Chapter 1
User Interface

User interaction with the system and his own programs is through the editor. The users' investment in learning these facilities is repaid in increased functionality and more uniform interface.

The user is primarily interested in manipulating the entities that make up the environment. The user interface is concerned with providing an orderly and convenient method of expressing these manipulations. The user communicates by typing characters (or function keys or moving a mouse). Though system entities are often presentable in a readable form, the objects themselves are not made up of the characters used to present them. As a result, the user interface is constructed to interact with the user through character editor and with the entities themselves in terms of their own representation. To accomplish this, the editor is separated into two layers:

1. The visible interface is a multi-window editor that provides a core set of facilities for handling user input, editing and screen management. This is called the Core Editor.

2. The type-specific, object-knowledgeable portion of the editor is called the Object Editor. Which object editor is used depends on the type of the object (entity). Although specific object types may require specific operations, there is a common set of operations requiring type knowledge that is provided by all (or most) object editors. These are referred to as object operations.

1.1. Core Editor Concepts

The Core Editor provides character editing facilities. This section is an attempt to define and briefly describe these.

1.1.1. Screen Structure

A Screen is the entire contents of the display at a particular time. Screens are made up of opaque rectangular areas, called Windows, arranged in a possibly overlapping pattern. More than one screen can be maintained by a session to facilitate changing from one multi-window activity to another (though not initially).

Windows are composed of character positions and, optionally, borders. Borders are used to visually delineate windows. The character positions represent a bounded rectangular region of the quarter-plane of an Image.

An image is an array (Natural) of lines, each consisting of an array (Natural) or characters. At any time, each image has a specific

number of lines, each of which consists of a specific number of characters. Lines beyond the end of the image and characters beyond the end of lines are treated as blanks on the window. A word is a portion of a line delimited by separator characters. Word boundaries are completely syntactic and are handled by the Core Editor.

An image is the user-readable representation of an entity in the system. One of the functions of the Core Editor-Object Editor combination is to provide mechanisms to reflect changes from the readable to the internal representation and back within the editing paradigm. The image is the Core Editor representation of the object.

Superwindows are collections of windows that are logically linked and maintained to be physically contiguous. Because of this logical connection, superwindows are commonly referred to as windows composed of windows. The most common example of a superwindow configuration has the following characteristics:

1. A window containing the image of an object to be edited. This is called an object window. It normally has top and side borders.
2. A banner window that explains the purpose and status of the object windows. Normally presented in a different font than its associated object window, with side borders. Although banners are implemented as windows, no editor operations will be provided initially for their manipulation.
3. A command window that is used for entering Ada statements to be compiled and executed to perform actions on the user's behalf. Normally has bottom and side borders.
4. The appearance of the whole is of a single box, surrounded by borders, with the command window separated from the object window by the banner.
5. The command window is an object window in its own right.

There is a system-managed output window that serves as the destination for general error messages and system output. Its associated banner is used to depict the state of the session.

1.1.2. Cursors

The physical screen has an apparent cursor, marking the current position of the user's focus of attention (from the editor's point of view). This is called the screen cursor. If the cursor is within a window, it represents the:

1. Image cursor: (line, column) in the image on the window.
2. Window cursor: (line, column) on the window.

If the cursor is not within a window, the image and window cursors, and operations that depend on them, are undefined.

For each type of cursor, there are a variety of operations to specify its position. Changing on the position of one type of cursor often, but not always, changes the position of others.

The window and image cursors are closely linked. When they move in concert, the screen cursor moves across the window; when they move separately, the image scrolls on the window (in addition to possible screen cursor motion). The rest of this section deals with image and screen cursors and their relation to each other, ignoring window cursors to simplify the discussion.

Moving the image cursor causes the screen cursor to move. Moving the image cursor to a position that is currently not on the window causes the window to be scrolled. The screen cursor will not leave the current window because of an image cursor motion.

