Notes on Definitive Notations

being extracts from Chapter 2 and Appendix A from the 3rd year project *3 D -Physical Modelling with Definitive Notations* by Andy MacDonald (1997).

2 DEFINITIVE NOTATIONS

This chapter describes a relatively new approach to computer based modelling, which uses a different type of software tool: the definitive notation. Firstly, I shall discuss the general concepts of definitive systems and notation, followed by a description of the particular definitive notation that I have investigated for this project. The chapter concludes with a section on how these notations can be used for modelling purposes. Most of the material that follows has been gained from personal experience during the course of the project, since very little reference material is available.

2.1 Concepts of Definitive Notations

Real world phenomena can be modelled on a computer by describing them as problems that can be solved using computer programs. These problems and methods of solving them need to be somehow represented in the programming language being used. Conventional languages (i.e. procedural languages) use variables and procedures to model the problem and have a set flow of control for solving it. The variables refer to storage locations that hold data about the objects being modelled and their values collectively represent the current state of the problem. Procedures describe what to do with the data stored in the variables and consist of sequences of instructions whose side effects change the state of the problem. The computer has to be told explicitly when to change certain variables and how to compute them. This puts added burden onto the programmer since he/she has to decide when it is appropriate to update certain variables and has to include instructions to do so in their program.

Imagine a model that contains a table and a lamp, which can be moved around by specifying their positions. To represent this you would need a program that contained, among others, variables holding the positions of the table and the lamp. To make the lamp sit at the centre of the table then the position of the lamp needs to be the current table position, plus an offset to the centre of the table. If you wanted to move the table to a new position then the position of the lamp also needs to be changed requiring a procedure to change the table and lamp positions. Now whenever the table is moved, the lamp is moved to the new table's centre. If you also want to move the lamp to any position on the table, the procedure needs to be changed to take into account the position of the lamp on the table. In addition, to get an up to date picture of this model, a redraw procedure needs to be called every time something is moved. All these dependencies need to be programmed in explicitly, which can become a complex task with many dependant objects and dependencies that are more complicated.

The above example shows that while procedural languages can be used for modelling (and are) they provide extra effort that could be better spent on the actual modelling itself. It would be more convenient to be able to provide abstract definitions of objects in term of other objects being modelled. The computer can then handle these definitions implicitly, making sure that they are always true. This is especially important during development of the model, when objects and the dependencies between them need to be frequently changed. Only the actual definitions need to be changed whereas, with procedural languages, entire sections of code may need to be altered, which takes time and is prone to errors. A system that can manage definitions between objects is called a *definitive system* or *definitive based system*.

Definitive systems consist of variables, which hold data representing different objects, and definitions, which describe the dependencies between objects. A definition usually defines a single target variable by an expression containing one or more source variables. The target is then said to *depend on* the source objects and the expression governs how the target should be computed from them. These definitions are handled by the system, which ensures that whenever a variable is changed, the variables that depend on it are re-evaluated to reflect this change.

The following illustrates a typical definition:

$$t \equiv \Phi(s_1, s_2, \cdots, s_n)$$

Where t is the target variable, $s_1 \ldots s_n$ are the source variables and Φ is an expression involving the source variables. This definition states that *t depends on* $s_1 \ldots s_n$ and is computed from these source variables using the expression Φ. Should any of the source variables change then the expression is recomputed automatically so that this definition will always be true.

Two examples of definitive principles in use are spreadsheet software and the UNIX* **make** command. A spreadsheet consists of a grid of cells (objects) which can hold values or formulas (definitions). The formulas state how the value of a particular cell can be computed from other cells automatically. Whenever any cells are changed, formulas referring to that cell are recomputed and a display action is invoked by the system. Using the same principles, the **make** command uses a script of dependencies to maintain files. An example dependency is as follows:

```
program.o : program.c
                        cc -c -o program.o program.c
```

This states that the file `program.o` depends on the file `program.c` and provides a UNIX command (`cc -c` ...) for updating `program.o`. This command is the *updating action* of the definition and in this case, since it has been given by the user, it is an *explicit action*. In the case of the spreadsheet, the display depends on all the cells and the display action is an *implicit action* since it is pre-defined by the system.

In order to use definitions in modelling something, a definitive system needs to be implemented. The implementation of a definitive system involves developing a dependency maintainer, which is not a trivial task. Therefore, it is much more sensible to use a general-purpose definitive system with a language for specifying the definitions in. A language, which interfaces with a definitive system, is called a *definitive notation*. In the next section, I will describe the definitive system that I have used in this project, and its definitive notations.

The advantages of using definitive notations are as follows:

- There is no need to remember which variables to update when a change has been made to the state of a model. This is especially useful when the environment is continuously changing.
- All variables will contain the most up to date value.

* Trademark of Bell Laboratories

- They allow easy interaction for a user since response to user's actions is immediate.
- They provide a natural way of describing the relationships between the objects being modelled.
- Most notations are run-time interpreted, so there are no long compilation delays.
- If they are interpreted then all objects and definitions can be modified at run-time, allowing a wide range of interaction and run-time changes when developing new models.

The main disadvantages are:

- Since most notations are interpreted and there are overheads in maintaining all the dependencies, they run slower than equivalent compiled programs.
- Pure definitive notations provide no means of communicating with other sequential programs (e.g. operating system routines) and so procedure-like actions are needed, which are triggered by variable changes. The side effects of these actions can be used to cause changes to the system's environment.
- When objects are represented by complex data types, the definitions can become more complex. A new definitive notation, which contains these types as primitive, is often needed.
- There are no commercially available definitive notations and most that have been written are experimental (although some have been well developed and tested in academic environments).

2.2 TkEden — A Definitive System

The definitive system I have examined and used for my project is called **TkEden**. It runs in the X Windows* environment and provides a graphical user interface through the **Tcl/Tk**† interface scripting language. The system consists of a dependency maintainer along with pre-defined dependencies and support routines, which provide the interface between the definitive system and the X Windows environment. Definitions can be supplied in three different definitive notations, **EDEN**, **DoNaLD** and **SCOUT**, which are described in more detail below. **TkEden** provides an interpreter for the **EDEN** notation and definitions written in the other two notations are converted to **EDEN** definitions before being interpreted. This means that definitions in **DoNaLD** and **SCOUT** can be manipulated in both there own notations and in **EDEN** itself.

Upon running, **TkEden** provides two windows, one for input and the other containing the current display, along with a set of menus. Definitions are given to the system either through scripts given on the command line or by directly entering them into the input window. This means that pre-written definitive programs can be run by **TkEden** and while running, the definitions can be changed by typing new ones into the input window. As well as giving a form of interaction with the program, this also eases development of models through experimentation. Input is given as blocks of definitions each proceeded by the keyword `%eden`, `%donald` or `%scout`, which indicates what notation the following definitions are written in. List of definitions and other information useful for debugging purposes are

---

* Trademark of the Massachusetts Institute of Technology

† Created by J. K. Ousterhout of Sun Microsystems Laboratories

available through menu commands that allow you to examine the objects being modelled and the dependencies between them. The display window shows the current state of the model as graphical objects defined with **DoNaLD** and **SCOUT**. These can be interacted with also, providing definitive programs with simple graphical user interfaces.

