FCL: a purely functional language for data-flow programming

Peter Michael Maurer
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/rtd

Part of the Computer Sciences Commons

Recommended Citation
http://lib.dr.iastate.edu/rtd/7057

This Dissertation is brought to you for free and open access by Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
INFORMATION TO USERS

This was produced from a copy of a document sent to us for microfilming. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help you understand markings or notations which may appear on this reproduction.

1. The sign or “target” for pages apparently lacking from the document photographed is “Missing Page(s)”. If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure you of complete continuity.

2. When an image on the film is obliterated with a round black mark it is an indication that the film inspector noticed either blurred copy because of movement during exposure, or duplicate copy. Unless we meant to delete copyrighted materials that should not have been filmed, you will find a good image of the page in the adjacent frame. If copyrighted materials were deleted you will find a target note listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed the photographer has followed a definite method in “sectioning” the material. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For any illustrations that cannot be reproduced satisfactorily by xerography, photographic prints can be purchased at additional cost and tipped into your xerographic copy. Requests can be made to our Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. in all cases we have filmed the best available copy.
Maurer, Peter Michael

FCL: A PURELY FUNCTIONAL LANGUAGE FOR DATA-FLOW PROGRAMMING

Iowa State University

University Microfilms International 300 N. Zeeb Road, Ann Arbor, MI 48106

Ph.D. 1982
FCL: A purely functional language for
data-flow programming

by

Peter Michael Maurer

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major: Computer Science

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa

1982
**TABLE OF CONTENTS**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>II. THE FCL LANGUAGE</td>
<td>8</td>
</tr>
<tr>
<td>A. Introduction</td>
<td>8</td>
</tr>
<tr>
<td>B. The Elements of the FCL Language</td>
<td>10</td>
</tr>
<tr>
<td>1. The basics</td>
<td>10</td>
</tr>
<tr>
<td>2. The FCL operators</td>
<td>12</td>
</tr>
<tr>
<td>3. Special parentheses</td>
<td>18</td>
</tr>
<tr>
<td>4. The CURRY and UNCURRY operations</td>
<td>20</td>
</tr>
<tr>
<td>5. Additional functions</td>
<td>20</td>
</tr>
<tr>
<td>6. SETS, and SET functions</td>
<td>24</td>
</tr>
<tr>
<td>7. Syntactic sugar</td>
<td>25</td>
</tr>
<tr>
<td>C. Data Structures</td>
<td>32</td>
</tr>
<tr>
<td>D. The UNDEFINED Element</td>
<td>37</td>
</tr>
<tr>
<td>E. Examples of FCL Programs</td>
<td>40</td>
</tr>
<tr>
<td>F. Concluding Remarks</td>
<td>43</td>
</tr>
<tr>
<td>III. COMPARISON WITH OTHER LANGUAGES</td>
<td>45</td>
</tr>
<tr>
<td>IV. THE TRANSLATION ALGORITHM</td>
<td>49</td>
</tr>
<tr>
<td>A. Introduction</td>
<td>49</td>
</tr>
<tr>
<td>B. Review of Existing Work</td>
<td>52</td>
</tr>
<tr>
<td>C. Parsing FCL</td>
<td>57</td>
</tr>
<tr>
<td>D. Preliminary Notions</td>
<td>61</td>
</tr>
<tr>
<td>E. Details of the Translation Algorithm</td>
<td>65</td>
</tr>
<tr>
<td>1. The overall structure</td>
<td>65</td>
</tr>
<tr>
<td>2. Phase 1: Combining mutually recursive functions</td>
<td>66</td>
</tr>
</tbody>
</table>
3. Phase 2: Eliminating recursion
4. Phase 3: Abstracting sub-expression names
5. Phase 4: Uncurrying functions
6. Phase 5: Abstracting argument names

F. Code Generation
G. Conclusion

V. THEORETICAL FOUNDATIONS FOR THE STUDY OF SHARED DATA
A. Introduction
B. Literature Review and Derivation of Requirements
C. Data Manager Syntax
D. Application Program Syntax
E. FCL Processes and Timing
F. Process Syntax
G. General Process Communication System
H. Specifications for a Shared-data Process Communication System
I. The Desugared Form of a Data Manager
J. Specifications for an Interleaved Data Manager
K. Application Program Graphs for Interleaved Data Manager
L. Conclusion

VI. CONCLUSION
VII. REFERENCES
VIII. ACKNOWLEDGMENTS
<table>
<thead>
<tr>
<th>Appendix</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>IX.</td>
<td>APPENDIX A: THE FCL GRAMMAR</td>
<td>200</td>
</tr>
<tr>
<td>X.</td>
<td>APPENDIX B: EXAMPLES AND DETAILS OF HALSTEAD-MEASURE CALCULATIONS</td>
<td>204</td>
</tr>
<tr>
<td>XI.</td>
<td>APPENDIX C: SEMANTIC ISSUES</td>
<td>317</td>
</tr>
</tbody>
</table>
I. INTRODUCTION

