OCTOLISP: An Experiment with Data Structures

R Dempster
Department of Computer Science, University of Natal, P O Box 375, Pietermatitzburg 3200

Abstract

OCTOLISP is a functional programming language which provides the programmer with a number of explicit data structures such as lists, stacks, vectors, sets, trees and forests. Each of these data structures also has a specific access mechanism associated with it which affects the manner in which computations using the structures behave. The language thus contains a relatively high level of abstraction and it is the purpose of this paper to show that this is the case. This will be done by firstly describing the language in enough detail to make a program written using the language understandable, and then by discussing an algorithm implemented in OCTOLISP.
Keywords: *Lambda expression, concurrency, parallellism.*
Computing Review Categories: *D.1.1, D.3.2, E.1.*

Received June 1989, Accepted April 1990

1. Introduction

OCTOLISP is a functional programming language which is a major extension of LISP. The language is currently under development in the Department of Computer Science at the University of Natal. One of the major features of OCTOLISP is the introduction of explicit data structures such as lists, stacks, sets, vectors, trees and forests as standard structures. Furthermore each of these structures has a specific evaluation strategy associated with it. Two of these strategies involve a degree of concurrency or parallelism which hopefully would make the language suitable for the solution of problems which require such a capability.

The basic ideas which led to the evolution of the language were that of combining the COND list of LISP [8] and the set-like *if - fi* and *do - od* control structures of Dijkstra [5]. The use of a structure-and-access-method to specify evaluation strategies was first proposed by Postma [10] and the resulting language was named QUADLISP at the Płock Conference in Poland [11]. These ideas were elaborated upon [12] resulting in a language which was complete, but rather inelegant. QUADLISP then became known as QL/8n as it went through a number of revisions. The final revision of QL/8n, now known as OCTOLISP, has been defined by Postma [13] for implementation purposes.

2. The Data Structures

In order to accommodate the extended data structures mentioned above OCTOLISP uses a nexus of four pointers (quad) rather than the bi-pointer cell of LISP. Two of these pointers are not unlike the 'car' and 'cdr' pointers of the LISP cell and bear the same names. They are primarily used to link quads in order to form composite structures. The other two pointers are the 'cvr' and 'cgr', pronounced 'cover' and 'cougar' respectively and are used primarily to associate data with the quad in question. The datum associated with the cgr is typically a guard or key while that associated with the cvr is a value or entity. The quad is also tagged in a manner similar to the LISP cell to allow the dynamic determination of its type.

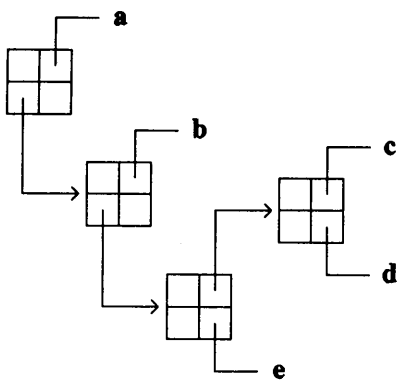
Quad Constants

These are paired key-entity structures created by using the Cons-Q operator. The dot notation of LISP is however extended by the use of four symbols known as *quadifiers.* The quadifiers in turn determine the structure of the quad constant in terms of the following precedence relationship which exits between them. A simple example of their use to create a quad constant is also shown:

Precedence Table

	:	↦	::	:=
:	<	>	>	>
↦	<	<	>	>
::	<	<	<	>

a : b : c ↦ d := e



Linear Data Structures

The simple linear data structures (lds) include lists, stacks, rings, sets and vectors of **entities** (values or elements) each denoted syntactically by means of a specific pair of brackets. Each of these structures has a specific mechanism allowing access to the contained values. The list and stack are accessed from the left and right respectively and are both based on singly linked lists of quads. The ring corresponds to a circularly linked list. The set is considered to be a random structure with all the entities (values or elements) equally accessible. It is sorted lexicographically and implemented as a doubly linked list of quads. The vector is considered to be a dense structure as the index-addressable values will by definition be considered to be stored in quads which are stored in contiguous memory locations. It is implemented as a singly linked circular list.