Since **TkEden** is an experimental language and not commercially available, there is very little reference material for it. Only **EDEN** has a reference manual, 'The EDEN Handbook' [1]. There is not much information available for the other two notations. This means that the best way of understanding these notations is to examine other peoples programs and experiment. I have provided, in appendix A, a description of the syntax of these notations and a short reference to **TkEden** in order to help the unfamiliar understand any definitions I may give.

2.2.1 EDEN

EDEN is a general-purpose language that supports the concept of definitions. It's name comes from the name of it's original interpreter, **EDEN**[*] – an **E**valuator of **DE**finitive **N**otations. The language is not a purely definitive one but a hybrid language combining definitive and procedural programming. Each statement in a typical program is either a procedural style statement or a definition. When a procedural statement is encountered by the interpreter it is executed and will have the effect of evaluating an expression, assigning a value to a variable or calling a procedure. On the other hand, when a definition is encountered, an equivalent definition is set up internally, which will then be evaluated by the interpreter when ever necessary. Since some of the statements are executable, the order of execution depends on the statement order. This is not the case between definitions where the interpreter handles the order of execution.

The syntax of **EDEN** is similar to that of **C**, in fact it can be considered as a subset of **C** with some additions. Like procedural languages, it has variables that represent storage locations, and can be assigned values from expressions that can contain other variables, operators and functions. These variables are known as *read/write variables* (RWV's). Unlike **C**, RWV's do not need to be declared before being used and are allocated storage when they first appear in the program. The type of a variable does not need to be given and depends on the type of the value assigned to it. If a variable of a given type is assigned a value of a different type then the variables type is changed to that of the new value. The following are types that are essentially the same as those in **C**:

- Integer — e.g. `123` (decimal) `0456` (octal) `0x1f` (hexadecimal)
- Character — e.g. 'a'
- Floating point — e.g. `1.23 .23 1.23e-15`
- Pointer — e.g. `&int_var` (address of int_var's storage location)

There are also some additional types as given below:

- @ — This represents *undefined* and is actually a constant, which says that the variable does not have a type. A variable will have this value if it is used before having a value assigned to it.

---

[*] Y W Yung, "EDEN – an evaluator of definitive notations", MSc Thesis, Department of Computer Science, University of Warwick, 1989

- String — A string is a sequence of characters, for example
  `"this is a string"`
  If `s` is a variable holding a string and `i` is an integer, then the expression `s[i]` is the ith character of `s`. Also `s#` is the length of string `s`.
- List — This is the only structured type in **EDEN**. It represents a list of data values of the types given and each element can be of a different type. An example of a list is...
  `[ 100, 'a', "string", [1,2,3] ]`
  If `l` is a variable holding a list, then `l[i]` is the ith element of the list and is of the same type as that element. The length of the list is given by `l#` which is the number of elements in the list. In addition, two lists can be concatenated e.g. `[1,2,3] // [3,4,5]` gives `[1,2,3,4,5]`.
- Function — This is similar to the **C** function type in that it represents the entry point to a sequence of statements but has a slightly different syntax. First of all when specifying a function, no parameter list is given. When the function is called, its arguments are put into a single list variable named '`$`'. The elements of this list can be given aliases by including a `para` keyword at the start of the function body, followed by a corresponding list of names. All local variables used in the function also need to be declared at the beginning of the function body. This is done by giving a list of them after an `auto` keyword although their types do not need to be given. The other main difference is that if no return value is given, `@` will be returned. Function definitions are indicated by the keyword `func` or `proc` (there is no difference between the two) for example:

```
func max {
    para m;         /* m is alias of first argument */
    auto i;         /* local variable */
    for (i = 2; i <= $#; i++)       /* for each arg */
        if ($[i] > m) m = $[i];     /* keep max */
    return m;        /* return max of all arguments */
};
```

  This function can be called by, for example, `max(1,3,5,2)` which will return `5`.

EDEN has the usual arithmetic, relational and logical operators as **C** for handling the numerical types (e.g. +, −, <, >, ==, `&&`, `||`). The main operators for handling the other types were given in the type descriptions above. Values can be assigned to expressions through assignment statements which are of the form `var = expression`, `var += expression` etc. Other statements in the language, used for control flow, are the usual **C** `if-else`, `while`, `for` and `switch`.

The language also has two additional statements for giving definitions, *formula definitions* and *action specifications*. As already mentioned these statements are not executed directly, but when encountered, set up the definitions internally. These definitions state that a variable, or action, depends on a set of source variables. Whenever a variable has a value assigned to it, any definition that includes it as a source variable is re-evaluated. If the target of the definition

is a variable then it is called a *formula variable* (FV). These hold data in a different way to the normal RWV's. An FV consists of two parts; the *data register* (DR) and the *formula expression* (FE). The data register holds the actual value of the FV and can be used in expressions but not written to (it's value is written to by the interpreter). In addition, the expression used in the definition is held by the formula expression and it indicates how to calculate the DR from the source variables. The FE can be written to and hence the formula can be redefined.

A formula definition is similar in format to a variable assignment statement except it provides an abstract definition of the variable. The syntax is the same as that of an assignment with the = symbol replaced by the keyword `is`. A typical formula definition is given as follows:

```
t is f(s1,s2, … ,sn)*
```

Where `t` is the target formula variable, `s1...sn` are the source variables and `f` is a formula, which may be an expression or a function. This statement forms a definition, which says that the FV `t` depends on the source variables `s1...sn`. The source variables can be either RWV's or FV's and this definition is re-evaluated if one of the RWV's is assigned to or one of the FV is recalculated. While it was stated that a formula definition is similar to an assignment they not the same. Formula definitions can be considered to always be true (target variable is equal to the expression) while, on the other hand, an assignment statement can only be assumed to be true directly after it's execution. As an example, if you wanted to define the following:

> "C lies at the mid point of line AB, where the end points A and B are defined independently"

Then this would be given as the following formula definition:

```
C is (A + B)/2
```

For more complex behaviour than possible with formula definitions, **EDEN** provides action specifications. These provide a way of specifying explicit actions, which are called when certain objects change. Action specifications are procedures, which depend on a set of source variables. Their syntax is similar to that of a function but with a list of variables, called the *dependency list*, which the function depends on. This is a comma-separated list, preceded by a colon, between the function name and its body. An example of an action specification follows:

```
proc print : a, b, c {
    writeln(a,' ', b,' ', c);
};
```

In this specification, the procedure, `print`, depends on the variables `a`, `b` and `c`. If any one of these variables changes then the procedure is called, and the values of the three variables are printed out. These specifications are useful when some procedural side effect is required as the updating action of a definition. They are usually used when various objects need to be changed by a definition or for display actions that need to be called when variables are

---

* Compare this to the typical definition on page 9

updated. Action specifications are usually called by the interpreter when all formula definitions have been re-evaluated so that all variables have up to date values.

2.2.2 DoNaLD

The name **DoNaLD** stands for **D**efinitive **N**otation for **L**ine **D**rawing and this notation allows the definition of two-dimensional shapes. It is a definitive notation for line drawing because once a shape is defined it is displayed in the form of a line drawing (i.e. outlines for shapes like rectangles and circles). This notation allows the specification of graphical diagrams with objects being modelled as graphical objects on the screen. With this graphical output, the results of modelling a problem can be seen immediately and in a convenient form. The notation itself consists only of definitions of shapes, and it is therefore a pure definitive notation. This means that there are no directly executable or procedure like statements and every statement is a definition. A script in this notation is therefore like a specification of the dependencies between different shapes and their elements.

