6

EdenLisp Implementation

The design of a new language is best tested by its implementation. The author's language is no exception. In this chapter the detail design of EdenLisp is described. EdenLisp is a complete language with its own syntax and semantics to which user input has to conform. The compiler design, consisting of lexical analyser, parser and definitive evaluator is delineated, including the structure and function of the underlying symbol table. That symbol table is crucial to the ability of EdenLisp to permit operation in various environments that are suited to the different design requirements of the user. Related design objects can be ascribed tree structured related environments permitting global and local variables; the input environment is within AutoCAD, either directly or from a file, and the CAD environment is via AutoCAD's own CAD commands

6.1 Introduction to AutoLisp

AutoLisp®, the programming interface to AutoCAD, is used to implement EdenLisp. The functional programming style of Lisp lends itself well to the prototyping of definitive notations. AutoLisp is interpretative: input Lisp instructions are immediately evaluated and the result actioned. Its syntax and structure are as Lisp, but it includes access to all commands that can be issued at the AutoCAD Command prompt. Files of AutoLisp code are loaded by typing at the Command prompt

```
(load "filename")
```

where *"filename.lsp"* is an ASCII text file of AutoLisp statements. The suffix *".lsp"* is necessary when writing the file, but is understood in the *load* statement. The parentheses are a necessary part of Lisp and rather awkward since nested statements may require large numbers of brackets (prompting some to say that LISP is an acronym, not for LISt Processing, but Lots of Irritating Silly Parentheses!)

The AutoLisp environment makes editing and debugging AutoLisp programs difficult as AutoCAD has to be running to interpret the code. The solution adopted was to use a TSR (terminate, stay resident) editor such as SideKick® or PCTools®. The latter has the advantage of allowing multiple files to be edited and both can be called whilst AutoCAD is running in text mode. With recent developments in Windows environments these problems will disappear.

A brief introduction to AutoLisp is useful as background to EdenLisp. Since AutoLisp has the same syntax as Lisp the reader is referred to texts on Lisp for more details, [*e.g.* Winston & Horn, 1989, Jones, Maynard & Stewart, 1990]. Lisp is one of the oldest computer programming languages and in some of its logic it actually antedates the modern digital computer. Its power arises from its ability to attach information to symbols. It only has one data-type, namely: *symbolic expression*, divided into *atoms* (numbers and symbols) and *lists* (of atoms or lists).

The following examples show how the Lisp interpreter evaluates symbolic expressions and prints the result. `$` is the Lisp prompt.

```
$ 4                 ; 4 is a numeric atom that evaluates to itself.
4
$ t                 ; t and nil are special atoms that evaluate to
T                   ;   themselves. nil is false; everything else is
true.
$ (+ 4 (* 7 8))     ; Inner arguments 7 and 8 are evaluated, the product
60                  ;   found and the result passed to the + function.
```

Naming is carried out by operators. `Setq` is an assignment operator that associates a variable with a value and returns that value. Because it makes a permanent change in a variable `setq` is one of a class of operators called *mutators* that are not part of a pure functional language. To associate the value `6` with the name `a` we write:

```
$ (setq a 6)
6
```

The addition of a single quote before a symbol forces the interpreter to take the name and not its value. The assignment.

```
$ (setq b 'a)
a
```

is interpreted as "set the value of *b* to the name *a*", hence it returns *a* and not *6*. `Setq` actually means *set quote,* hence "set the name `b` to the name *a*". That property is useful in being able to manipulate names, create macros and even create lisp within lisp. EdenLisp uses that property to advantage.

The task of keeping track of the assignations is carried out in the *environment.* Different assignments may be made if environments can be make local. AutoLisp allows local variables within functions.

Lisp has three basic list processing operators.

```
$ (cons 4 nil)    ; cons converts atoms to lists
(4)               ;   or adds an atom to a list
$ (car '(a b c))  ; car returns the head of any list.
a                 ; The head may be an atom as here, or a list
$ (cdr '(a b c))  ; cdr returns the tail, the remainder of a list after
(b c)             ;   removing the head. Result is a list (may be empty)
```

From these operators further lisp functions may be built up by the operator *`defun`*. Functions are made available for use in any program by simply calling it by its name with its arguments, *e.g.* the following defines the function *`member`*. Line numbers are added for reference; they are not part of Lisp.

```
(defun member (item lista)                                   1.
(cond                                                        2.
 ((null lista) nil)                                          3.
 ((equal item (car lista)) lista)                            4.
 (T (member item (cdr lista)))                               5.
))                                                           6.
```

The function tests if *`item`* is in *`lista`*. So

```
$ (member 'g '(b c d e f g h i))
(g h i)
```

takes as input the symbol *`g`* and tests if it is in the list *`(b c d e f g h i)`*. If *`g`* is found then it and the remainder of the list is returned. Since the answer is not *`nil`* it is also interpreted as *`TRUE`*.

The function uses *recursion*. The *`cond`* statement is similar to the *`case`* statement in procedural languages. There are two conditional cases and an "else" case.

- if *`lista`* is empty then exit the function returning *`nil`*
- if *`item`* is the same as the head of *`lista`* then exit returning *`lista`*
- if none of the other cases apply the "else" statement denoted by `T` causes the function to be invoked again but with *`lista`* without its head.

When `member` is first called the two cases in lines 3 and 4 fail so the ***T*** clause causes the function to be called again as `(member 'g '(c d e f g h i)),` i.*e.* without the first symbol `b`. That call also fails, so `(member 'g '(d e f g h i))` is invoked... and so on until `(member 'g '(g h i))` is invoked when line 4 will succeed, returning `lista = (g h i)`. At that point the function has been called 6 levels deep so has to climb out 6 times before delivering the result. Results of intermediate levels are stored on the stack.