Moving the screen cursor causes the physical cursor to move without changing the image cursor. Having moved the screen cursor to a position within a window, any operation involving either the image cursor or the underlying image causes the image cursor for this window to be moved to the screen cursor.

Each window has a current image cursor position. Operations that change the focus to a previously visited image (and do not specify a particular position in that image) will place the cursor at the previous image cursor position. Thus, moving away from a window using screen cursors leaves the image cursor at the point of last interest rather than at the exit position.

A mark is a saved image position. Marks are stored in terms of absolute image positions and do not change to adjust for inserted/deleted lines/characters.

1.1.3. Fonts and Designations

Each character that appears in a window is displayed in some font. The appearance characteristics of fonts vary from terminal device to terminal device, but different fonts on the same device commonly differ in boldness, brightness, video presentation (reverse or normal), underlining and blinking. More advanced devices allow traditional font distinctions such as italics. Specific choices are terminal-specific, but banners are typically represented in reverse-video, keywords are underlined or emboldened, etc.

Fonts are used to convey the usage of the characters displayed. In some cases the distinction is for user emphasis (e.g. keywords). More commonly, fonts are used impart a different meaning to the characters displayed. Each window has a default font. Characters that represent themselves and not otherwise special appear in this default font. There are, at least potentially, more different uses for fonts than a particular terminal supports. When this occurs, the same font will be used for more than one meaning, hopefully in a way that is not confusing.

Each non-printing ASCII character can be represented by its traditional position in the Control- sequence. Each of these is

printed as a font-changed version of its base character. For example, ASCII.SOH (aka Control-A) might be represented as a reverse-video A.

Many editor operations require one or more implicit operands to accomplish the desired goal. The current cursor is one such implied operand; the current selection is another. Two kinds of selection are available: text and object. Text selections are formed by marking the first and last character positions to be selected, thereby selecting the text in between. Object selections are accomplished by various Object Editor operations. These operations select a region of the image that corresponds to a meaningful portion of the underlying object.

For either form of selection, the region of the image corresponding to the selection is presented in a font to provide visual feedback as to the extent of the selection. It is possible to convert object selections to text selections, so either type is acceptable to text operations. Text selections need not have any relation to object boundaries and are not appropriate for object operations. Even so, the font used for the two types of selection is typically the same, relying on the user to remember how the selection was formed.

A *designation* is one of three forms of meta-text that object editors can insert into an image to convey special meaning and support structured text within the editor paradigm. Designations are presented in non-standard fonts.

Elision is the process of removing detail from an image. The editor supports this by allowing a section of the object to be elided and represented by an *Ellipsis* mark (typically "...", but more meaningful phrases are possible). The ellipsis mark is presented in a special font and is treated specially in Core Editor operations. The ellipsis is a placeholder for the elided section of the image. As such, the Core Editor treats the entire ellipsis as a object, rather than as a collection of characters. Specifically, it is not possible to change individual characters. Moving or copying the ellipsis only moves or copies the underlying object if done by object operations.

A *Prompt* is a placeholder for an empty place in the object that the user may want or need to fill. The prompt is an extension of the traditional notion of prompt as one or more characters printed at the beginning of a command line to signify readiness and remind the user of the program to which the command will be routed. Prompts are placed wherever the Object Editor expects the user to provide content. The prompt is printed in a distinguished font and disappears when any attempt is made to type over it. As a result, the prompt serves as a reminder and placeholder, but requires no effort to delete.

The contents of a prompt depends on the item to be entered and the amount of information that the underlying Object Editor has about reasonable values. The simplest form of prompt contains the name of the class of object that needs to be provided. For Ada, this would

likely be a nonterminal in the abstract grammar, e.g. expression. In more semantically defined situations, the prompt might contain a reasonable initial value. The default value of a parameter or the default initialization of for a field in an aggregate are examples of prompts that, left alone, become the values provided. An operation is provided to convert the prompt text to plain text, allowing normal edit operations without losing the entire text of the prompt.