A specification in the **DoNaLD** notation consists of variables and definitions between variables. Each variable in this notation represents a shape and holds data that describes that shape. The type of the variable indicates the shape that it represents and, unlike **EDEN**, each variables type needs to be declared before use. There are, essentially, three categories of types in **DoNaLD**: numerical scalars, primitive shapes and structured shapes. The numerical scalar types are simply types that hold single values such as `int` (integer values), `real` (floating point values), `char` (representative values of characters) and `bool` (binary values). These are not drawable objects, and so are not displayed, but can be used to define the elements of other objects. The primitive shapes are the actual drawable shapes of the notation and the type indicates what sort of shape to draw. A variable of one of these types holds values which indicate how the shape is to be drawn (the attributes of the shape e.g. width, height etc.): Below are three types in this category (there are more but these three give the idea):

- Point — This type represents a single point on the screen. A point, in two dimensions, is given by its x and y co-ordinates. Therefore, a variable of this type holds two scalar values specifying its position. Points are constructed by enclosing two scalars by braces `{` and `}`. These two values are either the two Cartesian co-ordinates separated by a comma or, in polar form, the modulus (distance along vector to point) followed by an `@` followed by the angle (that the vector makes with the x-axis). Two example points are ... `{30,50}` and `{10@0.3}`
- Line — A variable of this type represents a line between two points. The variable actually holds a list containing two point values and it can be specified the same way lists are in **EDEN**. For example, `[{10,20},{30,40}]` represents a line from (10,20) to (30,40)
- Circle — A circle has a certain radius and a position and hence variables of this type hold a point value, for the circles centre position, and a scalar for the radius. To construct a circle variable it is necessary to use the built in '`circle`' function that takes, as arguments, a point and a scalar and returns a circle. For example, `circle({10,20}, 5)` which gives a circle at point (10,20) with radius 5.

Complex shape types can be built which are essentially data structures holding many other shape types. They are constructed by opening a new shape type with the openshape keyword and then giving a block of definitions within that shape. Each of these definitions creates a variable of a given type, which then becomes part of the shape. An example of a shape is given as follows:

```
openshape cross                          ⇐ Opens new shape 'cross'
within cross {                           ⇐ Start of block defining cross
    line l1, l2
    l1 = [{10,-10},{-10,10}]             ⇐ Defines two lines
    l2 = [{10,10},{-10,-10}]
}                                        ⇐ End of block
```

These shape types group together their element shapes, which are drawn when displaying a shape variable. The individual elements can be accessed through path names similar to directory paths on file systems. For example, to reference the first of the line variables in the above you would use cross/l1.

Definitions in **DoNaLD** are syntactically the same as assignments in **EDEN** (i.e. using the = symbol). Every definition in a **DoNaLD** script sets up a dependency between the variable being defined and any other variable used to define it. Therefore, if the position of one object is dependent on another's it is useful to define the position of that object in term of the position of the other. This can be shown by defining a rectangle as follows:

```
openshape rectangle
within rectangle {
    int width, height
    point pos
    line top,bot,left,right

    width = 10
    height = 20
    pos = {20,30}
    top = [pos+{0,height}, pos+{width,height}]
    right = [pos+{width,height}, pos+{width,0}]
    bot = [pos+{width,0}, pos}]
    left = [pos, pos+{0,height}]
}
```

This example shows a rectangle being defined in terms of its width, height and position. The four lines, which represent the rectangle, are in turn defined in terms of these values. If the width, height or position of the rectangle is changed then the lines also change and the rectangle is redrawn in its new position. As can be seen **DoNaLD** provides a set of arithmetic operators for use in definitions but it also has other geometric operators and functions. For more information on the available operators, see the appendix.

DoNaLD can handle several viewports at once, where a viewport is like a canvas on which objects can be drawn. Variables are assigned to a viewport depending on which viewport is selected at the time the variable was defined. You can change to a different viewport by using

the `viewport` keyword followed by the name of the viewport. The image in the viewport is dependant on those variable assigned to it. If any of those variables change then the image in the viewport is redrawn. This means that there are no explicit drawing commands since the displays are handled implicitly. Also since several viewports can be handled, only one script is needed to specify many displays.

2.2.3 SCOUT

This is a definitive notation for **SC**reen lay**OUT**, hence the name. It is used for describing the geometry and layout of windows on a display screen using definitions. Since it defines the layout of the screen, it provides an interface between the output of other definitive notations and the actual display. For instance, a window can display the image from one of **DoNaLD**'s viewports. In **TkEden**, this notation indicates how windows containing images are placed in the display window of the system. It is useful to split the display into windows so that different types of images can be produced by different notations and each image may have different attributes (such as colours and co-ordinate systems). In addition, since different images are displayed in different windows, it provides a means of detecting interaction events (e.g. pressing a mouse or keyboard button) and which window they occurred in. This allows the development of simple graphical interfaces with each window handling its own interactions.

In a similar way to **DoNaLD**, this notation consists of variables, which are defined by expressions possibly containing other variables. All such definitions set up dependencies between the target and source variables, which makes this a pure definitive notation. **SCOUT** variables represent displays, windows and components of windows and their values describe these abstract objects. The type of these variables indicates what kind of object it represents and although the variables do not need to be declared, they have to remain of the same type. Of the types available, there are numerical types such as `integer` and `string`; types for describing window components such as `point`, `box` and `frame`, and; the actual `window` and `display` types. The numerical types are similar to the equivalent ones in **EDEN** and hold the same type of values. Points are equivalent to the point type in **DoNaLD** and represent a position on a display. They are defined in term of screen co-ordinates and can be constructed by giving the x and y co-ordinates in between braces. A box represents a rectangular region of the display and holds the position of the top left and bottom right corners of the region. Variables of this type are constructed as a list containing two points (which represent the two corners). A frame represents multiple regions and is just a list of boxes. Some examples of these types age given below:

Points:	`{100,200}` and `{x,y}`	(x and `y` are integer variables)
Boxes:	`[{10,20},{30,40}]` and `[p,q]`	(p and `q` are points)
Frames:	`([{1,2},{20,30}],[box1,box2])`	

A window specifies a region in which data can be displayed in some manner. This type can be described by the four following concepts:

- Type — This is the type of window and indicates what type of image it contains. The two main types are `donald` and `text` windows.
- Region — This represents the area of screen that the window covers. The image of the window will be placed in this region.

- Content Specifies what is to be displayed in the window.
- Attributes The attributes of the displayed image. Describes how it will be displayed.

The following gives an example of `text` and `donald` window variables, and shows how they are specified.

Text Window

TYPE: Text

REGION: Given by a frame variable

CONTENT: Given as a string variable, which is rendered into the image

ATTRIBUTES: Values that indicate the text and background colour; the windows border size and type, and; the alignment of text in the window

EXAMPLE:

```
window button = {
        type: TEXT
        frame: ([button_pos, 1, 8])
        string: "STOP"
        bgcolour: "red"
        fgcolour: "black"
        border: 2
        relief: "raised"
        alignment: CENTRE
        sensitive: ON
}
```

This definition creates a text window, called 'button', which is located at button_pos (a point value), is one character high and 8 characters wide. The image displayed in the window is the centred text, STOP, displayed in black on a red background. The window also has a raised border of width 2.

