, visual representations of them. They must also provide an *unparser* to execute those mappings in a running programming environment. We describe one such unparse specification language, called VIZ, and its unparser, called UAL. VIZ combines in a uniform descriptive framework a variety of capabilities to describe flexible views of a programming database using a library of high-level formatting routines that can be customized and extended by the implementor. The UAL unparser allows the highly conditional unparse mappings of VIZ to be executed efficiently. Its implementation is based on the automatic generation of explicit *display views*, together with a scheme for efficient incremental updating of them in response to arbitrary changes to objects in the programming environment.

This research is supported in part by the Software Engineering Division of CENTACS/CORADCOM, Fort Monmouth, NJ.

Table of Contents

List of Figures

1. Introduction

Programming environments frequently take as their model a structured database that is manipulated by a set of cooperating tools. As the user modifies the environment by invoking the tools, the current state of the environment is displayed to the user by mapping the internal structure of the database to concrete text/graphics, a process called *unparsing* [Medina-Mora 82]. Systems that allow an implementor to define the structure of the database with a grammar description also provide a way for the implementor to specify the unparse mapping. This is usually done by attaching to each production of the grammar one or more *unparse schemes* written in some *unparse specification language*. This language allows the implementor to indicate the placement of keywords, punctuation, production subcomponents, and formatting directives necessary to produce a concrete realization of the production. Unparse schemes are then interpreted by an *unparser* in the running environment to produce an interactive display. Figure 1-1 shows an example of a production, its unparse scheme, and an unparsed display of it.

The quality of the interaction between a user and a programming environment largely depends on the ability of the unparser to support flexible, powerful, and insightful projections of the objects in the environment. For that reason, much attention has been given to the nature of concrete representations in providing ellipsis, graphical views, browsing windows, flexible line breaking, and so on. Unfortunately these features have found their way into unparsers largely on the basis of their creators' whims; there has been no attempt to analyze them as particular instances of a more general theory for unparsing and, on a practical level, little attempt has been made to integrate the wide range of mechanisms into a single system.

As a starting point for such a theory this author has proposed a set of principles for unparsing, and based on them, a general framework for unparse specification languages [Garlan 85]. In this paper we present an unparse specification language, VIZ, that is an instantiation of that general framework. We also describe the implementation of an unparser, UAL, that efficiently interprets VIZ specifications to support the unparsing activity of interactive structure editing-based programming environments. Section 2 discusses the architecture of the programming environment and introduces basic terminology. Sections 3 and 4 sketch the design issues that VIZ and UAL address. Section 5 summarizes the capabilities of VIZ and UAL. Section 6 presents the VIZ language. Sections 7 through 11 discuss the UAL unparser. Finally, in Section 12 we evaluate the current implementation with respect to the basic goals that have driven our approach. The conclusions reached there point to the need for a more general theory of *views* and *view transformations*, such as the one that is currently being developed in this author's Ph.D. thesis [Garlan 86].

```
IF-ELSE ::=  cond-part:expression then-part:statement else-part:statement

        "If <cond-part> || @+(Then <then-part> || Else <else-part>)"

        If  ....
            Then ....
            Else ....
```

Figure 1-1: Grammar Entry, Unparse Scheme, and Unparsing of an IF Node

2. System Architecture and Basic Terminology

In this section we sketch the underlying architecture that supports the programming environments for which the VIZ/UAL system was written. We do this in order to introduce terminology that we will be using in the remainder of this report, and because the object-oriented point of view taken here is different from that adopted in most other structure editing work.

We view a programming environment as a structured database of program objects and a collection of tools that manipulate them. Objects in the database include "source code" objects (variables, statements, procedures, etc.), "annotations" (comments, marks, error indications, etc.), and "execution" objects (call stack, tracing information, etc.). Tools include, most prominently, a syntax-directed editor, semantic analyzers, an execution machine, a documentation manager, and a help facility.

The database is object-oriented (in the sense of Smalltalk [Goldberg 83]). By this we mean that its objects are instances of classes, and classes are organized in a class hierarchy. Behavior is inherited down the hierarchy so that operations defined for one class can be used by all of its subclasses. A simplified version of the class hierarchy is illustrated in Figure 2-1. (See [Gnome 85] for more details about the object-oriented architecture.)

Because of the prominence of syntax-directed tools, such as the program editor and the semantic analyzer, the database is organized as an annotated abstract syntax tree. The structure of the syntax tree is determined by a grammar that defines legal operators[1] and syntactic classes. Sample operator and syntactic class descriptions are shown below in figure 2-2. We refer to the person who writes the grammar as the *implementor*.

[1]We will use the terms "operator" and "production in the grammar" synonymously.

```
                          TreeNode
                             |
         ----------------------------------------------------
         |                |               |                  |
    NonTerminal       MetaNode        Terminal           Attribute
         |                                |                  |
      -------                     ----------------       ----------
      |    |                      |      |        |      | ......|
     IF...WHILE                 IdUse  IdDef  StrConst
```

Figure 2-1: The Class Hierarchy

A *user* creates new database objects, or as they are usually called, *nodes*[2], in a syntax-directed way through a program editor that guarantees the syntactic integrity of the database. Unexpanded portions of a syntax tree are represented by a special kind of node called a *meta node*. The user thus creates programs by filling meta nodes with operators chosen from the appropriate syntactic class. (For a fuller treatment, see [Medina-Mora 82].)

Objects in the database are manipulated by the user through a collection of *display views*. Views determine the visual appearance of objects, and also translate the user's operations on visualized objects into abstract operations on the database. The views of an environment are determined in the grammar by *unparse specifications* which define a mapping between database objects and concrete forms. This mapping is carried out by an entity called the *unparser*.

```
IF          ::=  bool-part:expression  then-part:statement
BLOCK       ::=  statements:list of (statement)
IDENTIFIER  ::=  id:variable

statement   ::= IF WHILE PROC-CALL .....
```

Figure 2-2: Sample Operator Descriptions

The abstract syntax for three *operators* is shown here. IF is referred to as a *fixed-arity* node, since it has exactly two sons. BLOCK is referred to as a *list* node and expands to a list of statements. IDENTIFIER is referred to as a *terminal* or *leaf* node. It is one of the system-provided types, namely "variable". The names on the right hand side of the productions are of the form "named-son:syntactic-class". A *syntactic class* is a set of one or more operators, any one of which can be used as an expansion for that class. For example, as shown, the class "statement" would contain the operators IF, WHILE, etc.

[2]We will use the terms "database object" and "node" synonymously.

3. The View Problem

A single visual representation for the program database is an insufficient basis for a powerful programming environment. As a collection of tools that interact with the user and with each other, the power of a programming environment depends critically on the ability of the tools to provide appropriate and different *views* of the database. To take a simple example, for any but the smallest program, it must be possible to display the program contents at varying levels of detail.

The need to provide multiple views in a programming environment leads to two problems: (1) it must be possible for an implementor of an environment to describe the desired views, and (2) the unparser for these views must be efficient enough to support multiple concurrent displays of the same objects in different views and to propagate the changes made in one view to all concurrently active views.

There is both a global and a local component to this. *Globally*, it should be possible for two tools to present different representations of a program to a user. For example, a "code" view of a program may not display all of the program documentation, while a "documentation" view may not display all of the program code. Or, an "outline" view of a program may show only the nested hierarchy of procedure declarations in a program and not their bodies. *Locally*, it should be possible for the display of an object to vary depending on the context in which it is shown. For example, an IF node might be displayed on one line if there is space for it, but on multiple lines otherwise. It might appear in the form "If a < b Then ..." if the user has asked to elide it. It might be highlighted in a particular way if it has an associated semantic error. Or, it might appear in a separate window if the user so indicates.

The use of global views, was pioneered in the multiple unparse schemes of Aloe structure editors [Medina-Mora 82]. In that system the implementor can associate a collection of numbered unparse schemes with every operator in the grammar. The user or the internal tools can set the scheme number, thus determining the global view of the abstract tree. More recently, PECAN [Reiss 84], has extended this notion in the direction of multiple *graphic* views of a program. In addition to attaching multiple schemes to each operator, PECAN provides a notification mechanism through which the views in the system are informed whenever a significant change takes place to a node in the program. The views themselves are then responsible for keeping their display current.

The use of local views has appeared in many contexts and in response to a wide variety of concerns. A number of environments have addressed the problem of ellipsis of programs including Hansen's [Hansen 71] *holophrasting* and Mikelsons [Mikelsons 81] automatic, focus-dependent ellipsis of programs. Others, such as Oppen [Oppen 79] and Coutaz [Coutaz 85], have explored techniques for flexible line breaking. Still others have looked into the use of local window contexts for browsing [Delisle 84] or to enforce scope boundaries [Garlan 84].

