http://www.dcs.warwick.ac.uk/modelling/

Table of Contents

- Introduction [p 3]
- Fundamental Philosophy [p 7]
  - Concept of EDEN Programming [p 8]
    - Formula Definition [p 8]
    - Action Specification [p 9]
  - The EDEN System [p 10]
    - Read/Write Variables [p 10]
    - Formula Variables [p 10]
    - Action Specification [p 12]
    - Execution Strategy [p 13]
- EDEN Language Guide [p 15]
  - Introduction [p 17]
  - Lexical Conventions [p 17]
  - Syntax notation [p 18]
  - Comments [p 18]
  - Identifiers (Names) [p 18]
  - Keywords [p 19]
  - Objects and Lvalues [p 19]
    - Read/Write Variables [p 20]
    - Function Definition [p 20]
    - Formula Variable [p 20]
    - Action Specification [p 21]
  - Data Structures [p 21]
    - @ (undefined) [p 21]
    - Integer [p 21]
    - Character [p 22]
    - Floating point [p 23]
    - String [p 24]
    - Pointer [p 24]
    - List [p 25]
    - Function [p 26]
  - Expressions [p 26]
    - Lvalue [p 26]
    - Primary expression [p 27]
  - Operators [p 27]
    - Arithmetic Operators [p 28]
    - String and List Operators [p 28]
    - Pointer Operator [p 29]
    - Relational Operators [p 29]
    - Equality Operators [p 29]
    - Logical Operators [p 29]
    - Conditional Operator [p 30]
    - Assignment Operators [p 30]
    - Precedence and Order of Evaluation [p 31]

- Procedural Statements [p 32]
  - Expression Statements [p 32]
  - Insert Statement [p 33]
  - Append Statement [p 33]
  - Delete Statement [p 33]
  - Shift Statement [p 33]
  - Compound Statement [p 34]
  - Condition Statement [p 34]
  - While Statement [p 34]
  - Do Statement [p 34]
  - For Statement [p 35]
  - Switch Statement [p 35]
  - Break Statement [p 35]
  - Continue Statement [p 36]
  - Return Statement [p 36]
  - Null Statement [p 36]
- User-defined Functions [p 36]
- Definitive Statements [p 38]
  - Formula Definition [p 39]
  - Action Specification [p 39]
- Miscellaneous Commands [p 40]
  - Dependency Link [p 40]
  - Query [p 41]
- Syntax Summary [p 41]
- Pre-defined Variables [p 44]
- Pre-defined Functions [p 44]

- Advanced Topics [p 53]
  - Adding C library functions to EDEN [p 54]
  - Programming Notes [p 57]
    - Memory Allocation [p 57]
    - Stack, Heap and Frame Stack [p 58]
    - Autocalc [p 61]

Introduction

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 (if not most) 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 telling 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 CAD packages, let the user manipulate graphical objects. These packages do the job quite well since they have highly interactive user-interfaces.

However the interior designer may require more than just refreshing the display of the objects on the screen; he may wish to specify some of their relative positions. For example, he may also like to put a lamp at the centre of the table no matter where it will be. That means the coordinates of the lamp are determined by the coordinates of the table. In this case, he has to calculate the new position of the lamp and redraw it after it has been moved. Since he may move the table several times to see which position is better, he will be required to repeat the same process Ñ calculate new coordinates and redraw the objects. Obviously it is a tiresome and error-prone job especially when the number of objects and definitions becomes large.

Some software packages, such as MacDraw, allow the user to group several objects into a single large object. The user can then manipulate the object using the operations provided by the packages. The grouping of objects preserves the relative coordinates and attributes of the objects among the same group during the manipulations. Hence, if a table and a lamp are grouped together, translation of the table will also translate the lamp by the same amount. However grouping objects also puts extra restrictions on the objects. For instance, the enlargement of the table will also resize the lamp in the same ratio, and this may not be the intention of the user. 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)

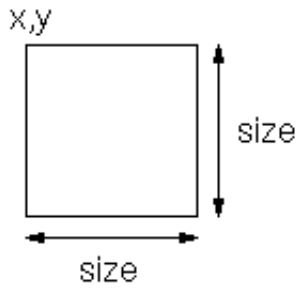
cannot be specified by this method. Thus the method of grouping objects can only be considered as a macro manipulation on a number of objects.

Therefore it is much more convenient to allow the user to specify the abstract definitions 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 below.

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

A system that can manage definitions is called a *definitive system* or definition-based system.

The use of a definitive system to describe geometric relationships also has another advantage. 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 objects are represented by small procedures 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. In the second example, the object specified by the definition:

$$S \equiv \text{square}(X, Y, SIZE)$$

would be re-defined if the value of the variable *SIZE* were to change.

Unfortunately, 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.

A well-known example of a system based on definitive principles is the *spreadsheet* 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 put in 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 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 `.o` is compiled from source 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 that 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 objects. 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. No cyclic definitions are allowed since recursively defined definitions lead to meaningless definitions, such as ‘‘$n \equiv n + 1$’’.

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

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` has illustrated the advantages of a definitive paradigm.

Another advantage of a definitive system is regarding 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 computer aided design (CAD) systems, that require intensive human interaction.

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

An 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 *EDEN - an evaluator of definitive notations [p 7]* - was designed to try to fulfil this purpose. It was designed to be a general purpose language supporting definitions. Users can define their own procedures and functions using some C-like statements. An interpreter can be customized for different applications. 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 is transparent not noticed by the user, and should not concern the naive user.

EDEN is a hybrid language that combines elements of traditional procedural and definitive languages. (note: some people use the term *definition-based* languages). It has both advantages and disadvantages of procedural and definitive languages. This user handbook is a guide to the EDEN programming language.

EDEN - an Evaluator of Definitive Notations

Y.W. Yung, *EDEN - an Evaluator of Definitive Notations*, MSc Thesis, Department of Computer Science, University of Warwick (1989)

Fundamental Philosophy

In this chapter, we will explain the fundamental philosophy of definitive programming and the language EDEN.

- Concept of EDEN Programming [p 8]
  - Formula Definition [p 8]
  - Action Specification [p 9]
- The EDEN System [p 10]
  - Read/Write Variables [p 10]
  - Formula Variables [p 10]
  - Action Specification [p 12]
  - Execution Strategy [p 13]

Concept of EDEN Programming

The important ingredients of the EDEN language are the formula definitions and action specifications. In this section, we explain these concepts.

- Formula Definition [p 8]
- Action Specification [p 9]

Formula Definition

A definition of an object is expressed as a mathematical expression. A variable is called a formula variable if its value is defined in terms of other variables, e.g:

$$v \equiv \Phi(v_1, v_2, \ldots, v_n)$$

where v is the name of the formula variable, and Φ represents an expression with the variables v_1, v_2, ... , v_n involved. The variables v_1, v_2, ... , v_n are called the source variables of this definition since the value of v is computed from their values. Hence the formula variable v, can also be called the target variable. EDEN supports the concept of definition. A formula definition has the form:

$$v \equiv \mathtt{is}(v_1, v_2, \ldots, v_n)$$

The keyword ‘‘is’’ defines a formula variable v whose value is computed from the values of source variables v_1, v_2, ..., v_n and the expression on the right hand side, denoted by Φ , is the formula of the variable v. In other words, a formula describes how the value of a variable is computed from other data. These formulae are permanently valid (unless they are re-defined). That is, no matter what the values of source variables are, the value of variable v is always equal to $\Phi(v_1, v_2, \ldots, v_n)$. Thus a formula gives an abstract definition of a variable rather than the explicit value of it. This is the major difference between formula definitions and assignment statements. For example, after executing the

assignment

$$v \equiv \Phi(v_1, v_2, \ldots, v_n)$$

v is equal to $\Phi(v_1, v_2, \ldots, v_n)$ only after the expression Φ is evaluated and before any of the values of the source variables is altered.

Unlike the assignment statement, the formula definition, itself, is not an executable statement. When and how the formula expression defined in the definition is managed by the system. However, the values of the formula variables are guaranteed to be up-to-date when the variables are examined.

Notice that a definition does not imply the inverse definitions (where these exist); for example, the definition (2-1) does not imply

$$v_i \equiv \Phi_i^{-1}(v, v_1, v_2, \ldots, v_i, v_{i+1}, \ldots, v_n)$$ where 1<=i<=n

A definition can be a representation of a real relationship among objects; for instance, ‘‘*a lamp is placed (somewhere) on a table*’’ can be formulated as:

```
lamp_position is table_position + something
```

If the table is re-positioned the lamp will also be moved because the definition has specified the position of the lamp in terms of the position of the table, the system can automatically keep track of these definitions and re-calculate `lamp_position` whenever `table_position` has been changed. But if the `lamp_position` is then redefined to

```
lamp_position is desk_position + something
```

we mean to put the lamp on the desk instead of the table. The table remains at the same position because it is not defined in terms of the lamp. Note that the new definition of `lamp_position` overwrites the previous definition. Hence the lamp can be moved independently by redefining its definition.

Action Specification

So far, we have talked about formula definitions without mentioning the updating actions (e.g. renew the display of the content of a variable when it is changed) since there is no default implicit action associated with each definition. However, EDEN provides a way of defining explicit actions. Thus the user can specify special updating actions for different devices. In the rest of this handbook, the term ‘‘*action*’’ refers to ‘‘*explicit action*’’ because it is the only kind of action available in EDEN.

The following statement illustrates a sample action definition.

```
proc display_v : v { display(v); }
```

The keyword ‘‘**proc**’’ defines an action, named as `display_v`, which invoked by the system when the value of variable `v`, (specified after the colon) is changed. The curly brackets `{}` enclose a list of statements to be executed sequentially. In this case, there is only one procedure call. The procedure `display` is defined separately. By calling different functions (with appropriate side-effects, e.g. `writeln(...)` prints the values of its arguments on the standard output) in the function libraries, display can do different updating actions. This makes EDEN more extensible.

Note that the value of `v` is not passed as a parameter to `display_v` when the action is invoked, but since it is a global variable, it can be accessed by the action directly.

The EDEN System

The EDEN language has definitions, actions and some conventional features. The conventional features are, for example, iterative loops, flow control statements, user-defined functions and read/write variables.

The variables can be classified into 4 types: read/write variables, functions, formula variables, and action specifications. EDEN has different statements to assign values to these variables.

- Read/Write Variables [p 10]
- Formula Variables [p 10]
- Action Specification [p 12]
- Execution Strategy [p 13]

Read/Write Variables

Read/write variables (RWV) are the same as the variables of conventional languages. As the name implied, the value of a RWV can be assigned and examined by the user. For instance,

```
A = 1
```

will assign integer `1` to the RWV `A`. The value of a variable is referenced if the variable appears in an expression - e.g. the right-hand side of the assignment statement. For example,

```
B = A
```

puts the value of `A` into RWV `B`.

Note that, in EDEN, a function is considered as a RWV (although there is a different instruction to define a function, and some restrictions on their definition). Functions will be discussed in the later chapters.

Formula Variables

A *formula variable* (FV) consists of a *data register* (DR) and a *formula expression* (FE) (or simply *formula*). The formula describes how the value of the *data register* (DR) can be computed from other variables (RWV or FV). The system will always update the DR using the formula whenever it is possible. This part of the system is called *formula data maintainer* (FDM).