Donald Window

TYPE: Donald

REGION: Given by a box variable

CONTENT: Given by the name of a **DoNaLD** viewport. The objects in the viewport are drawn in this window.

ATTRIBUTES: Values that give the co-ordinate system of the viewport; the foreground and background colours, and; the border size and type

EXAMPLE:

```
window display = {
    type: DONALD
    box: [topleft, botright]
    pict: "VIEW"
    xmin: 0
    xmax: 100
    ymin: 0
    ymax: 100
}
```

This definition creates a donald window as a rectangle whose top left corner depends on the point `topleft` and bottom right corner depends on `botright`. It displays a picture, which consists of objects in the viewport called '`VIEW`'. In addition, the co-ordinates are set up so that the x and y ranges of the viewport, from 0 to 100, are mapped into the window.

A display variable holds an ordered list of windows and indicates which windows are mapped to that display. If any two windows overlap in a display then the window earlier on in the display's list is drawn over the other. Display variables are constructed from a list of variables as `< window1 / window2 / ... / windown >`. In **TkEden** there is a pre-declared display variable called '`screen`' that represents the actual display window. Any **SCOUT** windows, which are in this displays list, are drawn onto the screen.

Although this notation is primarily for providing a specification of the display layout it also provides arithmetic, vector and list operators. In addition, it provides a means of dependency control through the `if` *`condition`* `then` *`expression`* `else` *`expression`* `endif` statement. This is not used as a flow of control expression; instead, it allows a variable to be defined by different expressions depending on a condition. An example of it use is:

```
string s_text = if s==1 then "s = 1" else "s <> 1" endif;
```

If the definition of a window includes the '`sensitive: ON`' statement then the window is sensitive to mouse and keyboard actions. When a mouse button is pressed and its pointer is in this window, an **EDEN** variable is created. This variable is given the name of the window followed by '`_mouse`' or, if it is a text window, '`_mouse_`*`box-number`*'. The variable is of type list and holds five values, which are the button number, type of action, state of keyboard, pointer x and y position in the window co-ordinates. If a key was pressed instead of a mouse button then the mouse part of the variable name is replaced by `key` and the first value of the list represents the key that was pressed. Using these automatic definitions, any interaction within a window can be detected and so an interface can be designed using **SCOUT** and **DoNaLD**.

2.3 Modelling with Definitive Notations

Models of real world objects are best described by their state and the relationships between individual objects. Therefore, when describing these objects in a computer-based model, the emphasis is upon defining each object's state in term of that of other objects. These definitive notations lend themselves well to this state based modelling, since the programmer is free to give abstract definitions of objects without worrying about the updating of each object's state. In essence, the programmer only has to state what is being modelled and not how to model it. In addition, because objects are automatically updated in term of their definitions, a definitive model can cope with an evolving environment. This allows a different approach to development, which encompasses observation, interaction and experiment. Modelling of this nature is generally known as *empirical modelling* and is based on modelling state as directly experienced rather than behaviour as circumscribed.

The first stage in any sort of modelling it to take the system being modelled and break it up into individual objects. These objects then need to be represented somehow by the data structures of the language being used. It is best to use structures that are closest in concept to the objects themselves since they lead to a simpler representation and more natural interface between objects. The next step is to identify relationships and interactions between objects. With definitive notations, these can be represented as dependencies between the variables that represent the objects. When using empirical modelling techniques these steps form an iterative task where, at first, only a small number of objects and dependencies are defined. Through observation and experiment, new objects and dependencies can be added which brings the model closer to the actual system being modelled.

When modelling with **TkEden**, objects can be specified in any of the three notations. Since each notation has its distinct flavour, it is best to represent the object in the notation most suited for it. This means that graphical objects are best defined in **DoNaLD**; conceptual objects in **EDEN,** and **SCOUT** is best used for describing the presentation of objects and how the user can interact with them. Since the built in types of these notations are often too primitive for the representation of objects, it is often necessary to represent them as lists or other structured types. It is important that the elements of these structured types are used to specify the attributes of the object and are the most appropriate for doing so. In addition, you should only represent relevant attributes and any dependencies need to be exploited. This is so that there is no repetitive or redundant information and all data is correct and up to date. It is also a good idea in **EDEN**, to provide functions for manipulating these types in order to make definitions as simple as possible.

In these notations, definitions are used for specifying a relationship between objects. They provide a way of telling the model that this relationship should always hold. Dependencies are fundamental in this kind of model, but it is also useful to have conceptual objects that interact with or act on various objects. These are called *actors* and can be created using **EDEN** action specifications. An actor then depends on the state of some object or objects and should the state change, the actor then performs its function. Since an actor is procedural in nature, it can perform any task that a user interacting with the system could. In particular they can change the value of one or more variables and even redefine dependencies between variables (this includes creating new dependencies). This gives the model the ability to redefine itself and makes this kind of modelling very powerful.

An important element of any real-time model is a clock. Without any form of clocking, all the definitions in a system would be evaluated and the system would wait until the state of any variable changes. To keep the state of the variables changing requires an actor to perform modifications to certain variables. This actor needs to be triggered and so requires a periodic change of variable on which it depends. An example of an **EDEN** action, which provides a periodically changing variable, is as follows:

```
proc clocking : clock {
        todo("clock++;");
};
```

This action updates a variable called 'clock' by incrementing it. Since the action also depends on clock then it will be called again and to prevent it being called instantly, the increment statement is placed on the 'to do' list. The 'to do' list is a list of statements that are to be executed after all definitions and actions have been dealt with. Now any actor that depends on clock will be executed periodically and the state of the model will continuously evolve.

APPENDIX A

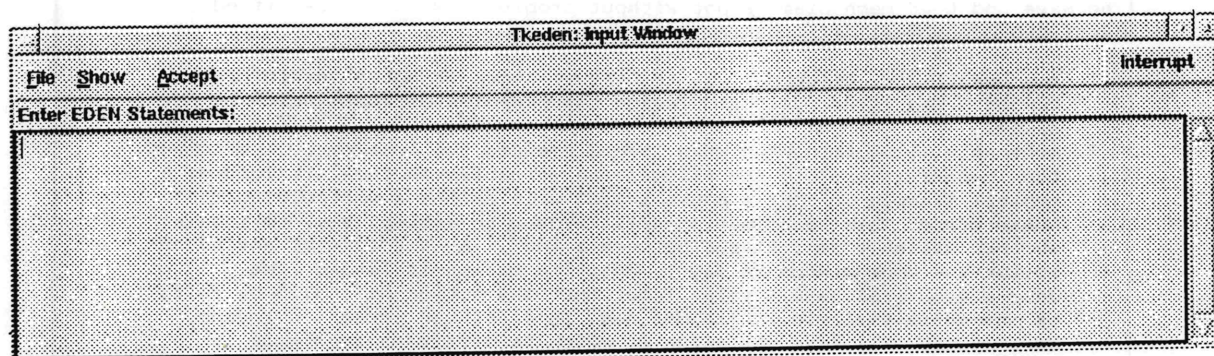
TkEden Reference

TkEden needs to be run in the X windows environment and can be invoked by the following UNIX command:

```
Tkeden [file-list]
```

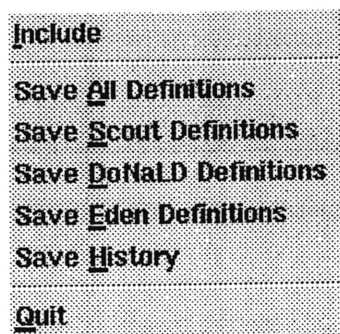
Where the file list is a list of filename which contain scripts of definitions in the **EDEN**, **DoNaLD** or **SCOUT** syntax.