Keyed Linear Data Structures

The keyed linear data structures (*kds*) are a variant of the simple linear structures which contain **key/entity** pairs as elements. The key may be used as a means of accessing the corresponding value. The key could also be treated as a **guard**, the value of which will determine whether access to the corresponding element will be permitted or not. Each key/entity pair is in fact an instance of the quad constant described previously.

Trees and Forests

These structures are non-linear data structures and are merely mentioned here, for they will have no impact on the rest of this paper.

Some Notation

It seems appropriate at this stage to introduce some of the notation associated with the data structures and then to illustrate the notation by means of an example. Each of the data structures mentioned other than the quad constant is denoted by means of a specific pair of brackets as follows:

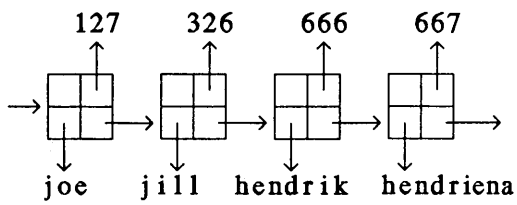
⦃list⦄ ⦃stack⦄ ⦃set⦄ ⦃vector⦄ ⦃ring⦄

The null, and universal, data structures are represented by means of the appropriate pair of brackets enclosing nothing, or two adjacent asterisks, respectively.

As an example we consider a keyed list containing the numbers and names of some fictitious delegates attending a conference. Syntactically this is represented as follows:

⦃127:joe 326:jill 666:hendrik 667:hendriena⦄

and structurally in terms of quads as follows:



3. Evaluation Strategies

OCTOLISP is by nature an applicative language and thus the execution (interpretation) of any program entails the application of a function to a number of arguments. In order to apply the function to the argument(s), the function itself and then each of the argument(s) in turn have to be evaluated if their values are required.

The argument(s) may take on a number of different forms.

The evaluation of forms which are specified by means of data structures is determined by the access mode associated with the data structure in question and the environment within which the data structure occurs. Three basic strategies are employed for the data structures: **sequential** in the appropriate direction for lists and stacks, and **determinate parallel** and **indeterminate parallel** for vectors and sets respectively. Before continuing with a description of the evaluation strategies it should be pointed out that OCTOLISP makes use of three valued logics employing the states true (t), false (f) and undefined (⊥). Any computation (process) which does not complete within some finite or arbitrarily defined period of time may be considered to be undefined. Once a computation has acquired the status, 'undefined' it may be terminated by its parent computation (process).

Sequential Evaluation
This mode of evaluation is associated with the list and stack and is similar to the familiar imperative mode of the procedural languages such as Pascal. The guards associated with the entities (values or elements) of the structure will be evaluated in an order determined by their position within the structure.

Parallel Evaluation
Computations which proceed in parallel must either do so as disjoint processes as described by Brinch Hansen [4], or if the intersection of their environments is non-empty they must employ a mechanism which ensures that this intersection remains consistent throughout the computation. OCTOLISP provides two intrinsic mechanisms for specifying the parallel computation to be associated with the evaluation of guards, determinate parallel and indeterminate parallel. These computations are by definition disjoint as no assignment is allowed in guards. The environment that is extant at the time that any one computation commences will remain consistent for the duration of that computation.

Determinate parallel processing is associated with the vector. It requires all the computations associated with the evaluation of the vector's guards to be defined if a result other than undefined itself is expected. These computations then proceed as parallel processes if sufficient processors are present. If insufficient processors are present then some arbitrary and not necessarily fair scheduling algorithm is used until all processes have terminated. No one process may make any assumptions regarding the progress or otherwise of any other process. By fair, we understand a scheduling algorithm which will distribute the processing capability equitably among the competing processes.

Indeterminate parallel processing is associated with the set. At least one of the computations associated with the evaluation of the guards of the set's elements should be defined if a result other than undefined itself is expected. These computations then proceed as in the determinate parallel case except that here the scheduling algorithm is required to be fair. The system aborts the balance of the processes as soon as one of them yields an acceptable result.