None of these efforts, however, has developed an unparse specification language that allows an implementor to easily *describe* views to meet the varying display needs of tools. Unparse descriptions in virtually all existing unparse specification systems are composed either of low level control constructs or of high level black boxes. In the first case, the specification burdens the implementor with unnecessarily complicated and often baroque detail at the level of assembly language programming. In the second case the implementor has little control over the unparsing process and the black boxes cannot be composed to build new unparsing features. Moreover, previous unparsing systems have not attempted to integrate in a uniform way a wide range of view mechanisms from which the implementor can choose to implement particular policies. Indeed, given the apparent cost of implementing some of the mechanisms individually (such as ellipsis), it has been an open question whether it is currently possible to provide an implementation of such a general integration that is efficient enough to support an interactive environment.

4. Other Design Issues

While the goal of providing multiple display views has been a central focus in the design of the unparse specification language and the unparser described in this report, three related goals have been the ability to support customization and extension of view mechanisms, the ability to map a user's actions on visible objects to operations on corresponding database objects, and the ability to support text editing.

4.1. Customization and Extension

Global and local views are necessary for a flexible unparser. But they are not sufficient. The unparser must also make it possible for the implementor and, in some instances, for the user to extend and customize those views. For example, given current high resolution display devices, it should be possible for the user to choose font, indentation levels, style of presentation of keywords and other textual programming entities. It should be possible for the implementor to define special purpose formatting and stylistic environments to support error presentation, varying styles of ellipsis, and graphics.

4.2. The Inverse Map

An implementation of views must support both an inward and an outward component. On the one hand, it must allow the efficient display of program objects. On the other hand, it must allow operations on the visible display to be translated into corresponding operations on the objects in the program database. For example, when a user points to an item that is visible on the screen, the position of the item must be mapped back to a corresponding program entity. More importantly, commands that operate on the structure of the database must be interpreted relative to the view on which they were applied. For instance, the command NEXT-SIBLING will have different meanings in different views since the order in which objects appear to the user may not be the same order that they are represented in the abstract syntax tree. Some components may not, in fact, appear at all.

4.3. In-place Text Editing

Structure editing-based systems have taken widely different positions on the need to incorporate text editing as a component of the structure editor. But even the most staunch "structuralist" recognizes the need for some form of text editing of comments, identifiers, and other strictly textual entities. An unparser must therefore provide some basic mechanisms for the text editing of program entities. Ideally, that editing can occur "in-place", i.e., directly at the display site rather than in some other window. The problem is significant in the design for the unparser since the representation that the unparser uses to store unparse information may determine the way in which it can handle text editing. This is particularly true in the face of multiple and proportional width fonts, the need to maintain multiple concurrent views, and the possibility that the text itself may not reside in any readily editable form.

5. Capabilities of VIZ and UAL

VIZ is an unparse specification language and UAL is an implementation of an unparser that supports it. The VIZ/UAL system runs on Apple Macintosh computers and supports a family of structure editors called GENIE environments.[3] The basic features of the VIZ/UAL system are these:

- Multiple simultaneous <u>global views</u> of the program database.

- Different mechanisms to support a wide class of <u>local views</u> including:

  - <u>Ellipsis</u> of various kinds. VIZ/UAL mechanisms allow simple textual replacement of objects with, say, "...", or a comment (in the style of the Cornell Program Synthesizer [Teitelbaum 81]). They also allow node-specific ellipsis (e.g., an IF node may be elided differently than a WHILE node), and several global ellipsis policies.

[3]GENIE environments are descendents of the GNOME family of introductory programming environments. [Garlan 84, Gnome 85]

- o Association of window boundaries with arbitrary nodes. For example, the implementor can indicate that he wants to see nested procedure declarations in separate windows, and the user can place arbitrary subtrees in separate windows.

- o Space-conditional formatting. For example, a statement can be displayed on one line if it fits and on multiple lines otherwise, or a list of elements can be formatted in a table of columns that conforms to the size of the elements.

- o Context-dependent formatting. The same type of object can be represented in different ways in different contexts. For example, a comment can be displayed one way if it is the documentation for a procedure, and another way if it is annotating a statement.

- o Attribute-dependent formatting. A node can use the value of attributes to determine its appearance. For example, comments in a programming language can be implemented as attributes of the objects they annotate.

- A library of high-level formatting environments. The implementor can specify the appearance of programming entities using formatting directives in the style of Scribe [Reid 80]. Formatting environments can be used, for example, to display keywords in a programming language in a special font, to highlight errors with a special kind of highlighting, to achieve flexible line breaking, to present layouts that depend on the amount of space available on the screen, or to format a list as a table.

- Extensibility. The implementor can add new formatting environments to the library.

- Customization. The implementor can customize the system by specifying global parameters of style, and by modifying the meaning of system-defined formatting environments.

- Abstraction. The implementor can associate unparse descriptions of program objects with various levels of the object hierarchy (Section 2), so that unparse behavior can be specified in one place but be used by many types of objects.

- In-place text editing of any subtree of the syntax tree. This can be used in combination with an incremental parser to process arbitrary textual input from the user, or it can be restricted to strictly "textual" items such as comments and identifier names.

- Fast node selection with a pointing device, and support for mapping of both structural (e.g., NEXT-SIBLING) and textual commands (e.g., NEXT-LINE) to operations on corresponding program objects.

- A largely device independent representation of unparse information. A single representation of a program can be written to the screen, a printer, a file, or a text buffer, without significant regeneration costs.

6. VIZ, An Unparse Specification Language

VIZ is the name of the unparse specification language for GENIE programming environments. As part of the definition of a programming environment, VIZ descriptions specify the mappings between each object in the database and its concrete, or visual, representations. VIZ is an instance of a more general framework for unparse specification languages, as developed for this author's thesis [Garlan 86]. A detailed rationale for using this framework is given there. Here we present the details of the VIZ language itself. In later sections we describe how VIZ can be implemented efficiently.

6.1. A Bird's Eye View of VIZ

The general form of a VIZ specification is shown in Figure 6-1. Its key components are:

- A collection of views is associated with each operator in a grammar for a programming language.
- Each view of an operator consists of a set of unparse descriptors and optional scene information.
- An unparse descriptor is a condition-scheme pair. The condition part of the pair is a boolean expression and determines under what conditions the associated scheme will be used. The scheme part of the pair determines how the operator will be displayed.
- The scene information associated with a view of a node is used to designate a window boundary with an operator.
- Associated with each grammar is a single unparse declaration section. Unparse declarations are used to customize global unparse parameters, to modify existing formatting environments, and to extend the unparse formatting library with new formatting environments.

A VIZ specification is interpreted by an unparser as follows: The unparser is given a start node and a view. The condition part of each descriptor for the start node in the given view is evaluated in order. The first condition to succeed causes the corresponding scheme part of the unparse descriptor to be interpreted as the concrete representation of the node. This scheme specifies how the node is to be formatted. It consists of literal strings, subcomponent names, and formatting directives. When a subcomponent[4] appears in a scheme the unparser is invoked recursively on it using the same view. Thus the view names determine the *global* aspects of a presentation of the database, and the descriptors within a view determine the *local* aspects. The descriptors associated with a view behave like a small production system [Charniak 80] in which the conditions act as triggers for schemes that cause

[4]A subcomponent is either a son of a node in the database -- such as the boolean-condition part of an IF node -- or an attribute -- such as a comment or other annotation.

```
Unparse Declarations:

        <Customizations and Extensions>

___________

Node xxx = component1 component2 ...

        View <viewname>

            Scene Info: <scene information>

            Descriptors:
                         condition_1  → scheme_1
                         condition_2  → scheme_2
                             .......
                         TRUE         → scheme_n

        View <viewname>
            .......
```

Figure 6-1: The VIZ Framework

a node to be formatted for display.[5]

We now consider in more detail each of the components of VIZ.

6.2. Views

Each view in a VIZ specification determines a mapping from database objects (nodes, attributes, etc.) to concrete structures (strings, newlines, icons, etc.). The mapping determines the global "look" of a program database; the unparsing process is started in a particular view and that view remains in force throughout the display of the visible subtree.[6] This allows the implementor to group the various concrete representations of operators in a grammar along the lines of the tools with which they will be associated. A documentation tool, for example, can define a "documentation" view that specifies the representations of nodes appropriate for interaction with pieces of documentation text. The representations in this view might be quite different from those used by a tool that allowed the user to manipulate the code of a program.

[5] The resolution of conflict between conditions that are simultaneously true is resolved by selecting the first condition to appear in the view description.

[6] There is no inherent reason that the ability to change views during the unparsing should be disallowed, but in practice it doesn't seem to be needed.

An operator need not have a separate view definition for each view in the system. One view is always designated the *default view*. If no specification for a particular view is given the default view for that operator will be used. Moreover, multiple view names can be associated with the same set of unparse descriptors.

