Lecture M3: Introduction to eden programming

M3.1. Background

The eden interpreter was designed and first developed by Y W (Edward) Yung in 1987. It has been extensively used in prototyping since. Designed to work in a UNIX/C environment, the interpreter was originally customised each time a new application was conceived; this involved recompilation of eden with a built-in interface to other UNIX/C-related utilities. (For instance, c.eden is eden with curses.h functions incorporated.) The interpreter has since been adapted to allow other applications to be driven without customisation. The principle exploited is illustrated by EX, the interface used to link eden to X-Windows.

The eden interpreter was originally developed with the evaluation of definitive notations in mind []. It provides a useful "hybrid" programming tool that allows definitive and procedural paradigms to be combined. (It is essential to link definitive systems with procedural mechanisms, since this is the conventional way in which most UNIX utilities and hardware devices are programmed.)

Some demonstration programs in eden are in ~wmb/public/demo/EDEN. This lecture gives only a brief introduction to eden: [1] has more details.

M3.2. Overview of basic eden characteristics

The eden syntactic conventions and data types are similar to those in C. The basic programming constructs are C-like: the for, while, if and switch. The main types for variables are float, integer, string and list. Lists can be recursive and need not be homogeneous in type. Comments are enclosed in /* .... */ parentheses.

There are two sorts of variables in eden: formula and value variables.

Formula variables are definitive variables.

Value variables are traditional procedural variables.

Variables do not have to be declared – the type of an eden variable is determined dynamically, and can be changed by re-assignment or re-definition.

The eden interpreter is a powerful programming tool that has to be used in a disciplined way for best results. Many eden programs can be built from three abstract programming features: definitions, functions and actions:

- definitions

  A formula variable v can be given a definition via

  v is f(a,b,c);

The eden interpreter maintains the values of definitive variables automatically, and records all the dependency information associated with a definitive script.

- functions

  The user can define special-purpose functions via

```
func fn /* function to compute result = fn (p1,p2,...,pn) */
{
        para p1, p2, ...., pn              /* parameters for the function */
        auto result, a1, a2, ...., am              /* local variables */
        <assignments and definitions>
        return result
}
```

  Functions specified in this way can appear on the RHS of formulae.

- actions

  An action is a "triggered procedure", specified via

```
proc pti : t1, t2, ... , tn /* procedure triggered by t1, t2,..., tn */
{
        auto result, a1, a2, ...., am              /* local variables */
        <assignments and definitions>
}
```

  An action is a generalised procedure (for a procedure, n=0). An action is a procedure that is triggered whenever one of its triggering variables ti is updated, whether or not the value of ti is changed in the update.

M3.3. Illustrative examples of eden use

1. How to use eden to implement a definitive notation (cf YPY thesis)

A definitive script in eden does not have a direct visualisation as a donald or scout script does. To interpret a donald script in eden, it is necessary to associate display actions with variables. Despite this, translation of a donald script into eden simplifies the interpretation of donald greatly, as all the maintenance of definitive variables is done automatically.

When generating an eden script, it is useful to be able to interrogate the values and current definitions of variables. The procedure writeln() can be used to display the current value of an eden variable, and the query

```
?v;
```

will return details of the defining formulae and dependency status of the variable v.

The eden interpreter allows the values of formula variables in definitive scripts to be undefined. There can still be technical problems in translating scripts that include partially defined values, however, since triggering in eden depends upon the values of trigger variables being defined. In this connection, it can be useful to use the boolean expression v==@ to test for undefinedness of the variable v.

2. The tank.e program

The tank.e program is an eden program developed to demonstrate the potential of eden as a means of portable specification. The eden program is based on an educational program used in schools that allows children to derive a specified integral target quantity of water by filling, emptying and pouring between 2 jugs of integral capacity. In our approach, the relationships between the significant quantities that define the display (the capacity and content of the jugs, the availability of the menu options, the target and status of the simulation) are specified using eden definitions. Any non-standard operators used are define in eden. Actions in eden are first used to keep the display up-to-date when key parameters are changed, and subsequently to specify the responses to menu selection.

M3.4. Using eden

A common mode of eden program development involves editing a program in one window whilst executing eden in another. Extracts can then be tested by cutting-and-pasting from the editor window into the interpreter window. In the development process, it is often useful to be able to undo design actions. Scripts of definitions can easily be restored to their original form, simply by restoring the original definitions. For this purpose, it's useful to keep old fragments of scripts, temporarily at any rate, by commenting them out rather than deleting them from the edited file.

Another useful feature of eden is the include facility, whereby

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

causes the eden file filename.e to be input. This can also be used to restore a definitive script to its original form.

When using eden in conjunction with the scout and/or donald translators, a typical command line is:

```
scout filename.s - | donald | eden -n
```

The "-" is a UNIX feature, signifying that input will be taken from the standard input after filename.s has been read. The "-n" option for eden suppresses the eden prompt.

M3.5. Miscellaneous additional issues for eden

The eden interpreter is best suited to representing scripts of fixed length, where the variables are persistent. This is appropriate when a designer is incrementally constructing a large model by refining and extending a set of definitions. It isn't so appropriate for dealing with activities where the set of observations changes dynamically as transient processes / agents are created and destroyed. Some degree of dynamism can be achieved in scripts by using eden features that directly generate or manipulate scripts. These include:

- execute() cf specification of the integrators in the VCCS
- forget()
- ' (turn string into variable name) cf digital watch

Common points of concern

- when an action is first defined, there is a one-off action call
- it is not possible to define the components of the list associated with a formula variable lv by treating lv[i] as an l-value
- the order of actions in an eden file can be significant, since it may determine the order in which sequences of triggered actions execute. This is a common problem of a rule-based programming paradigm.

Fundamental issues for eden

Experience has shown that eden is an exceptionally powerful programming tool. It is clear that the introduction of definitive scripts as an additional programming device in a procedural environment simplifies many programming tasks. But though eden can be a most effective tool, it can also be easily abused, and programming principles that might underlie eden are hard to identify. To understand what definitive scripts are good for, we need to be able to relate programming in eden to other paradigms, to seek a satisfactory abstract account of what eden programming entails and to identify what distinguishes between good and bad eden programs.

The fact that the definition "v is f(a,b,c)" can be interpreted as an assignment "v=f(a,b,c)" that is triggered by a, b and c might suggest that essentially eden is a rule-based paradigm. This is a misleading impression, since a major virtue of eden programming is that the use of definitions rather than actions imposes a discipline on execution of actions. For instance:

- evaluation associated with maintaining definitions takes priority over actions, so guaranteeing consistent relationships between variables
- definitions cannot be cyclic, whereas actions can trigger indefinitely through indirect self-reference.

These observations suggest that it is better to try to express state-transitions in a computation in terms of redefinition of definitive scripts, rather than to rely on the cumulative side-effects of loosely synchronised actions.

References

Forward reference to the ADM

Fix(ed) donald to operate in the donald filename.d - | eden -n mode?