4. Evaluation Environments

The primary factor determining the nature of a computation related to any one of the intrinsic data structures of OCTOLISP is the type of the data structure itself. A secondary, but equally important factor is the language construct within which the data structure, and more specifically the keyed data structure, is accessed. As all of the data structures are not permitted as operands within all the constructs, discussions dealing with a particular construct will prefix the term 'ds' with the name of the construct. The corresponding abbreviation for the named data structure using the first letters of the 'name' and 'data structure' will also be used. Some of the available language constructs are described below. Specific examples will not be given here as some illustrations of the use of these constructs will follow in a program segment presented later on.

The 𝕂-conditional: 𝕂 𝕀
The 𝕂-conditional, 𝕂 *kds* 𝕀 is an operator that takes as its operand a keyed data structure.

If the *kds* is a list or a stack then the guards are evaluated sequentially from the

appropriate side until a guard evaluates to true (t). The corresponding entity (element or value) is then returned as the result. If none of the guards is found to be true or if, in order to find a true guard, an attempt is made to evaluate a guard which returns a result of undefined then undefined (⊥) is the result.

If the *kds* is a vector, then the guards are evaluated in determinate parallel order. If all the guards are defined then, once all the guards have been evaluated, an arbitrary choice is made from amongst the true guards and the corresponding entity is returned as the result. If one or more guards is undefined or all are false then the value of the conditional is also undefined.

If the *kds* is a set, then the guards are evaluated in indeterminate parallel order. The value returned will be that corresponding to the first guard that evaluates to t. If all the guards are undefined or false then the value of the conditional is also undefined.

The block: 𝔹 𝔻 (or 𝔹-conditional)
The block (or 𝔹-conditional), 𝔹 d : i *bds* 𝔻 is an extension of the 𝕂-conditional which allows local variables to be declared and initialized before the *bds* is evaluated and all enabled values are returned. Unlike the 𝕂-conditional, the 𝔹-conditional returns a data structure of the same type as that used for 𝔹-conditional's body. This data structure will contain all the enabled values. The order of evaluation is once more determined by the evaluation strategy associated with the data *bds*. Lists correspond to bottom-up programming and stacks to top-down programming.

An iteration based on the block construct, but for value rather than effect, is the recurrence construct ℝ d : i *rds* ℜ. This is similar to the previous construct, but the evaluation or computation associated with the body is delayed rather than forced and thus will generally not result in an infinite computation. It naturally provides a stream of values as and when required.

Loops can also be defined in terms of recursive functions and the 𝕂-conditional to control the depth to which the recursion proceeds. The efficiency of this strategy can be considerably improved if the function lends itself to tail-end recursion.

5. The Lambda Expression and the Data Structures

OCTOLISP is a functional language and like LISP it makes use of lambda expressions to abstract functions from expressions. The most common form or representation of lambda expressions in λ-calculus involves the use of the Greek letter, lambda (λ) followed by a list of bound variables or formal arguments, followed by the expression itself typically separated from the bound variables by a period or dot. A typical lambda expression could be $\lambda xy.x^2+y^2$. The general form or syntax of the lambda expression in OCTOLISP makes use of a unique pair of brackets to denote the lambda expression:

ϕ λ-variables body where-clause ϕ

The λ-variables
The λ-variables correspond to the bound variables or formal arguments. The body of the lambda expression consists of an expression which may contain both bound and unbound or free variables. The where-clause provides a mechanism whereby the environment within which the lambda expression is going to be evaluated can be extended. All this is in effect really just syntactic sugar and should not detract from the real issue here, namely the way in which the OCTOLISP data structures affect the definition and evaluation of lambda expressions.

If the λ-variables are contained within a list or a stack then their evaluation is **delayed** and the lambda expression may be **curried**. That is, if the lambda expression is applied to a list or stack which contains fewer actual arguments than the number of formal arguments expected, then the result returned will be a new lambda expression. This lambda expression will contain the unevaluated actual arguments supplied, bound to their respective formal arguments. The remaining formal arguments constitute the new lambda variables. If the λ-variables are contained within a vector then currying is forbidden as the vector is a determinate structure. This also means that the evaluation of the λ-variables is **forced** and that the mode in effect corresponds to call-by-value. If the λ-variables are contained within a set then they must be

mutually independent as well as commutative. This will allow the evaluation of the λ-variables to be delayed. Currying is allowed. Guarded λ-variables are used when it is desirable to associate some form of type checking with the corresponding arguments.