The value of a formula is equivalent to the value of the DR of that FV. The user can (re-)define the formula of a definition but not the value of the DR. This means that the DR is read-only by the user. In EDEN, the formula of a formula variable is defined using the ‘‘**is**’’ operator, for example:

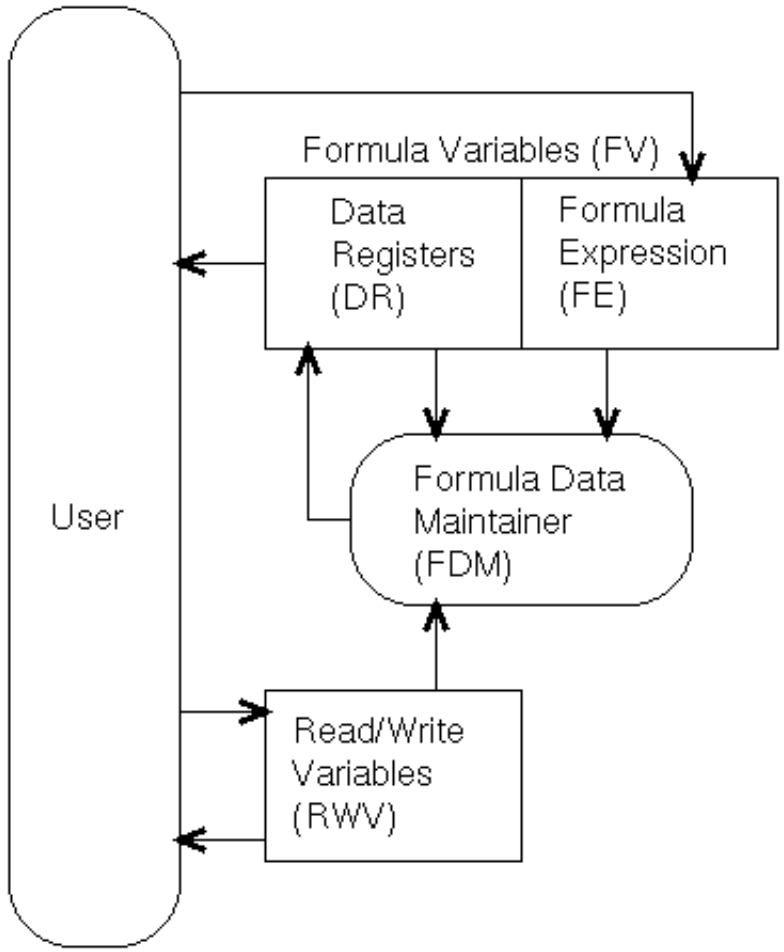
```
f is a + b
```

defines a FV `f` whose value is always equal to the sum of the values of variables `a` and `b`. The expression ‘‘`a + b`’’ is the FE of `f`. Note that all variables mentioned in the FE, in this case, `a` and `b`, can be either FV’s or RWV’s.

The value of a FV is referenced if the name of the FV appears in an expression, e.g. the right-hand side of an assignment statement.

The syntax of a formula definition is defined in the EDEN Guide [p 20] .

Block Diagram of a Formula Maintaining Sub-system



The arrows show how data might flow. E.g. the user can define a FV by giving the formula to it. As shown in the diagram, the user can examine the value of a DR but not write to it. When the user (re)defines a FV or writes to a RWV, the FDM will re-calculate those DR’s using their associated FE’s. The FDM will read the values of some RWV’s and/or DR’s if it is necessary. The results will be

assigned to the DR's by the FDM.

Action Specification

An *action specification* (AS) is a named sequence of instructions. This sequence of instructions will be invoked by the system whenever the values of any source variables, specified explicitly in a list, are *changed*. The term ''*changed*'' is causally defined. It may mean the value of a variable is different from the previous one, or the value of a variable is overwritten (by the user or by the system) though the value may be the same as the previous value. EDEN takes the latter definition. This definition of ''changed'' is used throughout this handbook unless otherwise specified.

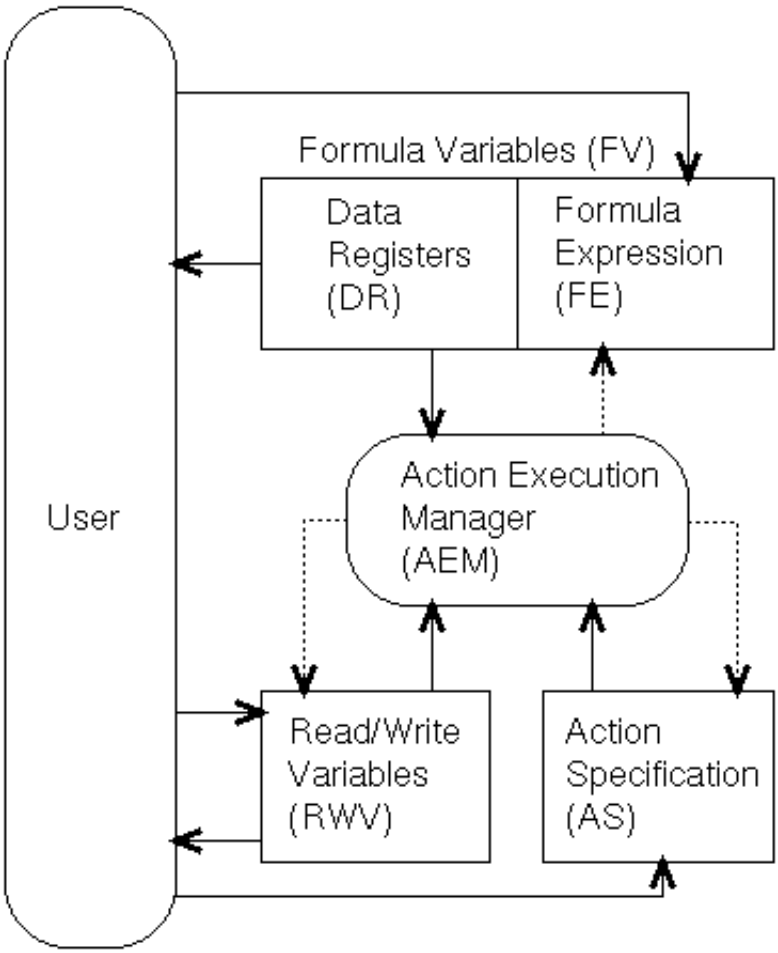
An example of an action definition is:

```
proc print_sum : a, b, c { writeln("a+b+c=", a+b+c); }
```

This action, named as `print_sum`, will invoked by the system whenever the values of the source variables `a`, `b` or `c` (listed after the colon) are changed. The action will print a string and the sum of the values of these variables on the screen due to the side-effect of the pre-defined function `writeln`.

The *action execution manager* (AEM) is responsible for invoking the action. This sub-system is called the *action execution management sub-system*.

Block Diagram of the Action Execution Management Sub-system



The arrows indicate the possible data flow in the system. Shaded arrows indicate the data flow due to the side-effects of action execution. From this diagram, we see that the actions are invoked by the AEM. AEM loads the procedure of the appropriate action from the AS and then executes it. The execution may cause a write to a (or some) RWV, or (re)define FV's or AS's (using some tricks). Of course, the action may also access I/O devices (not shown in the diagram).

Execution Strategy

EDEN is neither purely declarative (in which case the execution strategy does not affect the result) nor purely procedural (some side-effect are triggered rather than called explicitly). Therefore, the user needs to aware of the execution strategy in order to make the most out of EDEN.

EDEN takes precedence of formula evaluation over actions. This means that EDEN will update all the value changes to the variables first. The triggered actions are queued in an *action queue*. After all the variables are updated, those actions queued will be performed. This will ensure that any reference to a variable will get the most up-to-date value.

EDEN also uses a breath-first scheme of evaluation. Under such scheme, the interpreter spends more time on scheduling the order of evaluation than a depth-first scheme but it has an advantage of doing the evaluation of variable or the triggered actions only once in each phase. For example:

```
proc dummy : A {  /* dummy() will be executed when A changes */
        auto i;   /* local variable */
        for (i = 1; i <= 10; i++) {
                V = i;
        }
}

proc Print_V : V {
        write(V, ' ');
}

A = 1;     /* change A to trigger dummy() */
```

the interpreter will produces the output:

```
        10
```

rather than

```
        1 2 3 4 5 6 7 8 9 10
```

While the depth-first scheme is good at monitoring the change of values, the breath-first scheme the current implementation using is more efficient in producing the final result. Should the later output is preferred, the user has to force the interpreter to execute the actions in the action queue by a *eager()* statement. For the above example, the required result can be generated if dummy is alternatively defined as:

```
proc dummy : A {   /* dummy() will be executed when A changes */
        auto i;   /* local variable */
        for (i = 1; i <= 10; i++) {
                V = i;
                eager();   /* force execution of action queue */
        }
}
```

One should not presume the order of action execution. A general principle is that if the order of execution will significantly affect the result, it is likely that it is not a good program. However, some sort of ordering is necessary. A typical example is the simulation of a clocked system. In a clocked system, there should be actions such as:

```
proc clocking : clock {   /* a clocking process */
        clock++;           /* advance the clock */
}

proc device1: clock {  /* action synchronised by clock */
        ...   /* action body */
}

proc device2: clock {
        ...   /* action body */
}

...
```

In these cases, the `clocking` action should be executed last so that it would not block out other actions to be performed in this clock cycle. To solve this scheduling problem, EDEN has a to-do list (in fact two to-do lists, one currently executing and the other one is really to be executed). The clocking action above should be rewritten as:

```
proc clocking : clock {   /* a clocking process */
        todo("clock++;");        /* advance the clock in future */
}
```

This will cause the statement:

```
clock++;
```

to be saved in the (future) to-do list. After all the triggerred actions and things-to-do in the current to-do list have been executed, the system will perform those statements in the future (now becomes the current) to-do list. In this case, when `clock++` is executed, all other actions will be triggered and executed in this 'clock cycle' and another `clock++` statement is scheduled to do after all the necessary actions are performed in this clock cycle.

Clocking: Note

The clocking mechanism cannot be replaced by a for/while loop because within a loop the EDEN interpreter will not accept any user input. So the user cannot interact with the simulation.

EDEN Language Guide

- Introduction [p 17]
- Lexical Conventions [p 17]
- Syntax notation [p 18]
- Comments [p 18]
- Identifiers (Names) [p 18]
- Keywords [p 19]
- Objects and Lvalues [p 19]
  - Read/Write Variables [p 20]
  - Function Definition [p 20]
  - Formula Variable [p 20]
  - Action Specification [p 21]
- Data Structures [p 21]
  - @ (undefined) [p 21]
  - Integer [p 21]
  - Character [p 22]
  - Floating point [p 23]
  - String [p 24]
  - Pointer [p 24]
  - List [p 25]
  - Function [p 26]
- Expressions [p 26]
  - Lvalue [p 26]
  - Primary expression [p 27]
- Operators [p 27]
  - Arithmetic Operators [p 28]
  - String and List Operators [p 28]
  - Pointer Operator [p 29]
  - Relational Operators [p 29]
  - Equality Operators [p 29]
  - Logical Operators [p 29]
  - Conditional Operator [p 30]
  - Assignment Operators [p 30]
  - Precedence and Order of Evaluation [p 31]
- Procedural Statements [p 32]
  - Expression Statements [p 32]
  - Insert Statement [p 33]
  - Append Statement [p 33]
  - Delete Statement [p 33]
  - Shift Statement [p 33]
  - Compound Statement [p 34]
  - Condition Statement [p 34]
  - While Statement [p 34]
  - Do Statement [p 34]
  - For Statement [p 35]

    - Switch Statement [p 35]
    - Break Statement [p 35]
    - Continue Statement [p 36]
    - Return Statement [p 36]
    - Null Statement [p 36]
- User-defined Functions [p 36]
- Definitive Statements [p 38]
    - Formula Definition [p 39]
    - Action Specification [p 39]
- Miscellaneous Commands [p 40]
    - Dependency Link [p 40]
    - Query [p 41]
- Syntax Summary [p 41]
- Pre-defined Variables [p 44]
- Pre-defined Functions [p 44]

Introduction

The EDEN language has a number of C-like statements and operators. C programmers may find it familiar but they are still advised to go through this chapter since there are new data types and operators. Not all C operators have been implemented.

