Introduction to EDEN

From a practical perspective …

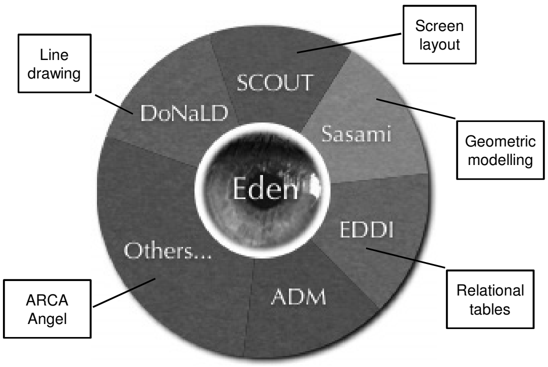
Background

EDEN interpreter due to Y W (Edward) Yung (1987)

Designed for UNIX/C environment
EDEN = evaluator for definitive notations

"hybrid" tool = definitive + procedural paradigms
… essential to drive UNIX utilities and hw devices

Extensions by Y P (Simon) Yung, Pi-Hwa Sun, Ashley Ward, Eric Chan and Ant Harfield



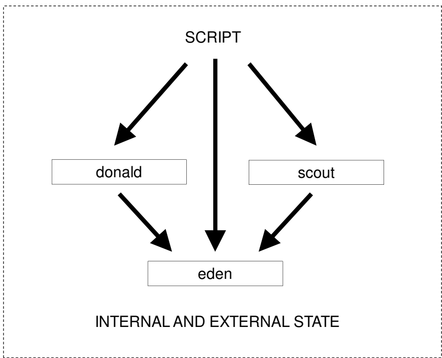
The EDEN interpreter as an engine /evaluator for definitive notations

The use of the word “definitive”

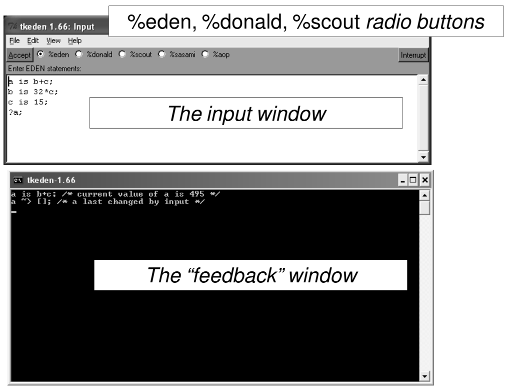
definitive = definition-based

a *definitive notation* = a notation within which definitions of variables can be made

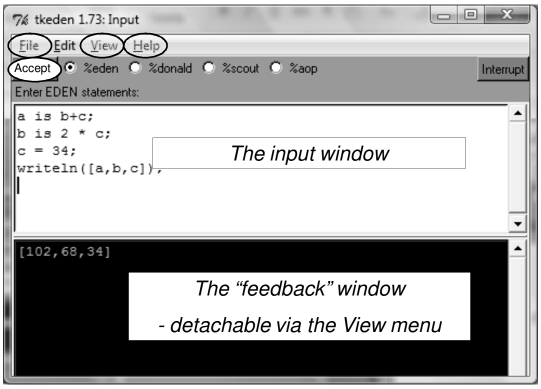
a *definitive script* = a set of definitions expressed in one or more definitive notations



The basic architecture of the EDEN interpreter



Interpreting the eden notation



Basic EDEN interaction

Use the File option to include scripts and to save the history of interaction

Use the View option to inspect the current contents of the script and the command history

Use the Help option to get quick reference information for eden, donald and scout

Use the Accept button (or alt-a) to process script in the input window

Use shortcuts (alt-p, alt-n) to recall previous input

Basic characteristics of EDEN 1

The eden notation uses C-like

- syntactic conventions and data types
- basic programming constructs:

    `for`, `while`, `if` and `switch`

Types: `float`, `integer`, `string`, `list`.

Lists can be recursive and need not be homogeneous in type. Comments are prefaced by ## or enclosed in /* .... */.