6.3. Scenes

Frequently it is desirable to associate window boundaries with operators in grammar. In a Pascal environment, for instance, it may be desirable to cause nested procedure declarations to be viewed in separate windows [Garlan 84]. This is illustrated below in figure 6-2. The collection of nodes that can be viewed contiguously on an output device is termed a *scene*. A view is therefore decomposable into a collection of overlapping scenes, each scene presenting some portion of the entire view. The starting node for a scene is called a *scene root*. Certain nodes in the scene may be designated *scene doors*. These act as "doorways" into other scenes in the sense that they may be "opened" or "entered" to produce a new scene. (Such would be the case for the "<body>" node in figure 6-2.)

```
Program ....

    Procedure a(....)
    <body>

    Function b(....): ...
    <body>

Begin ... End.
```

Figure 6-2: Procedure Scenes in Pascal
The bodies of nested procedure declarations are not displayed, but are represented by the string "<body>". When the user asks to expand a body (perhaps implicitly, by trying to visit one), the system provides a new window in which the nested procedure declaration is fully displayed.

In a VIZ specification, scenes are designated in the Scene Info portion of a node's view description. The specification of scene info is optional; in fact, most nodes will have no scene declarations at all. There are two kinds of scene declaration. The first designates an operator as a scene root for a given view. The second designates an operator as a scene door and specifies a (possibly new) view that should be used when the door is "entered". A typical scene declaration is shown below in figure 6-3. Here the operator **xxx** is both a scene root in view **aaa** and a scene door in view **bbb**.

The scene root/door mechanism works as follows: When the user "enters" a scene door[7], the

[7] The operation that the user invokes to "enter" a scene door or to "exit" a scene is considered a matter of user interface policy, and is not specified at this level.

```
Node xxx: ...

   View aaa
       Scene Info: Scene Root
       Descriptors: ....

   View bbb
       Scene Info: Scene Door -- View aaa
       Descriptors: ...
```

Figure 6-3: A Scene Declaration

unparser searches up the abstract syntax tree until it finds a node that has been designated a scene root for the new view associated with the scene door. The unparser opens a new window, unparsing the new scene in the new view. If the node is both a scene door and a scene root the "entered" node becomes the root of the new scene. In the above example, when the user causes the node xxx to be expanded in view bbb a new window will appear with the node xxx as the scene root and the scene will be unparsed in view aaa. Each scene stores the "previous" scene root and view so that on "exit" from a scene the previous scene can be restored.

The user may also explicitly designate an arbitrary node as a scene root. This will cause the subtree rooted at that node to be viewed in a separate window as if the implementor had designated it as a scene root in the grammar.

6.4. Conditions

The choice of unparse scheme within a view is controlled by the value of boolean expressions called *conditions*. A condition in a VIZ specification is composed of boolean connectives, '=', and conditional *terms*. A term is either a *qualified node* or a *perspective*. A qualified node is the node itself ("self"), its father, or a named son, and the qualifier can select either its value (such as the name of an identifier node) or its operator type (such as "IF"). A perspective is the name of a local unparsing context and will be described later (Section 6.6). The syntax for a conditional term is given in figure 6-4 and representative examples of conditions are shown in figure 6-5.

The choice of unparse scheme for a node thus depends on at most three things: the view in which it is being displayed, the perspectives that apply at that node, and *local* properties of the node. By "local properties" we mean the value, type, and attribute values of the node, its father or its sons (but not its brothers). A node's physical properties -- such as the amount of screen space it occupies

```
condition-term ::= node.qualifier | perspective
node           ::= "father" | named-son | "self" | ε
qualifier      ::= "value"  | "type"    | attribute-name
named-son      ::= identifier
attribute-name ::= identifier
perspective    ::= identifier
```

Figure 6-4: BNF for Condition Terms

-- are not used to determine the scheme with which a node will be unparsed.[8] As we will see, the fact that conditions don't name physical factors is important in making it possible for the unparser to perform efficient incremental updates. (Section 9).

```
father.type = "PROCDECL" AND self.value = "xxx"
self.err <> NIL
then-part.type = "BLOCK" OR bool-part.type = "META"
TRUE
```

Figure 6-5: Unparse Conditions

The first condition depends on the father's type and node's value; this could be used to have special unparsing for all procedures named "xxx". The second condition is true if the "err" attribute of a node is not NIL. The third condition depends on the operator of the "bool-part" and "then-part" sons of a node. The last condition is, of course, always true.

Since conditions are evaluated in the order in which they are specified, an implementor usually places the most specific conditions first. In fact, by convention the final condition in unparse schemes is the most general condition of all, namely "true".

6.5. Schemes

A *scheme* determines how a node will be displayed. In particular, it indicates the placement of keywords, punctuation, syntactic subcomponents, indentation, newlines, and so on. It also determines how the pieces of concrete syntax will be formatted on an output device. Figure 6-6 shows several unparse schemes for an IF node. The meaning of the first scheme, for example, is that the literal "If " is to be followed by the (recursive) unparsing of its boolean condition. A newline, indicated by '||', separates an indented "then" part.

More formally, a scheme consists of a list of unparse *phrases*. A phrase is a literal string, a syntactic

[8]As described in the next section, the formatting of a node *within a given scheme* can, however, depend on physical properties.

```
"If <bool-part> || @+(Then <then-part>)"
"@key(If ) <bool-part> || @+(@key(Then ) <then-part>)"
"@HV(If <bool-part> || @+(Then <then-part>))"
"@myhighlight(If <bool-part> || @+(Then <then-part>))"
```

Figure 6-6: Unparse Schemes for IF

In these unparse schemes for the IF operator, names enclosed in angle brackets are subcomponents. '||' indicates a newline. "key", "HV", "myhighlight", and " + " are formatting environments.

subcomponent, a nested phrase, or a newline.

- Literal strings consist of keywords and other syntactic "sugar". The "If " and "Then " strings in the schemes above are typical examples.

- A syntactic subcomponent is either a son name or an attribute name.[9] The "bool-part" and "then-part" are syntactic subcomponents of the IF node. In general, the subcomponents used in a scheme need not appear in the same order as they are described in the grammar for an operator. Some subcomponents may not appear at all, and others may occur more than once.[10]

- A nested phrase is of the form

  `@XXX(unparse-phrase).`

  The "XXX" is the name of either a *perspective* or else a *formatting environment*. Perspectives are described in the next section. A formatting environment determines the way strings and newlines will be interpreted on the output device. Formatting environments modify fonts, margin widths, indentation levels, etc., for the nested phrase. In the examples above " + ", "key", "myhighlight", and "HV" are formatting environments. The first increments the indentation level. The second changes the style in which keywords are displayed. The third is an implementor-defined environment for highlighting. The fourth formatting environment, "HV", is the "Horizontal-Vertical" environment. It will cause the IF statement to be displayed on one line if there is room in the target window and on multiple lines otherwise. Other system-defined environments include "flushleft", "outdent", "italic", "bold", "underline", "verbatim", "table". etc. Appendix II contains a list of the formatting environments that are available in VIZ.

- A newline is either *optional* or *required*, and is indicated by "||" or by "!!", respectively. Required newlines will always be displayed. Optional newlines, however, are used to guide the formatting environments. For example, the "HV" environment in the examples above will ignore optional newlines if by doing so the IF statement could be formatted on a single line.

[9] For list operators there is also a way to indicate "unparse all the sons in my list", and to specify a scheme that is used to separate the elements of the list.

[10] For example, in Pascal it is sometimes useful for a procedure name to appear with the closing "End" in the form: "Procedure xxx(...) Begin .... End; {xxx}".

One way of looking at schemes is that they provide an "interactive Scribe" for programming languages. Indeed, the choice of formatting environments, the definition and modification of these environments (Section 6.7), and even the phrase "formatting environment" has been borrowed directly from Scribe [Reid 80]. But VIZ is not simply an interactive document formatter. The use of multiple views, perspectives, and unparse conditions give it a conditional flavor that is not present in the current systems for displaying formatted documents (such as [Gutknecht 84] or [Lampson 78]). Using VIZ, programs are not viewed simply as static arrangements of text, but as dynamically changing, context-dependent, and history-sensitive visualizations of a program database.

6.6. Perspectives

While a view determines global context for display, *perspectives* are used to determine local context. Like views, perspectives are logical names defined by the implementor. But unlike views, they can be used in unparse conditions and they can be set or unset in unparse schemes. An example should help to clarify their use.

A typical perspective is "ellipsis". Nodes that fall within that perspective can be displayed in a variety of truncated forms. For example, an **IF** node might appear in the form "If a < b Then ..". Alternatively, it could be replaced by a user-supplied comment.[11] To use the ellipsis perspective the implementor would declare "ellipsis" to be a perspective in the unparse declarations (see Section 6.7). He could then set the perspective by using it in a scheme, such as:

```
..... → @ellipsis( .... )
```

A perspective applies throughout the unparsed subtree unless it is explicitly unset by a subcomponent. An implementor can use that perspective to choose an appropriate scheme for an operator by including the perspective in its unparse conditions. For example, an implementor would select schemes that should apply within the "ellipsis" perspective but not within the "error" perspective using the condition:

```
ellipsis AND NOT error → ...
```

Occasionally it is useful to set a perspective whenever a corresponding attribute value has a non-empty value. For example, an "error" perspective might be desired whenever a node has a value for its "error" attribute. This could be accomplished by having each operator check for the attribute in its condition parts. This would look like:

```
self.error <> NIL → @error( .... )
```

But this would have to be done for every operator in the grammar! VIZ allows the implementor to

[11] Other mechanisms to support ellipsis are discussed later.

achieve the same effect by designating a perspective as "automatic". The system guarantees that the "automatic" perspective will be set whenever an attribute of the same name is set.

