1 Introduction

1.1 Introduction of Definitive Paradigm

The purpose of a computer program is to describe methods of solving certain problems. It is necessary to represent a problem somehow in the program. Many programming languages have variables and procedures. The variables refer to some storage spaces to hold data representing objects (e.g. the coordinates of an object). A procedure is a sequence of instructions to tell the computer how to calculate the data stored in the variables. Due to the side effects of some instructions, the data can be converted into human readable forms, e.g. the `write` statement in **Fortran** can display data in a specified format.

An interior designer might like to draw a picture of a room with some furniture inside it on the display screen. Suppose there is a program which allows a user to enter the coordinates of these objects through the keyboard. This program can redraw the objects if the user issues a "refresh" instruction, or more conveniently, redraw the objects automatically after new coordinates have been entered or existing coordinates changed. It is not difficult to write this kind of program in conventional procedural languages, such as **Pascal** or **C**. Some existing software, like **MacDraw** or some computer aided design (CAD) packages, lets the user manipulate graphical objects. These packages do the job quite well since they have interactive user-interfaces.

These user-interfaces let the user access the pre-programmed utilities easily, and see the result almost instantly. However, these packages do not help a designer to *model* his/her design, i.e. to specify the design using expressions. For example, an interior designer may wish to specify some furniture's positions relative to the position of a room whose dimensions are still uncertain at this stage. For instance he may like to put the bed of a fixed size at one of the corners of the room no matter where and how big the room will be. That means the coordinates of the bed are determined by the coordinates of the room. In this

case, he has to calculate the new position of the bed and redraw it after the room has been positioned or resized. Since *design* is a sort of trial-and-error process, the designer will be required to repeat the same process—calculate new coordinates and redraw the objects. Obviously it is a tiresome and error-prone job especially when the numbers of objects and inter-object relationships become large. Unfortunately most CAD packages do not support much help to specify and maintain these relationships among objects.

Some software packages, such as **MacDraw**, allow the user to group several objects into a single complex object so that the user can manipulate all the grouped objects at once. The grouping of objects preserves the relative coordinates and attributes of the objects among the same group during the manipulations. Hence, if a room and a bed are grouped together, translation of the room will also translate the bed by the same amount. However grouping objects also puts extra restrictions on the objects. For instance, the enlargement of the room will also resize the bed in the same ratio, and hence violate the specification of the bed which must has a fixed dimension. Thus the method of grouping objects provides very restricted transformations on objects. Complicated definitions of objects, such as:

> *"C lies at the mid-point of line AB, where endpoints A and B are defined independently"* — (1-1)

cannot be specified by this method. Thus the method of grouping objects cannot model the relationship but can only be considered as a macro manipulation on a number of objects.

Therefore it is much more convenient to allow the user to specify the abstract *definitions* — the relationships — of the objects in terms of other objects. For example, the point C described in (1-1) can be defined in terms of points A and B mathematically as:

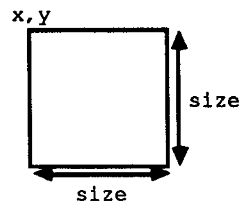
$$C \equiv \frac{A + B}{2}$$

The term, *definition*, is used in a narrow sense to refer to the mathematical expression of the relationship among objects. A definition specifies how to compute an item of data from the others but not vice versa; thus, a definition is a single directional relationship. If the expression implies the reverse relationships, then it is more commonly known as a *constraint*. In this thesis,

we are mainly concerned with the forward relationships as expressed by definitions.

In order to refer to the style of programming practice based on definitions, we introduce the following terms: A *definitive language* is a computer language that can specify definitions, and a *definitive notation* is a definitive language that is specialized for expressing data relationships between variables whose values lie in a particular *underlying algebra* of data types and operators (c.f. [Be85]). A *definitive system* is a computer system that not only expresses inter-object relationships in definitions but also enables the maintenance of these relationships. The programming paradigm which is based on definitions is, thus, called the *definitive paradigm.*