Basic characteristics of EDEN 2

Two sorts of variables in eden:

formula and *value* variables.

Formula variables are definitive variables.

Value variables are procedural variables.

The type of an eden variable is determined dynamically and can be changed by assignment or redefinition.

Programming / modelling in EDEN

The three primary concepts in EDEN are:

- definition
- function
- action

Informally

definition ~ spreadsheet definition

function ~ operator on values

action ~ triggered procedure

Definitions in eden

A formula variable v can be defined via

v is f(a,b,c);

EDEN maintains the values of definitive variables automatically and records all the dependency information in a definitive script.

Yellow text indicates eden keywords

Functions in eden

Functions can be defined via

```
func F
/* function to compute result = F(a,b,...,c) */
{
        para a, b, ..., c        /* pars for the function */
        auto result, x, y, …, z      /* local variables */
        <sequence of assignments and constructs>
        return result
}
```

Actions in eden

Actions can be defined via

```
proc P : r, s, …, t
/* proc triggered by variables r, s, …, t */
{
        auto x, y, …, z    /* local variables */
        <sequence of assignments and definitions>
}
```

Action P is triggered whenever one of its triggering variables r, s, … , t is updated / touched

Basic concepts of EDEN 1

Definitions are used to develop a definitive script to describe the current state: change of state is by adding a definition or redefining.

Functions are introduced to extend the range of operators used in definitions.

Actions are introduced to automate patterns of redefinition where this is appropriate.

Evaluator for DEfinitive Notations

Definitions are used to develop a definitive script to describe the current state: change of state is by adding a definition or redefining.

Functions are built-in for the operators in the underlying algebra of a definitive notation.

Actions are introduced to maintain the state of the graphical/perceptual entities specified by the definitive notation.

Basic concepts of EDEN 2

In model-building using EDEN, the key idea is to first build up definitive scripts to represent the current 'state-as-experienced'.

You then refine the script through observation and experiment, and rehearse meaningful patterns of redefinition you can perform.

Automating patterns of redefinition creates 'programs' within the modelling environment

Standard techniques in EDEN

Interrogating values and current definitions of variables in eden. To display:

- the current value of an eden variable v, invoke the procedure call

  `writeln(v)`

- the defining formulae & dependency status of v, invoke the query

  `?v;`

Typical EDEN model development

Edit a model in one window (e.g. using Textpad) and simultaneously execute EDEN in another

Cut-and-paste from editor window into interpreter window.

In development process, useful to be able to undo design actions: restore scripts of definitions by re-entering the original definitions.

To record the development history comment out old fragments of scripts in the edited file.

Managing EDEN files

Useful to build up a model in stages using different files.

Can include files using

```
include("filename.e");
```

or via the menu options in the input window.

Can consult / save entire history of interaction.

System also saves recent interaction histories.

Definitive scripts

Use scripts of definitions to represent state

Use redefinition to specify change of state

Scripts make use of definitive notations:

- DoNaLD - line drawing
- SCOUT - window layout
- ARCA - combinatorial graphs

Each notation is oriented towards a different metaphor

Definitive notations

Definitive notations are simple languages within which it is possible to formulate definitions for variables ("observables") of a particular type.

A definitive notation is defined by

- an underlying set of data types and operators
- a syntax for defining observables of these types.

Review/illustrate key features of DoNaLD and SCOUT

DoNaLD data types

Donald is a definitive notation for 2-d line-drawing

Its underlying algebra has 6 primary data types:

integer, **real**, **bool**ean, **point**, **line**, and **shape**

A **shape** = a set of points and lines

A **point** is represented by a pair of scalar values {x,y}.

Points can be treated as position vectors: they can be added (p+q) and multiplied by a scalar factor (p*k)

A **line** [p,q] is a line segment joining points p and q

DoNaLD operators

The DoNaLD operators include:

arithmetic operators:

+ * div float() trunc() if ... then ... else ...

basic geometric operators:

.1 .2 .x .y {,} [,] + *

dist() intersects() intersect()

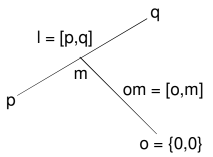
translate() rot() scale()

label() circle() ellipse()

A DoNaLD file should begin with a "%donald"

DoNaLD syntax – points and lines

declaring (NB) and defining points and lines
point o, p, q, m
line l
l = [p,q]
m = (p+q) div 2
line om
# new declarations can be introduced at any stage
o = {0,0}
om = [o,m]
.....