The advantage of recursion is the brevity of the code required, making prototyping very rapid, if rather dense to understand at first. The disadvantage is the stack growth. On AutoLisp the stack is 146 levels deep, beyond which the interpreter overloads. The code must then be written to make it "tail-recursive" (i.e. to make the recursion carry its own previous result so that the depth of recursion is reduced to the levels an iterative method would use).
AutoLisp has built-in functions that are common to most versions of Lisp to save defining them (*e.g.* `member` is a built-in function). A list of AutoLisp functions accessible to users of EdenLisp is included in Appendix 1.

The simplicity of an untyped language like Lisp is the ability to create abstractions on data that can capture primitive ideas in functions that in turn become building blocks for higher order structures. As Abelson and Sussman put it in their seminal book on Structure and Interpretation of Computer Programs:

> "As we confront increasingly complex problems, we find that Lisp, or indeed any fixed programming language, is not sufficient for our needs. We must constantly turn to new languages in order to express ideas more effectively. Establishing new languages is a powerful strategy for controlling complexity in Engineering design." *[Abelson & Sussman, 1985]*

6.2 The EdenLisp Compiler

EdenLisp consists of AutoLisp functions grouped into files as follows.

EDENLEX.LSP	Lexical Analyser
EDENPARS.LSP	Parser
EDENENG.LSP	Main Engine: input functions, interpreter and general control
EDENTYPE.LSP	Type checker
EDENENV.LSP	Environment control: Symbol table, object/frame management
EDENEVAL.LSP	Recursive Tree Evaluation of definitions

EDENUTIL.LSP	Utilities and error trapping

Geometrical and topological functions, including draw functions and operators are in files *EDENGEOM.LSP* and *EDENTOP.LSP*. Actions are in *EDENACTN.LSP*.

Fig 6.1 shows the organisation of the EdenLisp "compiler" in a block diagram. The lexical analyser scans the stream of EdenLisp source code, and separates it into tokens. The tokens are lisp *dotted-pairs* such as *(a . b)*. The latter have the property of being stored more compactly in memory than lists such as *(a b)*; furthermore, *(cdr '(a . b))* returns the atom *b* rather than list *(b)*. The head of the dotted pair identifies the nature of the token and its tail the actual keyword such as *IF, THEN,* or predicate symbol. The token stream is the input to the next module, the parser.

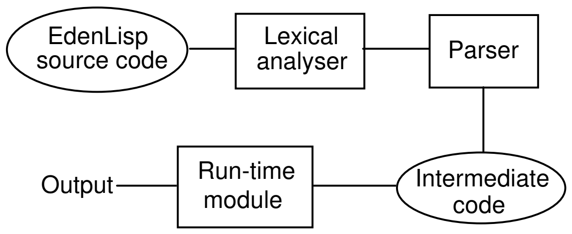

Fig.6.1 Organisation of the "compiler" stage of EdenLisp

The parser groups tokens together in accordance with their syntax into a parse tree. For instance an EdenLisp definition is a syntactic structure consisting of tokens arranged in the form: <variable identifier, assignment operator, formula>. A formula is any collection of symbols that are conventionally used to describe a relation. Input is expected using infix notation such as (a + b) rather than the prefix notation (+ a b) used in Lisp. In the parser the parse tree is traversed and the structure altered into the Lisp format. The parser also identifies dependent variables in the formula and the type of EdenLisp statement being input. The output lists the EdenLisp statement type, independent and dependent variables and the predicate in AutoLisp code. The lexical analyser and parser are discussed in the next two sections.

6.21 The Lexical Analyser

The lexical analyser "EDENLEX.LSP" accepts a string as input and reads it from left to right. Each character is read off the head of the string and the string replaced by

