THE DESIGN OF
O R I N J
* *
A VERY-HIGH-LEVEL NON VON NEUMANN LANGUAGE

by

W E S L E Y F . M A C K E Y

A Thesis
Presented to the Faculty of Graduate Studies
in partial fulfillment
of the Requirements for the Degree
DOCTOR OF PHILOSOPHY

Department of Computer Science
University of Manitoba
Winnipeg, Manitoba, Canada, R3T-2N2

The Design of ORINJ:

A Very-High-Level non von Neumann Language

BY

WESLEY FRANCIS MACKEY

A thesis submitted to the Faculty of Graduate Studies of
the University of Manitoba in partial fulfillment of the requirements
of the degree of

DOCTOR OF PHILOSOPHY

ABSTRACT

The Design of ORINJ:
A Very-High-Level non von Neumann Language
*
Copyright (C) 1980 -- WESLEY F. MACKEY
*

ORINJ is a programming language designed to aid in the specification and construction of algorithms and programs. It can serve the programmer both as a high-level specification language which is used to clarify the initial ideas of the problem, and it can also be used to completely specify a program at a greater level of detail. Thus, by a series of manual transformations on the text of a program, increasingly greater detail is given until it is within a compiler's ability to perform the rest of the transformations leading to a program in machine code.

ORINJ has several important features that distinguish it from other common languages. It is a non von Neumann language and is strictly value-oriented. The language is amenable to being run on a system with massive unstructured parallel execution, such as a data-flow machine. Consistent with current ideas in language design, it forbids trouble-causing features such as pointers, and permits the defini-

tion of abstract data types. ORINJ uses the generic function and generic selection as the basis of its means of making choices, and is essentially an expression language rather than a statement-oriented language.

Key words and phrases: abstract data types, data structures, encapsulation, generic functions, data-flow, non von Neumann languages, specification, synthesis, stepwise refinement.

CR categories: 4.29, 4.32, 4.34

ACKNOWLEDGEMENTS

I would like to thank my thesis advisor, Dr. P.R. King, for finding the time to read the many earlier versions of this thesis, and for many discussions of its material.

I would also like to thank the faculty members of the Mathematical Sciences Dept. of Florida International University for providing the incentive (i.e. a faculty position) to finish this project.

A scholarship from the (then) National Research Council of Canada (now NSERC) and a fellowship from the University of Manitoba provided financial assistance. This thesis was printed using the University of Waterloo SCRIPT text formatter.

TABLE OF CONTENTS

CHAPTER 2
DESCRIPTION OF 'ORINJ'

CHAPTER 3
EXAMPLES OF SPECIFICATIONS IN 'ORINJ'

LIST OF FIGURES

CHAPTER 1

THE DESIGN OF 'ORINJ'

In the days when computers cost millions of dollars and filled large rooms, programs were cheap by comparison and not very large by today's standards. But now, the situation is reversed, and hardware costs are very much smaller than software costs. As Dijkstra [028] pointed out during his 1972 Turing Award lecture:

> The increased power of the hardware, together with the perhaps even more dramatic increase in its reliability, made solutions feasible that the programmer had not dared to dream about a few years before. And now, he had to dream about them, and even worse, he had to transform such dreams into reality!

1.1. DESIGN GOALS

In order to facilitate this "transformation process", many languages have been created. Yet another language is being added to this babel. Naturally, the creation of a new language must be justified in terms of its design goals.

This new language, ORINJ* by name, is designed to aid in the specification and construction of algorithms and programs. ORINJ is intended to facilitate the development of programs by providing a tool which reduces the intellectual complexity of an algorithm by excluding irrelevant details until they become needed. ORINJ also serves as a notation in which the successive steps in a sequence of transformed programs can be expressed.

1.1(a) Hierarchical decomposition.

McKeeman [073] views the programming process as a series of levels, the highest of which is constituted by ideas in the mind of the problem solver. In order to make use of these ideas, they must be written down and clarified. These ideas are then re-expressed in increasingly more detail at increasingly lower levels until a program in machine code is realized.

This hierarchy can be divided into two parts -- the upper part is taken care of by humans, and the lower part is handled automatically by the computer. The exact position of the line dividing these two parts from each other varies according the sophistication of the programming system being used.

---

* Instead of naming the language after a person (ADA, EUCLID, EULER, PASCAL) or a star (ALGOL, ALPHARD), we follow convention (APL), and mis-spell the name of a fruit -- hence the name ORINJ!

The design goal of ORINJ is to raise this "line" to as high a level as possible, thus relieving the human of some of the burden of programming, and requiring the machine to do more work.

1.1(b) Specification languages.

Winograd [096] distinguishes three kinds of specifications, each of which is related to one or more of the levels in the hierarchy mentioned above.

1. Specification of a set of instructions, which can be used directly in a program. This is the usual procedural-language method.

2. Specification of inputs (preconditions) and outputs (postconditions), ignoring internal details, which will be filled in later. This style is used by ALPHARD [101] [102] in its "specification part".