The where-clause

The where-clause allows for the assignment of expressions to variables which occur free within the body of the lambda expression. This is achieved by means of an assignment statement, which assigns to a **set** of variables corresponding expressions. These expressions are declared by means of appropriate keyed data structures within which the keys correspond to the elements of the set. Once again the data structures are used to achieve effects in accordance with their associated evaluation strategies. If the data structure is a list or stack the variables are instantiated bottom-up or top-down respectively. In the case of the set the instantiations must be mutually independent while the ring allows mutual recursion. Finally the evaluation of the elements of a vector are all forced.

6. An Example

As an example of the use of OCTOLISP as a programming language we consider the coding of an algorithm for a predictive parsing table generator for LL(1) grammars which is well documented [1], [7], and [2]. As space is limited we will furthermore only present one of the routines used, namely **remuseless**, which was developed by Nevin [9]. These routines have been coded in a number of texts using various languages. The Pascal routines developed by Backhouse [3] could be used to provide a crude estimate of the expressive power of OCTOLISP.

The OCTOLISP routines

The grammar will be represented by a 4-tuple implemented in terms of a four element OCTOLISP list:

$$\{ \; V \;\; T \;\; P \;\; S \; \}$$

where V is the set of non-terminals, T the set of terminals, P the set of productions all stored in OCTOLISP sets and S is the non-terminal known as the starting symbol. The various symbols are represented by means of OCTOLISP text strings. Strings of grammar symbols will be represented by OCTOLISP lists of symbols. The null string is thus represented by means of the empty list $\{ \}$. The productions are represented by means of quad constants of the form $\alpha \rightarrow \beta$. The non-terminal α behaves as a key while β represents a string of grammar symbols represented as a list. The two are paired in a quad constant by means of the quadifier $\rightarrow$. A simple grammar for the sum or difference of two identifiers is represented as follows:

$$\{\{E\} \; \{id+-\} \; \{E \rightarrow \{id+id\} \; E \rightarrow \{id-id\}\} \; E\}$$

The inherent OCTOLISP data structures have not only made the process of setting up the structures for representing the grammar trivial, they have also resulted in a representation which is very similar to that used by the conventional mathematical notations.

We now examine the OCTOLISP implementation of the routine, **remuseless** which removes useless productions from the set of productions contained within a tuple specifying a grammar. The algorithm will be presented first followed by a detailed description of its behavior.

remuseless := *note: All italic text constitutes comments.*

Given a cfg G, returns a cfg G' where L(G) = L(G') and G' contains no useless symbols i.e. if G'=(V' T' P' S) then for all A in V' there exists a derivation $A \xrightarrow{*} w$ *where w is in* $(T')^*$

$\phi \; \{ \; G : \{ \; V \; T \; P \; S \; \} \} \; \phi$ **list$_4$ Vd T Pd S** ϕ *Create new grammar where*

$\{$ **Vd Pd isvalid findalive newalive** $\}$:=

$\{$**Vd** := $\phi_S \cap$ **V** ϕ **findalive T** ϕ ϕ *create new set of non-terminals*

Pd := $\phi_S |$ ϕ**isvalid** ϕ**findalive T** ϕ ϕ **P** ϕ *and productions set selector applies curried isvalid to each production in turn*

```
isvalid :=
        True iff all symbols in production p are in the set syms
        ϕ ∉ syms p : <lhs : rhs> ⊦ ϕ & ϕs∈ syms lhs ϕ
                                      ϕs⊑ syms ϕ mkSet rhs ϕϕϕ
        ϕ

{findalive} :=
        Returns as a set all non-terminals A st A ⟶* x
        for some x where all symbols in x are alive.
        ℭ findalive :=
                ϕ ∉ alive ⊦
                  𝕂 ∉ ϕs= alive ϕ newalive alive ϕϕ ⇒ alive
                       t ⇒ ϕ findalive : newalive alive ϕ
                    ⊦
                  𝕂
                ϕ
        ∋

newalive :=
        Returns as a set all non-terminals A st A ⟶* x is a
        production and all symbols in x are alive.
        ϕ ∉ alive ⊦
             ϕ μ ifalive ⇒ q↦ P ϕ
                [[ ifalive :=
                     ϕ ∉ p : <lhs :: rhs> ⊦ ϕs⊆ alive ϕ mkSet rhs ϕϕϕ
                     ϕ
                ]]
         ϕ
    ⊭
ϕ
```