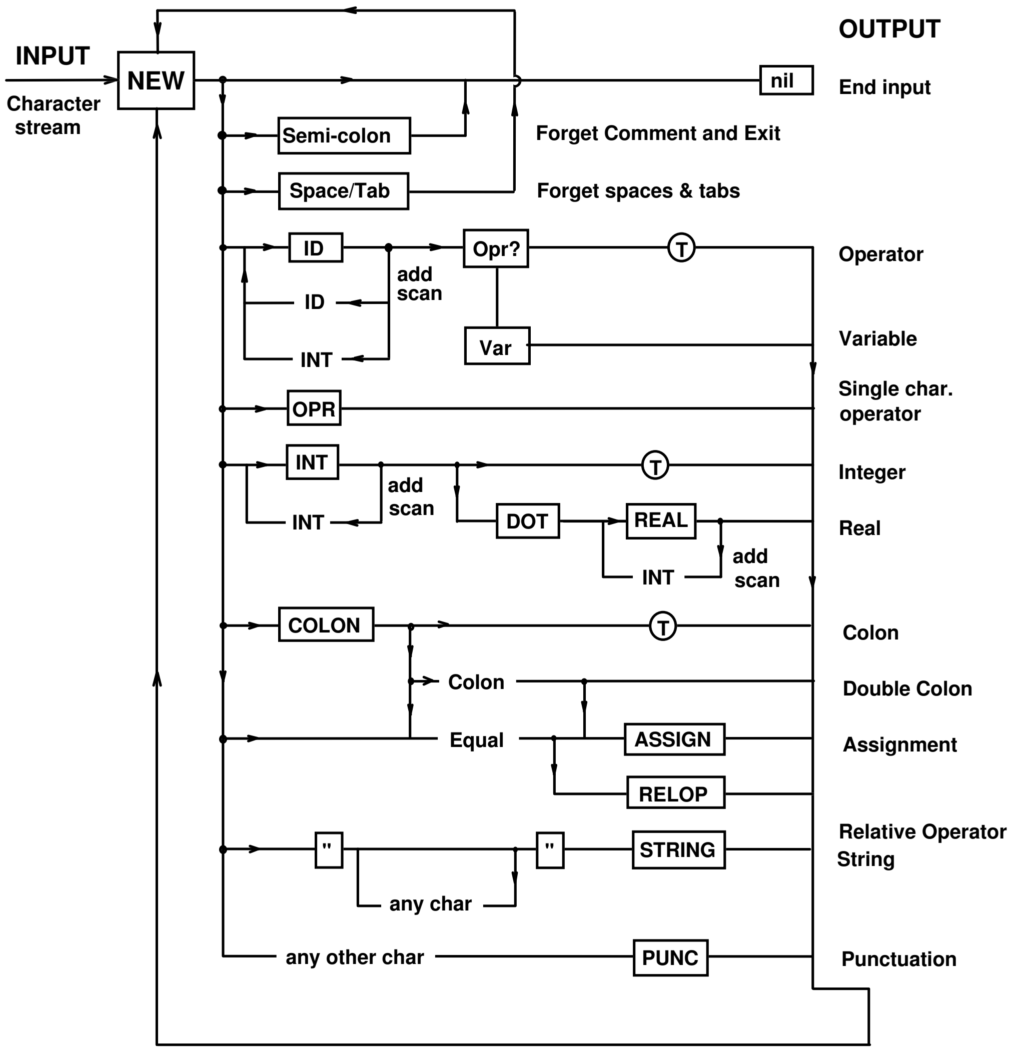

Fig 6.2 Finite State Machine for Lexical Analysis of String Input Stream

its tail. The process uses a `while` statement rather than recursion in order that a finite state machine (FSM) can be used to implement the lex stage. The structure of he FSM is shown in *fig. 6.2.*

Each character is identified by the lexical scanner into one of the character groups shown in *table 6.1.* Character groups are used as states in the FSM. For example, if the input string is *"phi = sin(a)"*, then the following state transitions occur within the FSM (see *fig. 6.3*). Starting in state NEW, character *p* causes the FSM to move to the state ID, and *p* is pushed onto a stack. The next character, depending on its character group, moves the FSM to a new state or keeps it in the current state. In our example, *h* is received and added to the stack, the FSM

RULE	**Token Name**
nothing	*nil*
" ", "\t"	'SPC
string char in [! ( ) , ? { } &]	'PUNC
"\\", "[", "]"	'PUNC
string char in [+ - * / ^]	'DEFOP
string char in [< > | ~]	'RELOP
"`"	'STR
string char in [a..z, A..Z, _ $ % @]	'ID
":"	'COLON
"="	'EQUAL
";"	'COMENT
string char in [0..9]	'INT
"."	'DOT
"'"	'SQUOTE
"\042"	'STR
any other character	'PUNC

***Table 6.1**. Characters are recognised into tokens named in the second column*

remaining in state ID. The same is done with the *i* character. Receipt of the *spc* (space) character ends the ID state: the current stack is emitted and the FSM returns to NEW. The *spc* character, as all white space, is ignored and the machine simply scans for the next character. This allows the input to be "pretty printed" by the user. Similar transitions are shown in *Fig 6.3* for the remainder of the definition. A definitive statement is ended by `return` (ASCII chr 13).

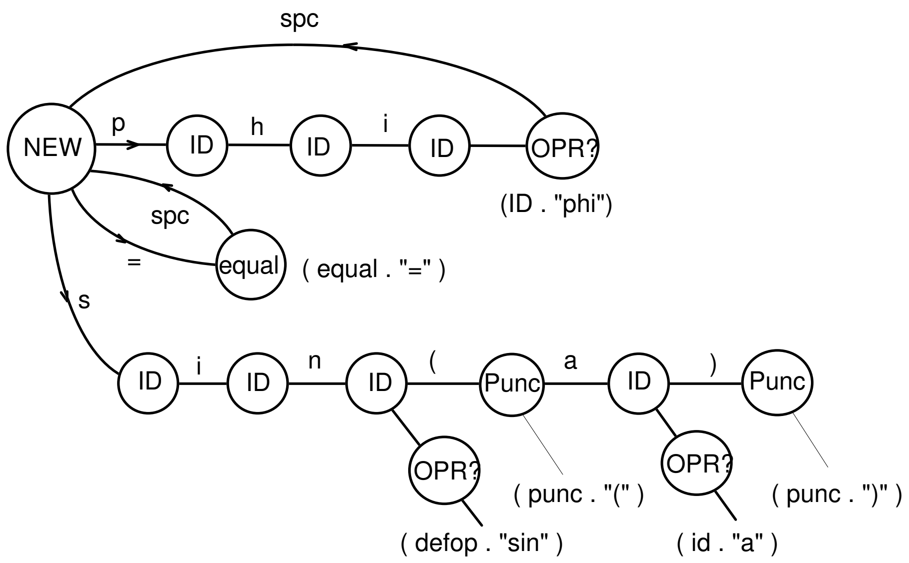

Fig. 6.3 State transitions for input stream `phi=sin(a)`

6.22 The Parser

The parser `"EDENPARS.LSP"` accepts a stream of tokens (dotted pairs) generated by the lexical analyser. The stream is of finite length and is first tested to be one of the following valid EdenLisp inputs.

1. a type definition of the form `type : id (id..)`
2. a definitive statement
   - with non-graphical output, of form `id = const|id|formula`
   - with graphical output, of form

     `id = draw_function|draw_function_formula`
3. Special definitive form: specified dependent variables are to be held at the values that are current on definition. E.g.

   `a : b c : b+c/d`

   where `b=8.0, c=7.5,` implies that `a = 8.0 + 7.5/d` is the definition to be evaluated thereafter, whatever subsequently happens to `b` and `c`.

Forms 2 and 3 are separated into left and right sides of the assignment symbol. Provided it is syntactically correct, the right side is converted into a parse tree using grammar rules; otherwise an appropriate error message is generated. The parser uses a *recursive descent parser* (c.f. Tanimoto, 1990) to change the input stream infix order into the corresponding prefix notation. The twelve non-terminals: *program, statement, expression, relational_expr, term, factor, opr1, opr2, int, real, id,* and *function* use production rules that are shown in Appendix 1.