An error is a section of text marked by the object editor to indicate a problem of some sort. An error is treated as a prompt for editing purposes. Correcting the problem detected will cause the error to go away when the object editor re-formats the presentation of the object.

1.1.4. Mechanisms

The following mechanisms are provided to support editing operations that are not primarily dependent on the apparent objects on the screen.

A keymap is a mechanism for binding a key or key-sequence to a specific action. Every key that the user hits is bound by this mechanism to some command. For example, the most common commands are character insertions that are mapped to the key labelled with the character. By changing the keymap, it would be possible to implement a Dvorak keyboard without modifying the terminal. Keys can be mapped to any statement list, but the most common mappings are to specific procedure invocations. Mechanisms are provided to accelerate functions that are mapped to known editor procedures, elaborated procedures that can be invoked independent of context and statement lists that are repeatedly executed in a context that has remained constant. Regardless of the level of acceleration provided, the semantics of a key are defined by the semantics of the Ada statements it maps to.

A macro is a sequence of saved editor commands that can be invoked together. Macros are appropriate for recording a set of actions for re-use later. It is expected that complicated operations, including those requiring parameter passing, will be done with Ada programs. Facilities are provided for saving macros with a session and for binding them to keys. Macros act like parameterless procedures with no local declarations and no control structures (except those internal to individual commands in the macro). There will, eventually, be a facility to convert macros into equivalent Ada procedures.

A Yank Buffer is a piece of an image that has been saved for later use, typically by a deletion operation.

A Command Image is any Ada fragment that is prepared to be compiled and executed on the user's behalf. The Core Editor has no information about what each of these commands does, but saves the image in case the user wishes to repeat the same or similar operation. Command images can reference any of the builtin commands by their Ada names.

A Stack is a structure for saving a set of objects based on usage patterns. Stacks are used to store marks, windows, images, selections, yank buffer, and command images. The operations described below make it possible to cycle through the previous instances of each type in an orderly manner. The primitive operations are:

1. Push. Add/move an item at the top of the stack.
2. Next. Examine the next item down the stack.
3. Previous. Examine the previous item up the stack.
4. Top. Examine the item at the top of the stack.

Next (previous) "wraps" to the top (bottom) when applied to the bottom (top).

1.2. Object Editor

The object editor provides the transformations between the object and its image. This is done by incremental parsing and pretty-printing operations. Four basic operations are supported for viewing and changing objects.

1. Display. Create the image of an object.

2. Format. Parse text changes made to the image into the object and update the image to reflect the changes. This provides an opportunity for incremental syntax checking and correction and pretty-printing.

3. Commit. The object is in a user-desired state. Take the appropriate actions to reflect this intent. For most object types, this means saving the object. For commands, it causes the command to be executed.

4. Revert. Bring the image back to the state it had following the last commit. This provides a coarse-grain undo facility.

The object editor provides selection operations that understand the structure of the object being edited. These operations provide the ability to select objects, their parents (the containing object), next and previous brothers, and children. These selected objects serve as operands to move, copy, delete, elide and expand operations, as well as to type-specific tools outside of the editor.

1.2.1. Pointing

One of the basic notions of the environment is that objects are interconnected and that it is easier for the user to point at an object of interest and request information than it is to formulate a specific procedural request naming the object and the desired information. Having selected an object of interest (either explicitly or by simple cursor placement), at least the following broad categories of information can be requested:

Definition
: Show the definition of this object. For a reference to an Ada object, this move the cursor to the declaration of the object. From the defining occurrence, it moves the cursor to the definition in the body or private part.

Completion
: Provide infcrmation about the possible correct completions for the object of interest. Fill out all or part of a name on the basis of a prefix or pattern. Fill out the remainder of a syntactic structure. Provide prompts and/or values based on the type of the object that will make it possible for the user to complete the object. An example of all of these would be entry of the prefix of a procedure name and having it complete to a procedure call with full named-parameter notation for the call prompt-designated presentations of the defaults and nonterminal prompts for parameters without default values. An advanced form of completion is to provide prompt values that are the results of evaluating default value functions. The result of the function will often mean more to the user than the process for determining it. There is an associated ability to cycle through choices be repeatedly evaluating these functions.