A perspective can be explicitly turned off in a scheme. For example,

```
marked → @!ellipsis( .... ) .
```

uses the existence of a "marked" perspective to turn off the "ellipsis" perspective.

Perspectives resemble views in that both are used as logical names for partitioning schemes. Perspectives are unlike views in that they apply locally within a scene; while all nodes in the scene are in the same view, perspectives can be set and unset in local subtrees. Perspectives allow nodes to influence the behavior of the nodes in the subtree below them. But they do so in a non-binding kind of way. When a node sets a perspective it makes a certain condition known to its subcomponents; these subcomponents can either choose to make use of the perspective, or they can ignore it.

6.7. Unparse Declarations

The implementor customizes and extends the *unparse environment* using unparse declarations. There are four kinds of declaration: style, modify, define, and perspective declarations. *Style* declarations set the system's global default display parameters. *Modify* declarations are used to customize system-defined formatting environments. *Define* declarations declare new formatting environments. *Perspective* declarations make perspective names known to the system.

We have characterized formatting environments in terms of their behavior in controlling the display of text (Section 6.5). In fact, a formatting environment is simply a collection of attributes that determine the way text will be interpreted on an output device. Thus, a new formatting environment can be defined by associating a set of attribute-value pairs with a name. Similarly, an existing environment can be modified simply by changing the values of previously defined attributes.

```
Unparse Declarations:

    Style  -- font = Helvetica9, indent = 5 spaces, face = bold,
              spacing = 2 lines, wrap = on
    Define -- mytable: face = (underlined), columnize = true,
              spacing = 1 line
    Modify -- key: face = (underlined, italic)
    Perspectives -- ellipsis (auto), error (auto), marked
```

Figure 6-7: A Typical Unparse Declaration Section

There are two kinds of display attributes. *Unconditional attributes* determine the current state of the output device. VIZ recognizes the following unconditional attributes: font (e.g., Helvetica9), face (e.g., bold, italic, underlined), left-margin, right-margin, indent-increment, and spacing (between lines). *Conditional attributes* determine the way strings are formatted with respect to each other and with respect to the output device. VIZ supports the following conditional attributes: *horiz-vert* (format on one line if possible, otherwise on multiple lines), *wrap* (break long lines), *columnize* (format a list as a table), and *center* (position text in the center of a line), . Appendix II provides a complete list of the attributes that determine a formatting environment in VIZ. The distinction between conditional and unconditional attributes is, in fact, irrelevant to the implementor who only needs to know what attributes exist and what values they can take. The distinction is, however, of great importance to the implementation of formatting environments (discussed in Section 8), and so we briefly mention it here.

The use of unparse declarations is illustrated above. The *style* declaration will cause a program to be displayed using the Helvetica9 font family, with an indentation increment of 5 spaces, in bold face, double spaced, and wrapping long lines. The *define* declaration defines the "mytable" formatting environment. It will columnize a list, underline the elements, and use single spacing between lines. The *modify* declaration changes the system-defined "key" formatting environment so that it will use italics and underlining in displaying program keywords. Finally, the *perspective* declarations define "ellipsis", "error", and "marked" perspectives. The first two are marked automatic, meaning that the perspective will be set by the system automatically whenever a node has a non-nil value for an attribute of the same name.(See Section 6.6).

7. A Model of Unparsing

UAL is an implementation of an unparser that efficiently executes the mappings described in a VIZ specification. Before discussing this implementation we first present the model on which UAL is based.

Figure 7-1 represents a diagramatic view of the principal components of UAL. The model consists of three kinds of structures, and two active agents. The structures are:

1. the database, represented as an abstract syntax tree,

2. a collection of unparse trees, or *u-trees*, that represent current scenes,

3. a collection of output devices, such as screen windows, files, printers.

In order to produce a display the database is mapped into the current active scenes and the current scenes are mapped onto output devices. The active agents that cause these two mappings to occur are:

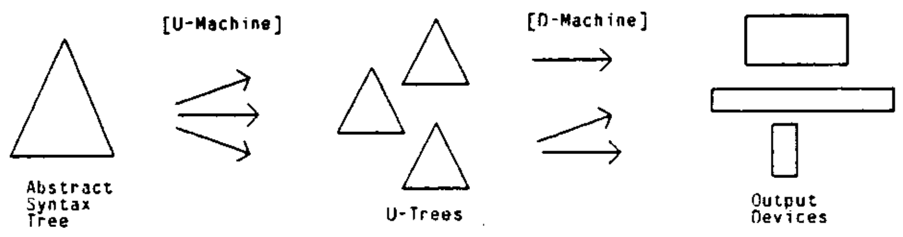

Figure 7-1: A Model of Unparsing
Views determine a mapping between the database (represented as an abstract syntax tree) and a collection of unparse trees [*u-trees*]. Each view may associate database objects with several u-trees and each u-tree can be mapped to zero, one, or many *output devices*. The mapping between abstract syntax tree and u-trees is carried out by an unparse machine [*U-machine*], and the mapping between u-trees and output devices is carried out by a display machine [*D-machine*].

1. the *U-machine*, or unparse machine, and
2. the *D-machine*, or display machine.

The U-machine translates objects in the program database into U-tokens that are used to build u-trees. The D-machine translates u-trees into D-tokens that are used to drive output devices.

The data structures representing a u-tree are described in the next section. For the time being, however, a u-tree can be thought of as a *concrete* syntax tree (as opposed to an *abstract* syntax tree). In other words, it contains in tree form all of the elements of concrete syntax needed to display the abstract syntax tree, including literal strings, newlines, and formatting information.

Thus the UAL model the process of unparsing consists of 6 stages.

1. <u>The database is changed.</u> A tool modifies the program database directly or a user modifies it through a displayed view of it.

2. <u>"Affected" nodes in the database are marked</u> Because of the dependencies between nodes introduced by the condition clauses of unparse descriptors, a change to one node can affect the scheme chosen by others. For example, consider a node that uses the descriptor "father.xxx = yyy → ...". If the father's "xxx" attribute changes, the son's unparse specification must be reevaluated. That is to say, a change to the father cause a change to the son.

3. <u>Changes are propagated to the u-trees by the U-machine.</u> Nodes that have been dirtied in the database (either directly or through the dependencies just mentioned) are reevaluated. The reevaluation consists of recalculating the scheme for each dirtied node and reinstantiating the new scheme as concrete syntax in the appropriate u-trees. This is done by the U-machine.

4. U-trees are resolved. Each modified u-tree is now traversed to resolve the space-dependent formatting of the conditional attributes of formatting environments (wrap, horiz-vert, center, columnize). This consists of annotating the u-tree with newlines and padding strings to satisfy the constraints imposed by limitations in physical space. During this stage certain kinds of size information are calculated. Size information is used (a) by the formatters in determining the "fit" of the u-tree, (b) by the D-machine to determine what part of the u-tree is will be used to generate D-tokens, (c) to map display coordinates to nodes in the u-tree, and (d) to highlight the user's current focus of attention.

5. Changes to the u-trees are propagated to virtual output devices. Output devices are *virtual* in the sense that several output devices may be mapped to the same physical device (say, appearing as several windows). Propagation to output devices is handled by the D-machine which calculates the correct portion of each u-tree to map to the output device and then generates D-tokens for it. D-tokens are strings, newlines, or tokens used to control the behavior of the output device (e.g., setting margin widths).

6. Actual output devices are updated. Display tokens must be interpreted for each active physical output device. This involves updating bitmaps, opening files, etc.

The important implementation issue in this paradigm is efficiency. In particular, changes must be propagated incrementally because it is too costly to regenerate the entire visible representation of the database for every view each time the database changes. Also, a user's operations on the visible representation of the database must be mapped to operations on the database itself. We will now describe how this is done. First we look more closely at the structure of the u-tree as it is the pivotal structure in the model. Next we look at each of the six stages and show how the propagations and resolutions can be done efficiently.

8. The U-tree

A u-tree is a concrete instantiation of the schemes associated with a scene[12] in some view of the database. It is a tree composed of unparse nodes, or *u-nodes*, that can be interpreted on demand to produce displayable tokens.

To illustrate the structure of a u-tree consider the following scheme for an IF node

```
... → "@key(If ) <bool-part> || @key(Then ) <then-part>"
```

The corresponding abstract syntax tree and u-tree are shown below in figure 8-1.

There are six types of u-nodes: scene, ref, environment, formatter, string, and newline. All u-nodes have father, left-brother, and right-brother fields for linking into the u-tree. Additionally each u-node

[12] Recall that a scene is the portion of the database that can be viewed contiguously in one window, starting at a given node (the *scene root*) in a given view.