An input submitted to these rules is treated recursively. The output, for example from `a+b+sin(c)+d` returns the binary form `(+ a (+ b (+ (sin c) d)))`, rather than the more compact form `(+ a b (sin c) d)`. Although both are Lisp compatible the binary is easier to generate. The recursive form is not the most efficient parsing method but has the advantage of compact coding, hence easier to debug.

The parser also allows input of standard AutoLisp. If an opening `"("` is detected in the definitive formula then the formula is handled directly by the lisp interpreter in the same way as Basic allows Peek and Poke. Errors are then handled by AutoLisp. Another AutoLisp form allowed is an opening `"!"` such as `"!a"`, which AutoLisp interprets as `(print a)`, useful for debugging and interrogation of the variables.
Finally the parser makes one or two other translations in order to format the grammar more conveniently. This is best illustrated by an example.

```
$ (parse (lex "a = b+(c - d/e)"))
(DEFN "a" (b c d e) ("(+ b (- c (/ d e)))") )
```

At the Lisp prompt the command *(parse (lex ..))* causes the parsing of the lexical tokens obtained from the definition, returning the form shown in the next line. The parsed form has four or five components

1. the kind of EdenLisp statement (type definition, definitive statement, restricted definition)
2. the actual variable or type depending on the kind of statement as a string
3. a list of dependent variables
4. the formula, definition or list of variables to be typecast, in string form
5. if restricted a list of restricted variables

6.3 Definitive Interpreter and Symbol Table

6.31 Identification of Statements

The intermediate code from the parser input is passed to the EdenLisp "engine", *EDENENG.LSP.* The engine identifies the type of statement using information at the head of the code and then deals with each as appropriate. As observed above, a statement must be a Lisp statement, a type declaration, a definition, or a restricted definition. If it is not an input error is returned and the interpreter ignores the whole statement.

1. **LISP** statement

Lisp statements must begin with an opening parenthesis **"("** so that is always taken as the signal that the statement following is in AutoLisp. The statement is passed through to the AutoLisp interpreter without being actioned or stored by EdenLisp. In that respect such statements are akin to the commands in PADL-2. Care is required with the use of such statements. Rather like using assembler from within a conventional language unexpected things may happen because EdenLisp does not know about any changes made by Lisp statements. Thus subsequent EdenLisp instructions will be interpreted in the new environment.

2. **DECL** = Type declaration, input is of the form

< DECL, name of type, string containing a list of variables to be declared >

The parser has already checked the name of the type is legal, so the string is now converted into a list of identifiers. The existence of an identifier is checked by scanning the symbol table. If a record is found in the symbol table the interpreter refuses that declaration and returns a warning to the user. If the declaration is valid the engine passes it to the environment control for creating a new record to hold information about the variable. (The environment and symbol table are dealt with below.)

3. **`ID`** is legal
Regardless of form of the statement, if it is not one of the first two kinds then it is some kind of definition. That being so it must be of the form **`ID`** `= statement`. The interpreter can therefore check at this stage to see if **`ID`** had been declared. If it has not, then an error is returned and the interpreter ignores that statement.

4. **`DEFN.`** Definition or redefinition of form **`ID`** `= statement`.
The interpreter first decides what kind of statement is being defined and actions each slightly differently.

a) If **`ID`** is a constant or a string that is accepted and sent on to the evaluator

b) **`ID`** is a formula or a draw function. Dependent variables are extracted from the parsed input string and checked to see if they have been declared. If they are declared then the interpretation proceeds; otherwise an error is returned and the interpreter ignores that statement.

5. **`RDEFN.`** Definition or redefinition of the form **`ID`** `: Dvars : statement`
This indicates a restriction is placed on certain of the dependent variables (`Dvars`) in the definition. The purpose is that those `Dvars` between the colons are to be evaluated immediately and the current values of those variables entered in the definition as constants. For example the `RDEFN` defined as follows

```
a : b c : b - c * d
```

would be evaluated by setting `b` and `c` to their current values and then passing the resulting formula on to the evaluator. If for example `b=8` and `c=b-4` then `a` would be defined as

```
a = 8 - 4 * d
```

No subsequent change in `b` or `c` would affect the definition of a since neither `b` nor `c` appears in the formula.

If the input takes the form of a definition then the interpreter has a number of functions to carry out

1. The ID may occur on both sides of the definition. This is a circular definition and would cause an infinite loop. Reject.
2. The definition is a constant. Test the type of the constant against that of the ID. If OK then proceed at step 5.
3. The definition is a formula. Check dependent variables and those variables that depend upon ID and list them, checking for hidden circular definitions. Go to 5.
4. The definition is to replace an existing one. Store the previous symbol entry in case an error is generated during evaluation. Go to 5.

5. Submit the new formula to the evaluator

Circular definitions such as `a=a+1`, unlike the equivalent procedural statement `a:=a+1` that merely adds one to the variable, would cause the evaluator to try to update variable `a` over and over, forever. Circular definitions may be caused unwittingly by a series of related definitions where a variable ends up after several definitions being defined in terms of itself. Thus the interpreter has to test all dependent variables for circularity before allowing evaluation to proceed.

6.32 Type Checking

Type is tested by functions in `EDENTYPE.LSP`. This was a formidable task. EdenLisp, unlike Lisp itself, is strongly typed. Each variable must have a declared type and operators may be defined with meanings appropriate to particular types. This has numerous advantages. For example one could unambiguously define the following on the operator +

`int+int=int | int+real=real | real+real=real`

`point+point=vector_sum` or `point+point=line`

`line+line=polyline`

The method adopted was to structure type checking by means of "manifest types" [Abelson & Sussman, 1985, 2.3.2] whereby a data object has a type that can be recognised and tested. That means that all variables must have their type stored alongside the identifier. The symbol table record looks after that so the type checker can replace the variables in a formula by their pre-declared types and then evaluate the type that results from rules such as those quoted above for the "+" operator. The type evaluation can be as complex as the evaluation of the formulae themselves, since one has effectively the same number of variables and operators as the other. That makes the task of evaluation up to about twice as difficult in the worst case although certain simplifying operations such as coercion may be possible. Coercion is included in EdenLisp: `INT` may be coerced to `REAL` if appropriate (*e.g.* `3+6.9` gets coerced to `3.0+6.9`). Lists of integers are automatically coerced to `LREAL`.