Help
: Explain the object. As distinguished from definition, show a description of the object and its use. For an error, show an explanation of what was wrong, associated rules, etc.

Attributes
: Display attributes of the object that are not part of its image. Instances of this sort of information would be modification date, creator, and installation/elaboration status.

1.2.2. Operations

Object Editors provide a number of common operations that depend on the form and content of the objects presented. The basic ability is to read and format the object and, in many cases, take the modified image and convert it back into its object equivalent. In addition, object editors provide movement/selection operations that depend on the structure of the object. The assumption is that the object can be viewed as a tree-structure in which each object has a parent (the object containing it), siblings (objects with a common direct parent), and children (objects that it contains). For Ada programs, these operations follow the logical nesting structure of the language; for text, the correspondance might be sentences, paragraphs, sections, chapters, etc.

1.3. Ada Editor

While it possible to conceive of object editors for many types, the first and most important is the one for Ada. Because of its interaction with system structure and semantics, the Ada object editor provides operations and, in some cases, imposes restrictions that have no parallel in other objects.

Editing Ada source objects follows the Core Editor-Object Editor paradigm. Changes are made to the source as text. The Ada Object Editor provides syntactic completion, structural motion (parent, child, sibling), etc. based on its knowledge of Ada. Though the object-specific operations differ, there is no conceptual difference between these Ada source objects and text objects. The meaning and variety of operations differ because of the intrinsic differences in the two types of objects, the fundamental reason for the existence of type-specific object editors.

1.3.1. Insertion Points, Installation and Elaboration

Elaborated and, to a lesser degree, installed objects are fundamentally different from source objects and the editing operations that are appropriate are correspondingly different. Elaborated packages contain Ada declarations that are referenced by other installed or elaborated unit, affecting both their compilation state and any active threads executing in the corresponding code.

To control the changes and make it clear what was intended, operations are provided to explictly withdraw the elaborated version and install/elaborate its replacement. By limiting the scope of what is withdrawn or replaced, it is possible to restrict the impact of the change to the specific objects that were changed. Object deletion provides an unambiguous way of removing a precisely specified set of objects. Insertion points provide a similarly explicit way of specifying where new objects are to be created. An insertion point is represented to the user as an ellipsis, that when inspected is represented in a source Ada window. The user can then enter the declaration for the object (using Ada source editing). When the source object is installed, it assumes its position at the insertion point, either as an object itself or as a "separate" reference to the newly created separate object.

1.3.2. Directory View and Attributes

Traditional directory services provide access to a variety of information to help remind the user of what is contained in the directory, when it was created or changed, how large it is, etc. The principal support for this is a procedure that will print a list of objects in a directory accompanied by the appropriate attributes. The list will appear on the screen as an output window; changing the output has no effect on the underlying directory. This will eventually be supplanted by a read-only object editor that provides

the same information along with additional control and display facilities. A limited form of writeable object editor could be provided to allow object deletion, changing names, and changing attributes that are user-changeable.

1.4. Program Execution

User actions are performed by executing Ada statements. These statements can be executed by creating a command window, entering the desired Ada code, and commiting the command window. This causes the statement to be compiled and executed. It is also possible to bind statements to keys in ways that shortcut the compilation without changing the semantics (1.1.4). The sections below describe the context in which statements are executed, the forms of binding and how they interact with execution, and the runtime environment provided for statements.

1.4.1. Context

Ada statements are semanticized and executed from a particular context. For user commands, the environment constructs a context that provides convenient access to user, system and object-type-specific objects and procedures.