An EDEN program is a list of statements. Each statement can be one of the followings: *formula definitions [p 20]* , *action specifications [p 21]* , *function (procedure) definitions [p 26]* and C-like *procedural statements [p 32]* .

```
program:
        statement program
```

When a procedural statement is encountered, this statement will be executed. The effect of execution is to evaluate an expression, assign values to variables or call other functions/procedures (these must be defined earlier). The order of statements reflects the order of execution, and thus affects the results of computation. The user is responsible for arranging the statements in proper order to get the correct results.

In addition, the EDEN interpreter will do the (re-)calculation of the formula definitions and/or invoke the procedures defined by the action specifications automatically. The order of calculation of formula definitions and execution of actions should not be of concern to the user, and is fully controlled by the interpreter.

The *dependency-link [p 40]* command is an alternative way of specifying actions. The *query [p 41]* command inspects the current definitions of the objects.

Lexical Conventions

There are five classes of tokens: *identifiers [p 18]* , *keywords [p 19]* , *constants [p 21]* , *operators [p 27]* , and *other separators*. *Blanks*, *tabs*, *newlines*, and *comments [p 18]* (collectively, ‘‘*white space*’’) as described below are ignored except as they serve to separate tokens. Some white space is required

to separate otherwise adjacent identifiers, keywords, and constants. If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

Syntax Notation

In the syntax notation used in this chapter, syntactic categories are indicated by *italic type*, literal words and characters in `fixed-width font`, and keywords in **`bold fixed-width font`**. Alternative categories are listed on separate lines. An optional terminal or non-terminal symbol is indicated by the subscript ‘‘*opt*’’, so that

[*expression-list* $_{opt}$]

indicates an optional expression list enclosed in square brackets.

Comments

Comments are arbitrary strings of symbols placed between the delimiters `/*` and `*/`. The whole sequence is equivalent to a white space. Note that `/* */` can be nested. It is useful to comment a block of program with comments in it. Lines beginning with `%` (no even spaces or tabs before it) are also comment lines.

```
/* This is a comment */
/* --- Begin of comment ---
        /* This is a nested comment */
--- End of comment --- */
% This is a single-line comment
```

Comments are not parts of the executable program, but are used by the programmer as a documentation aid.

Identifiers (Names)

An identifier (name) consists of a sequence of letters and digits. The first character must be a letter. The underscore character _ is considered a letter. EDEN imposes no limit on the number of characters in a name, but the implementation of the EDEN interpreter does (about 255 characters). An EDEN keyword cannot be used as an identifier.

Examples of identifiers

```
hello
this_is_a_most_unusually_long_name
IF
foO
bAr
HorseSense
var1
var2
_auto_
_
```

Upper- and lowercase letters are distinct, so `Count` and `count` are different identifiers. An identifier is a symbolic name of a variable [p 20] , function [p 20] or other objects.

Examples of character sequences that *cannot* be used as identifiers:

```
012
a fool
$sys
auto
3var
pay.due
foo~bar
.name
if
```

Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

```
append
auto
break
case
continue
default
delete
do
else
for
func
if
insert
is
para
proc
return
shift
switch
while
```

Objects and Lvalues

An *object [p 20]* is a manipulatable region of storage; an *lvalue* is an expression referring to an object. An obvious example of an lvalue expression is an identifier. There are operators which yield lvalues: for examples, if `E` is an expression of pointer type, then `*E` is an lvalue expression referring to the object to which `E` points. The name ‘‘lvalue’’ comes from the assignment expression `E1 = E2` in which the left operand `E1` must be an lvalue expression. The discussion of each operator below indicates whether it expects lvalue operands and whether it yields an lvalue.

An object can be any one of the following:

- Read/Write Variable [p 20]
- Function Definition [p 20]
- Formula Variable [p 20]

- Action Specification [p 21]

Objects: Note

The vocabulary we use is based on the C reference manual. The term *object* is *not* used in the OOP sense.

Read/Write Variables

A *read/write variable* (RWV) is a named storage that holds a data. The data can be any EDEN data structure [p 21] . There is no need to declare a variable before it is used. Memory storage is allocated when the variable name first appears in a program. Data can be assigned to RWV's using assignment statements [p 30] . For example,

```
V = 1;
```

The identifier (name) `V` denotes an RWV whose value is assigned to integer `1` by the operator =. The semi-colon terminates the statement.

The value of an RWV is referenced if its name appears in an expression. If the RWV has not been assigned a value before its value is referenced, the value `@ [p 21]` (meaning ‘‘undefined’’) is assigned to it automatically.

Different typed data can be assigned to the same variable at different time. The EDEN interpreter checks for data type clash at run-time.

The interpreter assumes all objects are RWV's until they are defined to be *functions [p 20]* , *formula definitions [p 20]* or *action specifications [p 21]* .

Function Definition

A *function definition* is an object that stores an entry point (address) of a sequence of instructions (the function) which computes a result from the parameters. There are some functions pre-defined [p 44] by the intepreter. The user can define his/her own functions [p 36] using some procedural statements.

The value of a function definition is equivalent to the function [p 26] (address of the instruction sequence).

Note that the interpreter allows an identifier designating a RWV [p 20] to be re-used as a function name but not vice versa, because the intepreter assumes all objects are RWV [p 20] 's initially.

Formula Variable

A *formula variable* is an object that stores a formula expression (unlike a RWV [p 20] which stores a value) and is bound to that expression using a *formula definition [p 39]* . The value of a formula variable is equivalent to the current value of the formula expression that the variable stores. To optimize the computation speed, the formula variable also stores the up-to-date value of the formula

expression. So, whenever, the value of the formula variable is referenced, this value is used. The interpreter is responsible for updating this value.

An identifier designating a RWV can be re-used to designate a formula variable and vice versa.

Action Specification

An *action specification* is an object that stores an entry point (address) of an instruction sequence - the *action procedure [p 39]* . It also stores a list of objects on which the action depends. Whenever the values of these objects (usually are RWV's or formula variables) are changed the action procedure will be invoked by the system.

Note that the value of an action specification is equivalent to the address of the action procedure.

An identifier designating a RWV can be re-used as an action name but not vice versa.

Data Structures

There are 8 different data types, as follows (integer and character types are sometimes collectively called *integral type*.

- @ (undefined) [p 21]
- Integer [p 21]
- Character [p 22]
- Floating point [p 23]
- String [p 24]
- Pointer [p 24]
- List [p 25]
- Function [p 26]

@ (undefined)

The constant `@` (undefined [p 21]) is a special value. It has a unique (un-named) data type. Some operators [p 27] , such as arithmetic operators [p 28] accept `@` as their operands, and usually `@` will be returned. Other operators which do not take `@` operands will generate an error.

@: Note

`@` is similar to the ‘‘bottom’’ of some algebras.

Integer

An *integer* can be specified using one of three formats.

Decimal Integer Constants

A *decimal integer constant* is a stream of digits with non-leading zero, for example:

`123`	*valid*
`A123`	*invalid (begins with non-digit)*
`0123`	*invalid (begins with zero but is a valid octal)*

Octal Integer Constants

An *octal integer constant* is a stream of digits with a leading zero. The digits `8` and `9` have octal values 10 and 11 respectively. For example:

`0456`	*valid (= 302 decimal)*
`018`	*valid (= 020 octal = 16 decimal)*
`456`	*invalid (but is a valid decimal)*
`A456`	*invalid (begins with non-digit)*

Hexadecimal Integer Constants

An *hexadecimal integer constant* is a stream of hexadecimal digits starting with `0x` (zero followed by the character x). Letters `A` to `F` have the hexadecimal values 10 to 15 respectively. Lowercase letters `a` to `f` are equivalent to their corresponding uppercase letters, for instance:

`0xAB`	*valid (= 171 decimal)*
`0x1f`	*valid (= 31 decimal)*
`AB`	*invalid (begins with an alphabet letter)*
`01f`	*invalid (missing x)*

Character

A *character constant* is a character enclosed in single quotes, as in `'x'`. The value of a character constant is the numerical value of the character in the machine's character set, for example:

`'A'`	*valid (value is 65 for the ASCII character set)*
`'AB'`	*invalid (more than one character)*

Certain special characters, such as the single quote `'` and the backslash `\`, may be represented according to the following table of escape sequences:

newline	NL(LF)	`'\n'`
horizontal tab	HT	`'\t'`
backspace	BS	`'\b'`
carriage return	CR	`'\r'`
form feed	FF	`'\f'`
backslash	\	`'\\'`
single quote	’	`'\''`
double quote	"	`'\"'`
null	NUL	`'\0'`
bit pattern	ddd	`'\`*ddd*`'`

The escape `\`*ddd* consists of the backslash followed by a stream of octal digits which are taken to specify the value of the desired character. If the character following a backslash is not one of those specified, the backslash is ignored.

Floating Point

A *floating point constant* consists of an integer part, a decimal point, a fraction part, and an exponent part; where an exponent part is an `e` or `E`, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the `e` (`E`) and the exponent (not both) may be missing.

Note that a floating point constant may not contain any embedded blanks or special characters.

Some floating point constants are:

```
1.23
.23
0.23
1.
1.0
1.2e10
1.23e-15
```

but not

`1,000.00`	*comma not allowed*
`1 000.00`	*embedded space not allowed*
`1000`	*decimal point or exponential part needed*
`.e0`	*integer part or fractional part needed*
`-3.14159`	*this is a floating point expression, not a constant*

String

A *string constant* is a character sequence enclosed in double quotes:

```
"this is a string"
```

The backslash convention for representing special characters can also be used within a string. This makes it possible to represent the double quote and the escape character backslash `\` within a string. It is not possible for a string to contain the `\0` (NUL) character since it is used to represent the end of string internally. Therefore,

```
"this is the end\0here we passed the end of string"
```

is equivalent to

```
"this is the end"
```

If *s* denotes a string expression, and *i* an index (an integer expression) then *s*`[`*i*`]` denotes the *i*th character in the string.

An individual character of a string can be accessed randomly by giving the index - an integer expression enclosed in `[]` - after the string expression. The first character is numbered 1 (not zero). For example, if

```
s = "abcdef";
```

then

`s[1]` is the first character of s, i.e. 'a'.

`s[i]` is the ith character of s.

It is illegal to index beyond the current number of characters in a string. So `s[7]` is an error since `s` has only six characters. The suffix operator `#` returns the current number of characters in the string. In the last example, `s#` is 6. For the empty string, `""#` is 0.

Pointer

A *pointer* is the address of an object in the memory space. The prefix operator `&` gets the address of an object. For example,

```
iptr = & int_variable;
cptr = & s[5];    /* if s is a string, cptr points to the 5th char of s */
```

The prefix operator `*` refers to the object which the pointer is pointing to. For example,

```
i = * iptr ;    /* i = int_variable; */
c = * cptr ;    /* c = s[5]; */
```

There are no pointer constants, and no pointer operations except the pointer equality checks.

List

The *list* is the only structured data type in EDEN. Data items of different types can be grouped to form a list using the *list formation operator [p 28]* `[]`. Commas are used to separate the data. The whole list is considered as a single data item. For example, the list

```
[ 100, 'a', "string", [1,2,3] ]
```

holds four items: an *integer [p 21]* (`100`), a *character [p 22]* (`'a'`), a *string [p 24]* (`"string"`) and a *list* (`[1,2,3]`).