3. Specification of the general behaviour of the system as a whole, without giving any idea of how to solve the problem. This is the main idea introduced in Winograd's [096] report.

ORINJ is primarily focussed on the second level, but is related to each of the other two levels, and provides a bridge between them. In this way, we hope to make it possible for the problem solver to do more work in a single language, rather than changing languages at each level boundary is is done, for example, by Walker, Kemmerer, and Popek [093] in the specification and implementation of an

operating system. This single-language approach also figures prominently in the construction of SETL [025] [026] programs.

1.2. DESIGN CRITERIA (EXCEPT TYPES)

In designing a new language, careful consideration must be given to what features to include in it. Whorf [095] has hypothesized that the nature of a language constrains the various possible ways of considering a given problem. { Consider doing list processing in FORTRAN -- No thank you! }

There is, of course, no such thing as the perfect programming language, but an approximation to it must make the expression of solutions to problems as easy as possible. ORINJ is to be a general purpose specification language, or very high level language, capable of describing the essential details of a problem, without necessarily giving any consideration to implementation details.

And we must not forget McKeeman's advice [073]:

> Having decided not to do very much, and to copy most of it, the problem reduces to achieving the necessary features in a consistent manner.

In other words, we should keep the language as small as possible without unduly restricting its domain of application.

1.2.1. Some necessary features

The previous paragraph raised the question of what constitutes the desirable features of the language. In choosing features for the language, we must try to ensure that there are neither too few nor too many.

1.2.1(a) Elimination of variables.

One problem with many programming languages is the way in which variables are used. An early discussion by Wulf and Shaw [103] censured the global variable, which can be changed in unexpected ways at unexpected places. Bauer [010], in discussing program development, suggests that variables are introduced much too early in the programming process, since they lead to remote connections of data, just as the GOTO leads to remote connections of commands.

Runciman [084], using PASCAL as a vehicle for explanation, mentions several good reasons to restrict the use of variables as much as possible, using recursion instead, even in the final implementation:

1. Static concepts are easier to understand than dynamic ones, since, with static concepts, we do not have to "mentally execute" a program to discover its meaning.
2. Novice programmers can lose their grip on program construction by introducing too many variables.

3. A small piece of code can have far-reaching effects, such as changing a global variable which is referenced many pages later.

4. Formal descriptions of programs with variables are complicated.

5. The value of a variable can be undefined.

It should be noted here that we are referring to what ALGOL 68 [063] [092] calls references -- we are not objecting to symbolic constants or "read-only" variables.

Thus, for the reasons mentioned above, we would like to delete, or at least restrict as far as possible, the use of variables. The language would then be what is often called a "single assignment" language. Of course, some variables are necessary, if only at the top level, as in LISP [072] [094], so our ideal of no variables can not be completely realized.

Restricting variables makes it easier to transform programs in an algebraic manner without regard to complicated instructions between statements, thus reducing the intellectual complexity of a piece of code.

1.2.1(b) Executable specifications.

Besides having a specification which describes a problem, our design goals require a specification which is directly executable, however inefficient it may be. It follows from this that we can test the specification on some sample

problems, and if necessary change the specification until a "correct" version is achieved -- where "correct" means that version which is intuitively what we think we want.

We are thus saved the necessity of recoding an implementation in whole or in part, as would be necessary in most other languages. We can also delay choosing the most efficient concrete data structures until we have a sufficient knowledge about the problem to be able to make an intelligent choice.

Having written the specifications of the solution to a given problem, we would like to be able to make some changes in the specifications in order to increase the efficiency of the resulting program. The result could be a sequence of ever more refined specifications such as is mentioned in the paper referenced above. More probably, however, the set of successively more detailed specifications will form a tree, or family, of programs, such as Parnas [079] discusses.

1.2.1(c) "DO considered OD". [048]

An important question that must be answered is whether or not there is to be a looping construct in the language. If we follow our decision to eliminate variables, then any looping construct is also prohibited, since loops can not return values. Some such attempts [099] have been made, but the result is not very convincing. And, if loops can not modify variables, then there is no point in having them.

If we delete loops from ORINJ, then all iteration must be done by recursion. Many people will object to recursion on two grounds:

1. Recursion is expensive, because procedure calls take a lot of time and memory.

2. Recursion is harder to understand than looping.

We agree with Hehner [048] in answering these objections:

1. Agreed -- if you are using PL/I. In a language without environment nesting, i.e. no global variables, recursion can be made very cheap -- in the case of tail recursion, as cheap as looping. LISP [094] and VAL [002] have this property. In any case, a specification language does not address the problem of efficiency.

2. Again agreed -- if you make the mistake of explaining it in terms of procedure calls and returns. In an applicative non von Neumann language (see Section 1.2.3), it can be explained in terms of textual substitution using algebraic rules, and there is no problem. In fact, when explained properly, recursion can even be included in a (von Neumann) beginner's language such as MABEL [062].

As we shall soon see, the decision to prohibit loops and use recursion fits in very well with other features of non von Neumann languages.