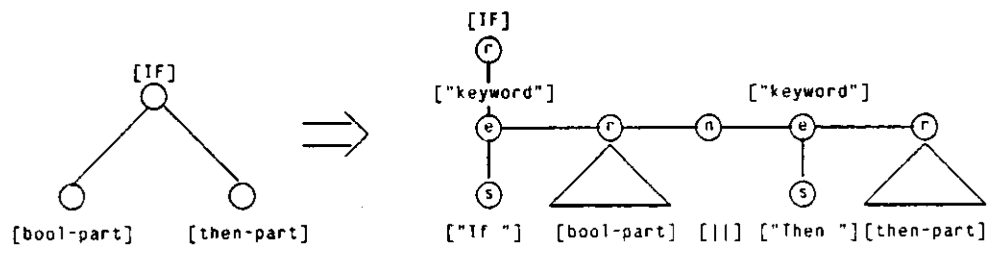

Figure 8-1: U-tree for an IF node

has a status field used for bookkeeping.

Each u-tree has *scene* u-node as its topmost node. Scene u-nodes link together all active scenes of the database. They also contain status information such as the user's focus of attention (described later), whether the u-tree has been modified since the last redisplay, and so on. *Ref* u-nodes point back to a corresponding object in the database. In the figure below, the top u-node is a ref u-node; it points back to the corresponding **IF** node in the abstract syntax tree. The u-nodes for the **bool-part** and **then-part** are similarly ref u-nodes. *String* u-nodes point to a literal string. In the example, string literals "If " and "Then " are associated with string nodes.

U-nodes that represent *newlines* are of three types: optional, required, and annotated. The first two types correspond to the use of optional and required newlines in unparse schemes (see Section 6.5). Optional newlines can be turned either "on" or "off" during the resolution of conditional formatting environments. Required newlines are always displayed. Annotated newlines are added to the u-tree by routines associated with conditional formatting. Setting the "wrap" attribute, for example, will cause newlines to be placed in lines that would otherwise extend past the right margin of the output device.

Environment and *formatter* u-nodes encode information associated with formatting environments. Recall that a formatting environment consists of a collection of attribute-value pairs and that attributes are either conditional or unconditional (Section 6.7). At the level of the u-tree the treatment of these two kinds of attributes is quite different. An unconditional attribute primarily affects the state of the output device itself. A conditional attribute, however, affects the relationship between strings and newlines in the u-tree. For example, setting the unconditional font face attribute to "italic" requires that the current font for the output device be set during the process of generating D-tokens. Setting the conditional attribute "wrap" to "on", however, requires that a function be called to modify the placement of newlines in the u-tree *before* any D-tokens are generated. Consequently, in the u-tree the setting of a conditional attribute is represented by a *formatter* u-node and the setting of a collection of unconditional attributes is represented by an *environment* u-node. The former is associated with a function that is called to resolve the space-conditional formatting of the u-tree. The latter contains a pointer to a *display vector* that contains the current values of the unconditional display attributes.

To see how the u-tree behaves during unparsing it is useful to consider each u-node in a u-tree as progressing through three stages: creation, resolution, and interpretation. These correspond to stages 3, 4, and 5 of the 6-stage unparse process outlined above. In the *creation* stage the U-machine generates tokens that cause a u-node to be allocated, linked into a u-tree and given initial values that will depend on the kind of u-node. In the *resolution* stage functions associated with conditional formatters annotate the u-tree with newlines, and sizes of nodes are generated. In the *interpretation* stage u-nodes are interpreted to produce D-tokens that are used to display strings or control the behavior of display devices.

During these stages the behavior of each type of u-node is as follows:

- root: A root serves primarily to link together all active scenes and has no function during resolution and interpretation stages.

- ref: A ref node's primary purpose is to point back to an object in the database, and to store size information. Each object in the database that appears in a scene will have a corresponding ref node for each appearance in a scene. In this way an operation performed on a u-node can be mapped back to an operation on the corresponding node in the database. The size associated with the ref u-node is used to map display coordinates to nodes in the database and is also used to know the region of display that must be highlighted when the user's cursor moves to that node. We call this size the "actual size" of the node and calculate it as part of the resolution phase. We also store a "desired size" at the node that represents the size the node would occupy in the absence of optional line breaks. Desired size can be calculated when the u-tree is created since it depends only on the existence of required newlines and the display size of strings in a subtree; it is used by the formatters during resolution (see Section 10.1). During the interpretation phase ref nodes do not require any special activity on the part of the D-machine since they do not themselves generate any concrete syntax, and the size information associated with them has already been calculated in the resolution phase. However, the subtree rooted at a ref node will, in general, contain nodes that generate D-tokens, and so this subtree is traversed by the D-machine.

- environment: On creation we associate a display vector with each environment u-node. During interpretation the D-machine uses that display vector to reset a target output device with new parameters. The node's display vector is derived from the display vector stored at the closest enclosing environment u-node. For example, if an unparse scheme uses a formatting environment that sets line spacing to 2 lines, the "spacing" value of the new display vector will have the "spacing" field set to 2. Below the root u-node for a scene is an environment u-node that establishes the default display characteristics. Its display vector is determined by system-defined default values that can be modified by the implementor in his "style" declarations (Section 6.7).

- formatter: Formatter u-nodes are responsible for implementing conditional attributes. There is one formatter u-node for each setting of a conditional attribute caused by the invocation of a formatting environment. During the resolution phase routines associated with formatter u-nodes massage the u-tree below them by annotating them with newlines and padding strings or by disabling existing newlines. The algorithms that the formatters use are sketched later (Section 10). Formatter u-nodes have no function during the interpretation phase.

- string: String u-nodes contain a string "display size" and a pointer to the text that they represent. The display size of the string will depend on the enclosing environment since this environment determines the font size for strings below it. This size is calculated during the creation phase. In the resolution phase string u-nodes have no function. During the interpretation phase a string emits a "string" D-token.

- newline: Optional and required newlines are inserted in the u-tree as schemes are instantiated. Subsequently, in the resolution stage some optional newlines will be turned "off", and annotated newlines may be added to the u-tree. During the interpretation phase newlines produce a "newline" D-token.

In the current implementation there is at least one u-tree for each "active" scene. Scenes are considered to be active until they are explicitly deactivated. This usually occurs if a window is removed, a tool is finished unparsing to file, or a printer finishes printing a scene. The same scene may be represented by more than one u-tree if it appears in different windows. In general the same u-tree cannot be used for two virtual output devices (such as two windows on the screen) since the resolution of conditional formatting environments may depend on the width of the device. However, in some situations, such as printing the contents of a window on the printer, it may be desirable to use the same u-tree for multiple devices. UAL allows this.

Viewed as an abstract data type, the important properties of a u-tree are these:

- Line-oriented textual operations. These include: "move back k lines from a given u-node"; "generate n lines of text"; "calculate the number of lines between two nodes". These operations are possible because newlines are explicitly represented in a u-tree. The u-tree does not, however, support textual commands at the character level -- such as "insert character". (See Section 10.2.)

- Structure-oriented tree operations. A user's commands NEXT-SIBLING, or GOTO-FATHER, must be interpreted relative to the representation given in the u-tree since unparse schemes can reorder, repeat, and eliminate nodes that appear in the abstract syntax tree. This is done by ignoring all but the ref u-nodes in the u-tree.

- Screen-to-node mapping. When the user points to a position on the screen it is necessary to determine the appropriate selected object. The u-tree makes this possible because node display sizes are kept explicitly as fields of the ref nodes.

- Incremental updating. As we will see, random changes to the database can be interpreted as replacement and reevaluation of u-node subtrees; the other information in the u-tree can be used without regeneration. In other words, it serves as a cache of display information.

9. Incremental Updating in UAL

We presented the model on which the UAL implementation is based and we decomposed the process of unparsing into six stages. Then we described the key data structure in the model, the u-tree. Now we return to the six stages and show how the u-tree is used to implement incremental updating of the display.

9.1. Stage 1: Changes to the Database

As we have pointed out, a user's operations on a database must be interpreted relative to the u-tree representation. A user's operation can be decomposed into the steps:

- The user applies a command to a current focus of attention.

- The user's focus of attention is translated into an object or objects in the database.

- The current command is translated into a sequence of database operations.

The user's current focus of attention is indicated externally by highlighting. Internally a separate *focus* is maintained for each u-tree.[13] A focus is either a single ref node or a pair of ref nodes. In the first case, the ref node points back to a node in the abstract syntax tree, or an attribute of a node. In the second case the two ref nodes refer to elements of a list in the abstract syntax tree, and represent a selection of the nodes in a list between the two nodes (inclusive).

The choice of legal operations available to the user will, in general, depend on the current focus. For example, at a meta node for representing a statement, the user has the option of constructing a **WHILE**, *IF*, etc., but not a variable declaration. The currently legal operations are primarily determined by consulting grammar tables for the language being generated.[14] The way in which the commands are presented to the user and the way in which the user indicates the choice of command is not

[13]A *focus* is often called the *cursor* in other structure editors. Some u-trees, such as a u-tree for printer, may not have a focus.