The use of a definitive system to describe geometric relationships also has some other advantages. Many CAD systems allow the user to specify parametric objects, i.e. generic objects whose size and position can be determined by giving parameters. In conventional systems, such as **DOGS** [PAFEC-1; PAFEC-2], parametric objects are represented by some **BASIC**-like programs comprising appropriate sequences of drawing instructions, but a method of specifying the relationships between parts of the object explicitly is preferable. For example, contrast the two specifications of a square with given corner and size below.




```
(1)  Procedural
moveto x, y
lineto x+size, y
lineto x+size, y-size
lineto x, y-size
lineto x, y
```

```
(2)  Definitive
parametric shape square(x,y,size)
      point NE, SE, SW, NW
      line  N, S, E, W

      N  ≡ [NW, NE]
      E  ≡ [NE, SE]
      S  ≡ [SE, SW]
      W  ≡ [SW, NW]
      NW ≡ {x, y}
      NE ≡ {x+size, y}
      SE ≡ {x+size, y-size}
      SW ≡ {x, y-size}
end
```

In the first example, the size and position of the corner are fixed when the symbol is drawn. Hence the user has to re-construct the symbol when any parameter has changed. In the second example, the object specified by the definition:

$$S \equiv square(X, Y, SIZE)$$

would be re-defined automatically if, for example, the value of the variable SIZE were to change. The specification is written in a DoNaLD-like notation. DoNaLD is a notation for describing graphical objects in an abstract way, i.e. by specifying the relationships among objects in definitions [BABH86]. The primitive graphical objects are points and lines. Complex objects are composed of these primitives by grouping them into openshapes — similar to the parametric shapes listed in the specification.

Notice that the shape S is "re-defined" rather than "re-constructed" because S is a set of definitions rather than a graphical image. Each definition represents an item of geometric data of the point or line type. The re-construction of the graphical image is handled by the system. This means that the user is now dealing with the geometric model of the graphical objects instead of a set of graphics instructions (as well as the geometric model).

One of the advantages of the DoNaLD notation is that the order of the definitions is insignificant in the specification. For example, the two lines of the definitions of the points N and E can be interchanged without affecting the layout of the graphical image. On the other hand, it will be a disaster if any two lines in the procedural specification are swapped.

Another advantage of the definitive specification is that individual definitions can be re-defined afterwards. The change of the geometric model is incremental, i.e. only parts of the model are affected. We take it for granted that the graphics package is sophisticated enough to handle incremental change of the graphics image.

Conventional languages do not provide any built-in facility to help writing a definitive system. Thus the programmer must write a detailed definition manager to handle the definitions.

1.2 Examples of Definitive Systems

A well-known example of a system based on definitive principles is the spreadsheet [Am86] software which has recently become so popular in business and personal accounting applications. A spreadsheet is a program that provides us with a large grid of cells into which we can insert numbers and formulae. A cell that contains a formula causes the system to re-calculate the formula whenever any cell on which it depends is changed, and display the new value on the screen automatically. The display action is implicitly invoked by

the system after the re-calculation. These actions can be called *implicit actions* because they are pre-defined in the system.

The **make** command [Fe78] in UNIX† is a good example of file maintenance software. Unlike the spreadsheet software, the user has to specify the updating action in a file (the *makefile*) explicitly. These actions can be called *explicit actions*. For example, to express the fact that

"object file file.o is compiled from C source file file.c"

the appropriate makefile is:

```
file.o : file.c
        cc -c file.c
```

The first line indicates the dependence and the second line contains a UNIX command showing how the object file can be compiled from the source file. The explicit actions allow less restricted updating actions to be defined. For example, the user can give different options to the compiler or use different compilers to compile the target object. However the user has to make sure there are no destructive side-effects.

In a definitive system, each definition has one and only one target object (such as a cell or an object file). Usually there is at least one source object. The target object is said to "depend on" the source objects because it is computed from them only and whenever the values of the source objects change the target object must be re-computed. Cyclic definitions, such as "$n \equiv n + 1$", are meaningless, and usually prohibited by definitive systems.

A spreadsheet program and the **make** command are driven by definitions. Although **make** is more like a *dependency maintainer* (it keeps track of objects by referencing the dependence, but does not refresh the target object), it combines representation of data dependencies with explicit actions resembling those needed to maintain definitions. Actually a definitive system can be implemented using a dependency maintainer with some built-in computing utilities. Internally, a system builds a dependency graph according to the definitions given and uses this information to determine which data has to be

† UNIX is a trademark of AT & T Bell Laboratories

updated; from the formula of the definitions, it knows how to compute the targets.