In the current implementation the only binary operators are simple arithmetic on numbers. Most operations are carried out using functions on the given types. Lists of these functions and the types of variables passed as arguments are listed in Appendix 1 together with the types returned by those functions. AutoLisp functions that are compatible the EdenLisp types are included. Those correspond to common numeric operations (such as square, root, log and trigonometric functions) and certain list operations on which EdenLisp puts a greater structure than just *list*. Realisation functions currently return LLreal with display done by side effect.

List properties are used to advantage in checking formulae that have many operations of the same type. A formula such as

```
    thetam = log(4*K*R+Omax) / cos(Kt*E-th)
```

has all real operations and the result is real. Thus all that is needed is to find the types of all variables and check they are all the same. The same technique is also used where there are different types in the input but some are repeated.

The main difficulty encountered in type checking was dealing with one of Lisp's most powerful devices the quote function. Quoted atoms or quoted lists are not evaluated under any operation so `('a+'b)` has no meaning, whilst the function `(openWin 'Swin)` is interpreted as "open a window called `Swin`". In the latter case `Swin` does not have a value unless one is assigned separately. Since variables in a quoted list may well be declared and have a type it is vital that the quoted list is excluded from the type check and replaced by `QATOM` or `QLIST`.

6.33 Symbol Records

EdenLisp symbols are held in a *symbol table* that has the AutoLisp variable name **sym**. Symbols are variables with types defined by the user through the type definition statement. On receipt of a valid declaration statement a *constructor* function creates an empty symbol entry that is a **record** containing first the name or identifier of a variable and then a series of at least one list of attributes. Each set of attributes appertains to the properties of a variable of that name in a particular window and lists of attributes are stacked so that the current attributes are those in the current environment (see below for environment). Attributes are **fields** in the record as follows.

`IDval:`	The current value of the `ID`. If undefined then symbol `@` is used.
`IDtype:`	The declared type
`Def:`	The definition itself
`DefType:`	The definition type `(const, formula, drawfunction,` etc.)
`DepVar:`	Name(s) of dependent variable(s) in the definition
`DepOnIt:`	Name(s) of variable(s) that depend on this `IDval`
`handle:`	This contains a value that is generated by an AutoCAD command if `DefType` is a Draw Function. Its value is the address of the entity information in the AutoCAD database.

For the input `"int : a"` the constructor will create the record

```
(a (@ int nil nil nil nil nil))
```

This record is appended to the symbol table. Values of individual fields of a record of a single variable can be inspected by means of *selector* functions. (All functions of this kind commence with an underscore character.) All these:

```
_idVal, _idtyp, _Def, _Dtype, _Dvar, _DOnIt, _Handl
```

take the `id` name as argument. One has to be in the correct environment to access that information as the current window is assumed.

To change the values of any field in the record we use a *mutator* function, so-called to emphasise that an assignment (a permanent change) is being made by means of that function. These functions all have an exclamation character to emphasise they are mutators. The function calls need the `id` name, the new value and the name of the window (`win`) that is the home of the variable.

```
IdVal! (id idVal win)
Idtyp! (id idtyp win)
Def!   (id Def win)
Dtype! (id DType win)
Dvar!  (id DVar win)
DonIt! (id Donit win)
handl! (id handle win)
```

6.34 Evaluation

Evaluation of definitions is carried out by *EDENEVAL.LSP* functions and calling the mutator functions. The first task is to check whether the definition of a variable **ID** over-writes a previous one. If it does, then the symbol table needs to be amended to cancel the previous definition. The previous form of the record is put in a temporary store in case it is required to be put back if an error is encountered during evaluation. All records that appertain to the previous definition need amending. The previous dependent variable list is used to find those records and the name `ID` is deleted from each of the "DepOnIt" fields of the dependent variables. (The significance of the "DepOnIt" field is that the new variable must be changed by any subsequent change in a dependent variable.) We now proceed with the new definition. First the records of the new dependent variables are fetched, `ID` added to their "DepOnIt" field and then the new definition is passed to the evaluator.

The evaluation proceeds downward and upwards: downward to find values of dependent variables and upward to amend variables that depend upon the value of the new definition. Moving down the dependency graph simply means finding values of dependent variables by using the Lisp interpreter on the formulae in their definition. However the upward traversal may be complex as the new value of `ID` may mean that those variables that depend on it now have values, they previously being undefined. The only way to check that is to evaluate the variables that depend upon `ID` on the whole path to the root. To do that will mean invoking downward evaluation for each variable on the upward path. Recursion is used extensively for this process.

6.4 Environment

6.41 Window Environment

The evaluation strategy described satisfies the requirement for a single set of definitions for an object. However in our algebra we wish to group different objects that have common properties. That in turn requires that we group symbols in places that are according to the object to which it appertains. In EdenLisp we follow a similar arrangement to that suggested by [Abelson & Sussman, 1985, §3.2] where these places are maintained in structures called *environments*. An environment is a sequence of *windows*. Each window is a table in which the first entry is the name of the window, the second entry is the name of the parent window (or nil if it is the root window), and further entries are *records* of variables containing their names and their *bindings* that associate variables with values & definitions. (A window may contain at most one binding for any variable; if there are more than one the variable is coerced to the last definition of that variable to be input.) The *value* of a variable with respect to an environment is the value of the variable in the first window in the environment that contains a binding for that variable. If no window in the environment specifies a binding then that variable is *unbound* in that environment. In practical terms that means if there is no record in the symbol table the variable is unbound. A variable may be declared and not be given an explicit value. The value returned in that case is the special cipher `@`.