The last few years have seen an ever increasing interest in functional computer architectures and high-level languages. One such architecture is the data-driven architecture [Dennis 1974]. A number of different data-driven architectures have been proposed [Arvind and Gostelow 1977, Davis 1978, Dennis and Misunas 1975, Dennis and Weng 1979, Johnson 1980, Plas et al. 1976, Sleep 1980, Watson and Gurd 1979], but the unifying concept behind all such architectures is the idea that an operation may execute as soon as its operands become available. This, of course, leads to a potential massive increase in parallelism. For example, consider the high-level expression \((a+b) + (c+d)\). In data-flow machine language this expression would be represented by the graph of Figure 1.1.

![Figure 1.1. Example of a data-flow graph](image-url)
The values of a, b, c, and d would be represented by tokens which flow along the arcs of the graph. In this example the sub-expressions (a+b) and (c+d) may be evaluated in parallel.

An important variation of the data-flow architecture is the demand-driven or lazy-evaluator architecture [Friedman and Wise 1976, Keller, Lindstrom and Patil 1979, Turner 1979a,b]. Although these architectures have some interesting features, this dissertation will focus on data-driven architectures.

The massive potential parallelism of data-driven machines introduces some complications into the translation of high-level languages. Consider the following program segment:

\[
X := 1; \\
X := 2; \\
A := X+1
\]

If this program segment were evaluated strictly on the basis of availability of operands, the value of X would be ambiguous when X+1 is evaluated. Within the context of a von Neumann language and translatibility to data flow, it has been shown that this ambiguity can be resolved using a flow-analysis technique [Allan and Oldehoeft 1980], but the exploitable parallelism is limited. Another response to
this problem is to enforce a single-assignment rule on the high-level code [Chamberlin 1971] (i.e., no variable name may appear more than once on the left hand side of an assignment statement).

Another problem imposed by conventional high-level languages is the restriction of parallelism for certain algorithms. For example, consider the algorithm of Figure 1.2 which squares each element of a K-element vector, A.

\begin{verbatim}
I := 1;
WHILE I <= K DO
BEGIN
    B(I) := A(I) * A(I);
    I := I + 1
END
\end{verbatim}

Figure 1.2. Squaring every element of a vector

Although the squaring operations of Figure 1.2 may all take place in parallel the statement I := I + 1 would force them to be done serially if the algorithm were translated precisely. The response to this problem has been to design programming constructs which allow the values of I to be generated in a more parallel fashion. One such construct is the FORALL construct of the VAL language
(Ackerman and Dennis 1978), which is illustrated in Figure 1.3.

\[
\begin{align*}
B: & = \text{FORALL } I \text{ IN } [1,K] \text{ DO} \\
C: & = A(I) * A(I) \\
& \text{CONSTRUCT } C \\
\end{align*}
\]

Figure 1.3. The FORALL construct of VAL

The FORALL construct is intellectually appealing because it does not specify how the index values of I are to be generated, only that they are to be generated in some reasonable fashion. This allows the programmer to ignore a certain amount of irrelevant detail.

In functional languages [Backus 1978, Friedman and Wise 1978, Henderson 1980, McCarthy et al. 1962, Turner 1976a,b] such as Backus' FP systems [Backus 1978], it is possible for the programmer to ignore even more detail. For example, in FP systems the program segment of Figure 1.2 could be written: \( \alpha(x_0[id, id]) \). This example is particularly appealing because it is highly compact and does not even mention an index.

There is actually very little difference between the functional languages cited above and high-level data-flow languages [Ackerman and Dennis 1978, Arvind, Gostelow and
Plouffe 1978]. Note that in Figure 1.3, the FORALL construct appears on the right-hand side of an assignment statement. This is permissable because the FORALL has a well-defined value, as do all other constructs in VAL. Other high-level data-flow languages also have this property. In this respect high-level data-flow languages are identical to the functional languages cited above.

On the other side of the coin, functional or applicative languages have been studied because of their compactness and potential for exposure of parallelism [Backus 1978, Friedman and Wise 1978]. It has been shown that applicative languages may be easily translated to pure combinatory code [Turner 1979a,b]. The pure combinatory code is elementary enough to serve as a machine language, and the translation algorithm is so simple that it is an easy exercise to prove it correct.

Although the concept of pure combinatory code was first put to use in a lazy-evaluation or demand-driven environment [Turner 1979a,b], it has been shown that this concept leads naturally into the concept of data-driven evaluation [Sleep 1980].