Upon loading the following window is displayed along with the output window for the **SCOUT** display variable `screen`:



Definitions in the three notations can be input directly into this input window and accepted by clicking on the 'Accept' button. The 'Interrupt' button is provided to stall the dependency maintainer in case an error causes it to 'run away'.

The 'File' button brings up a menu of the following file options:

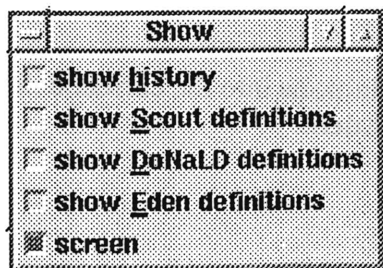


This menu allows you to include a file containing definitions into the current definitive store, to save the state of the definitive store and to quit **TkEden**.

Saving and loading the definition store

```
The main aim of the save and load facility is to save the current stage of
script development so that it can be worked on in a separate session.

To save to a file, you can use either the save menu options under the File
main menu, or choose the save option in the definition views.

Saving DoNaLD definitions or the history has no options other than choosing
file name. Saving Eden, Scout or all definitions, however, provides an
option for Save as reusable definitions. When selected (default),
definitions will be saved in such a way that the file can be loaded back.
When selected:

    * for Scout, there will be variable declarations added to the top of the
      file.
    * for Eden, the definitions generated by the DoNaLD and Scout translators
      will not be saved, neither will the system definitions as they will be
      automatically included each time the system starts up.
    * for All definitions, DoNaLD and Scout definitions will be saved first
      followed by Eden definitions.

The Save and Load mechanism is not without problems. We have identified at
least two conceptual difficulties with it:

  1. All DoNaLD loci disappear. It is because locus is not a `definitive'
     concept. The locus is stored inside the graphics database which is not
     saved.
  2. Actions will be triggered when loaded because the interpreter perceives
     the `redefinitions' to the triggerring variables as they are loaded. A
     typical example is that a mouse action will define a variable to the
     mouse state. This redefinition will trigger an action, say, to toggle
     another variable door_status between 1 and 0. At the time of saving the
     script the door_status is say 1. When the script is loaded, the
     door_status will have a value 1 but then the action will also be fired
     as the mouse variable is (re)defined. So after the script is loaded,
     door_status will have the value 0, which is different from what is
     saved.

In a definitive system, the visual state is definitively linked to the
script of definitions, or the definitive state. Therefore, loading a script
should regenerate the visual state at the time of saving. However, locus
breaks this definitive relationship between definitive state and the visual
state. A seemingly obvious solution would be to save both the definitive
state and the visual state, and on retrieval, disable the Eden evaluation
mechanism. This will result in including a lot of implementation-dependent
visual information in the saved file, which is not a desirable way of
handling non-definitive features.

More appropriately, we should avoid using loci. Using DoNaLD graphs should
minimise the need for it.

We have some ideas of tackling the second problem. At the moment, we have to
live with it.
```

The second button along the menu bar of the **TkEden** input window brings up a dialog box for opening up different views:



Viewing and saving history

Tkeden remembers the history of interaction. This includes the scripts entered through the input window and the definitions generated by the mouse or key actions in the Scout windows. Error messages are also considered as part of history.

The history can be examined by selecting the Show/show history option. When the system catches an error, the history view will automatically pop up with the error message shown at the rear.

The history can be saved in a file by selecting the File/save history option or the Save option in the history view.

A copy of the history is also saved automatically to the file named .tkeden-history in your home directory. It is a preventive measure for unexpected clashes or deadlocks of the system. Since this file is cleared every time tkeden is run, it is a good practice to use the Save option explicitly if you want to keep history.

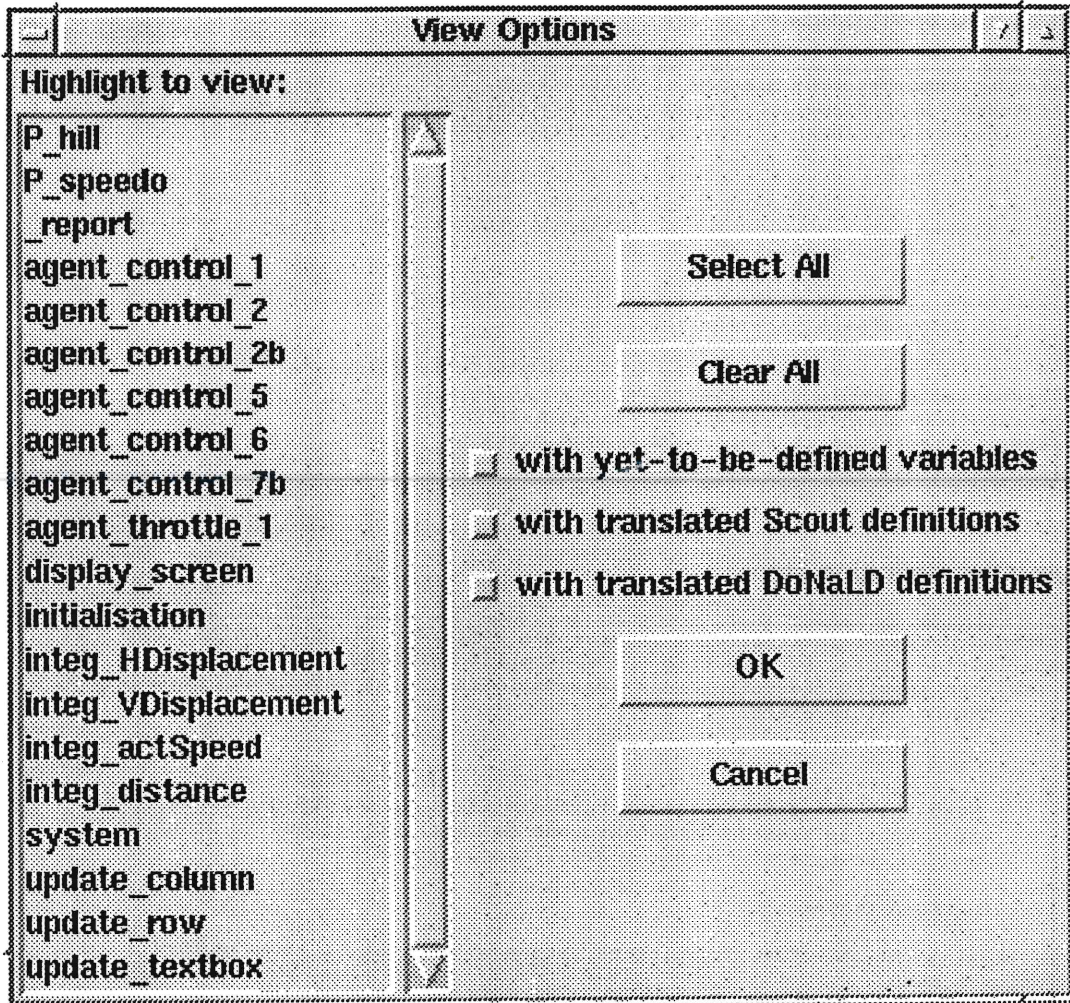
Viewing the definition store

Tkeden provides facilities to examine the different kinds of definitions in definition store. By selecting the appropriate options in the Show menu, the user can examine the definitions storing in the Scout and DoNaLD translators and the Eden interpreter.