The context is a declare block at the end of the body of the elaborated package corresponding the current object window. For a command window attached to an elaborated package, this is simply the end of the package body. All other objects are considered to be rooted where they are declared. Dynamically created objects have the context of their creator, e.g. a Text_IO window for an object with no underlying file has the context that was active when the command that created it was started. The initial (and any other for which no predictable dynamic predecessor exists) context for a session is set to the home package of the user. Subsequent session continuations resume the context saved at shutdown.

The declare block that is generated as the default for a particular command window has the following form:

```
declare
    [global declarations]
begin
    declare
        [object editor-specific declarations]
    begin
        [statements]
    end;
end;
```

The global declarations are typically use clauses and renames that make system and core-editor commands more accessible; the object

editor-specific declarations provide the same facility based on the type of the object being editted. The user can change the declarations for a particular execution simply by editing these declarations. Changing the declarations persists with the particular command window, but does not change the underlying defaults. Any legal Ada declaration is possible, but the declaration is elaborated once for each execution, so it is not possible to retain state from one execution to another, only between the statements of the block. The nested declarations are required to allow object-specific operations to hide global operations. The declarations to be used in each context are defined by declarations in the definitions package of the user's home package. Appropriate default values from the system definitions package are used if no user definitions are provided.

1.4.2. Command Windows

Each object window has (or can have) a command window associated with it, from which it is possible to type Ada statements to be executed. The term, command is used to mean the set of statements in a command window. For the case where a single statement has been entered, this coincides with the traditional command paradigm; for more complicated command windows, it can be very different. The full facilities of the Ada Object Editor are available to edit commands. The command image initially contains of the context described above. The cursor is positioned on the statement prompt in recognition of the relative frequency of simple statement entry, but it is possible change the declarative part of the block with normal editing. Commands are executed by commiting the current command window. After the command has terminated, the declarative portion of the command image will remain. The statement portion of the window will be converted to a prompt in preparation for new statements.

The standard arrangement for command windows is to have one under each object window (or set of object windows). When a command window is treated as an object window (i.e. the user enters commands to modify it), a new command window is created. This new command window operates on the command window as an object, not on the original object. Its context is rooted in the same place as the base object window.

Command windows are automatically placed at the bottom of the object window they deal with and are not generally separable from their object windows. Command windows persist, disappearing only when explicitly requested or when their object window is removed or replaced.

A history of command window entries is kept to allow the user to examine and re-use previous commands. The history is retained as a stack of entries that were actually executed from a command window; commands that are directly bound to keys do not appear in this history. One history is kept for all command windows. Consecutive repetitions of commands are reduced to a single instance. The history

stack has a fixed (though possibly very large) depth. Later versions will provide facilities for history commands that deal with the history of a particular command window.

1.4.3. Execution and Concurrency

When the contents of a command window are executed, the editor buffers input until one of following happen:

1. The command finishes. This is the sequential command execution case. The buffering provides traditional command type-ahead. While the command is executing in this form, the user is said to be connected to the command.

2. The command requests input. Input requests from commands are handled by editing into an input window attached to the executing command. If the user is connected to the command, a request for input causes the cursor to be moved in this input window. At this point, the command is waiting (though associated tasks are certainly not stopped) for input. The user can provide that input or perform any other editing operations. When input is provided and commited, the user is again in the connected state with input buffered.

3. The user disconnects from the command execution. In many cases, the user will not want to wait for the command to complete before going on to do something else. In these cases, it is possible to disconnect from the command, causing it to run asynchronously. Disconnection can occur either before or after the command has started execution. Prior disconnection is possible by issuing the disconnect in conjuction with commiting the command window. This removes any chance of race conditions between the user and the program as to where the cursor ends up or other state transitions. Note that disconnection doesn't inhibit the program from either writing output to a window or requesting input. Input requests no longer automatically move the cursor into the input window. See PROGRAMIO for more details.