1.3 Advantages of the Definitive Paradigm

A definitive system saves us from remembering what has to be updated when certain data have been changed. It is especially useful in a continuously changing environment; for instance, a programmer would use **make** to compile those modified modules of a large project under development. The success of spreadsheet programs and **make** illustrate the advantages of the definitive paradigm.

Another advantage of a definitive system is with regard to human-computer interaction. Some definitive systems can be programmed to echo the up-to-date data on the display as soon as certain data has been changed by the user. The user can know the result of the change immediately and then continue to modify the definitions. This interactive behaviour of the definitive system may contribute to applications, such as CAD systems, that require intensive human interaction. Some discussion on human-computer interaction in different programming paradigms appears in [Me87].

Definitions are good for specifying relationships between objects. The effects of updating actions are able to simulate the actions of objects responding to a change in the environment.

1.4 Motivation of the Design of EDEN

The implementation of a definitive system is not a trivial task. As mentioned earlier, it may involve developing a complicated dependency maintainer. It is more sensible to make a general purpose definitive system and build a special-purpose system front-end on top of it. Unfortunately no such systems are publicly available. An experimental language, called EDEN, was designed to try to fulfil this purpose [Yu87]. It was designed to be a general-purpose language supporting definitions. Users can define their own procedures and functions using some C-like statements. The first prototype of such an interpreter was implemented in 1987. Since that time, another prototype of the same language has been implemented. The difference between these two versions is mainly concerned with the internal evaluation strategy and should not concern the naive user.

This thesis succeeds the discussion in [Yu87]. We concentrate on the implementation techniques and the applications of the EDEN language. We also evaluate the design of the language through some experimental programming practice, and give some possible suggestions to improve the current design.

One way of viewing the EDEN language is that it is a C-like language with the built-in mechanism of handling definitions between variables. However, it will give us the wrong impression of the language as an extension of a procedural language. The correct way of looking at the language is to view it as a definition-based language with the procedural statements as an extension since the definition management mechanism is the basic feature of the EDEN system. The procedural statements provide a simple way of defining a function and implementing (explicit) *actions*. The idea of actions is borrowed from UNIX's **make** command. By associating an action procedure with a variable, we can echo the value of the variable onto the screen (by the side-effects of a function call within the action procedure) whenever it has changed. The EDEN interpreter schedules and invokes actions according to the specification defined by the user.

In EDEN, a definition has the form:

identifier **is** *expression* ;

The keyword "**is**" defines the formula variable named as *identifier* to be the formula *expression*, NOT the value of the *expression*. The value of the variable is equivalent to the value of the *expression* at the time of evaluation. For example,

```
F is A + B;
```

If `A` and `B` have the values `2` and `3` respectively, the value of `F` is `5`. But if the value of `B` is redefined to `5`, the value of `F` becomes `7`.

The following action specification will print the up-to-date value of variable `G` on the screen by the side-effect of function `writeln`.

```
proc print_G : G { writeln("G is now: ", G); }
```

So,

Input	*Output*

`G is I + J;`
`I is 5;`
`J is 10;`	

	`G is now: 15`
`I is 100;`	
	`G is now: 110`

Unfortunately, the EDEN language does not encourage/enforce people to write definition-based programs; conventional programmers would tend to use the procedural features intensively, for instance. On the other hand, the procedural features provide a handy way to implement routines efficiently (since they match the von Neumann architecture). In addition, procedure statements provide a natural way to communicate or control I/O interfaces.

Scout[Yu88] is a definitive notation which describes the text layout on the screen. A screen consists of a number of text windows. The dimensions and content of a window can be specified by definitions. **Scout** was implemented as a preprocessor of the EDEN interpreter. The **Scout** preprocessor translates **Scout** notations into equivalent EDEN definitions and action specifications. **Scout** operators are implemented as EDEN functions. The action specifications calls a number of routines of an existing window package through the EDEN interpreter, and hence produces the display.