To explain the windowing environment we use the following example of

```
(openW 'S1)
int : a b c d
a = b + c
b = 4
d = a
     (OpenW 'S2)
     c = 3
     b = c
          (OpenW 'S3)
          a = 9
          c = 10
          (closeW)
          (OpenW 'S4)
          a = 8
          b = 14
          (closeW)
     (closeW)
(closeW)
```

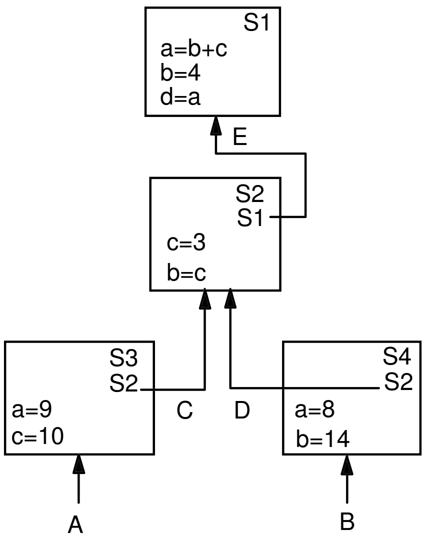

Fig.6.4 A Simple Environment Structure

EdenLisp.

Fig 6.4 shows the environment structure consisting of four windows S1 to S4. In the figure A, B, C, D and E are pointers to environments with C and D pointing to the same one. Environment paths are indicated by having the name of the window followed by its parent. Variables **c** and **a** are bound in the S3 window, while **b** and **d** are bound in windows S2 and S1. The value of **c** in environment D is 3. The value of **c** in environment B is also 3. This is because **c** is not bound in B so a binding is sought in the enclosing environment D and we find a binding in the parent window S2. On the other hand the value of **c** in environment A is 10 because the first window binds **c** to 10.

There are a number of problems that make the environment structure difficult to implement in AutoLisp. First, variables accessed outside functions are necessarily global. If we enter the lisp statement

```
  (setq a 8)
```

then the variable **a** is set to 8 and cannot be changed locally in the manner described. The way that this was addressed was to have copies of the local value stored in the record appertaining to the variable and the record stored in the particular window where it is declared. When a window is opened, the function scans all the existing variables in the window and sets them to the values in the appropriate record. A variable can then be global for the purposes of the evaluator but its current value is always stored back in the record whenever it changes.
The second problem arises in relation to variables such as **a** being changed in a child window. How do we make sure that the value of **a** is different in environments A and B? In the implementation described above the record of **a** will be changed by the redefinition of `a=9` in S3 or the redefinition `a=8` in S4. That means that separate definitions are required for S3 and S4. Worse, when in S2 there is yet another definition for **a**. So where do we store these different records and make sure they are accessible?

In the above scenario the variables are only declared in the root window S1 so all the changes are stored in S1 and none in the other windows. So what solutions are possible?

1. LOCAL: If a redefinition is made of a variable from an antecedent window in a child window, then create a new record to be kept in that child window. Such a *local variable* would have to be declared locally and would be separate from the one of that name in the antecedent window. The difficulty of that is seen in window S3. Here c=10 is defined, but in S2 b=c. Thus the evaluation of c=10 will necessarily update b to have the new value of b. In S2 b=3 whereas in A b=10. The question then arises as to where to record the value of b in S3. It would not have been explicitly declared as local in S3 since there is no direct redefinition of b in S3. If the record in S2 is updated there is no way of re-establishing the value of b when returning to S2 except by reconstructing the script appropriate to that environment and evaluating its definitions all over again. We can do that by *undoing* new definitions in the current window before moving to any antecedent and re-evaluating using the definitions found there.

2. An ENVIRONMENT based system would have the environment path as the only current one that can be changed. When a new window is opened all antecedent variables have a new attribute set added to their current record, a copy of the existing attribute set. For example the record

```
(a (3 int 3 const nil nil nil))
```

would become the record

```
(a (3 int 3 const nil nil nil) (3 int 3 const nil nil nil))
```

when a new child window is created and entered. The current record is now regarded as the one immediately following the id. This process is called 'pushing' the stack of attributes. That process means that all records in a given environment will be updated to be appropriate to that environment and any subsequent changes will be correctly entered into the most recent of the records regardless of where the record is in the environment. Thus in environment A the values of (a,b,c,d) will be (9,10,10,9) while in environment B they will be (8,14,3,3). On exit to a parent window the most recent record attributes may be 'popped' off and the records are then in the correct state for that window. If the window already exists when the use transfers to it, then the definition in that window will need to be re-evaluated to update the records on entry to that window. An alternative to that would be to carry the window title along with the attribute stack so that popping is not done but the attribute set appropriately to a given environment can be detected.

The multiple attribute system is implemented in the current EdenLisp.

The operations on environments that are required are implemented in EDENENV.LSP in terms of operations on windows and bindings. These functions are set up as far as possible as constructors, selectors and mutators. *Constructors* set up data in the correct format, selectors allow inspection of data and mutators change the data. Examples of each kind are as follows.

Constructors

```
(Mk-Win W-name)
(Mk-Rec id idval idtyp Def Dtype Dvar DonIt Handle)
```

Mk-Win constructs a new window as the child of existing environment and Mk-Rec constructs the empty record described above *e.g.* `(a (@ int nil nil nil nil nil))`.