* Note that the definitions captured in Scout or DoNaLD can be different from what is stored in Eden because definitions defined in Scout or DoNaLD can be overwritten by redefining through Eden.

Viewing Eden Definitions

Selecting the show Eden definitions option will give you another View Options menu.



with yet-to-be-defined variables

Force the viewer to display the definitions of those variables yet-to-be-defined. These variables bear undefined values (@), so the definitions shown are of the same form as a variable defined as @. However, variables defined as @ and not yet defined mean something different to Eden in relation to action triggering. If this option is not selected, those variables defined as undefined values will still be displayed.

with translated Scout definitions

Those definitions generated by the Scout translator will be displayed in red.

with translated DoNaLD definitions

Those definitions generated by the Scout translator will be displayed in blue.

the "agent" list

Select an agent by pressing the first mouse button. First mouse button with the Shift key will extend the selection. First mouse button with the Control key will toggle the selection. You can also use the Select All or the Clear All button to select all agents or clear all selections.

Tkeden associates with each Eden variable (including functions and procedures) a cause (the agent) of last redefinition. The causes (agents) may be:

system	The variables, functions and procedures preloaded every time tkeden is run, with the exceptions of the functions

	column(), row() and textbox() (used by Scout) that are defined by Eden actions update_column(), update_row() and update_textbox() representatively. These functions will carry these action names as their master label.
initialisation	Similar to system, but these symbols come from the files specified in the command line when tkeden is invoked. The same situation as in the last case, any symbols defined by Eden actions during this phase will not render the initialisation label but the Eden action name label instead.
input	Any symbols directly modified as a result of the user input through the input window.
interface	Any symbols directly modified as a result of key and mouse button actions through the Scout interface.

or any Eden action.

In the definition view, definitions are sorted first by the agent name, then sort by the type of definition and then in alphabetical order of the variable names. The types are sorted in the sequence of explicit definitions, formula definitions, functions and procedures or actions.

Tkeden: Eden Definitions

Save Find Rebuild Close

```
proc P_speedo_node2_8 : _speedo_node2_8, A_speedo_node2_8, SPEEDO {
      plot_line(SPEEDO, &_speedo_node2_8, &A_speedo_node2_8);
}
AGENT _report
_brakF = -0;
_curSpeed = 0;
_gradient = 0;
_gravF = 24525;
_sampleThrottlePos = 0;
_tracF = 0;
_windF = -2e-07;
sampleClk = 0;
sampleHDisp = @;
sampleVDisp = @;
AGENT agent_control_1
onBtn_prev = 1;
AGENT agent_control_2
cruiseStts = 3;
offBtn_prev = 2;
onBtn = 1;
```

Viewing DoNaLD Definitions

The DoNaLD definitions view will show all the definitions stored in the DoNaLD translator. It will first show the declarations of all the variables then their definitions. Similar to the Eden definition view, the agents defining the variables will also be shown.

Viewing Scout Definitions

The Scout definitions view will show all the definitions stored in the Scout translator. Unlike the Eden definition view or the DoNaLD definition view, showing of the agents defining the variables is a feature to be implemented.

The Save Option

* See Save and Load

The Find Option

To search for any words in the current view.

The Rebuild Option

Reconstruct the definition view. In the case of viewing Eden definitions, the user needs to select the view options again.

```
        primary-expression
        - expression
        ! expression
        & lvalue
        expression #
        ++ lvalue
        lvalue ++
        -- lvalue
        lvalue --
        [ expression-listopt ]
        expression binop expression
        expression ? expression : expression
        lvalue asgnop expression

primary-expression:
        lvalue
        ( expression )
        primary-expression [ expression ]
        primary-expression ( expression-listopt )

lvalue:
        identifier
        $
        $number
        lvalue [ expression ]
        * expression
        ` expression `
        ( lvalue )

expression-list:
        expression
        expression , expression-list

statement:
        expression ;
        function-definition
        formula-definition
        action-specification
        dependency-link
        query-command
        compound-statement
        insert lvalue , expression-1 , expression-2 ;
        append lvalue , expression ;
        delete lvalue , expression ;
        shift lvalueopt ;
        if ( expression ) statement
        if ( expression ) statement else statement
        while ( expression ) statement
        do statement while ( expression ) ;
        for ( expressionopt ; expressionopt ; expressionopt ) statement
        switch ( expression ) statement
        case constant : statement
        default : statement
        break ;
        continue ;
        return expressionopt ;
        ;

compound-statement:
        { statement-listopt }

statement-list:
        statement
        statement  statement-list

function-definition:
        function-declarator  function-body

function-declarator:
        func identifier
        proc identifier

function-body:
        { para-aliasopt  local-var-declopt  statement-listopt }

para-alias:
        para identifier-listopt ;
```

```
function-body:
        { para-aliasopt  local-var-declopt  statement-listopt }

para-alias:
        para identifier-listopt ;

local-var-decl:
        auto identifier-listopt ;

identifier-list:
        identifier
        identifier , identifier-list

action-specification:
        function-declarator dependency-list function-body

dependency-list:
        : identifier-list

dependency-link:
        identifier ~> [ identifier ] ;

query-command:
        ? lvalue ;
```

Operator Precedence

The primary-expression operators

```
()          []        ``
```

have highest priority and group left-to-right. The unary operators

```
*     &     -     !     not     #     ++     --
```

have priority below the primary operators but higher than any binary operator, and group right-to-left.

Binary priority decreasing as indicated below.

```
binop:
        *     /     %
        +     -     //
        >     <     >=     <=
        ==    !=
        &&    and
        ||    or
```

The conditional operator

```
?:
```

has priority lower than the binary operators, and groups right-to-left.

