APPENDIX E

MORE ABOUT FORMULA AND ACTION

This appendix is a supplementary of the paper; it summarizes the properties of formula and actions, and tries to explain the idea of action in another way.

1. Formula variable

Formula variables are defined using the keyword "**is**". For example, **F is A+B;** defines a formula variable **F** to be the formula **A+B** (not the value of **A+B**).

A and B may or may not be defined (or assigned) before F is defined. The interpreter will create the variables if they have never appeared before. In such case the variable will be initialized to **@** (means undefined). In the example, F is undefined until both of **A** and **B** are defined (or assigned).

The syntax of a formula definition is

identifier **is** expression

The identifier is the formula variable name.

Unlike a value variable, the value of a formula variable is dynamically changing. (The value of a value variable is assigned by an assignment statement, once it is set the value will keep constant unless the variable is re-assigned by other assignment statements.) The formula variable is dependent on all the variables mentioned in the expression. So **F is A+B;** makes **F** depends on **A** and **B.** Whenever the values of A or B changes, F will be re-calculated. Therefore if A and B has the values 1 and 2 respectively, F has the value 3. Suppose B is now set to 10, F then has the value 11 (1+10). F is recalculated by the interpreter implicitly (just after B is redefined or recalculated).

Formulae are not necessary to be arithmetic expressions; they can be expressions of any type. Actually the interpreter does not check the type of the expression or the variable since there is no way to know the data type of a variable in advance. It is user's responsibility to avoid type crash.

If there is a type crash when the evaluation takes place, it will cause an error.

Formula can nest but cannot form loops. Otherwise the evaluation may loop forever. For examples,

```
F1 is F1;
```

is illegal definition because F1 is defined on itself which is somewhat nonsense.

The following definitions

```
F1 is F2;
F2 is F1;
```

the first definition is legal but the second one is not because it makes F2 (and so does F1) indirectly recursive defined.

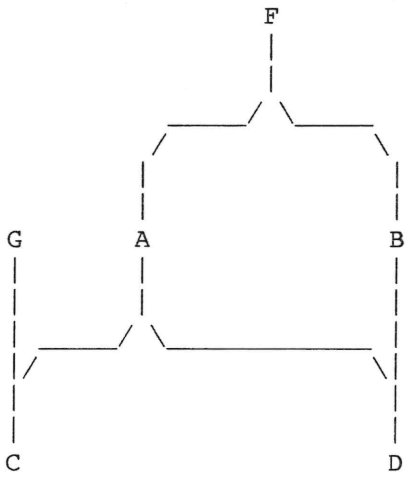
```
F is A;
A is X;
```

are legal formula variable definitions (if X does not directly or indirectly dependent on F and A) F depends on A, and A depends on X. F is said to be indirectly dependent on X. When X is redefined (or reassigned), the value of A is recalculated. So the value of A is changed and hence causes the value of F to be recalculated.

The dependency of the formulae and be represented by a directed graph; consider the following definitions:

```
F is A + B;
A is C * D;
G is C - 1;
B is D / 3;
```

The graph will look like this



Every variable is represented as a node in the graph. The variables are joined by edges if they have relationships; the direction (implies by the position: from top to bottom) shows the dependency: variables are dependent on those variables below them.

Note that there is no loops in this graph (don't forget it is a digraph); hence these definitions are valid.

Assume that the value of C is now changed, say, by an assignment; the interpreter will search upward along the edges and recalculate those variables visited. Let's say the interpreter uses a depth-first search, and its goes from left to right. The nodes are visited in the order: [C,G,A,F]. That is also the order of recalculation. (The other variables are not affected.)

However if the interpreter searches from right to left instead, then the order will becomes [C,A,F,G].

Similarly, if the interpreter use different method to travel upward, different order of evaluation will come off. Say a breadth-first search will give [C,G,A,F] or [C,A,G,F].

However no matter which method the interpreter uses, the related variables will be updated at the end of the assignment.

The same example is considered. The interpreter is assumed to use depth-first search and go from left to right. If the variable D is now changed, the order of evaluation will be [D,A,F,B,F]. F is evaluated twice. The breath-first search will not prevent F from evaluating twice. There is nothing wrong but just a waste of computing time.

2. Actions

An action variable is a function except it can be invoked by the interpreter implicitly when one of the related variable of the action is changed.

The syntax of an action definition in EDEN is

proc action-name : dependency-list function-body

(The keyword **func** can be used in place of **proc**.) The dependency list is a list of identifiers (variable names) separated by commas. (If the list is empty the interpreter considers the definition as a function definition rather than an action definition.) The action is dependent on these variables; the variables used in function body will not affect the dependency. that means when the value of any variable in the dependency list is changed, the action will be called by the interpreter implicitly. The following example is a definition of an action which depends on three variables:

```
proc ACT : A, B, C
{
    writeln("A=",A, " B=",B, " C=",C);
}
```

ACT is an action depends on A, B, and C. ACT will be invoked when the value of A, B, or C is changed. (** ACT will not be invoked unless all of the value of A, B, and C are defined.) In this example, ACT will printed the three values of A, B, and C on screen using the builtin function **writeln**.

The action

```
proc A : A { /* anything */ }
```

is illegal because it is depends on itself.

The following action is legal but will cause crash:

```
proc A : B { B = 0; }
```

Such a definition is allowed because, for example,

```
proc A : B
{
    if (B < 100) writeln(B++);
}
```

2.1. Simulation of Formula and Action

Actually an action variable is a hybridization of formula and function. It has both the properties of formula variable and function - the properties of formula variable: auto-execution (c.f. auto-recalculation), nested dependency, prohibition of recursive definition, the order problem of execution, etc; the properties of function: can be called manually (e.g. **ACT()** to print the three variables).

Formula and action variables can simulate each other.

3. Simulate Action using Formula

The following action

```
proc A: X,Y,Z { FUN(); }
```

can be simulated by a formula variable:

```
A is FUN(X,Y,Z);
```

where FUN() is the function body of the action A.

3.1. Simulate Formula using Action

The formula

```
F is G(X,Y,Z);
```

can be simulated by action A:

```
proc A : X,Y,Z { F = G(X,Y,Z); }
```

where F is value variable but functions as a formula variable with the co-operation of the action A.

4. Order of Evaluation

Last section gives another way of understanding the nature of actions. Action is nothing magic in the language because it just using the side effects of functions to do tricks. How good (or bad) the trick

is depends on how the user programs it. It leaves the user the responsibility of avoiding desirous side effects.

The biggest problem which a EDEN programmer facts is the unknown order of evaluation (including execution). It is a problem because we are using the side effects of actions (or functions). Unexpected results may appear if the order of evaluation is unpredictable. The double calculation (described in section 1) cause the action executed twice (may be more depends on the degree of the nodes).

Although the language has the re-ordering command, but it may not solve all the problems. For instant, the language does not specify how to travel the graph. One interpreter may use depth-first search, and the other one may use breadth-first search, or may even do it parallelly in concurrent environments. The reorder command can do little thing about it. Therefore the order of evaluation is generally non-deterministic.

The simple "trigger propagation" strategy solves the double evaluation problem but it only works in sequential environment; complicated evaluation strategy is needed in concurrent environment.

Fortunately the problems are not so crude because it is a bad style of using such (order-dependent) side effects and it is usually prevented from experienced programmers.