[14]It would be possible to present the legal operations as a view of the grammar entry for the node that represents the current focus (see, e.g., [Feiler 83]), but we have not done this in the current implementation.

discussed here. (See [Gnome 85] for more details.)

The translation of operations on the u-tree to operations on the database is straightforward. Typically the operation is either one that can be executed by the u-tree directly (such as scroll, redisplay, text edit), or else it can simply be passed on to the abstract syntax tree.

Operations that change the database, such as deleting or inserting a node, cause immediate propagations to all u-trees in which the changed node(s) have some representation.[15] To make this updating efficient each node in the syntax tree has a "scene chain". This is a list of pointers threaded through corresponding ref u-nodes in the u-trees. The U-machine walks this list updating the occurrences of the corresponding u-nodes in a way to be explained presently.

9.2. Stage 2: Propagation within the Database

Direct changes to a node or attribute in the syntax tree may have consequent effects on other nodes whose unparse schemes depend on the changed node. In this stage we handle these side effects. In general, such propagations can be handled by an attribute propagation mechanism such as the one proposed by Kaiser [Kaiser 85]. In VIZ, however, the dependencies between nodes are sufficiently constrained that a more elementary mechanism can be used. Recall that the choice of unparse scheme for a node can only depend on the value or type of the node itself, its father, or its sons. It is sufficient, therefore, to reevaluate each of the immediate neighbors of a changed node. Thus, when the user changes a node the U-machine is called to update the u-nodes corresponding to the father[16] and each son of the changed node.

9.3. Stage 3: Propagation to the u-tree

During this stage changes to the database must cause corresponding changes to the u-trees. For the purposes of unparsing, any operation on a program database is equivalent to a sequence of three elementary operations:

1. Delete: a node is deleted from the database.

2. Insert: a meta node is inserted into the database.

3. Replace: one subtree is replaced by another subtree.

The first and second operations can only occur in a list or attribute list, since a son of a fixed-arity node can not be removed from or added to a node in the database. The third operation covers a wide

[15] The changes will not, however, be visible to the user until the display devices are updated.

[16] The "father" of an attribute is taken to be its "owner".

class of events including: replacing a meta node by an operator, transforming some portion of a tree into another tree, and replacing a subtree by a meta node. The operations on the u-tree for each of these actions consists of a corresponding deletion, insertion, or replacement of u-nodes.

As we have already discussed, unparse scheme-related dependencies between nodes in the abstract syntax tree may require the reevaluation of unparse scheme for neighboring nodes. One way to accomplish this would be to treat each of the secondary effects as an instance of the "replace" operation, replacing a node by itself. But this would be extremely expensive. Changing an attribute of the scene root, for example, would require a complete recalculation of the entire view. Instead, we introduce a fourth operation on a u-node, "Reevaluate". The effect of the operation is to reevaluate the u-node subtree as incrementally and minimally as possible. We now explain how this is done.

Recall that the choice of scheme for a node depends on exactly three things:

1. The current view.
2. The structural context of the abstract syntax (namely, a node's value or type of itself, its attributes, its father, and its sons).
3. Inherited perspectives.

To facilitate incremental reevaluation of a node's scheme, a ref node stores the view and scheme that were used to produce it and set of perspectives that were in force when its scheme was calculated. This information is used to determine if the changes to the database result in a new unparse scheme for a node. We reevaluate the node's unparse conditions using the previous view and perspective set. If the resulting unparse scheme is the same as it was before the changes to the database, we can ignore the changes. Otherwise, we replace the corresponding u-subtree with the new version of the scheme.

9.4. Stage 4: Resolution of the u-tree

At the end of the previous stage, schemes have been transformed into corresponding u-nodes in the u-trees; from this point on we can ignore the abstract syntax tree and the unparse schemes altogether. Within the u-tree, however, there is still work to be done to resolve conditional formatting and recalculate node sizes.

As nodes are modified in the previous stage, we mark them as "new". When a node is marked we also mark its ancestors as "dirty". We now use these marks to "resolve" the u-tree in a walk of the "dirty" portion. Starting at the root we walk all "dirty" paths applying formatters and recalculating "actual" sizes. The action of a formatter depends on the kind of formatter involved and is sketched later in Section 10.1. Incrementality at this stage is gained by only having to reprocess the "dirty" paths of the u-tree.

9.5. Stage 5: Propagation to the Display

During this phase u-nodes are interpreted by the D-machine to produce D-tokens suitable for display on an output device. It is important to note that we need not update all output devices continuously. While the U-machine keeps all "active" scenes current, it is likely that at any given time only a subset of the output devices will need to be kept completely up-to-date. Thus D-tokens are typically generated for only some of the scenes.

We have already described the action of each type of u-node during "interpretation". What remains to be discussed is how the D-machine calculates which u-nodes to interpret for a given device. Typically this is done in terms of lines and the current focus. Recall that a u-tree stores a "focus" which represents the users focus of attention. In addition, with each output device we store the first and last u-nodes currently displayed. The output device can thus make the following kinds of requests of the D-machine:

- generate display tokens for k lines above the top u-node currently displayed.
- generate display tokens for k lines following the bottom u-node currently displayed.
- generate display tokens for k lines starting n lines above the current focus.
- generate display tokens for k lines starting at the scene root.

9.6. Stage 6: Updating Output Devices

There are three kinds of D-tokens: strings, newlines, and parameter-settings. It is up to the UAL display module to interpret these appropriately for the intended output device. To accomplish this the display module consists of a collection of high level device drivers. D-tokens are multiplexed to the correct driver(s) as they are received. Each device maintains a vector of parameters which completely characterizes the current state of the output device. These parameters correspond to the unconditional attributes of formatting environments discussed earlier (namely, font, margins, indentation, etc.). They are set and reset using "parameter-setting" D-tokens. "String" and "newline" tokens are then interpreted relative to the current parameter settings of the output device.

10. Further Details

In this section we tie up a collection of loose ends. First, we describe how the system formatters work. Second, we describe how UAL supports in-place editing. Third, we briefly indicate how various policies of ellipsis can be implemented with the mechanisms in UAL. Fourth, we consider the issue of device independence. Finally, we discuss the algorithm used to map from display coordinates to objects in the database.

10.1. Formatters

Formatters are responsible for resolving the space-conditional aspects of unparsing. As we described (Section 6.7), VIZ/UAL currently supports four conditional attributes: horiz-vert, wrap, columnize, and center. Each of these has an associated formatting routine. The "horiz-vert" formatter adds up the desired sizes of each node in its subtree. If that sum is greater than the space left on the line it does nothing. Otherwise, it traverses the subtree marking newlines as inactive. The "wrap" formatter traverses the subtree annotating it with newlines whenever sizes exceed the available line width. The "columnize" formatter is restricted to lists of terminal nodes. It scans the list to find the maximum element size. Then it annotates the list with blank padding strings so that items line up in multiple columns the size of the maximum width. The "center" formatter acts like the "wrap" formatter except that it annotates the u-tree with blank padding strings so that text is positioned in the center of the line.

We are planning to add a number of other formatters to the system. A smarter "columnize" will be able to format using variable width columns. We are also looking into using a "matrix" formatter which can format a list of lists of elements. Finally there are a number of graphic formatters that will be added including a "box" formatter for drawing boxes around subtrees, and various "line" formatters for drawing lines.

10.2. Text Editing

From the user's point of view, text editing occurs as follows: The user selects a subtree to edit. He invokes the "edit" command. The editable region is redrawn using a distinctive font, indicating the boundaries of the editable region. The user then edits that region. The system complains if an attempt is made to edit anything outside this region, although the user is allowed to browse through the entire scene. When he is finished editing the user gives a "done-editing" command. The editable region is then redrawn using the standard method of displaying schemes.

To implement this, UAL uses an auxiliary buffer structure. As we have pointed out, the u-tree supports line-oriented operations, but not the usual character-oriented operations required by text editors (such as to insert or to delete character). Moreover, the presence of multiple and proportional width fonts makes it difficult to add this capability directly to the u-tree mechanisms. Thus we do the following: when the user invokes the text editor on a subtree UAL unparses the entire scene into a text buffer. The text buffer supports multiple fonts. However, the portion of the u-tree that is to be edited is unparsed using a special, non-proportional width font. This delimits the editable region visually from the surrounding text and also makes incremental screen updating easier.[17] Two markers are kept in the text buffer to indicate the editable boundaries. Changes to the buffer are updated directly by the buffer mechanism; they do not involve UAL or the u-tree at all. Long lines can be wrapped at this stage, but otherwise no automatic formatting is done. When editing is finished, the editable region is given to an incremental parser[18] which transforms the text into a new abstract syntax subtree. This subtree replaces the previous one in the syntax tree, and UAL is then called to reunparse it onto the screen.