Assignment operators all have the same priority, and all group right-to-left.

```
asgnop:
        =     +=     -=
```

DoNaLD Syntax

```
Variable names

x
     refers to the variable x in the immediate context
~/x
     refer to the variable x in the context one level up
/x
     refers to the variable x in the root (topmost) context
x!
     refer to an Eden variable

Data types

 Type      Example
 int       34
 real      10.0
 char      "abc"
 boolean   true, false
 point     {50, 100}, {modulo @ angle}
 line      [{10,10}, {80, 90}]
 circle    circle(centre, radius), circle({500,500}, 400)
 ellipse   ellipse(centre, major, minor)
 image     I!ImageFile("gif", "hill.gif")
 label     label L1, L2
           L1 = label("abc", {100,100})
           image img
           L2 = label(img, {200,100})
 openshapeopenshape cross
           within cross {
             line l1, l2
             l1 = [{10, -10}, {-10, 10}]
             l2 = [{10, 10}, {-10, -10}]
           }
 shape     shape S
           S = trans(cross, 100, 200)
 graph     func sqr { return $1 * $1; }
           %donald
           graph g
           within g {
             x<i> = 1.0 + <i>
             f<i> = sqr!(x<i>)
             nSegment = 9
             node = [circle:circle({x<i> * 100, f<i> * 10}, 10);
               label:label("(" // rtos(x<i>, ".0f") // "," // rtos(f<i>,
           ".0f") // ")", {x<i> * 100, f<i> * 10 + 20})]
             segment = [line:[{x<i-1>*100, f<i-1>*10}, {x<i>*100, f<i>*10}]]

           }

Functions

arithmetic operators and functions

+
-
*
div (division)
mod
sqrt (square root)
log
exp
trunc (convert real to integer)
float (convert an integer to real)
rand (random number generator)
     - int rand(int) or real rand(real)
     if the argument is an integer, say 10, rand(10) returns a random number
     ranged from 0 to 9 inclusive;
     if the argument is a real, say 1.0, rand(1.0) returns a random number
     ranged from 0.0 to 1.0 inclusive.

trigonometry functions
```

```
sin
cos
tan
asin
acos
atan
     - accept only real arguments

relational operators

&& (and)
|| (or)
! (not)
<
<=
== (equal)
>
>=

geometric functions

point midpoint(line)
     - mid-point of a line
point intersect(line, line)
     - intersection point of two lines
line perpend(point, line)
     - the perpendicular line intersecting the point
real dist(point, point)
     - distance between two points
bool intersects(line, line)
     - whether the lines intersect
bool separates(line, point, point)
     - whether the line separates the two points
bool includes(circle, point)
     - whether the point is inside the circle
bool incident(line, point)
bool incident(circle, point)
     - whether the point coincide with the path
bool pt_betwn_pts(point, point, point)
     - whether the 2nd point is within the box bounding the 1st and the 3rd
     point
bool colinear(point, point, point)
     - whether the points are colinear
bool distlarger(point, point, value)
bool distlarger(line, point, value)
bool distsmaller(point, point, value)
bool distsmaller(line, point, value)
     - whether the distance from the point to the point/line is larger than
     or smaller than the specified value

shape transformations

trans(shape, x, y)
     - translate a shape (or openshape)
scale(shape, ratio)
     - scale a shape (or openshape) wrt the origin of the coordinate system;
     would not change the font size or the image size of labels
rot(shape, point, angle)
     - rotate a shape round a point by certain angle

string functions

//
     string concatenation
itos(int)
     - integer to string conversion
rtos(real, format-string)
     - real to string conversion. format-string as would be required by the
     C function fprintf().

image functions

Example:

%donald
real xscale, yscale
```

```
image zoom, source
source = I!ImageScale(I!ImageFile("gif", "logo.gif"), xscale, yscale)
zoom = I!ImageScale(I!ImageCut(source, 20, 0, 200 * xscale, 200 * yscale),
        xscale, yscale)

label imgzoom, imgsrc
imgzoom = label(zoom, {500, 200})
imgsrc = label(source, {500, 700})

other functions

circle(center, radius)
ellipse(center, extreme_point1, extreme_point2)
label(string, position)
if boolean_expression then expr else expr
.1, .2
     - 1st and 2nd points of a line or 1st and 2nd coordinate of a point,
     for example:

     line l
     l = [{0,0}, {100,100}]
     point p
     p = l.2          # i.e. {100,100}
     real x
     x = p.1          # i.e. 100

.x, .y
     - the projection of a point onto the x- and y- axles, e.g.:

     point p, q
     p = {100, 200}
     q = p.x          # i.e. {100, 0}

Graphs

   * A graph comprises of nSegment number of segments and nSegment + 1
     number of nodes. But there may be multiple visualisation of the same
     node or segment.
   * x<i> and f<i> have to be defined even though they may not be used in
     the node / segment definitions.
   * node and segment have to be defined. Should no visualisation associate
     with any of node or segment, define the variable to []
   * Due to the limitation of DoNaLD that once a variable is declared as one
     type, it cannot be redeclared to another type. So one the first
     visualisation of the node is circle, any subsequent redefinition to
     node must have a first element declared as a circle. However, you need
     not give any definition to it.
   * A DoNaLD graph will declare a series of node and segment entities
     accessible in DoNaLD. The nodes are named as node1_1, node1_2, etc. and
     the segments are named as segment1_0, segment1_1, etc., where the first
     number represent which visualisation of the entity and the second
     number concerns which node or segment. You can redefine the individual
     nodes and segments without any complaint from the system (BUG?).

Attributes

The attribute variables are EDEN variables, and are named after the
translated name of the DoNaLD variables in EDEN. The attribute variable name
for the DoNaLD variable Obj/line1 would be A_Obj_line1. Attributes are of
the form:

"attr1=value1,attr2=value2..."

The set of attributes available may vary from one implementation to another.
For tkeden, legal attributes are:

color
     applicable to: any shape
     acceptable values: any X-Windows recognised colour name + transparent
linewidth
     applicable to: line, arc, circle, shape
     acceptable values: positive integers, 0 = minimum line width
linestyle
     applicable to: line
     acceptable values: dotted, dashed, solid
arrow
```

```
     acceptable values: dotted, dashed, solid
arrow
     applicable to: line
     acceptable values: first, last, both, none followed by 2 spaces)
locus
     applicable to: any shape
     acceptable values: true, false
fill
     applicable to: any shape
     acceptable values: solid, hollow

Viewports

viewport VIEW1

means that from that line onwards the graphical objects to be declared (not
defined) will be associated with the viewport VIEW1.

Comments

DoNaLD uses # as the comment symbol. Comment starts from # to the end of the
line.

Escape to EDEN

There are two ways of escaping from DoNaLD to EDEN.

  1. A line beginning with ? will be passed to EDEN directly.

  2. Switching to EDEN and back
```

SCOUT Syntax

```
Scout describes a display as (potentially) overlapping windows. For example,
if display disp is defined as

disp = < win1 / win2 >

this means that display disp consists of two windows win1 and win2, should
win1 and win2 overlap, win1 overlays win2.

window = region X content X attributes

For a text window,

region (called a frame) = list of boxes
     The string is filled into the first box, the remaining characters are
     filled into the second box and so on.
content = a character string
     - a character string
attribute = { fgcolour, bgcolour, border, bdcolour, relief, alignment }
     These attributes indicate the colour of the text string, the colour of
     the background, whether the boxes have borders, the border colour, the
     kind of border and the alignment of strings in relation to the boxes
     respectively. The acceptable reliefs are raised, sunken flat, ridge and
     groove.

For a DoNaLD window,

region = a box
     - a box
content = a drawing (name of the drawing)
     - a drawing
attribute = { xmin, ymin, xmax, ymax, fgcolour, bgcolour, border, bdcolour
relief }
     xmin, ymin, xmax, ymax defines the coordinate system of the drawing;
     fgcolour and bgcolour defines the foreground and background colour,
     border determines whether to draw borders of the box, bdcolour defines
     the border colour and relief how the border is drawn.

For an image window,

region = a box
```

- an image

attribute = { bgcolour, border, bdcolour relief }

bgcolour defines the colour to be filled when the image is not large enough to cover the whole area of the window. border determines whether to draw borders of the box. bdcolour defines the border colour and relief how the border is drawn.