Existing applicative languages provide a wide variety of features which vary from language to language. The objective of this dissertation is to define a new applicative
language which combines the more useful features of various existing languages. These features will be extended or generalized wherever possible. This new language is called Functional Computing Language (FCL). A second, equally important objective is to propose a translation algorithm which is an enhancement of the algorithm used by Turner [Turner 1979a,b]. This algorithm will produce an intermediate code which is purely combinatory in form, but can be translated to data flow graphs more easily than the pure combinatory code of Turner. A third objective of secondary importance is to show that FCL may be used to provide a theoretical foundation for the study of shared data in a functional environment.

Some of the features of existing applicative languages which are included in FCL are:

1. Compact syntax,
2. The ability to support a wide variety of data structures,
3. The ability to introduce name-value bindings at any convenient place in the program,
4. The ability to define higher-order functions,
5. The ability to introduce function definitions at any convenient point of the program.

The syntax of FCL resembles that of FP systems [Backus 1978],
the Friedman and Wise applicative language [Friedman and Wise 1978], and SASL [Turner 1976]. This was done in an effort to realize the benefits of compact code exhibited by these languages. FCL's binding operators are syntactically equivalent to any other operator. This allows name-value bindings and function definitions at any convenient place in the program. This also enhances the programmer's ability to define higher-order functions. The ability to define data-structures will be discussed in depth in Chapter II.

The remainder of this dissertation is organized into five chapters. Chapter II presents the FCL language and some examples of its use. Chapter III presents a short comparison between FCL and other high-level languages. Chapter IV presents the details of an algebraic translation from FCL to low-level data-flow code. Chapter V presents a theoretical foundation for the study of shared data. Chapter VI presents conclusions.

In addition to the main text of this dissertation, three appendices have also been provided. Appendix A gives the grammar of FCL. Appendix B contains examples of FCL programs, as well as the details of the calculations summarized in Chapter III. Appendix C briefly discusses semantic issues.
II. THE FCL LANGUAGE

A. Introduction

It is generally accepted that traditional von-Neumann languages such as FORTRAN and PASCAL do not allow the full capabilities of the various data flow architectures to be realized. For this reason, a number of high-level data-flow languages have been proposed [Ackerman and Dennis 1978, Arvind, Gostelow and Plouffe 1978]. One thing that these new languages have in common is that they are value oriented. That is, one programs by defining output values in terms of input values. This definitional form of coding is also used to write programs in purely functional languages such as SASL [Turner 1976], LISP [McCarthy et al., 1962], and others [Backus 1978, Friedman and Wise 1978, Henderson 1980]. In fact, it has been shown that the concept of execution of purely functional languages leads naturally into the concept of data-flow graphs [Sleep 1980].

All of the languages cited above have their deficiencies. Even though high-level data-flow programs consist of nothing but functional applications, there are so many different ways to specify function applications that most high-level data-flow programs tend to resemble von Neumann programs, except in minor details. Backus has pointed out that there are significant improvements to be made over von Neumann-type
coding, and it seems incumbent on all language designers to make some effort to realize this goal. On the other hand, purely functional languages typically handle only a very restricted range of data types. This is particularly true for structured data types, since functional languages typically allow only one type of structure: the list. However, practical programming has found a variety of uses for other structures such as records (in the PASCAL sense) and arrays (to include those whose origin is other than 1).

One major objective of this research is the design of a language which is purely functional in nature (similar to SASL, LISP and others) and allows the use of arbitrary structured data-objects. It is further desired to introduce these objects with a minimum of new syntactic features to handle them. The language FCL (Functional Computing Language) meets these goals.

Other objectives of the FCL design are virtually unrestricted coding, the generalization of certain features found in other languages, and architectural independence. The goal of architectural independence will, at times, give way to the need to translate high-level expressions to data-flow flow graphs, although such instances are rare. It is hoped that FCL will transcend data-flow architectures in the sense that it will suggest new hardware features to be
implemented. However, this possibility will not be explored here.

It is to be remembered that FCL is to be considered a tool for research rather than an end in itself. Although the features and syntax presented here are firmly established, it is expected that significant extensions will be proposed in the future.

The remaining sections of this chapter are organized as follows: Section B presents the basic features of the language. Section C discusses data structures, Section D presents the undefined element. Section E gives a few examples, and Section F presents concluding remarks.

B. The Elements of the FCL Language

1. The basics

Backus [1978] has pointed out that the syntax of conventional programming languages could be made less cumbersome by extending the concept of expressions, and removing the concept of program statements. The typical applicative language has two classes of syntactic objects: expressions and definitions. In FCL, the concept of an expression has been extended to include the concept of a definition, thus FCL has only one syntactic object: the expression. Every FCL expression denotes an object from one of the three
semantic classes: sets, scalars, or functions. Sets are the FCL equivalent of types, and the words "set" and "type" are to be considered synonymous throughout the remainder of this dissertation. Scalars and functions serve the usual purpose.