Interestingly, although the syntax and motivation of definitions and actions are very different, they have so many common characteristics that they can be unified, i.e. the definitions can be emulated by actions and vice versa. In this thesis, the similarities and the differences between them are discussed. The unification of definitions and actions is important because some restrictions imposed on definitions, such as non-cyclically defined restriction, can be got around by the equivalent action emulations. On the other hand, the procedural style of action specification can be replaced by the "pure" definitive style — the equivalent definition emulation. For example,

Definition	***Equivalent action emulation***	
`F is A + B;`	`proc formula_F : A, B { F = A + B; }`	(1-2)
`Sum is Sum + Data;`	`proc formula_Sum : Data { Sum = Sum + Data;`	(1-3)

In (1-2), `F` becomes a conventional variable; the action `formula_F` is responsible for updating the value of `F`, i.e. emulating the formula variable on the L.H.S. In (1-3), the definition is invalid in EDEN since it is cyclically defined. The action specification on the R.H.S. resembles the definition but has no cyclic dependence. The "definition" computes the new `Sum` by adding the new `Data` to the current value of `Sum`.

Consider the following action specification, for which there is an equivalent definition emulation.

Action Specification	***Equivalent definition emulation***	
`proc print_F : F { writeln(F); }`	`func echo { writeln($1); }` `print_F is echo(F);`	(1-4)

In (1-4), a function `echo` is defined to print the first argument (`$1`) on the screen. The dummy definition `print_F` calls this function when the value of `F` is changed.

1.5 Methodologies of Implementing Definition Manager

So far, there is no general formal method of implementing a definition manager. One theme of this thesis is to consider some methods of implementation.

The simplest way of handling definitions is to treat them as functions. Every time we require the value of a formula variable, we evaluate its formula expression. This ensures the value is up-to-date. But obviously it is not an efficient way of evaluation.

The current implementation of EDEN definition manager is based on the "data dependency" approach as discussed earlier. The definition manager maintains a dependency graph. It uses this information to sort out which variables have to be updated and schedules their order of evaluation.

Other possible ways of implementing a definition manager are based on data-flow machines, rule-based schemes, etc.

The EDEN language hides all the details of how the definition manager operates. The order of definitions and action specifications is insignificant in a program because the order of evaluation of definitions and invocation of actions is scheduled by the system. Since the action procedures are executed in arbitrary order, it is the responsibility of the programmer to write well-behaved procedures.

There are no reasons why we cannot implement the scheduler and evaluator on a data-flow machine or a multi-processor machine. The possibilities of porting the EDEN language to concurrent systems are investigated in this thesis. We begin with the analysis of current scheduling mechanisms which are based on the sequential system architecture, and try to develop a scheme for concurrent system architecture.

1.6 Application of EDEN

Implementing other definitive notations is one of the aims of the EDEN language. A description of the implementation of a small subset of DoNaLD notation, using EDEN interpreter as the back-end, is given to illustrate this ability. Though the EDEN language is not mature enough to be able to implement the whole set of DoNaLD notation, the subset we have implemented is enough to illustrate the strong and weak points of the definitive programming paradigm.

We also try to apply definitive paradigms to different fields of interest, such as user-interface specification, constraint maintenance, and digital circuit simulation. We shall explain how a window system can be specified in definitions. We shall also look at a simple constraint example presented in [Le87], and present an EDEN program to maintain this constraint. The last example is a simulation of a pipe-line processor written in EDEN.

The rest of this thesis is divided into:

Chapter 2: Comparison of Programming Paradigms

A comparison of programming paradigms is presented. We compare definitive languages mainly with their close cousins, functional languages.

Chapter 3: Design of EDEN

A description of the fundamental philosophy of the design of EDEN and followed by a brief summary of EDEN language. We also present the unification of definitions and actions.

Chapter 4: EDEN Programming

We present three examples to demonstrate the application of EDEN. The first example uses definitions to specify the layout of a window system. The second example tries to implement a constraint satisfaction system using definitions and actions. The final example uses definitions to simulate a digital circuit.

Chapter 5: Implementation of EDEN Interpreter

This chapter outlines some important components of the EDEN interpreter. Particular emphasis is given to the definition management scheme.

Chapter 6: Implementation of DoNaLD

This chapter describes the project that uses EDEN as a back-end to implement the DoNaLD notation. It is the first step towards the application of definitive principles to CAD.

Chapter 7: Parallel Definitive System

Because definitions implicitly describe data dependency, we believe that definitive languages are good languages in parallel computation. We try to exploit the potential for parallel computation in definitive systems.

Chapter 8: Future Research

The EDEN project suggests many ideas for improving the EDEN language. We present a brief proposal of future research.

Chapter 9: Conclusion