remuseless := *note: All italic text constitutes comments.*

The variable **remuseless** is bound to the function definition which follows.

ϕ ∉ G : ∉ V T P S ⊦⊦ ϕ **list$_4$ Vd T Pd S** ϕ
Create new grammar where

This is the definition of a lambda expression which is always enclosed within the special brackets (hereafter *sbs)*, ϕ ϕ. The bound variables (or formal arguments) have been specified by means of a quad constant. This is a form of variant which allows the programmer to attach a handle so to speak to the OCTOLISP list containing the arguments. Because the formal arguments are specified by means of a list the actual arguments will be evaluated sequentially from the left. The body of the lambda expression consists of a function application contained within the *sbs* ϕ ϕ. The function is the **list** constructor which will create a list from the values of its arguments, **Vd, T, Pd** and **S**. As **Vd** and **Pd** do not appear amongst the bound variables they must either be free (i.e. previously defined in the environment within which **remuseless** is defined) or defined by means of a where clause as is the case here. The where clause is a means of defining additional values and functions required within the current environment.

{Vd Pd isvalid findalive newalive } := {...⊭

This where clause defines a set of values by means of an OCTOLISP stack which corresponds to a bottom-up evaluation. They will thus be described in that order here.

```
newalive :=
Returns as a set all non-terminals
A st A ⟶* x is a production and all symbols
in x are alive.
        ϕ ∉ alive ⊦
             ϕ μ ifalive ⇒ q↦ P ϕ
                [[ ifalive :=
                     ϕ ∉ p : <lhs :: rhs> ⊦
                       ϕs⊆ alive ϕ mkSet rhs ϕϕ
                     ϕ
                ]]
        ϕ
```

newalive is also a lambda expression which has one bound variable, namely **alive**, and a function application as a body. The function application consists of the functional μ, which maps the guarded function **ifalive** $:\rightarrow {}_{q\Gamma\rightarrow}$ onto **P**, the set of productions. **ifalive** is thus applied to each of the productions in **P** in turn and if the result is true causes $_{q\Gamma\rightarrow}$ (extract the left hand side of a quad structure) to be applied to the tested production from **P**. If the test returns true then the left hand side of the production is included within the set of alive non-terminals, **alive**. **ifalive** is a lambda expression which takes as an argument the production, converts the right hand side to a set of grammar symbols and then determines whether or not the grammar symbols are contained within **alive** or not.

```
{findalive} :=
Returns as a set all non-term's A st A -*-> x
for some x where all symbols in x are
alive.
ɕ findalive :=
 ϕ ∉ alive }
   𝕂 ∉ ϕs= alive ϕ newalive alive ϕϕ :⇒ alive
        t :⇒ ϕ findalive : newalive alive ϕ
     }
   𝕀
 ϕ
ɔ
```

findalive is a lambda expression defined within the context of a ring (denoted by the *sbs*, ɕɔ) which results in the body of the lambda expression being evaluated recursively. In this case the recursion is controlled by the 𝕂-conditional. The first clause in the conditional determines whether the set of alive symbols and the set derived by applying **newalive** to **alive** is equivalent and returns **alive** if they are. If not **findalive** is recursively applied to **newalive** which is in turn applied to **alive**.

```
isvalid :=
True iff all symbols in production p are in
the set syms
ϕ∉ syms p :<lhs : rhs>}
 ϕ & ϕs∈ syms lhs ϕ ϕs⊑ syms ϕ mkSet rhs ϕϕϕ
ϕ
```

isvalid is a lambda function which has two bound variables, **syms**, the set of symbols and p, a production represented as a quad with a left and right hand side. The body forms the logical and of the results of two set operations. The first determines whether the symbol or left hand side of the production is an element of **syms** while the second determines whether the symbols on the right hand side of the production considered as a set are contained within the set **syms**. The set **syms** is of course the set of alive symbols.

```
Pd := ϕs| ϕisvalid ϕfindalive T ϕ ϕ P ϕ
create a new set of productions the set
selector applies the curried isvalid
function to each production in turn
```