10.3. Ellipsis

UAL supports three general types of ellipsis: "windowing", node-specific ellipsis, and global ellipsis. We have already described how windows can be attached to nodes to produce scenes (Section 6.3). This is a form of ellipsis since it hides details of nested subtrees by placing them in another window. The second form of ellipsis depends on the use of an "ellipsis" perspective. As discussed earlier, the implementor can indicate in a node-specific way how each node is to behave in the presence of ellipsis using the condition "ellipsis → ...." (Section 6.6). The third form of ellipsis can be achieved in conjunction with the setting of attributes. Since the scheme in which a node is unparsed can depend on the value of an attribute, a variety of global policies of ellipsis based on properties of the abstract syntax tree can be implemented by assigning an ellipsis "weight" to each node. Weights can be assigned on the basis of a variety of policies including those derived from the distance from current focus of attention and from flow analysis.[19] Nodes can then use the value of their ellipsis weight to determine their formatting as shown in the example below. UAL does not provide the style of ellipsis that automatically elides a program based on sizes stored in the u-tree (as in Mikelsons [Mikelsons 81], for example), although it would be possible to do this using an "ellipsis" formatting environment.

[17] But there is no inherent reason why multiple font and multi-font editing could not take place using this buffer mechanism.

[18] This assumes, of course, that the implementor is willing to invest effort in building or generating an incremental parser for the language of the environment. If not, the implementor can restrict the text editing to identifiers, comments, and other string nodes.

[19] It may be desirable, for example, to show all nodes on the path from the root to the current focus of attention that could influence the flow of control in an executing program.

```
ellipsis <  5 → "If <cond-part> Then <then-part>"
ellipsis < 10 → "If <cond-part> Then ..."
ellipsis < 15 → "If ... Then ..."
TRUE          → ""
```

Figure 10-1: Variable Ellipsis for an IF Node

10.4. Device Independence

It is decidedly *not* the case that all u-trees are device independent. The formatting of schemes that use conditional formatting environments will usually depend on the width of the target output device. On the other hand, once the formatting information in a u-tree has been resolved, the tokens generated by the D-machine *are* device independent. This allows the display module to multiplex the same tokens across a variety of output devices, provided they do not depend on the width of the lines produced. For example, a u-tree that is to be displayed in one window can also be written to a file, a printer, or a text buffer without change. Moreover, even in the worst case, an existing u-tree can be retargeted for a new device by reresolving the u-tree; it is not necessary to regenerate the u-tree from scratch.

10.5. Backmap

To support a pointing device it is necessary to be able to map screen coordinates to an object in the abstract syntax tree. To do this UAL uses the sizes of nodes stored at the ref u-nodes, and the fact that each output device stores the top u-node that is currently displayed. The algorithm attempts to match display coordinates with a "closest" ref node in the u-tree. It goes roughly as follows: given coordinates (x,y) and a starting u-node, we walk the list of ref sons below it. If a son's "size" contains the (x,y) pair we continue the process recursively on that son. The recursion stops when (x,y) is found to lie (a) within the display of a terminal node (b) between two sons, (c) before the list of sons, or (d) after the list of sons. In case (a) the terminal node is the desired node. In case (b) the desired node is either the father or the left son, depending on whether the father is a fixed-arity node or a list node in the abstract syntax tree.[20] In case (c) or (d) the desired node is the father.

[20] For example, if the user selects a semicolon separating two statements, the algorithm will select the statement preceding the semicolon. However, if the user selects the '+' separating an expression such as "a + b", the algorithm selects the "plus" node.

11. Efficiency Considerations

UAL has been designed with intent of allowing efficient and random updates to unparsing information in the presence of multiple views and highly conditional unparsing. There are, however, three aspects of UAL that may lead to problems with efficiency.

The first concerns the problem of start up. In the current implementation the u-tree for the entire scene is generated as soon as that scene becomes active (appears in a window on the screen, for instance). If the scene is large (textually speaking), then the time to create the u-tree can be significant, although at worst, it is proportional to the number of nodes in the scene.

The second concerns the problem of u-tree deletion. The cost of deleting a u-tree is proportional to the number of u-nodes in the u-tree, since we must free each node and for each ref node unlink it from its "scene chain" (Section 9.1). Again, this cost can be significant if the u-tree is particularly large.

The third concerns the problem of space utilization. Since all active views are maintained concurrently (whether or not their display is continuously updated), and since each view contains many more nodes than the portion of the abstract syntax tree that it represents, a large amount of space may be required for the u-trees.

We have partial solutions to each of these problems. The first problem can be ameliorated in two ways. First, we can make the instantiation of new u-nodes relatively fast. To do this we compile the unparse schemes for each operator into blocks of pre-initialized u-nodes. When an operator is to be instantiated in the u-tree we insert the block of u-nodes as a unit, "relocating" internal pointers to point to absolute u-node addresses. Second, we can make use of the notion of "pseudo-scenes" to limit the amount of u-tree that we have to build at one time. A pseudo-scene is an operator in the grammar that acts like a scene root, except that no window is attached to it. In particular, we assume that there are no formatting interactions between pseudo-scenes so that a pseudo-scene can be unparsed in isolation from all others. Thus we can unparse a particularly large portion of the database in blocks of pseudo-scenes as they are needed.

The second problem, that of u-tree deletion, is alleviated in the same way. Using blocks of u-nodes, the cost of deleting a u-tree is proportional to the number of *ref* nodes.[21] And, if pseudo scenes are used to limit the size of a u-tree, the cost of deleting it will decrease proportionally.

[21] The number of ref u-nodes will, in general, be 3-4 times less than the total number of u-nodes.

A solution to the third problem, that of space utilization, rests on a solution to the two previous problems. Rather than incrementally maintaining all active scenes, whether or not they are actually being updated on the screen, the U-machine need only maintain all scenes that are currently needed by the display manager. Other scenes can be freed and regenerated as they are needed. Of course, this is an acceptable solution only if the cost of creating and deleting a u-tree is not high.

12. The View Problem Revisited

We return now to the central issue that has motivated the development of VIZ and UAL, namely, the view problem. It seems fair to ask, at this point, just how well have we solved the basic problem of providing multiple, powerful, flexible, and efficient views to support the collection of tools that compose a programming environment.

VIZ and UAL are currently being used to build a variety of programming language environments. One of these is a novice learning environment for Pascal [Gnome 85]. In this environment UAL supports a number of global views including several "code" views, an "outline" view, and a variety of "execution" views. In the code views, the user can interact with a program using scoped scenes (see Section 6.3), or as a single textual entity. In the outline view the user is shown a hierarchical outline of the procedure declaration structure. In the execution views the user can observe dynamically changing values of monitored variables, the call stack, or the value of arbitrary arithmetic expressions. At the local view level, the views take advantage of all of VIZ's capabilities for supporting multiple fonts, styles, highlighting, ellipsis, windowing, space-efficient unparsing, in-place editing, etc. The user has considerable control over the appearance of his programs and can dynamically change many of stylistic parameters available to the implementor.

Given the impressive combination of capabilities, can it be said that VIZ solves the view problem for these programming environments? The answer, we believe, is "No". Despite its flexibility, power, and efficiency, there are some fundamental shortcomings in this approach, and indeed, in all approaches to views taken to date, including Aloe [Medina-Mora 82] and PECAN [Reiss 84] environments.

The shortcomings of current approaches to views arise from two assumptions:

- views exist to provide *multiple visual representations for the user* of a programming environment, and
- the database is organized as an *abstract syntax tree.*

The first assumption has led to a situation in which tools in a programming environment can present multiple abstractions of the database to the user, but cannot partake of these abstractions directly themselves. Every tool is thus required either to use the abstract syntax tree as it stands, or to build and maintain specialized data structures outside the database for its own needs. The symbol table used by a semantic tool for type checking illustrates the problem. A symbol table can be thought of as a "semantic" view of the database. The information needed in a symbol table is already in the abstract syntax tree, but it is not in a suitable "tabular" form. It should be possible for a tool to define a "table" view of the database that would let the tool manipulate objects as if they were part of a symbol table. If a semantic tool cannot define such a view it has to use the tree itself as a symbol table or else copy duplicate information into its own special structures.[22] Neither solution is ideal.

The second assumption puts severe limits in the ability to provide flexible projections of the database. In particular, it is difficult or impossible for a mapping directly from the syntax tree alone to produce:

- views that collect all nodes satisfying some property,
- views that use non-hierarchical data representations, such as graphs, sets, hash tables, etc.,
- views that *are* hierarchical but have significantly different structure from the abstract syntax tree.

The solution to these problems rests on a generalization of the notion of views and is being developed as this author's thesis work [Garlan 86]. Views are defined there as abstract mappings from an unstructured database of objects onto a set of typed structures. One of these mappings produces the abstract syntax tree itself. But there are many others, constrained only by the requirements of consistency and reasonable efficiency. In particular, tools can define abstract views based on a generalized set of structures including graphs, sets, arrays, and other views. View mechanisms support multiple abstractions of the database, provide a language for describing these views and sharing between views, and yield an efficient implementation that guarantees view consistency. They are already being used to support research in providing semantic tools for structure editing environments [Kaiser 85].