Selector functions interrogate the symbol table by window, record or field as follows. Some names have been introduced without the underscore because they are familiar names for those kinds of operations.

```
(_win W-name)                ; get window from *SYM* table
```

`(_pwin W-name)`	; get full parent window of win
`(_ChWin W-name env)`	; get names of child windows of win from *sym* table
`(_ids Win)`	; list all entries of variables in current window
`(dir W-name)`	; list window name, parent, all variables and child windows
`(path Win)`	; list the path from win to the root
`(_rec id W-name)`	; get complete record of id from *sym*
`(_fld fldNo id W-name)`	; get a particular field from a record of id

Mutators are functions that change an environment, window, record or field

a) Environment Mutators

`(MkEnv!)`	; sets up startup global Symbol table *sym* and *win*
`(ClrEnv!)`	
`(setV! Reclst)`	; sets vars in each rec to the value in a second position
`(reval! parent)`	; re-evaluate on closing a window

b) Window Mutators

`(OpenW W-name)`	; Open a window called W-name if it does not already. Insert the new window in `*sym*`. If the window exists then set variables to local values and set the constant `*win*` to the new window.
`(CloseW )`	; Close current window and move to parent. Reset variables to local values of parent and set `*win*` to new window = parent
`(Ins-W! W-name win)`	; Include a new copy of existing window `W-name` in current `win`

c) Record Mutators

`(New-R! rec win)`	; Insert a complete new record in `win`
`(ins-R! id rec win)`	; Change complete record in `win`. That involves getting the existing window and record for changing, replacing the old record with `rec`, putting `rec` back in `win` and then replacing win in *`sym`*
`(PushR! id)`	; Make a second copy of the record (other then the id) before the current record such that rec = (id (copy rec) (previous rec)). Amend the previous record by appending the current window name (*`win`*) to indicate the position of current rec in its environment

`(PopR! id childW)`	; Remove the most recent attribute list in the `id` record. (undoes the effect of `PushR!`)
`(ins-F! newfld fldNo id env)`	; Replace the field *`fldNo`* in record *`id`* in current window

6.42 Programming environment

The user environment for EdenLisp is within AutoCAD itself. The EdenLisp interpreter is loaded from the AutoCAD command prompt by means of a Lisp command file. The user simply types

```
(load  "Edenu")
```

and it gets loaded. The parentheses are essential as that signals an AutoLisp command. The AutoCAD command `load` without parentheses has quite a different meaning.

(The above assumes the relevant files are in C:\edenlsp. If not the full path name must be written between the double quotes. AutoCAD uses the backslash to signal that a special character follows so it is necessary to type two backslashes within quotes *viz.*

```
    (load  "c:\\edenlsp\\Edenu")
```

AutoCAD also accepts a single forward slash in such circumstances, as in Unix; *i.e.*

```
    (load  "c:/edenlsp/Edenu")
```

is equally acceptable.)

Once EdenLisp is loaded the interpreter is run by the AutoLisp command

```
(eden)
```

after which the user is presented with the EdenLisp prompt `:>` Only valid EdenLisp or AutoLisp statements are allowed at the prompt. Previously written EdenLisp scripts must be text files (ASCII files in MSDOS) and may be loaded either from the Command or the EdenLisp prompts by the `DLOAD` function described earlier.

Exiting EdenLisp simply requires the ENTER key to be struck.

The method described is the formal way to load EdenLisp. AutoCAD can be customised in many ways and the pull down menus in Version 10, 11 and 12 can be used to make it easier to load not only the EdenLisp interpreter but also any EdenLisp programs.

Since EdenLisp consists only of AutoLisp functions it is possible for it to coexist with any other AutoLisp functions, although care is required to prevent functions of the same name being loaded, with perhaps very peculiar consequences!

The symbol table is set up at the first invocation of `(eden)`. If it is necessary to clear the symbol table, as one might if a new set of objects or a new program is to be entered then `(ClrEnv!)(MkEnv!)`are the functions used to clear the Symbol table and to create a new one. These functions do not clear the screen so it may be necessary to invoke AutoCAD commands to do that. `Undo back` will undo all commands made in AutoCAD up to the previous IO command. EdenLisp itself, as with all AutoLisp commands, can only be unloaded by explicitly setting all functions to nil. That fearsome task is best done simply by exiting AutoCAD and starting a new drawing.

The program structure for a script is easy if there is only a single (global) environment. Variables are declared and definitions made on those variables. If a windowed environment is desired then the `(openW WinName)` function sets up the window in the symbol table and sets the global variable `*win*` to be the current window. All subsequent declarations will be within that window and a new variable with the same name as one in a parent window is allowed and coerces the variable to have the child definition whilst in that environment. Windows opened must be closed by `(CloseW)` to cause the environment to revert to its parent window. If the `openW` command is explicitly used to open a parent, the child window is automatically closed. `(CloseW)` does not need an argument as the current one is implied.

6.43 CAD Environment

The CAD display environment is AutoCAD. Objects are defined, manipulated and displayed by means of AutoLisp functions intended to be accessed from within EdenLisp. These functions are arranged into the files `EDENTOP.LSP`, `EDENGEOM.LSP`, `EDENDISP.LSP`. The first of these, `EDENTOP.LSP`, contains graph theory functions such as `PEDGE`, `CEDGE`, `OBJECT`. These functions manipulate labels as lists of strings or quoted names. `ISOMORPH` is an interesting function that creates a new list of strings identical in arrangement to the input but distinguished by concatenating the second argument (a string) to each of the component strings. As the function name implies the output has the same shape (or graph) as the input. This provides us with the means of creating instances of the same topology whilst making each have its own distinct variable names. As discussed in chapter 5 that enables us to change the graph structure of generating strings so causing all derived variables to change their graph structure in an identical manner.