There are no list constants in the language. The characteristics of the list data type are:

- Different typed data can be stored in the same list. For instance,

  ```
  L = [ 100, 'a', "string", [1,2,3] ];
  ```

  `L` holds four items: an *integer [p 21]* (`100`), a *character [p 22]* (`'a'`), a *string [p 24]* (`"string"`) and a *list* (`[1,2,3]`).
- The individual items of a list can be accessed randomly by giving the index (an integer expression quoted by `[]`) after the list. The first item is numbered 1 (not zero). For example,

`L[1]`	is the first item of `L`, i.e. 100 (in the previous example).
`L[4][2]`	is 2

- It is illegal to index beyond the current number of items of a list. So `L[5]` is an error since `L` has only four items.
- The suffix operator # returns the current number of items in the list. In the last example, `L#` is 4, and `L[4]#` is 3. For the empty list, `[]#` is 0.
- The infix operator `//` returns the concatenated list of two lists; for example,

  ```
  [1,2,3] // [5,6,7]
  ```

  produces

```
[1,2,3,5,6,7]
```

- List-related statements are: **append [p 33]** , **insert [p 33]** , **delete [p 33]** and **shift [p 33]** statements.
- There are no list constants.

Function

EDEN allows the user to define [p 36] functions (or procedures). There are also pre-defined functions [p 44] , such as `write`, `writeln`, and `exit`.

Expressions

The precedence of expression operators is as follows, highest precedence first.

- Arithmetic operators [p 28]
- String and list operators [p 28]
- Pointer operator [p 29]
- Relational operators [p 29]
- Equality operators [p 29]
- Logical operators [p 29]
- Conditional operator [p 30]
- Assignment operators [p 30]

All integer overflows are ignored in the current implementation. Division by 0 causes a run-time error.

- Lvalue [p 26]
- Primary expression [p 27]

Lvalue

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

The dollar sign `$` is the actual argument list of a function; `$`*number* is the *number*-th argument where *number* is a decimal integer constant.

An lvalue followed by an expression in square brackets is an lvalue. The intuitive meaning is that of a subscript. The lvalue must have type string or list. The expression must be of integer type.

The unary `*` operator means indirection: the expression must be a pointer, the result is an lvalue.

An expression enclosed in a pair of `` ` `` (open quotes) is a lvalue where the expression must be of string type. The object, referred to by it, is the object having the name identical to the string. For instance, `` `"A"` `` is equivalent to the object `A`.

A *parenthesized lvalue* is a lvalue which is identical to the unadorned lvalue.

Primary Expression

```
primary-expression:
        lvalue
        ( expression )
        primary-expression [ expression ]
        primary-expression ( expression-list opt )
```

An *lvalue [p 26]* is an simple expression. The value of the object is returned.

A *parenthesized expression* is a primary expression whose type and value are identical to those of the unadorned expression.

A *primary expression* followed by an expression in square brackets is a primary expression. The intuitive meaning is that of subscript. The primary expression must have type string [p 24] or list [p 24] .

A *function call* is a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the actual arguments to the function. The primary expression must be of type function [p 20] .

In preparing for the call to a function, a copy is made of each actual parameter (even it is a string or a list); thus all argument-passing in EDEN is strictly by value. A function may change the values of its formal parameters, but these changes cannot affect the values of the actual parameters. On the other hand, it is possible to pass a pointer on the understanding that the function may change the value of the object to which the pointer points. The order of evaluation of arguments is undefined by the language. (Although the current implementation evaluates parameters from left to right, the user should not assume this fact.)

Operators

- Arithmetic Operators [p 28]
- String and List Operators [p 28]
- Pointer Operator [p 29]
- Relational Operators [p 29]
- Equality Operators [p 29]
- Logical Operators [p 29]
- Conditional Operator [p 30]
- Assignment Operators [p 30]
- Precedence and Order of Evaluation [p 31]

Arithmetic Operators

Expressions with binary operators group left-to-right. The usual arithmetic conversions are performed.

```
arithmetic-expression:
        expression + expression
        expression - expression
        expression * expression
        expression / expression
        expression % expression
        - expression
```

The result of the + operator is the sum of the operands.

The result of the − operator is the difference of the operands.

The binary * operator indicates multiplication.

The binary / operator indicates division.

The binary % operator yields the remainder from the division of the first expression by the second. The two operands must be of integral type (i.e. integers [p 21] or characters [p 22]).

The unary − operator is the negative of its operand.

All arithmetic operators are strict, i.e. they return @ [p 21] if either operand is @ [p 21] .

If both operands of an arithmetic expression are of integral type, the result is always an integer [p 21] . If any operand is a floating point [p 23] , the result is a floating point [p 23] (except for the % operator which takes integral operands only).

String and List Operators

```
string-expression:
        expression // expression

list-formation-expression:
        expression // expression
        [ expression-list opt ]

expression-list:
        expression
        expression , expression-list

integer-expression:
        expression #
```

If the operands of // are of string [p 24] or character [p 22] type, the result will be a string [p 24] which is the concatenation of the two operands. If the operands are of list type, the result will be a list which is the concatenation of the two operands. If either or both of the operands is @ [p 21] , the result will be @ [p 21] (see pre-defined functions substr and sublist). The [] operator groups the values of the expressions into a list. The expression list is a list of expressions separated by commas. If the expression list is omitted the list returns a null list.

Pointer Operator

```
pointer-expression:
        & lvalue
```

The result of the unary `&` operator is a pointer [p 24] to the object referred to by the lvalue.

Relational Operators

The relational operators group left-to-right, but this fact is not very useful; `a<b<c` (meaning `(a<b)<c`) does not mean what it seems to be ((a<b) and `(b<c)`), as in a normal mathematical expression.

```
relational-expression:
        expression < expression
        expression > expression
        expression <= expression
        expression >= expression
```

The operators < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to) all yield `0` if the specified relation is false and `1` if it is true. The type of the result is integer [p 21] . The usual arithmetic conversions are performed.

Two strings [p 24] may be compared. Single characters [p 22] are compared from the left to the right according to their codes in the machine's character set. Note that a string is always terminated by `\0`, so the shortest string is considered the smaller. Strings are equal only if their lengths as well as their contents are identical.

It is an error if the two operands are of different types, but if either argument in an relational expression is `@ [p 21]` then the value is `@ [p 21]` . Lists [p 25] and pointers [p 24] cannot be compared using the relational operators.

Equality Operators

```
equality-expression:
        expression == expression
        expression != expression
```

The == (equal to) and `!=` (not equal to) operators are exactly analogous to the relational operators [p 29] except for their lower precedence (thus `a<b == c<d` is `1` whenever `a<b` and `c<d` have the same truth-value).

Note that two lists [p 25] can be compared for equality. Lists are equal only if their lengths as well as their contents are identical. Two pointers [p 24] can be compared for equality. Pointers are equal only if they points to the same object.

Logical Operators

logical-expression:
        expression && *expression*
        expression and *expression*
        expression || *expression*
        expression or *expression*
        ! *expression*
        not *expression*

EDEN operates in 3-value logic (true, false and @ [p 21]). Any non-zero value means true; zero means false. In addition to the lazy operators (as in C, left-to-right evaluation, the second operand is evaluated only if necessary), eager operators - and, or, not - are also defined. Notice that the truth tables for the lazy and the corresponding eager operators are not identical.

@ && @ = @	@ and @ = @	@ || @ = @	@ or @ = @
@ && F = @	@ and F = @	@ || F = @	@ or F = @
@ && T = @	@ and T = @	@ || T = @	@ or T = @
F && @ = F	F and @ = @	F || @ = @	F or @ = @
F && F = F	F and F = F	F || F = F	F or F = F
F && T = F	F and T = F	F || T = T	F or T = T
T && @ = @	T and @ = @	T || @ = T	T or @ = @
T && F = F	T and F = F	T || F = T	T or F = T
T && T = T	T and T = T	T || T = T	T or T = T

! @ = T	not @ = @
! F = T	not F = T
! T = F	not T = F

Truth Tables for operators: &&, and, ||, or, !, not.

Conditional Operators

conditional-expression:
        expression ? *expression* : *expression*

Conditional expressions group right-to-left. The first expression is evaluated and if it is true, the result is the value of the second expression, otherwise (i.e. false or @ [p 21]) that of third expression. The second and third expressions need not have the same type; moreover only one of them is evaluated.

Assignment Operators

```
assignment-expression:
        lvalue = expression
        lvalue += expression
        lvalue -= expression
        ++ lvalue
        lvalue ++
        -- lvalue
        lvalue --
```

There are three binary assignment operators, =, +=, and −=, all of which group right-to-left. All require an lvalue as their left operand. The right operand is evaluated and the value stored in the left operand after the assignment has taken place.

In the simple assignment with =, the value of the expression replaces that of the object referred to by the lvalue.

The behaviour of an expression of the form `E1` *op*`= E2` may be inferred by taking it as equivalent to `E1 = E1` *op* `(E2)`; however, `E1` is evaluated only once. Both of the operands must be of integral types.

The object referred to by the lvalue operand of prefix ++ is incremented. The value is the new value of the operand, but is not an lvalue. The expression `++x` is equivalent to `x+=1`.

When postfix ++ is applied to an lvalue the result is the value of the object referred to by the lvalue. After the result is noted, the object is incremented in the same manner as for the prefix ++ operator. The type of the result is the same as the type of the lvalue expression.

The lvalue operands of the prefix and postfix −− is decremented analogously to the prefix and postfix ++ operators respectively.

All the objects referred to by the lvalues of these assignment operators must be read/write variables, not formula variables.

Precedence and Order of Evaluation

The table below summarizes the rules for precedence and associativity of all operators. Operators on the same line have the same precedence; rows are in order of decreasing precedence, so, for example, `*`, `/`, and `%` all have the same precedence which is higher than that of + and −.

Operator	*Precedence*
`() [] ```	left to right
`! not ++ -- - * & #`	right to left
`* / %`	left to right
`+ - //`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`&& and`	left to right
`|| or`	left to right
`?:`	right to left
`= += -=`	right to left

Procedural Statements

Procedural Statements are executable statements. The statements are executed in sequence except as indicated.

- Expression Statements [p 32]
- Insert Statement [p 33]
- Append Statement [p 33]
- Delete Statement [p 33]
- Shift Statement [p 33]
- Compound Statement [p 34]
- Condition Statement [p 34]
- While Statement [p 34]
- Do Statement [p 34]
- For Statement [p 35]
- Switch Statement [p 35]
- Break Statement [p 35]
- Continue Statement [p 36]
- Return Statement [p 36]
- Null Statement [p 36]

Expression Statement

Most statements are *expression statements*, which have the form:

expression ;

Usually expression statements are assignments [p 30] or function/procedure [p 26] calls. For instance:

```
I = 1;                /* assignment statement*/
J++;                  /* another form of assignment */
DoSomething(I, J);    /* procedure call */
```

Insert Statement

The statement

insert *lvalue* , *expression-1* , *expression-2* ;

inserts a value evaluated from *expression-2* into the list [p 25] object referred to by lvalue at the position *expression-1*; *expression-1* must be of integral type and cannot be smaller than `1` or be greater than the current number of items of the list plus one. The list object referred to by *lvalue* must be a read/write variable [p 20] .

Append Statement

The statement

append *lvalue* , *expression* ;

appends a value evaluated from *expression* to the end of the list [p 25] referred to by lvalue. It is equivalent to do

insert *lvalue* , *lvalue* # + 1 , *expression* ;

but the *lvalue* expression is evaluated only once.