4. The user cancels the command. This causes execution of the command to be terminated. Buffered input is also lost.

1.4.4. Binding and Builtin Commands

Two different methods for causing statements to be executed have been discussed: commands bound to keys and execution of the contents of a command window. With the exception of side-effects on various components of editor state, either method results in the same execution. The effect is that of executing the designated Ada code in the proper environment. This does not imply that all execution uses the most general mechanism. Rather, the acceleration mechanisms provided to make builtin commands execute quickly are the result of

careful binding of the fixed command set to externally visible procedure instances.

Ada names are bound to internal commands by means of keymaps (KEYMAPS). When a key is indexed, the bound command is executed. The process used to determine what to execute depends on the Ada name, the type of the binding, and the ability of the command object editor to detect equivalence to a previously used command.

Builtin commands are Ada procedures that have a fixed location in the envrionment. Binding keys to these procedures involves the selection of a fixed functionality, independent of the executing context. As a result, once the correspondence has been established, it is possible to shortcut the key to execution process without even calling the indicated procedure. Obviously, if it were possible to change the bodies of the fixed procedures, without changing the internal operation of the procedures, an inconsistency would arise. Similarly, it is always possible to introduce hiding into any Ada scope such that a "fixed" name references a new procedure. Since there is no chance that this would occur inadvertently, no steps are currently envisioned to protect against such a confusion.

Even if a procedure is not one of those implemented inside the editor, it is possible to bind a key to its execution in a way that is context independent. This simply requires that the binding be to a fully-qualified name in a context that is very unlikely to be hidden. This form of binding is treated just like builtin commands, except that a more sophisticated invocation method is required. It suffers from the same unlikely inconsistencies in the face of concerted attempts hide the intial definition.

By binding keys to simple (or not fully qualified) names, it is possible to provide keys that execute different functions depending on the context in which they are invoked. This form of binding is very likely to require semantic analysis and possibly code generation to be successful. Some acceleration is available by recognition of known names from the semanticized name in context or by recognition of previous use of the same procedure in the same context.

One last form of binding allows keys to be bound to commands with the purpose of prompting for the command (placing it in context in the command window), rather than to execute it. This allows keys that provide the command and prompts for the parameters, saving command entry, but still allowing complete parameter flexibility. This mechanism is invoked whenever the Ada that was bound to the keys was incorrect and/or incomplete, allowing the user to see the problem and correct it.

1.4.5. Jobs

Ada execution takes place within tasks. A single user command, even one that is apparently sequential, may be implemented by more than one

task. Jobs (JOBS) make it possible to treat the command execution as a single entity, without worrying about the precise implementation.

Jobs provide a basic level of execution control. The tasks of the job can be scheduled together or terminated together. The job serves as an identifiable entity for these purposes, where for individual tasks, there is no guarantee that an Ada name exist for the task throughout its execution. The environment also uses jobs as the basis for determining the current user focus: the user is either waiting for the completion of a job (command) or not. The execution priority of the command and the course of the user's interaction with it can be different in the two cases.

Each job has associated state corresponding to traditional program or process state. Although some of this state is system control information, salient pieces are of interest to the user. A good example of this is the standard input and output files defined by Text_IO. The location and status of the windows allocated to these files is part of job state.

A series of user commands executed serially will share (serially) the job and its state. Continuing with the Text_IO example, a series of commands executed will share input and output windows, creating a single script of the various command executions.

Disconnecting from a job creates a new job with its own state. In the Text_IO window example, two asynchronous jobs would update different windows. Once a command is started, its job number doesn't change (though it may create other subordinate jobs). As a result, a job that is started, then disconnected will act upon the inherited state and any new commands that are entered start over from scratch (in the Text_IO example, the disconnected job uses existing windows and new commands get new windows). Disconnecting a job before it is started causes the newly started job to create its own state and leaves the user attached to the same job as before the command was started.

1.4.6. Naming Objects

Naming objects in the package directory system follows the naming and visibility structure defined in Ada. The following factors are involved in extending these simple names:

1. Versions and configurations. Because there are multiple versions of most objects, Ada naming is interpreted within the context of the current configuration. Procedures and functions that provide access to specific versions will do so by explicit version parameters.