The sensitive attribute is common to all three types of windows. It is used to declare that a window is sensitive to mouse and keypress actions. When this attribute is ON, a mouse action or a keypress action within the region of this window will cause a definition to be generated. If a mouse action occurs in a window and it is a DoNaLD or image window, then the window name concatenated with _mouse will be the name of the variable to be defined; if it occurs in a text window, the window name concatenated with _mouse_ followed by the box number will be the variable name. The value assigned to the appropriate variable records the nature and the location of the mouse action. It is a 5-tuple of (button, type, state, x, y) where

button
: the button number pressed or released;

type
: the button action (4 = pressed, 5 = released);

state
: the state before the button action occurred (shift (+1), caplock (+2), control (+4), meta (+8) and was-pressed (+256)). For example, if a button is released while the shift and control keys are depressing, state will be 1 + 4 + 256 = 261;

x, y
: the x- and y- coordinates of the mouse in the coordinate system of the window in which the mouse action occurred.

As with mouse events, a stroke on the keyboard will generate a definition. Instead of _mouse or _mouse_ followed by a box number, the variable name of the generated definition will end with _key or _key_ followed by a box number. The value defined will also be a 5-tuple: (key, type, state, x, y), where key is the ascii code of the key pressed.

Data Types and Operators

Operators: +, -, *, /, % (remainder), - (unary minus)
    Meaning: Normal integer arithmetics
    Example: 10 % 3 gives 1

Constructor: {x, y}
    Meaning: Construct a point
    Example: {10, 20} is a point with x-coordinate 10 and y-coordinate 20

Operators: +, -
    Meaning: Vector sum and vector subtraction
    Example: {10, 20} - {20, 5} gives {-10, 15}

Selector: .1, .2
    Meaning: Return the 1st (x-) coordinate and the 2nd (y-) coordinate respectively
    Example: {10, 20}.1 gives 10

Constructor: {field-name: formula, field-name: formula, ..., field-name: formula}
    Meaning: Constructing a window
    Example: { type: DONALD, box: b, pict: "FIGURE1"}

Selector: .field-name
    Meaning: Return the value of the field
    Example: { type: DONALD, box: b, pict: "FIGURE1"}.box gives b

Constructor: < W1 / W2 / ... / Wn >
    Meaning: Constructing a display, if W1 and W2 overlap, W1 overlays W2
    Example: < don1 / don2 >

List function: insert(L, pos, exp)
    Meaning: Insert the expression exp in the position pos of list L
    Example: insert(<w1, w2, w3>, 2, new) gives <w1, new, w2, w3>

List function: delete(L, pos)
    Meaning: Delete the posth element of list L
    Example: delete(<w1, w2, w3>, 2) gives <w1, w3>

Operator: if cond then exp1 else exp2 endif
    Meaning: if cond gives non-zero value (true) then returns exp1 else returns exp2, in this context, exp1 and exp2 must have the same type.
    Example: if 1 then "Open" else "Close" endif gives "Open"

```
statement ::
    declaration | definition
declaration ::
    type_name var_list ;
```

```
     declaration | definition
declaration ::
     type_name var_list ;
type_name ::
     string | integer | point | box | frame | window | display
var_list ::
     var | var_list , var
var ::
     string_var | integer_var | point_var | box_var | frame_var
     | window_var | display_var
definition ::
     [ string ] string_var = string_exp ;
     | [ integer ] integer_var = integer_exp ;
     | [ point ] point_var = point_exp ;
     | [ box ] box_var = box_exp ;
     | [ frame ] frame_var = frame_exp ;
     | [ window ] window_var = window_exp ;
     | [ display ] display_var = display_exp ;
string_exp ::
     string
     | string_var
     | string_exp // string_exp
     | strcat( string_exp , string_exp )
     | substr( string_exp , integer_exp , integer_exp )
     | itos( integer_exp )
     | window_exp .string
     | window_exp .pict
     | window_exp .font
     | window_exp .bgcolor
     | window_exp .fgcolor
     | window_exp .bdcolor
     | window_exp .relief
     | if integer_exp then string_exp else string_exp endif
image_exp ::
     image_var
     | ImageFile( string_exp , string_exp )
     | ImageScale( image_exp , integer , integer )
     | if integer_exp then image_exp else image_exp endif
integer_exp ::
     integer
     | integer_var
     | integer_exp .c
     | integer_exp .r
     | integer_exp int_op integer_exp
     | - integer_exp
     | ( integer_exp )
     | strlen( string_exp )
     | point_exp . integer_exp
     | window_exp .xmin
     | window_exp .ymin
     | window_exp .xmax
     | window_exp .ymax
     | if integer_exp then integer_exp else integer_exp endif
int_op ::
     + | - | * | / | % | == | != | > | >= | < | <= | && | ||
point_exp ::
     point_var
     | { integer_exp , integer_exp }
     | point_exp + point_exp
     | point_exp - point_exp
     | box_exp . direction
     | if integer_exp then point_exp else point_exp endif
direction ::
     n | e | s | w | ne | nw | se | sw
box_exp ::
     box_var
     | [ point_exp , point_exp ]
     | [ point_exp , integer_exp , integer_exp ]
     | frame_exp . integer_exp
     | window_exp . box
     | shift( box_exp , integer_exp , integer_exp )
     | intersect( box_exp , box_exp )
     | centre( box_exp , box_exp )
     | enclose( box_exp , box_exp )
     | reduce( box_exp , box_exp )
     | if integer_exp then box_exp else box_exp endif
```

```
     | ( box_list )
     | window_exp .frame
     | append( frame_exp , integer_exp , box_exp )
     | delete( frame_exp , integer_exp )
     | frame_exp & frame_exp
     | if integer_exp then frame_exp else frame_exp endif
window_exp ::
     window_var
     | { window_field_list }
     | display_exp . integer_exp
     | if integer_exp then window_exp else window_exp endif
window_field_list ::
     window_field | window_field_list , window_field
window_field ::
     type : { TEXT | DONALD | ARCA }
     | frame : frame_exp
     | string : string_exp
     | font : string_exp
     | box : box_exp
     | pict : string_exp
     | xmin : integer_exp
     | ymin : integer_exp
     | xmax : integer_exp
     | ymax : integer_exp
     | { bgcolour | bgcolor } : string_exp
     | { fgcolour | fgcolor } : string_exp
     | { bdcolour | bdcolor } : string_exp
     | border : integer_exp
     | relief : string_exp
     | alignment : justification
     | sensitive : { ON | OFF }
justification ::
     NOADJ | LEFT | RIGHT | EXPAND | CENTRE
window_list ::
     window_exp | window_list / window_exp
display_exp ::
     display_var
     | < window_list >
     | append ( display_exp , integer_exp , window_exp )
     | delete ( display_exp , integer_exp )
     | if integer_exp then display_exp else display_exp endif
```