DoNaLD syntax – shapes

```
openshape S
within S {
        int m  # this is equivalent to declaring int S/m outside S
        point p, q
        openshape T
        p = {m, 2*m}
        within T {
                point p, q        # this point has the identifier S/T/p
                p, q = ~/q, ~/p
                # a multiple definition: p = ~/q and q=~/p
                # ~/... refers to the enclosing context for T
                # viz. S, so that ~/p refers to the variable S/p
                .....
        }
        ...
}
```


DoNaLD extras

Can define shapes in another way also: e.g.
    shape rotsquare = rotate(SQ,….)
where SQ is defined to be a square

The “within X { …” context is reflected in the input window in EDEN

A syntax error in a ‘within’ context resets to the root context …

… there are NO SEMI-COLONS (;) *in DoNaLD !!!*

SCOUT types

SCOUT is a definitive notation for screen layout

Its primary data type is the **window**
Other types include: **display** (collection of windows, ordered according top to bottom); **integer, point** and **string**.

Windows are generally used to display text or DoNaLD pictures.

SCOUT screen definition

Overall concept

a SCOUT script defines the current computer screen state
screen is a special variable of type *display*
the display is made up out of windows

Simplest definition of **screen** has the form
    screen = < win1 / win2 / win3 / win4 / win5 / .... >
where ordering of windows determines how they overlay

Alternatively can define **screen** as union of displays
    screen = disp1 & disp2 & disp3 & disp4 & ....

SCOUT window definitions

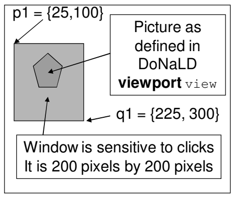
A SCOUT window definition takes the form
    window X = {
        fieldname1: …
        fieldname2: …
        …
    }
where the choice of *fieldnames* depends on the nature of the window content.

Defining a window to hold a DoNaLD picture

DoNaLD Window		
field name	*type*	*description*
type	content	Must be the value DONALD
box	box	The region in which the DoNaLD picture is shown
border	integer	Set the border width of the bounding box
pict	string	The name of the DoNaLD picture
xmin	point	
ymin	point	Show the portion of the DoNaLD picture
xmax	point	bounded by the points (xmin, ymin) and (xmax, ymax)
ymax	point	

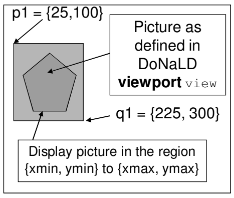
A simple SCOUT DONALD-window

```
point p1 = {25, 100};
point q1 = {225, 300};
window don1 = {
    box: [p1, q1],
    pict: "view",
    type: DONALD,
    border: 1
    bgcolor: “green”
    sensitive: ON
};
# locations of points are in pixels from top left of screen
#  coordinates of DONALD picture {0,0} to {1000, 1000}
```



Another SCOUT DONALD-window

```
window don2 = {
    box: [p1, q1],
    pict: "view",
    type: DONALD,
    xmin: zoomPos.1 - zoomSize/2,
    ymin: zoomPos.2 - zoomSize/2,
    xmax: zoomPos.1 + zoomSize/2,
    ymax: zoomPos.2 + zoomSize/2,
    border: 1
    sensitive: ON
}
```



Defining a window to hold text

Text Window		
field name	*type*	*description*
type	content	Must be the value TEXT
string	string	The string to be displayed
frame	frame	The region in which the string is shown
border	integer	Width of the border of the boxes of the frame
alignment	just	NOADJ, LEFT, RIGHT, EXPAND and CENTRE are the possible values to denote no alignment, left justification, right justification, left and right justification and centre of the text inside each box in the frame
bgcolour	string	Colour name for the background colour of the text
fgcolour	string	Colour name for the (foreground) colour of the text