2. Ada program objects. Ada (quite reasonably) provides no way for programs to name their source components. To provide self-reference, attributes have been provided for each program object that allow designation of the principal parts of the

object. Names consisting of an Ada name attributed to indicate the part are called source names.

3. Nascent objects. If the object doesn't exist yet, it can't be named. Strings are used to provide the name the object will take on. Strings are also used as in traditional systems to provice deferred naming within programs.

4. Convenient aggregation. Users often perform operations on groups of objects whose names are textually related. This is done by providing wildcard characters to be used in conjuction with string names.

1.4.7. Source Names

Conceptually, each program object has a visible part and a body. For a object, Ada_Name, the visible part is Ada_Name'Spec and the body is Ada_Name'Body. If the object has only a visible part or only a body, the attributes are interchangeable. For a type completed in the private part, 'Body refers to the completion of the type and 'Spec refers to the incomplete declaration.

Overloading makes procedure and function names without parameter specifications insufficient. For each overloading of a name, a nickname is provided. The nickname is used to index the 'Spec and 'Body attributes. The system assigns numeric values as the nicknames on the basis of their order of occurrence in the visible part and body. The same nickname value is assigned for both the body and visible part of a single object. A facility will be added to allow users to designate nicknames using pragma Nickname (Mumble).

User nicknames define an enumeration type, package_name'Nicknames. As such, the nicknames for the subprograms of a package are unique for the package, not just for the subprogram name that is overloaded.

1.4.8. Strings as Names

Strings can be resolved as Ada names. Whatever could be provided as an Ada or source name can be placed in a string and resolved as a name. Most procedures and functions for direct user use will provide string names in addition to direct object references. The spirit of the environment is for names to be Ada names, so though it is not possible to keep string names from being interpreted differently, there is considerable advantage to uniformity.

Strings are used to provide the name for new objects. All names are Ada names, so the string must contain a name that will be legal after its introduction into context.

The restrictions on naming Ada objects are not as severe for strings as for direct Ada names. Specifically, program objects need not have 'Spec or 'Body unless the operation requires it; the unattributed name refers to both the visible part and the body if both exist.

A string referring to an overloaded set of objects refers to all of the objects. If the context requires a unique object reference, this is an error.

1.4.9. Context

As describe in the section on command windows, name resolution depends on the context of the current object. The assumption in that section was that current object is part of the package directory system. This assumption works well until the object under study is the execution instance of a running program. Assume that the user is stopped at an invocation of function F, initially executed from package P. Different contexts are of possible interest:

1. The initial package context. Objects in this context are available from a command window on the same object as the one from which execution was initiated.

2. The local execution context. This is the context of the current command window. References to objects or procedures local to the current invocation of F are available by the same names as they would be within F, though other procedures or functions in the same package may be available even though their declarations are not visible to F.

3. The global execution context. If F is part of the execution of program created from a library, these are references to objects in other packages in the closure of the packages necessary for the program to execute. For units that are with'ed in the context of F, direct references are available.

4. Dynamic execution context. This is the same as the local execution context above, except it refers to a context other than the current one. Examples would be previous invocations of F (or other units) in the dynamic call history or invocations in a different task thread.

All but the last two of these contexts are available without extension of Ada naming. All of them are available by establishing context at the appropriate context. To do this, there must be a naming convention for contexts. The current debugger uses a preceding "." to indicate global names to be accessed independent of with list and "@n" to indicate relative stack frames. There are also explicit names for execution contexts, e.g. specific tasks instances. The current debugger notation for these is Ada-like. To carry them forward into the environment, it might actually be better if the notation were explicitly non-Ada.

1.4.10. Context, Creation and Deletion

Ada makes names more available than typical directory systems, partially because names are only referenced, not created or deleted.