Delete Statement

The statement

delete *lvalue* , *expression* ;

deletes a value from the list [p 25] object referred to by lvalue at the position evaluated from *expression*; *expression* must be of integral type and cannot be smaller than `1` or be greater than the current number of items of the list; whence the list cannot be a null list. The list object referred to by *lvalue* must be a read/write variable [p 20] .

Shift Statement

The syntax of the shift statement is:

shift *lvalue*$_{opt}$;

The shift statement deletes the first value from the list [p 25] referred to by *lvalue*; whence the list cannot be a null list. If *lvalue* is omitted, the argument list `$` is assumed; hence it can only be used within a function body. The equivalent **`delete [p 33]`** statements of the shift statements are:

```
delete $ , 1 ;      /* shift ; */
delete lvalue , 1 ;         /* shift lvalue ; */
```

Compound Statement

So that several statements can be used where one is expected, the compound statement is provided:

```
compound-statement:
        { statement-list opt }

statement-list:
        statement
        statement  statement-list
```

Condition Statement

The two forms of the conditional statement are:

```
if ( expression ) statement
if ( expression ) statement else statement
```

In both cases the expression is evaluated and if it is true, the first sub-statement is executed. In the second case the second sub-statement is executed if the expression is false or `@ [p 21]` . As usual the ‘‘else’’ ambiguity is resolved by connecting an **`else`** with the last encountered **`else`**-less **`if`**.

While Statement

The while statement has the form

```
while ( expression ) statement
```

The sub-statement is executed repeatedly so long as the value of the expression remains true. The test takes place before each execution of the statement.

Do Statement

The do statement has the form

```
do statement while ( expression ) ;
```

The sub-statement is executed repeatedly until the value of the expression becomes false or `@ [p 21]` . The test takes place after each execution of the statement.

For Statement

The for statement has the form

for (*expression-1* $_{opt}$ **;** *expression-2* $_{opt}$ **;** *expression-3* $_{opt}$) *statement*

This statement is equivalent to

```
expression-1 ;
while ( expression-2 ) {
        statement
        expression-3 ;
}
```

Thus the first expression specifies initialization for the loop; the second specifies a test, made before each iteration, such that the loop is exited when the expression becomes false or `@ [p 21]` ; the third expression often specifies an incrementation which is performed after each iteration.

Any or all of the expressions may be dropped. A missing *expression-2* makes the implied **while** clause equivalent to

```
while (1);
```

other missing expressions are simply dropped from the expansion above.

Switch Statement

The switch statement causes control to be transferred to one of several statements depending on the value of an expression. It has the form

switch (*expression*) *statement*

The statement is typically compound [p 34] . Any statement within the statement may be labelled with one or more case prefixes as follows:

case *constant* **:** *statement*

There may also be prefix of the form:

default **:** *statement*

When the **switch** statement is executed, its expression is evaluated and compared with each **case** constant. If one of the **case** constants is equal to the value of the expression, control is passed to the statement following the matched **case** prefix. If no **case** constant matches the expression, and there is a **default** prefix, control passes to the prefixed statement. If no **case** matches and if there is no **default** then none of the statements in the **switch** is executed.

Break Statement

The statement

```
break ;
```

causes termination of the smallest enclosing **while [p 34]** , **do [p 34]** , **for [p 35]** , or **switch [p 35]** statement; control passes to the statement following the terminated statement.

Continue Statement

The statement

```
continue ;
```

causes control to pass to the loop-continuation portion of the smallest enclosing **while [p 34]** , **do [p 34]** , or **for [p 35]** statement; that is to the end of the loop.

Return Statement

A function [p 26] returns to its caller by means of the return statement, which has one of the forms:

```
return expression opt ;
```

In the optional expression is omitted the value returned is `@ [p 21]` . Otherwise, the value of the expression is returned to the caller of the function. Flowing off the end of a function is equivalent to return `@ [p 21]` .

Null Statement

The null statement has the form:

```
;
```

A null statement is useful to carry a label just before the } of a compound statement [p 34] , for example:

```
switch (...) { ... case 1: ; }
```

or to supply a null body to a looping statement such as while, for example:

```
while (1) ; /* an infinite waiting loop */
```

User-defined Functions

Function definitions have the form:

```
function-definition:
        function-declarator  function-body

function-declarator:
        func identifier
        proc identifier

function-body:
        { para-alias opt  local-var-decl opt  statement-list opt  }

para-alias:
        para identifier-list opt  ;

local-var-decl:
        auto identifier-list opt  ;

identifier-list:
        identifier
        identifier , identifier-list
```

The identifier is declared to be a function [p 20] using the keywords **func** or **proc**. There is no difference between **func** and **proc**. The word **proc** is more meaningful when the function serves as a procedure, i.e. a function which does not return a value.

- All parameters are passed by value. The actual parameters forms a list called `$` (dollar sign); notice that there is no parameter list after the identifier. The first argument is `$[1]`, the second argument is `$[2]`, and so forth. For convenience, `$[1]`, `$[2]`, ... can be replaced by `$1`, `$2`, ...
- **para** gives nicknames to the parameters. The first identifier in *para-alias* matches `$1` and so forth. The number of identifiers does not required to match the number of actual parameters. `$`*n* can be referenced by either `$`*n*, `$[`*n*`]` or the nickname given.
- `$#` is the current number of arguments. List operations are applicable to `$`.
- All parameters are passed by value.
- Local variables must be declared at the beginning of the function body and are preceded by the keyword **auto** (means dynamically allocated). All local variables are RWV's. There is no need to declare the type of local variables because EDEN does the run-time type checking. The syntax is:

  ```
  auto identifier-list ;
  ```

  where identifier-list is a list of identifiers separated by commas.
- The return statement returns the value of the expression. The syntax is:

  ```
  return expression opt  ;
  ```

  If the expression is omitted, `@` will be returned. Flowing off the end of a function is equivalent to return `@`.
- The value returned can be ignored by the caller. In this case, the function acts as a procedure. The keyword **func** can be replaced by **proc**. There are no differences between these two keywords. Thus they can be interchanged. The purpose of having another keyword is to self-comment the

program. It is intended (not restricted) that func should be used for defining side-effect-free operators.
For example:

```
func max    /* returns the max. value of its arguments */
{
    para m;              /* m is the first argument $1 */
    auto i;              /* declare local variables */

    for (i = 2; i <= $#; i=i+1) {
                         /* for the other arguments */
        if ($[i] > m)    /* the ith argument */
            m = $[i];
    }
    return m;            /* returns max to the caller */
}
```

defines a function named `max` which returns the maximum value of its arguments.

- To call a function, the arguments must be put in parentheses preceded by the function name. For example,

  ```
  MaxNumber = max(0,i,j,k);
  ```

  evaluates the maximum value of `0`, `i`, `j` and `k`, and stores the results in the RWV `MaxNumber`. The parentheses cannot be omitted even if no arguments are passed to the function.
- The function name, itself, denotes an entry point (a pointer) of the function code. It is a valid value (a pointer to a function) that can be used in an expression. Thus, if

  ```
  F = max;
  ```

  then the statement

  ```
  MaxNumber = F(1,3,2);
  ```

  assigns `3` to `MaxNumber`.
  Also a function can be put into a list. For example, if

  ```
  G = [ min, max ];
  ```

  where min is supposed to be a function that returns the minimum value of its arguments, then

  ```
  Num = G[n](1,3,2);
  ```

  `Num` will have the value `1` if `n` is `1` (`G[1]=min`), `3` if `n` is `2` (`G[2]=max`).
- Once an identifier is used as a name of a function/procedure/action, it cannot be re-used to designate a RWV/formula variable. This restriction is posed by the interpreter (not by the language) to prevent destroying a function definition accidentally.

Definitive Statements

These two types of statement are crucial to EDEN, distinguishing it from a non-definitive language.

- Formula Definition [p 39]
- Action Specification [p 39]

Formula Definition

A formula definition has the form:

```
formula-definition:
        identifier is expression ;
```

It is normally expected that the operators appearing in the *expression* are ‘‘pure’’ functions [p 26] , i.e. without side-effect. The interpreter does not allow the use of assignment operators [p 30] in the *expression*, but does not otherwise detect the use of ‘‘impure’’ functions. It is not advisable to use such functions.

In addition, a formula variable cannot be (directly or indirectly) circularly defined. For example,

```
f is f + 1;
```

is an error because `f` is defined in terms of itself, and not logically meaningful. The definitions:

```
i is j;
j is i;
```

are syntactically correct, but are conflicting with the restriction. Thus the EDEN interpreter rejects the second definition, and produces the error message:

```
j : CYCLIC DEF : ABORTED near line ...
```

The examples below are valid definitions of formula variables (assuming they are not circularly defined):

```
f is a + b;
M is max(a,b,c);
```

The EDEN interpreter controls all the evaluations of formulae. The order of evaluation is not specified by the language, i.e. can happen in any order. One version of EDEN interpreter may evaluate the formula of `f` before that of `M`, whenever the value of `a`, for instance, is changed, but some other versions may do it in the reverse order. A concurrent EDEN system might do both evaluations in parallel. Therefore, the user should not make any assumption on the order of evaluation.

Action Specification

An action specification has the form:

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

dependency-list:
        : identifier-list
```

The *function declarator* and *function body* are the same as those of function definition [p 36] . An identifier is declared to be an action using the keywords **`proc`** or **`func`**. There is no difference between **`proc`** and **`func`**; but the user is recommended to use **`proc`** because an action usually serves as a procedure.

The dependency list is a list of comma-separated identifiers preceded by a colon.

An action is normally a procedure invoked automatically by the system when any object specified in the dependency list is changed. No parameters will be passed to the procedure; so the argument list `$` is always empty when called by the system. All values returned by the actions are ignored by the system.

The order of invoking actions is controlled by the EDEN interpreter, but the user can invoke an action explicitly by giving a pair of parenthesis (a null argument list) after the action name (exactly like a procedure/function call). This is useful for debugging.

Notice that if the identifier list is omitted in the dependency list the action is just an ordinary procedure.

The use of actions (and the use of ‘‘impure’’ functions as operators in formulae) leads to procedural activity outside the user’s direct control when expressions are evaluated. The user should make no assumptions about the order in which such procedural actions are performed. For instance, no two actions that can be invoked in the same time should write to the same RWV. EDEN does not provide any facility to detect or prevent such ‘‘interference’’ between actions.

Miscellaneous Commands

- Dependency Link [p 40]
- Query [p 41]

Dependency Link

An alternative way of defining an action specification is first to declare the action as a procedure/function, and then specify the dependency link using the command:

```
dependency-link:
        identifier ~> [ identifier-list ] ;
```

where each *identifier* in the *identifier list* refers to an action (procedure) that depends on the value of the object referred to by *identifier*. So the action:

```
proc p : a, b, c { /* action body */ }
```

is equivalent to

```
a ~> [ p ] ;
b ~> [ p ] ;
c ~> [ p ] ;
```

Note that the dependency link command adds dependency linkage to actions. If the user wants to remove a particular linkage, the whole action must be re-specified. For example, to remove c from the specification of action p above, p must be re-defined as:

```
proc p : a, b { /* action body */ }
```

Query

The query command has the form:

```
? lvalue ;
```

The query command prints the definition of the object referred to by lvalue on the stdout of the UNIX environment. The format of printing a RWV [p 20] is:

```
value
RWV-name ~> [ identifier-list ] ;
```

The format of printing a formula variable is:

```
formula-variable-name is expression ;
formula-variable-name ~> [ identifier-list ] ;
```

The format of printing a function definition is:

```
func function-name
{ function-body }
function-name ~> [ identifier-list ] ;
```

where **proc** may be printed instead of **func** depending on which word was used by the user.

The format of printing an action specification is:

```
proc action-name : identifier-list
{ action-body }
action-name ~> [ identifier-list ] ;
```

where **func** may be printed instead of **proc** depending on which word was used by the user.

The query command is an *ad hoc* feature in the language and contains bugs in the current implementation of EDEN; for instance,

```
? L[1];
```

prints the definition of `L` instead of `L[1]`.

Syntax Summary

```
expression:
        primary-expression
        - expression
        ! expression
        & lvalue
        expression #
        ++ lvalue
        lvalue ++
        -- lvalue
        lvalue --
        [ expression-list opt ]
```

```
        expression binop expression
        expression ? expression : expression
        lvalue asgnop expression

primary-expression:
        lvalue
        ( expression )
        primary-expression [ expression ]
        primary-expression ( expression-list opt )

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 lvalue opt ;
        if ( expression ) statement
        if ( expression ) statement else statement
        while ( expression ) statement
        do statement while ( expression ) ;
        for ( expression opt ; expression opt ; expression opt ) statement
        switch ( expression ) statement
        case constant : statement
        default : statement
        break ;
        continue ;
        return expression opt ;
        ;

compound-statement:
        { statement-list opt }

statement-list:
        statement
        statement  statement-list

function-definition:
        function-declarator  function-body

function-declarator:
        func identifier
        proc identifier
```

```
function-body:
        { para-alias opt   local-var-decl opt   statement-list opt  }

para-alias:
        para identifier-list opt  ;

local-var-decl:
        auto identifier-list opt  ;

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 [p 27] 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:
        =     +=     -=
```

Pre-defined Variables

The following is a list of pre-defined read/write variables. The user should not alter the value of these variables, except `autocalc`.

`stdin`, `stdout`, `stderr`
: These three variables are pre-defined file pointers equivalent to those file pointers in C. All of them are of integer type.

`autocalc`
: If the value of `autocalc` is set to 0, the mechanism of auto-recalculate of formula definitions will be switched off. In such a situation, formulae and actions will not be recalculated or invoked when the values of those variables on which they are dependent are changed. However, when `autocalc` is set to a non-zero value, the auto-recalculation mechanism will be back in action. By default, `autocalc` is set to 1.

`calc_list`
: In the earliest version of EDEN interpreter, the pointers pointing to the formula variable and action specifications are queued in the list variable `calc_list`. The interpreter would not update or invoke those queued formula variables and actions when `autocalc` is switched ‘‘off’’. The following action specification can update these queued formulae and actions automatically when `autocalc` is on again:

```
proc _autocalc : autocalc {
    while (calc_list#) {
        *calc_list[1];       /* update formula var's */
        shift calc_list;     /* remove from list */
    }
}
```

Note that this action will only be invoked when the auto-calculate mechanism is on (i.e. `autocalc` is non-zero). This action forces the formula variables to be updated by simply evaluating them (the interpreter always updates the formula variables if they are evaluated). Although this action does not invoke queued actions, the updating of the formula variables will cause the related actions to be invoked.

In the new version of EDEN interpreter, all queued formulae and actions would be updated and invoked as soon as the auto-recalculate mechanism is switched on. The references to the formula variables and action specifications are stored in an internal format and hence the variable `calc_list` is not needed. Thus the user cannot alter the list. To inspect the formula variable and action specification queues while `autocalc` is ‘‘off’’, the user can call the pre-defined functions, `formula_list() [p 44]` and `action_list() [p 44]` , which transform the internal information into EDEN’s list type.

Pre-defined Functions

A number of functions are pre-defined by the EDEN interpreter. These pre-defined functions cannot be re-defined by the users to prevent ruining their definition accidently. The following is a list of pre-defined functions.

I/O Functions

`write(...)`, `writeln(...)`

`write` and `writeln` print the arguments on *stdout* (standard output file). `writeln` appends a newline `\n` at the end. For example,

```
for (i = 1; i <= 10; i++) write(i, ' ');
```

gives

```
1 2 3 4 5 6 7 8 9 10
```

```
writeln("sum of ", 1, '+', 2, " = ", 1 + 2);
```

gives

```
sum of 1+2 = 3
```

Type Conversion Functions

`type` (*data*)

It returns the type of data as a string:

Data Type	String Returned
`@ [p 21]` ‘‘undefined’’	`"@"`
integer	`"int"`
character	`"char"`
string	`"string"`
floating point	`"float"`
list	`"list"`
function	`"func"`
procedure	`"proc"`
pre-defined function	`"builtin"`
pre-bound C function	`"C-func"`

For example, `type(1)` gives `"int"`.

`int` (*data*)

If *data* is an integer, the same value will be returned. If it is a character, the code of the character will be returned. If it is a string, the string will be assumed as a stream of digits, and these digits are converted into an integer which will be returned. If it is a floating point, it will be truncated into an integer before it is returned. If it is a pointer, the address is casted to an integer which will be returned. The function returns `@ [p 21]` otherwise.

`char` (*data*)

If *data* is a character, the same character will be returned. If it is an integer, the character having the code equals to the integer will be returned. If it is a string, its first character will be returned.

If it is a floating point, it will be truncated into an integer and then cast to a character before it is returned. The function returns `@ [p 21]` otherwise.

`str`(*data*)

If *data* is a string, the same string will be returned. If it is `@ [p 21]` the string "@" will be returned. If it is a character, the character will be converted into a string and returned. If it is an integer or a floating point, the string of digits corresponding to the value will be returned. The function returns `@ [p 21]` otherwise.

`float`(*data*)

If *data* is a floating point, the same value will be returned. If it is an integer, the floating point having the same value will be returned. If it is a character, the floating point having the same value as the code of the character will be returned. If it is a string, the string will be assumed as a stream of digits, and these digits are converted into a floating point which will be returned. If it is a pointer, it will be casted into an integer and then converted to a floating point before it is returned. The function returns `@ [p 21]` otherwise.

String Functions

`substr`(*string string, int from, int to*)

`substr` returns a substring of *string*. *from* and *to* are the starting and ending positions respectively; they are integer values. If *from* is greater than *to*, the null string will be returned. *from* must not be small than 1. If to is greater than the number of characters of string, spaces will be added. For example,

```
substr("1234567890", 4, 8)
```

evaluates to `"45678"`.

Examples:

Example	Result
`substr("1234567890", 4, 8)`	`"45678"`
`substr("1234567890", 5, 3)`	`""`
`substr("1234567890", 7, 12)`	`"7890 "`

`strcat`(*string string,...*)

`strcat` returns the string of the concatenation of its arguments. All arguments must be of string or character types. Characters are considered as a string of length 1. If it is called with no arguments, the null string will be returned. For example,

```
s = strcat("Garden", " of ", "Eden", '.');
```

`s` has the value `"Garden of Eden."`.

`nameof`(*pointer pointer*)

`nameof` returns the name of object to which the first argument (of pointer type) points. The function returns a string. This function is an ad hoc function. **Bugs**: If the pointer points to a character of a string object or an item of a list object, the string or the list object name is returned. For example,

Example	Results
`nameof(&Object)`	`"Object"`
`nameof(&A_String[1])`	`"A_String"`
`nameof(&A_List[5])`	`"A_List"`

List Functions

`sublist` (*list list, int from, int to*)

`sublist` returns a sublist of *list*. *from* and *to* are the starting and ending item positions respectively; they are integer values. If *from* is greater than *to*, the null list `[]` will be returned. *from* must not be small than 1. If *to* is greater than the number of items in list, `@ [p 21]` will be added. For example,

Example	Result
`sublist([1,2,3,4,5], 2, 4)`	`[2,3,4]`
`sublist([1,2,3,4,5], 5, 3)`	`[]`
`sublist([1,2,3,4,5], 4, 7)`	`[4,5,@,@]`

`listcat` (*list list, ...*)

`listcat` returns the list of the concatenation of its arguments. For example,

```
listcat([1,2,3],[4,5,6],[7,8,9])
```

gives

```
[1,2,3,4,5,6,7,8,9]
```

`array` (*int n, data*)

`array` returns a list consists of *n* items that each is *data*. If *data* is omitted, `@ [p 21]` will be assumed. *data* can have any type. For example,

Example	Result
`array(3, 0)`	`[0,0,0]`
`array(4)`	`[@,@,@,@]`

Time Functions

`time()`

`time` returns the current time in seconds since Jan 1, 1970

`ftime()`

`ftime` returns the current time in terms of the time elapsed since Jan 1, 1970. ftime is accurate up to milli-seconds. The return value is a list [*second*, *milli*] where *second* = the number of seconds and *milli* = the number of milli-seconds in addition to the time elapsed since Jan 1, 1970.

`gettime()`

`gettime` returns the current time. The return value is a list of seven integers, the meaning of these integers are respectively:

second	0-59
minute	0-59
hour	0-23
day of month	1-31
month of year	1-12
year	year-1900
day of week	0-6 (0 = Sunday)

Script Functions

`apply` (*func function, list list*)

`apply` calls the function specified in the first argument with the second argument as its actual argument (i.e. `$`). Thus the first and second arguments must be of type function and list respectively. For example,

```
apply(writeln, [1,2,3]);
```

is equivalent to call

```
writeln(1,2,3);
```

`execute` (*string string*)

`execute` executes a string as Eden statements. It returns 0 if no errors occurred in the execution of the string, non-zero otherwise. The execution terminates as soon as it encounters a syntax error or run-time error. Note that it is not a macro; the string must be some valid and complete EDEN statements. Also this function can be executed within a function body. For example,

```
proc reset_F { execute("F is A+B;"); }
```

Every time `reset_F` is called, `F` will be defined to be a formula variable contains the expression `"A+B"` unless there are run-time errors, such as the circular definition conflict.

`todo` (*string string*)

There are inside EDEN two todo lists, one current one and the other is a store for statements to be executed when the current todo list exhausted. `todo` is similar to `execute` except that the todo-statements will only be executed after the current thread of control terminates. For example, in one session you enter:

```
todo("writeln(1);"); writeln("Hello world");
```

The result will be

```
Hello world
1
```

`include` (*string filename*)

`include` works in the same way as `execute` except it takes in a file instead of a string; the file is specified by the first argument which is a string expression. If the execution is successful, include returns 0. The execution terminates as soon as it encounters a syntax error or run-time error. The file name and line number where the error occurs will be reported by the interpreter. For example,

```
include("utility");
```

executes the file called `utility`.

`exit` (*int status*)

`exit` terminates the program and returns *status*, an integer, to the parent process or operating system. If *status* is omitted, 0 is assumed.

`forget` (*string name*)

`forget` remove the entry of variable whose name is indicated by the string name. `forget` returns 0 if success, 1 if it can't find the variable that matches the name, 2 if it refuses to remove the entry because there are some definitions or actions depends on this variable.

Note: it is very dangerous to remove a variable since there may be some references on the variable that cannot be detected (e.g. a reference within a function body). In such case, the system may produce unpredicable result. (This function may be removed or replaced in future releases).

`eager()`

`eager` eagerly evaluate/execute all queued definitions/actions despite of the status of `autocalc [p 44]` .

`touch` (*pointer vp1, vp2, vp3, ...*)

vp1, ... are pointers to variables. touch puts the targets of variables, pointed to by *vp1*, *vp2*, *vp3*, ..., to the evaluation queue.

`formula_list()`

`formula_list` returns the current list of addresses of formula variables queued. This function is useful to inspect the internal queue of formulae when the `autocalc [p 44]` is 0 (off). This function is implemented in the later versions of the EDEN interpreter.

`action_list()`

`action_list` returns the current list of addresses of action specifications queued. This function is useful to inspect the internal queue of actions when the `autocalc [p 44]` is ‘‘off’’. This function is implemented in the later versions of the EDEN interpreter.

`symboltable()`

`symboltable` returns the current internal symbol table of the interpreter as a list. Each symbol entry is of the form:

```
[ name, type, text, targets, sources ]
```

where

name
: the object name (a string)

type
: a string indicates the type of the variable: "var", "formula", "proc", "func", "builtin", "Real-func", "C-func". Note that these types are different from the output from type.

text
: if the object is a function definition, formula variable, or action specification, then text stores the definitions in textual form.

targets
: a list of variable names (strings) by which this object is referenced directly.

sources
a list of variable names (strings) to which this object refers directly.
For example, if the following definitions are defined

```
...
a is b + c;
b = 3;
f is b * g;
...
```

and,

```
S = symboltable());
```

then, `S` is:

```
[ ... [ "a", "formula", "b + c;", [], ["b","c"] ],
    [ "b", "var", "", ["a"], [] ], ... ]
```

This function is useful to inspect the internal queue of actions when the `autocalc [p 44]` is ‘‘off’’. This function is implemented in the new versions of the EDEN interpreter.

`symbols` (*string type*)
`symbols` returns a list of symbol names which are of the type required. The *type* may be:
`"@ [p 21] ", "int", "char", "string", "float", "list", "var"`
any read/write variable (defined by assignments),
`"formula"`
any definitive variable (defined by definitions),
`"func", "proc"`
user-defined functions and procedures,
`"builtin"`
EDEN functions or procedures,
`"Real-func"`
Real C functions,
`"C-func"`
other C functions,
`"any"`
any symbols

`symboldetail` (*string name*), `symboldetail` (*pointer symbol*)
`symboldetail` returns the information of a particular symbol. The information is in the same format as that of symboltable.

Unix Functions

`getenv` (*string env*)
`getenv` returns the string of the environment variable *env*. See `getenv(3)`.

`putenv` (*string env*)
`putenv` sets the environment variable. *env* should have the form: `"name=value"`. See `putenv(3)`.

`error` (*string err_msg*)
`error` generates an EDEN error and prints the error message *err_msg*.

`error_no()`
`error_no` returns the last system (not EDEN) error number (an integer).

`backgnd(`*string path, cmd, arg1, arg2, ...*`)`

`backgnd` executes a process at background named by *path* `backgnd` returns the process id (-1 if fail). **Bug**: `backgnd` may returns a +ve id even if it can't execute the command.

`pipe(`*string path, cmd, arg1, arg2, ...*`)`

`pipe` pipes output (stdout) to process named by *path*. `pipe` returns the process id (-1 if fail). **Bug**: `pipe` may return a +ve id even if it can't execute the command.

`get_msgq(`*int key, flag*`)`

`get_msgq` gets a message queue using key. flag denotes the permission and options (see `msgget(2)`). It returns the message queue id (integer), -1 if fail.

`remove_msgq(`*msqid*`)`

`remove_msgq` removes a message queue whose id is *msgid*. It returns -1 if fail (see `msgctl(2)`).

`send_msg(`*int msqid, [msg_type, string msg_text], flag*`)`

`send_msg` sends a message (text string) to message queue *msqid*. *msg_type* denotes the message type. *msg_text* is a string (terminated by `\0`, i.e. at least having length 1. *flag* is 0 (for wait) and 04000 (octal) for no wait (see `msgsnd(2)`). It returns -1 if fail.

`receive_msg(`*int msqid, msg_type, flag*`)`

`receive_msg` receives message of *msg_type* from message queue *msqid*. flag: c.f. `send_msg` and `msgsnd(2)`. It returns `@ [p 21]` if fail, else [*m_type*, *m_text*] where *m_type* is the actual message type received and *m_text* is the text string received. The following functions are data type related functions. `int`, `char`, `str`, and `float` are type conversion functions.

Predefined C Procedures

The following list of functions are pre-bound C functions. Users should refer to their own reference manual.

Function	Description
`fclose`	close a file
`fgetc`	get a character from an opened file
`fgets`	get a string from a file
`fopen`	open a file
`fprintf`	print to a file
`fputc`	print a character to a file
`fscanf`	formated read data from a file
`gets`	get a string from stdin
`pclose`	close a pipe
`popen`	open a pipe
`putw`	put a machine word to a file
`setbuf`	set the buffer size of a file
`sprintf`	print to a string
`sscanf`	formated read data from a string
`system`	execute command in a sub-shell
`ungetc`	unget a character
`srand`	random number generator
`rand`	random number generator

Functions defined in ‘‘math.h’’

`sin cos tan asin acos atan atan2 sqrt pow log log2 log10 exp exp2 exp10`

Different releases of the EDEN interpreter support different C function packages, such as CURSES (standard UNIX windowing package) and *SunCore* (standard SUN workstation graphics package). The user should consult to the release notes and the packages’ reference manuals for the descriptions of the functions.

Miscellaneous Functions

`debug` (*int status*)

If *status* is a non-zero integer, debug turns on the debugging mode. When the debugging mode is on the interpreter prints out some information showing the status of the machine. Default: off (0). For example,

```
debug(1);        /* turn on debugging mode */
```

It is a subroutine for developing the interpreter. It is available to the user only if the interpreter was compiled with the flag `-DDEBUG`.

`pack(...)`

`pack` allocates a continuous memory space on the heap and stores the data in this memory block. The values are packed in a machine-dependent fashion, so the function is not portable. Data is packed as C type:-

integer	int	(4 bytes)
real	float	(4)
character	char	(1)
string/pointer/function/etc	char *	(4)

`pack` returns the address of the beginning of memory block as an integer. Example:

```
x = [0.0, 100.0, 100.0, 0.0];        /* x-coordinates */
y = [0.0, 0.0, 100.0, 100.0];        /* y-coordinates */
polygon_abs_2(pack(x), pack(y), 4);
                     /* will draw a filled square */
/* in fact, you could do:
     polygon_abs_2(pack([0.0, 100.0, 100.0, 0.0]),
                   pack([0.0, 0.0, 100.0, 100.0]),
                   4);
*/
move_abs_2(0.0, 0.0);
polyline_abs_2(pack(x), pack(y), 4);
                     /* will draw a hollow square */
/* c.f. SunCore Reference Manual */
```

Bugs: It is very *ad hoc*, and not universal, i.e. cannot handle all struct. It converts real numbers into "float"s instead of "double"s (so we can call Suncore polyline and polygon primitives). There should be an "unpack" function.

Advanced Topics

- Adding C library functions to EDEN [p 54]
- Programming Notes [p 57]
  - Memory Allocation [p 57]
  - Stack, Heap and Frame Stack [p 58]
  - Autocalc [p 61]

Adding C Library Functions to EDEN

Although the EDEN interpreter has a number of pre-defined functions, such as substr or writeln, and allows the user to define his own functions within the EDEN environment, the user may find restrictions in special purpose applications, such as a graphics application. The user may find some useful functions which are already written in C. Unfortunately, with the current implementation, there are no ways for the EDEN user to call C (or other languages) functions without recompiling the interpreter.

Some efforts had been done to minimise the difficulties of interfacing with C functions so that the user can call a C function directly from the EDEN environment. A built-in EDEN/C interface can bind a C function to an EDEN name. However, due to some technical problems and the differences in the method of argument passing, not all C functions can be called by the EDEN interpreter through the EDEN/C interface.

Limitations

- The EDEN/C interface assumes that all C functions return values of integer type. So, the C functions must return integers or characters since C automatically casts characters to integers. Functions which return pointers, including character pointers, are accepted only if the pointers are compatible with integers (i.e. they occupy the same amount of memory). Functions which return floating points or structures are not allowed by the current implementation.
- The interface cannot handle macros (e.g. `getchar()`) unless they are rewritten as true C functions.
- Arguments of integer, character, string and pointer type can be passed to the C functions. If the argument is of string type, the character pointer which points to the first character of the string is passed. Arguments cannot contain values of list type. All arguments are cast to integers before they are passed to the C functions. If data types are integer-incompatible, i.e. require a different amount of storage, then they may cause problems. Since the compatibility of data types varies from compiler to compiler, the user should check it out themselves.
- Only the first 10 arguments can be passed (floating point number counts two).

Despite these limitations, the EDEN/C interface can handle a large number of existing C functions. Sometimes, the user can write a simple C function to ‘‘bridge’’ with a C function excluded by the limitations. For instance, the user can write a C function, e.g. `BridgeFunc`, to interface with the actual C function, e.g. `ActualFunc`, which requires a structure as its argument:

```
struct point {
        int x;
        int y;
        };
```

```
ActualFunc(struct point p)
{ ... }

BridgeFunc(intx, int y) {
        struct point p;
        p.x = x;
        p.y = y;
        return ActualFunc(p);
        }
```

Then the bridge function is bound instead of the actual function.

To bind a C function

1. The user must edit the header file, `$PUBLIC/tkeden/Misc/customlib.h`, to add the C function declarations which must be enclosed in the two lines:

   ```
   #if INCLUDE == 'H'
   #endif
   ```

   For example,

   ```
   #if INCLUDE == 'H'
           extern system();
           extern fprintf();
           extern char * fgets();
           extern double sin();
           extern double cos();
   #endif
   ```

2. In the same file, `customlib.h`, add the binding information entry of the form:

   ```
   { "EDEN_name", (int(*)())C_function_name },
   ```

   where *EDEN_name* is a valid EDEN identifier and *C_function_name* is the C function name declared in (1). The two names can be different.
   ‘‘`(int(*)())`’’ is used to cast the C function to an integer function. Don’t forget the commas at the end.
   The binding information must enclosed in

   ```
   #if INCLUDE == 'T'
   #endif
   ```

   For example:

   ```
   #if INCLUDE == 'T'
           { "system", system },
           { "fprintf", fprintf },
           { "fgets", (int(*)())fgets },
           /* ... etc ... */
   #endif
   ```

   If the function returns a double precision floating point, the binding information shall be enclosed in

```
#if INCLUDE == 'R'
#endif
```

For example:

```
#if INCLUDE == 'R'
        { "sin", sin },
        { "cos", cos },
        /* ... etc ... */
#endif
```

3. Recompile EDEN following instructions in the file `$PUBLIC/tkeden/INSTALL`. Don't forget to add the links in `Imakefile` (or `Makefile` as appropriate) if the new functions added require links to object libraries other than what have already included.

A set of macros [p 57] is defined, at the beginning of `customlib.h`, to reduce the complexity of binding C functions to EDEN names. These macros are `SameFunc`, `Function`, `SameReal`, `RealFunc`, and `SpecialF`:

`SameFunc(`*Name*`)`
: Bind a C function named as Name to the EDEN name identical to Name. The value returned by the C function must be of integer type.

`Function(`*Ename,Cname*`)`
: Bind a C function named as *Cname* to the EDEN name *Ename*. The value returned by the C function must be of integer type.

`SpecialF(`*Ename,Type,Cname*`)`
: Bind a C function named as *Cname* to the EDEN name *Ename*. *Type* specifies the type of value returned by the C function.

`RealFunc(`*Ename,Cname*`)`
: Bind a C function named as *Cname* to the EDEN name *Ename*. The C function must be a double precision floating point function.

`SameReal(`*Name*`)`
: Bind a C function named as *Name* to the EDEN name identical to *Name*. The C function must be a double precision floating point function.

These macros reduces the two-part information into a single macro. To specify a binding, just append the macros at the end of `customlib.h`. For example:

```
/* customlib.h */
...
SameFunc(system)
SameFunc(fprintf)
SpecialF(fopen,FILE *,fopen)
Function(getchar,my_getchar)
RealFunc(sin,sin)
SameReal(cos)
...
```

Warning

No white space is allowed in the macros (except in `Type`), and no other punctuations (including commas and semi-colons, but excluding comments) can follow these macros.

Macros for Binding C Functions to EDEN

Note that these macros only works with ANSI C.