In string manipulation, the "value" of labels is irrelevant, the output of these functions is the arrangement of the labels, for example according to the way that points are to be joined together. Objects are formed as instantiations of prototype sets of labels, now treated as variables. This is a difficult stage as it is necessary to move from a manipulation of labels to manipulation of the *values* of variables under those labels. That is achieved by means of the Lisp function `READ` that takes a string and returns the first word or list as a Lisp atom or list. Alternatively we can use the Lisp function `QUOTE`. A quoted atom or list is treated in Lisp as an object that is not evaluated. Quoted lists may be quite long and may indeed be embedded Lisp programs treated as unevaluated text. It is easy to see the power of such an ability. We exploit that property in the functions `COMPLEX` and `OBJECT`. The first converts strings into labels, the second attaches the values to the newly created variables and also creates an instantiation by creating lists of reals that can represent 2D or 3D points. This input list of reals (called a *frame*) is shifted with respect to the world origin and scaled, both according to the arguments of `OBJECT`. The output, another frame, is a list of reals.

`EDENGEOM.LSP` deals with common manipulations of geometrical entities. Geometrical operations described in chapter 5 are formed by invoking functions for vector manipulation (addition, subtraction, dot and vector products, scaling, midpoint) and matrix transformations such as translation, rotation, reflection. Again these functions are strongly typed and are checked as such by the EdenLisp interpreter. Selection of elements of lists may be made by use of `PROJN`, a function that accepts a list of reals, a list of real-lists, or a frame and returns the nth member. Common geometrical shapes can also be generated, *e.g.* rectangle, n-sided regular polygon.

`EDENDISP.LSP` contains the functions that call upon AutoLisp Commands to perform drawing operations. The main functions have been described in Chapter 5. If called from AutoLisp the AutoCAD `COMMAND` function must have its arguments encapsulated as a series of strings. The `COMMAND` function is complicated and each call must be precisely in the format that a user would enter the data at the user interface. A number of problems local to AutoCAD were encountered. For example many commands require the user to point to the entity to be manipulated. Giving the correct reference to that entity often proves tricky, particularly if a number of other operations have already been done to that entity. The most foolproof method involves identifying how AutoCAD itself references the entities. That is done by means of an entity reference called the *handle*. The handle is a unique address given by AutoCAD that accesses the details of that entity in the AutoCAD database. AutoLisp provides a number of handle manipulation commands that return the handle name, delete or edit the handle. In EdenLisp we add the handle address to the `*SYM*` table for each `DRAWFN`, *i.e.* to each definition that causes an entity to be displayed. A further advantage of the handle is the AutoLisp command `ENTDEL:` that totally removes the entity from both the AutoCAD database and from the screen. `ENTDEL` is invoked within the implementation of EdenLisp to remove all trace of a previous definition when that is redefined. It is easy to see the benefit of that in EdenLisp: redefinition does not entail explicitly finding and deleting the previous entity from the screen. The redisplay is thus much faster than is common in traditional methods of animation sequencing for example. (Interestingly, AutoCAD "remembers" it has created those entities despite killing the handles. The command `UNDO` will still sequence though previous instantiations, providing a high-speed sequence through the design history. It remains to be seen how the `UNDO` command may be exploited for precisely that purpose!)

6.44 Actions

The Abstract Definitive Machine that is the conceptual framework of EdenLisp has the notion of *guarded action.* The implementation of the guard in EdenLisp is by a conditional definition. The definition is entered at the EdenLisp interface with a predicate in the conventional procedural form of the **`if`** statement,

```
Defa = if relative statement then statement else statement
```

Care is needed interpreting the meaning of this definition: the problem is the same as in the Eden interpreter. Consider the following definitions in EdenLisp

```
f = if c then x else y
g = if c then b=x else b=y
```

Here f depends on c, x and y. In the second definition, if c is true then `g` depends on x only, if c is not true `g` depends on y only. In both cases changes in x and y will affect `g` in conditions where they should not do so. If c is true then `f=x` and `g=(b=x)` in the second case.

Actions are implemented either by manipulating quoted atoms or lists and evaluating them as needed, or by using string manipulation followed by a `read`, again as a way of evaluating. Essentially an action triggers Lisp functions that generate the text of new definitions from fragments of input strings or dummy definitions. For example, a simple way to manipulate an input string is to make a function that replaces each occurrence of `"?n"` (where `n` is an integer) in the string by the element of the replacement list corresponding in position to that integer. The resulting text or dummy definition is handed to EdenLisp as an input that is effectively an alternative to the keyboard or disc file. The computer can therefore be considered to be the agent inputting the definitions. It is that idea that is developed in the multi-agent system envisaged in the ADM.

6.5 Discussion

The evaluation procedure in EdenLisp is strict forward. It traverses the tree according to the order of the dependent variables. If a variable is redefined it is important that it be added to the beginning of the 'deponits' of records of dependent variables otherwise old values could get used in evaluations prior to dealing with the new definition. The operation of adding to the front is foreign to Lisp so care is necessary in doing it and it costs some time in evaluation.

Type processing is very difficult to do if we allow the wide powers of Lisp to be used. Many Lisp operations are on lists with members that may be of any type; those operations (even the basic ones of `car` and `cdr` - returning the head and tail of a list respectively) are virtually impossible to use explicitly in EdenLisp because of the typing problem. The set of "command" operations in Lisp that deal directly with lists are therefore not implemented, although they can be accessed via embedded AutoLisp.

Type processing is computationally expensive. With the tree traversal involved with evaluation EdenLisp becomes slow in action. Care has been taken to make each process computationally as simple as possible but the sum of the processes makes updating take considerable time. Although that is acceptable for prototype activities to test the ideas of EdenLisp some work is necessary to make it useful to engineering users. Creating new functions means typing them. Some provision for that has been implemented in the form of a Lisp function `newtype(name, type, input type)` that can be invoked at the time of creating the function in AutoLisp.

The windowing environment is complicated and makes large programs potentially extremely verbose. One possibility is to consider all environments except the current one to be static or historical. That would not permit parallel environments to get changed when a common parent variable gets changed, although the task of updating will need to be redone each time a window is re-entered. The problem is to balance computational time versus space, a problem identified earlier in our discussion.