All Ada references are also from fixed lexical positions in the program.

Consider the following characteristics of names:
1. Simple, qualified.
2. Local, contextual, absolute.

Notes on all of the combinations of these name characteristics.

1. Simple local names are those available in a closed scope.

2. There is only one simple absolute name, Universe; all absolute qualified names start with Universe. There is a standard abbreviation, U, introduced by the declaration "package U renames Universe" immediately inside Universe.

3. Qualified local names start with simple local names.

4. Contextual names denote objects that are available through Ada visibility that are neither local nor absolute. Examples include names in containing scopes or in library units that are referenced in use clauses.

5. Object deletion and creation are unsurprising for local or absolute names.

6. Creation or deletion of objects referenced by contextual introduces the problem of "capturing" unintended objects. This is solved by expanding the name and allowing the user to proceed if that is the intent. Create and Delete will have parameters controlling how automatic capture should be.

1.4.11. Advanced Topics

A potential problem is the number of different types of names the user can specify, requiring common operations to be heavily overloaded and potentially leading to user-access inconsistency because not all functions are overloaded on all of the common methods. Note that the concern here is on the form of the name the user enters, not the resolution of the name. The types of naming the user has access to:

1. Direct object. The name directly specifies a particular object. This is the case where the most Ada type context is available to aid in resolution.

2. Vector aggregate. The Ada answer to procedures that want multiple objects as arguments. Potential inconveniences are introduced by the difficulties inherent in resolving (A, B, C) to be Array_of_File_of_Integer'(A, B, C) as required by the program. [2]

3. Strings.

4. Wildcard. The user provides a string with wildcard characters that resolves to a selection or a vector. Except for user convenience, string names could be replaced by a function that takes a string and returns a vector of objects. Note that it is not feasible to have an array aggregate whose elements are each strings.

5. Pattern. The user provides a syntax/semantic pattern for an Ada object with terminals, non-terminals and wildcards. The is the object editor extension of regular expression matching and is very powerful for editing Ada programs. [3]

Immediacy of interpretation is an issue. Simple completion is done by resolving a wildcard to a single name with user interaction to iterate over multiple possibilities. Wildcards will also resolve to vector aggregates in appropriate circumstances [2]. This corresponds to expanding the wildcard for the user prior to executing the command, rather than during command execution.

1.5. Keys and Command Factoring

The builtin command set of the editor has been factored into a sets of operations and sets of objects. Each group of operations can be applied to a group of types by using the key that specifies the object type followed by the key for the operation. Default object types have been chosen to reduce the frequency of two-key sequences, and since the factoring doesn't occupy all possible keys (especially for terminals with function keys, etc.), it is possible to place commonly used sequences on single keys.

1.5.1. Types

The set of object types is:

1. Character cursor. Character insertion, image position and marks.
2. Command. Command window and history.
3. Designation. Elisions, error and prompts.
4. Macro.
5. Line.
6. Screen cursor. Motion on the screen.
7. Selection. Both object and text selection.
8. Window.
9. Yank buffer.

1.5.2. Operations

The following is a brief description of each of the classes of operations and the types that each applies to.

1. Planar movement (up, down, left, right)
    1. Cursor. Move user cursor on the image

2. Screen cursor. Move user cursor on screen.
3. Selection. Select parent, child or brothers.
4. Window. Scroll the window over the image.

2. Relative positioning (next, previous, beginning_of, end_of)
   1. Designation.
   2. Line.
   3. Word.

3. Modification. (copy, delete, insert, move, transpose; capitalize, lower-case, upper-case)
   1. Character.
   2. Line.
   3. Selection.
   4. Word.

4. Stack. (next, previous, push, top)
   1. Command. Manipulate history. Push is implied by execution.
   2. Mark.
   3. Selection.
   4. Yank buffer.

More detail, including an initial key assignment for QWERTY-only keyboards is available in [BLS.CE.DOC]R1000_Commands.MSS.

Table of Contents