```
#if INCLUDE == 'H'
#define SameFunc(Name)              extern Name();
#define Function(Ename,Cname)       extern Cname();
#define SpecialF(Ename,Type,Cname) extern Type Cname();
#define RealFunc(Ename,Cname)       extern double Cname();
#define SameReal(Name)              extern double Name();
#endif

#if INCLUDE == 'T'
#define SameFunc(Name)             {#Name,Name},
#define Function(Ename,Cname)      {#Ename,Cname},
#define SpecialF(Ename,Type,Cname){#Ename,(int(*)())Cname},
#define RealFunc(Ename,Cname)
#define SameReal(Name)
#endif

#if INCLUDE == 'R'
#define SameFunc(Name)
#define Function(Ename,Cname)
#define SpecialF(Ename,Type,Cname)
#define RealFunc(Ename,Cname)      {#Ename,Cname},
#define SameReal(Name)             {#Name,Name},
#endif
```

Programming Notes

- Memory Allocation [p 57]
- Stack, Heap and Frame Stack [p 58]
- Autocalc [p 61]

Memory Allocation

Memory allocation is automatically handled by the EDEN interpreter. The string and list data types are assigned by copy (i.e. each string has its own copy instead of sharing the same string through pointers as in C). For example,

```
S1 = "1234567890";
S2 = S1;
```

`S2` is now the string `"1234567890"`. If we now do

```
S2[1] = "X";
```

then `S2` becomes `"X234567890"`, but `S1` is still `"1234567890"`.

Also the EDEN interpreter allocates the exact amount of memory for the string. Reference beyond the string's or list's memory block (probably through C function calls) may have unpredictable results.

If we pass a string as an argument to an EDEN function, the string is copied, i.e. passed *by* value. But if we pass a string to a C function, only the address of the memory is passed, i.e. passed *by reference*. So, the user should be careful in calling C functions.

For instance, if we want to use the C function `gets` to read in a string from *stdin*, we must first allocate some memory to a string variable before we call `gets`:

```
S = substr("", 1, 255);
gets(S);
```

Here we call the function `substr` to assign 256 spaces (strings are terminated by an invisible NUL character) to `S`. Then we call gets to read a string into the memory allocated to `S`. By doing so, we not only allocate memory for `gets` to used, but also set the data type of `S` to the string type.

Stack, Heap and Frame Stack

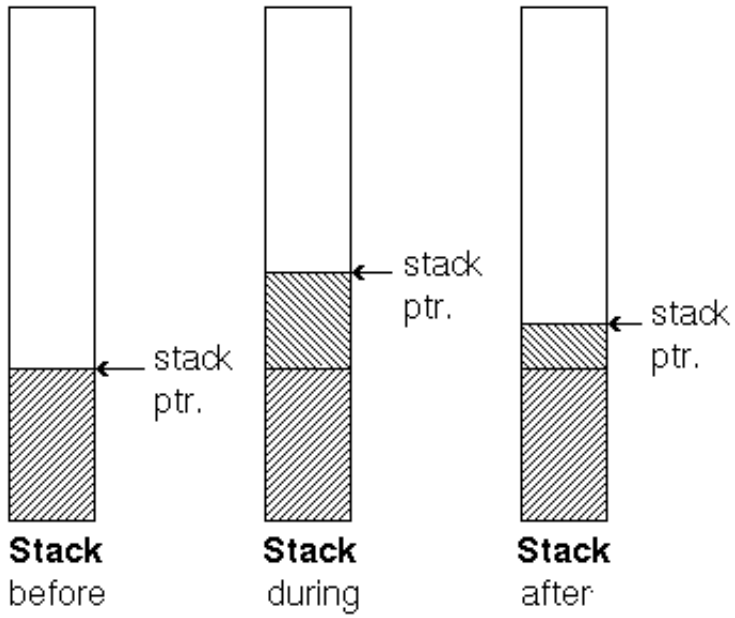
The pseduo machine of the EDEN interpreter is a simple stack machine.

Stack

A *stack* holds the values used and computed during evaluation of an EDEN program.

When the machine calls a function, the parameters are pushed on the top of the stack. Actually, the parameters are merged to form a single list datum. If the function requires any local variable, the machine allocates space for their values. When the function returns, the allocated local variables and the arguments stored on the stack are poped; then the value is returned by pushing it onto the stack. See the figure below which shows the stack

- *before* a function is called,
- *during* a function call, the arguments having been pushed onto the stack, and
- *after* the function has returned, the arguments and local variables having been popped, and the answer pushed onto the stack.



Since the stack is a fixed size memory block, a deep function call may cause the ‘‘stack overflow’’ error.

Heap

To reduce the frequency of copying the contents of strings and lists, we add another data structure, called the *heap*, to hold the contents of temporary strings or lists during the string/list operations. The overhead for allocating/deallocating memory on the heap is less than that of `malloc()/free()`.

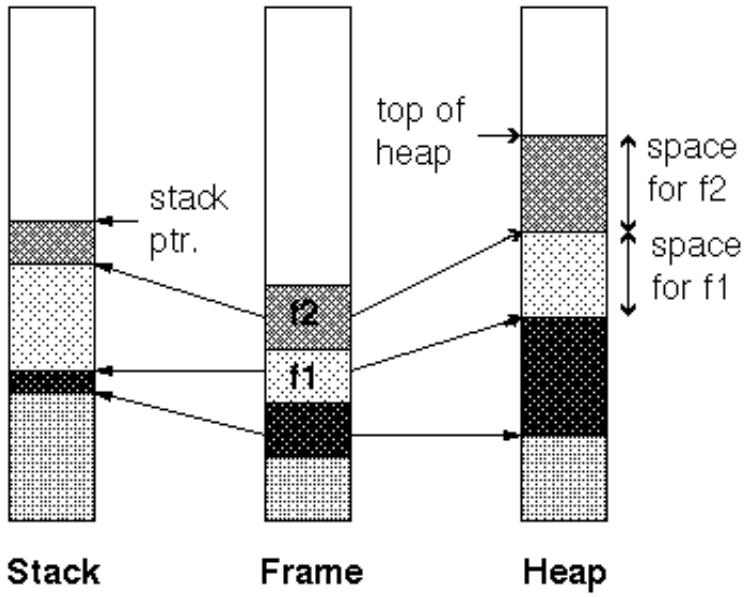
The heap is a large memory block. Memory is allocated within this block. A pointer keeps track the top of heap similar to the stack pointer of the stack. The pseduo machine instruction, *freeheap*, releases all the memory allocated by previous computation. The heap is not affected by the return of the function call. Freeheap instructions are automatically inserted by the interpreter at the points that it thinks safe to free memory. However, if a computation involves long strings, lists or too deep function called, the machine may not have the chance to free the memory and thus causes the ‘‘heap overflow’’ error.

The user may have to modify their program to minimise the load of stack and heap to avoid memory overflow errors. For instance, an iterative function has less demand on stack and heap than its recursive equivalent.

Frame Stack

In returning from a function call, the machine has to stored the previous machine status, e.g. stack pointer, (pseduo machine) program counter, number of local variables (since local variables were put on the stack), etc. A frame stack holds these information.

When a function is called, the current machine status, a *frame*, is pushed on the *frame stack*. For example, if `f1()` calls `f2()`, the status of stack, heap and frame stack is shown below.



The frame stack keeps track of the stack and heap pointers. A *freeheap* instruction (generated by EDEN) causes the top-of-heap to return to the bottom pointed to by the top-most frame (e.g. `f2`). On the other hand, the stack pointer is lowered only if the function returns.

A frame is popped when a function returns and the machine status resumes to the previous state. Since the frame stack is implemented as a small stack, about 64 entries, the number of level of nested function call is 64 approx. This maximum limit is enough for general use, but if a function nested too deep, the machine will generate a ‘‘nested too deep’’ error.

Avoiding Memory Overflow

Since the EDEN interpreter passes list "by value", i.e. copies the entire list on heap, it is very eay to overflow the heap if you pass long lists as parameters of a recursive function. To get round this problem, you may consider the use of iterative loops instead of recursive functions, or you may pass the "pointer" of that list to the function, and let the function to copy that list to its local variable. For example, instead of:

```
func foo /* (list) */
{
        ...
        foo($1);
        ...
}
```

do:

```
func foo /* (list *) */
{
        auto list;
        list = *$1;     /* copy the list to local variable */
        ...
        list = ...;     /* do the calculation here */
        foo(&list); /* pass the pointer of that list */
        ...
}
```

Also not that the interpreter will try to free the heap space at the end of each statement. Therefore, it would be helpful space-wise to break a long expression into smaller expressions and use temperary variables to hold intermediate computation results. For instance:

```
a = expression;
foo(a);
```

uses less heap space than

```
foo(expression);
```

The machine uses `malloc()` to allocate space for strings and lists rather than claiming space from the heap. So the use of strings and lists would not normally causes memory overflow.

Autocalc

The `autocalc` variable controls the evaluation scheme of the EDEN interpreter. When it is 1 (default) or non-zero, the interpreter will try to evaluate all definitions (if they are evaluable, i.e. all their source variables are defined); otherwise (autocalc is zero) the interpreter puts the suspended definitions in a queue.

Before the interpreter evaluates a definition, it tests whether all its source variables are defined, i.e. their UpToDate flags are true. However, the interpreter is not intelligent enough to identify variables referenced by a definition indirectly. For instance,

```
func F { para N; return N + Z; }
Y is F(1);
```

Function `F` references the (global) variable `Z`, but the interpreter does not know about that. When `Y` is defined, the interpreter thinks that `Y` depends on `F` only, and proceeds to evaluate `Y` and thus evaluate `F`. However, `F` finds that `Z` is undefined. In this example, `F` returns the sum of `Z` and the first parameter which produces `@` when `Z` is `@` However, some operators and most of the statements produce an error when the expected data types are not met.

Of course, if we defined `Z` before the introduction of `Y`, it may not cause an error. However, if we introduce `Z` after that of `Y`, we are in the risk. Hence, the result of computation seems to be sensitive to the order of definitions. Despite of this ordering problem, redefining `Z` would not cause `Y` to be updated. This may not be the intended behaviour sometimes. There are two ways to get around it:

- Set `autocalc` to `0` before the introduction of all definitions. After all definitions are defined, set `autocalc` to `1` again. By doing so, the evaluation of definitions is delayed. For instance,

```
autocalc = 0;
/* ... other statements ... */
func F { para N; return N + Z; }
Y is F(1);
Z is 10;
/* ... other statements ... */
autocalc = 1;
```

However, this method does not solve the re-evaluation problem.

- Explicitly give the dependency of `Z` to `F`. For example,

```
func F { para N; return N + Z; }
proc touch_F : Z { touch(&F); }
Y is F(1);
Z is 10;
```

Because now change of `Z` will induce a change to `F`, `Y` is indirectly dependent upon `Z`, and `Y` can't be evaluated until `Z` is defined. Also Y will be re-evaluated when `Z` is re-defined. This point is not true for the former solution. In fact, the interpreter should generate such dependency for function specifications.
Note that such attempts as:

```
func F: Z { para N; return N + Z; }
```

or

```
func F { para N; return N + Z; }
Z ~> [F]
```

to impose dependency would not work because in both cases a change of `Z` will call the function `F` (as if it is an action). This is an error because a parameter is needed to calculate `F`.