The function **isvalid** which requires two arguments is called with one, namely the result of applying **findalive** to **T**. The result of this is a function expecting the remaining arguments. This curried function is then applied to each of the productions in **P** in turn by the set selector resulting in a set which does not contain useless productions.

```
Vd := ϕs∩ V ϕ findalive T ϕ ϕ create new set
of non-terminals
```

Apply the set intersection operator to the set of non-terminals and the set of those grammar symbols which have been determined to be alive to form the set of alive non-terminals.

Discussion

The most striking feature of the implementation is the length of the routines. This should not be surprising as functional programming languages are known for their expressive power and the resulting brevity of programs [6]. If the brevity is more marked in the case of the OCTOLISP implementation it can probably be attributed to the fact that the **remuseless** algorithm is based on pattern matching and that one of the OCTOLISP design objectives was to provide the programmer with intrinsic pattern matching capabilities.

The next most striking feature is the almost natural translation of the algorithm into OCTOLISP. This together with the functional paradigm results in routines which are more easily proved correct [6]. An implementation using structures and routines defined using a procedural language will, despite the best of intentions, rely heavily on documentation in form of comments to relate the

implementation to the algorithm. This marked difference greatly increases the possibility of misrepresentation and makes any attempt to reason about the routines in a manner consistent with the algorithm a difficult one.

7. Conclusions

LISP has been criticized in the past for being synonymous with Lots of Irritating Silly Parentheses. If that is the case how much more so for OCTOLISP which introduces a fairly extensive set of rather unusual brackets. We argue that while it may be more difficult to learn the language initially, the brackets and notation provide a concise and powerful form for expressing algorithms. This is in particular true for algorithms related to grammars, automata, algebraic forms and languages generally. It is also intended that the language be used for the formal specification of systems and as a prototyping tool for complex systems.

References

[1] A V Aho and J D Ullman, [1972]. *The Theory of Parsing, Translation and Compiling*, Vol. 1, Prentice-Hall, Englewood Cliffs, N. J.

[2] A V Aho and J D Ullman, [1977], *Principles of Compiler Design*, Addison-Wesley, Reading, Mass.

[3] R C Backhouse, [1979], *Syntax of Programming Languages*, Prentice-Hall, London.

[4] P Brinch Hansen, [1973], *Operating Systems Principles*, Prentice-Hall, Englewood Cliffs, N. J.

[5] E W Dijkstra, [1976], *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N. J.

[6] P Henderson, [1980], *Functional Programming Application and Implementation*, Prentice-Hall, Englewood Cliffs, N. J.

[7] P M Lewis II, D J Rosenkrantz, and R E Stearns, [1976], *Compiler Design Theory*, Addison-Wesley, Reading, Massachusetts.

[8] J McCarthy et al., [1962], *Lisp 1.5 Programmer's Manual*, The M.I.T. Press, Cambridge, Mass.

[9] N J Nevin, [1989], OCTOLISP programs for the transformation of Context Free Grammars, Honours Project, U. N. P.

[10] S W Postma, [1979], 'n Kritiese Evaluering van die Programmeertaal LISP met Voorstelle vir Moontlike Verbeteringe. Thesis, Rand Afrikaans University, Johannesburg.

[11] S W Postma, [1980], LYSPE/2 - a program language for theoretical program science in Mathszewski, R: Conference on Technology of Science, Płock, Poland. pp 39-59.

[12] S W Postma, [1984], On the Definition and Implementation of the Program Language QUADLISP, Thesis, UNISA, Pretoria.

[13] S W Postma, OCTOLISP syntax and Language; Technical Report in preparation.