A simple SCOUT TEXT-window

```
window doorButton = {
    frame: ([doorButtonPos, 1, strlen(doorMenu)]),
    string: doorMenu,
    border: 1
    sensitive: ON
};
string doorMenu = if _door_open then "Close
    Door" else "Open Door" endif;
```


SCOUT extras

When aspects of the screen are undefined by the SCOUT script, it will not be drawn / redrawn

Sensitive SCOUT windows generate definitions of associated mouseButton variables: they supply information about the mouse state and location & can be used to trigger EDEN actions

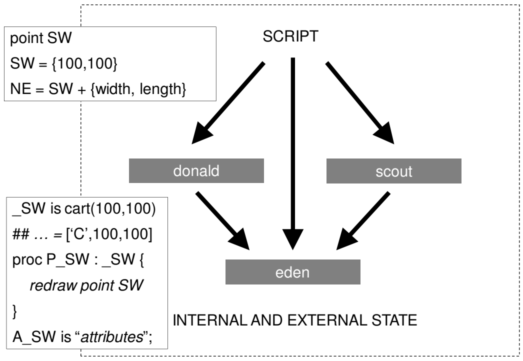
Mouse clicks show up in the command history

SCOUT & DoNaLD extras

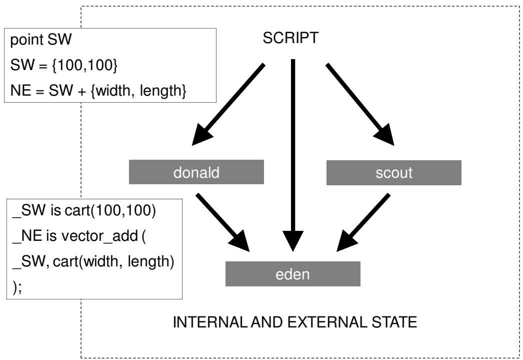
By default, a DoNaLD picture is displayed in a system generated SCOUT window, and has coordinates between {0,0} and {1000,1000}

SCOUT observables can be accessed in EDEN by the same names

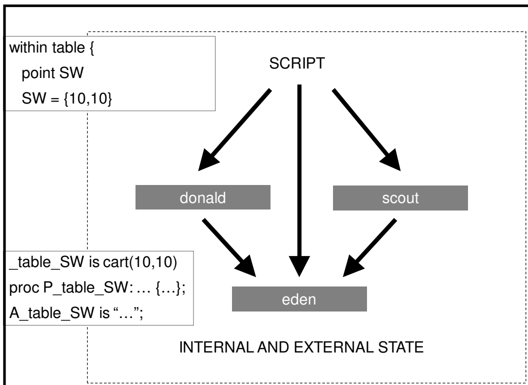
A DoNaLD observable X/t can be accessed in EDEN and SCOUT by _X_t etc.



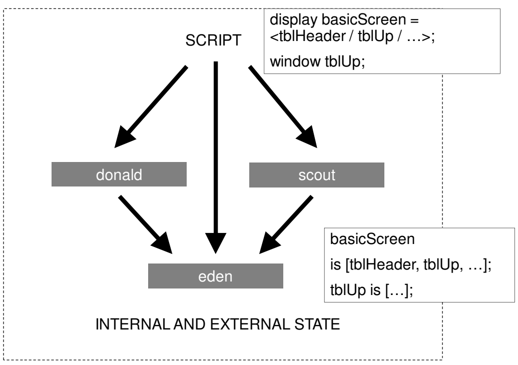
Translating donald to eden



Translating donald to eden



Translating donald to eden



Translating scout to eden

Examples of definitive notations

Notation	*Basis for underlying algebra*
eden	scalars, recursive lists, strings
donald	points, lines, shapes
scout	windows, displays (window = template + content)
arca	diagrams, vertices, incidences
sasami	polygonal meshes, renderings
eddi	relational database tables and views

Each notation is adapted to the metaphorical representation of different kinds of observable