The model of views underlying VIZ is thus a special case of this more general notion. In particular, u-trees supported by VIZ are an explicit representation of a special kind of view called a *display view*, and the unparse descriptions of VIZ define a composite mapping from the special case of an abstract syntax tree view to the special case of a display view. In the more general setting, however, display

[22] Some editor systems will allow the implementor to define a symbol table with a separate grammar and manipulate it as a syntax tree [Ambriola 84], but this does not solve the problem of duplicated information and maintenance effort.

views can be composed with *any* abstract view, not just the abstract syntax tree view. This provides a more powerful basis for a user interface as well as yielding a uniform framework in which the benefits of views can be reaped both by the tools and by the user.

13. Conclusions

We have described an unparse specification language, VIZ, and an implementation of an unparser, UAL. VIZ provides a uniform descriptive framework for defining views that can take advantage of a powerful and extensible set of facilities for flexible unparsing. UAL provides an efficient implementation that supports multiple concurrent views that can be updated incrementally in response to arbitrary changes in the programming environment. While an important measure of the value of this system will be the success of the programming environments that are now being built using it, early experience with prototype environments has shown that from an implementor's perspective, it represents a quantum jump in improvement over exiting unparse mechanisms. On a more fundamental level, however, we believe that the approach taken here represents an important initial step in providing a general framework of views for programming environments.

Acknowledgements

The current version of this report has benefited greatly from suggestions made by numerous people. In particular, we thank Nico Habermann, Rob Chandhok, Gail Kaiser, Charles Krueger, Barbara Staudt, Richard Cohn, Pedro Szekely, and especially Vincenzo Ambriola and David Notkin for their critical reading of earlier versions.

I. The Grammar for VIZ

The description for VIZ uses the following meta syntax:

boldface	indicates syntactic sugar.
italics	indicates a nonterminal symbol of the grammar.
{ }	indicates a list of one or more items.
[ ]	indicates an optional symbol inside a production.
|	separates choices.

Views

view-spec	::=	{*view*}
view	::=	[**Default**] **View** *view-names*
		[**Scene Info:** *scene-info*]
		Descriptors: *descriptors*
view-names	::=	{*view-name* , }

Scenes

scene-info	::=	[*scene-root*] [*scene-door*]
scene-root	::=	**Scene root**
scene-door	::=	**Scene door** [-- **View** *view-name*]

Descriptors

descriptors	::=	{*cond-scheme-pair*}
cond-scheme-pair	::=	*condition* → *scheme*

Unparse Conditions

condition	::=	*condition-term* |
		(*condition-term*) |
		condition-term bool-op condition-term
bool-op	::=	**AND** | **OR** | **NOT** | **=**
condition-term	::=	[*node*] . *qualifier* | *perspective*
node	::=	**father** | *named-son* | **self**
qualifier	::=	**value** | **type** | *attribute-name*

Unparse Schemes

scheme	::=	" {*phrase* } "
phrase	::=	*string* | *subcomponent* | *newline* | *nested-phrase*
subcomponent	::=	< *named-son* > | < *attribute-name* >
newline	::=	*required-newline* | *optional-newline*
optional-newline	::=	‖
required-newline	::=	!!

nested-phrase	::=	@ *envt* (*phrase*)
envt	::=	*perspective* | *formatting-envt*

Identifiers

view-name	::=	*identifier*
named-son	::=	*identifier*
attribute-name	::=	*identifier*
perspective	::=	*identifier*
formatting-envt	::=	*identifier*

II. Formatting Environments

Formatting environments determine the way an *unparse phrase* (Appendix I) will be displayed. Each formatting environment consists of collection of attribute-value pairs. In the first subsection of this appendix we list the formatting environments defined by the VIZ/UAL system. In the second subsection we list the attributes that are used to determine a formatting environment.

II.1. System Formatting Environments

Name	*Result*
HV	Display a phrase on a single line if it will fit.
Table	Format in a table of columns adjusted to maximum width of largest element in the table. Can only be applied to a list of identifiers.
Center	Center the display in the middle of the output device line.
Verbatim	Suppress conditional formatting. Display as written.
+, Indent	Indent.
-, Outdent	Outdent.
<, Flushleft	Align with the left margin.
u, Underline	Underline without breaks.
b, Bold	Display using **boldface**.
s, Shadow	Display using shadowed face.

II.2. Display Attributes

Name	*Type and Meaning*
font	Font name.Values depend on output device. Specifies font *family* and *size*.
face	Set of face attributes. Values depend on output device. For the Macintosh VIZ/UAL supports *italic*, *bold*, *shadow*, *underline*, and *outline*.
left-margin	Number of spaces, or size in pixels. Examples: "2 spaces", "25 pixels". Determines the leftmost position to which text will be written. A *space* has a fixed width so the specification of left margin is guaranteed to be the same for all fonts.
right-margin	Number of spaces, or size in pixels. Determines the distance from the left border of the output device to the right hand margin. Depending on the line wrap policy for the given device, it may or may not be the case that no characters extend beyond this margin.

indent-increment	Number of spaces, or size in pixels. Determines the effect of the **Indent** formatting environment.
spacing	Number of lines, or size in pixels. Examples: "2 lines", "25 pixels", "0 lines". A value of 0 lines implies single spacing. A value of 1 line implies double spacing.

References

[Ambriola 84] Vincenzo Ambriola, Gail E. Kaiser and Robert J. Ellison.
An Action Routine Model for ALOE.
Technical Report CMU-CS-84-156, Carnegie-Mellon University, Computer Science Department, August, 1984.

[Charniak 80] Eugene Charniak, Christopher K. Riesbeck and Drew V. McDermott.
Artificial Intelligence Programming.
Lawrence Erlbaum Ass., Hilsdale, NJ, 1980.

[Coutaz 85] Coutaz, Joelle.
A Paradigm for User Interface Architecture.
Technical Report CMU-CS-84-124, Carnegie-Mellon University, Computer Science Department, May, 1985.

[Delisle 84] Delisle, Menicosy, and Schwartz.
Viewing a Programming Environment as a Single Tool.
In *Proceedings of the Software Engineering Symposium on Practical Software Development Environments.* ACM-SIGSOFT/SIGPLAN, April, 1984.

[Feiler 83] Peter H. Feiler and Gail E. Kaiser.
Display-Oriented Structure Manipulation in a Multi-Purpose System.
In *Proceedings of the IEEE Computer Society's Seventh International Computer Software and Applications Conference (COMPSAC '83).* November, 1983.

[Garlan 84] Garlan, David B. and Miller, Phillip L.
GNOME: An Introductory Programming Environment Based on a Family of Structure Editors.
In *Proceedings of the Software Engineering Symposium on Practical Software Development Environments.* ACM-SIGSOFT/SIGPLAN, April, 1984.

[Garlan 85] Garlan, David B.
A Framework for Unparse Specification Languages.
1985.
To appear.

[Garlan 86] Garlan, David B.
Views for Tools in Software Development Environments.
PhD thesis, Carnegie-Mellon University, 1986.
In progress.

[Gnome 85] Chandhok, Garlan, Goldenson, Tucker and Miller.
Structure Editing-Based Programming Environments: The GNOME Approach.
In *Proceedings of NCC85.* IFIPS, July, 1985.

[Goldberg 83] Goldberg, A. J. and Robson, D.
Smalltalk-80: The Langauge and Its Implementation.
Addison-Wesley Publishing Co., Reading, 1983.

[Gutknecht 84] J. Gutknecht and W. Winiger.
Andra: The Document Preparation System of the Personal Workstation Lilith.
Software-Practice and Experience :73-100, January, 1984.

[Hansen 71] Hansen, W. J.
Creation of Hierarchic Text with a Computer Display.
PhD thesis, Stanford University Department of Computer Science, June 1971.
Also Argonne National Laboratory ANL7818, July 1971.

[Kaiser 85] Kaiser, Gail E.
Semantics for Structure Editing Environments.
PhD thesis, Carnegie-Mellon University, June, 1985.

[Lampson 78] B. W. Lampson.
Bravo Manual
Xerox Corporation, Palo Alto, CA, 1978.
in *Alto User's Handbook.*

[Medina-Mora 82] Medina-Mora, Raul.
Syntax-Directed Editing: Towards Integrated Programming Environments.
PhD thesis, Carnegie-Mellon University, March 1982.

[Mikelsons 81] Mikelsons, Martin.
Prettyprinting in an Interactive Programming Environment.
In *Proceedings of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation.* SIGPLAN/SIGOA, June, 1981.

[Oppen 79] Oppen, Derek C.
Pretty Printing.
Technical Report, Stanford University, 1979.

[Reid 80] Reid, Brian K.
Scribe: A Document Specification Language and its Compiler.
PhD thesis, Carnegie-Mellon University, 1980.

[Reiss 84] Reiss, Steven P.
Graphical Program Development with PECAN Program Development Systems.
In *Proceedings of the Software Engineering Symposium on Practical Software Development Environments.* ACM-SIGSOFT/SIGPLAN, April, 1984.

[Teitelbaum 81] Teitelbaum and Reps.
The Cornell Program Synthesizer: A Syntax-Directed Programming Environment.
CACM 24(9), Sept. 1981.