The simplest expressions are literals and names. Literals are self-defining and denote members of the five basic sets INTEGER, REAL, BOOLEAN, CHARACTER, and STRING. Examples of the five different types of literals are 20, 3.5, TRUE, 'A', and "ABC". Names must be defined by some type of binding operation, and are formed according to the following rules:

1. A name may be of any length and may include any of the characters A,...,Z,0,...,9, and '-' (underline).
2. A name must contain at least one letter or underline character.
3. No name may begin or end with an underline.
4. Capitals and lower case letters are considered equivalent.

More complex expressions are defined by applying functions to simpler expressions. Function applications are denoted in five ways:
1. Juxtaposition of a function definition and a parenthesized list of one or more expressions.
   For example: \( F(2, 3, A) \).

2. Combining two expressions with an infix operator.
   Example: \( A + B \).

3. Placing a prefix operator before an expression.
   Example: \( \text{Not } P \).

4. Enclosing an expression or list of expressions in special parentheses. Example: \([F, G]\).

5. Implicitly in the notation. Examples will be given where appropriate (see the CURRY operation).

Juxtaposition of a function definition with its arguments is the preferred form of notation in the sense that all function applications may be written in this form. Alternative function names are provided for this purpose.

2. The FCL operators

The FCL operators from high to low precedence are:
\(\#, \circ, \text{(Unary } +, \text{Unary } -), **, (/, *, \text{DIV, MOD}), (+, -),
(<, >, =, <=, >=, \neq), \text{NOT, AND, OR, . . . } /, :, \uparrow, |, \Xi, ;,\)
where operators of equal precedence are enclosed in parens. Most of the above operators are familiar and require no further explanation. The new operators are \(\uparrow, |, \Xi, ;, /, .., \circ, \) and \(\#\). All operators are left-associative except ** and + which are right-associative.
The "\rightarrow" or function-definition operator is used to define new functions and new sets of functions. The form of this operator is \text{EXP}1\rightarrow\text{EXP}2. If \text{EXP}2 denotes a set, then \text{EXP}1\rightarrow\text{EXP}2 denotes a set of functions, otherwise it denotes a single function. For example, INTEGRER\rightarrow\text{REAL} denotes the set of all functions from INTEGRER to REAL while INTEGRER\rightarrow 3.5 denotes the constant function which returns 3.5 for every integer argument. In the expression 2\rightarrow\text{REAL} the literal "2" is taken to mean the singleton set containing 2. The expression 2\rightarrow 3 denotes the function which maps 2 to 3 and is undefined elsewhere. This example demonstrates that FCL functions may have explicitly finite domains. This is an important distinction between FCL and some other applicative languages. The implications of this fact will be discussed in section D.

The "|" or extension operator is used to extend the definition of a function. Its form is \text{F}|\text{G} where \text{F} and \text{G} are functions. The value of (\text{F}|\text{G})(x) is equal to \text{F}(x) if \text{F}(x) is defined, and equal to \text{G}(x) otherwise. For example, 2\rightarrow 3|4\rightarrow 8 denotes the function which maps 2 to 3 and 4 to 8 and is undefined elsewhere. If both \text{F} and \text{G} denote sets of functions, then \text{F}|\text{G} denotes the set of functions obtained by applying the extension operator to all pairs of functions \text{f}\in\text{F} and \text{g}\in\text{G}. If one of \text{F} and \text{G} denotes a single function and the other denotes a set of functions, the single function is treated as a singleton set. For example, 2\rightarrow\text{INTEGER}|3\rightarrow\text{INTEGER} is the
set of functions which map the set \{2, 3\} onto the set INTEGER and are undefined elsewhere. Similarly, \(2\rightarrow 5|3\rightarrow \text{INTEGER}\) is the set of functions which map 2 to 5 and 3 to an integer.

The ":" or argument-binding operator is used to define function argument names. It has four forms: \(N:\text{EXP1},\ N:\text{EXP1}\rightarrow \text{EXP2},\ (N_1,\ldots,N_K)\rightarrow \text{EXP1},\) and \((N_1,\ldots,N_K)\rightarrow \text{EXP1}\rightarrow \text{EXP2},\) where \(N,N_1,\ldots,N_K\) denote names, \(\text{EXP1}\) denotes a set, and \(\text{EXP2}\) is an arbitrary expression. For example, the expression \(x:\text{INTEGER}\rightarrow x^2\) denotes the function which maps each integer onto its square. In this example, the name "x" denotes an arbitrary element of the set INTEGER. Examples of the other three forms of this operator will be given later. In the form \(x:\text{EXP1}\rightarrow \text{EXP2}\) the scope of \(x\) is \(\text{EXP1}\) and \(\text{EXP2}\).

The "\(=\)" or sub-expression-name binding operator is used to name expressions. Its form is \(N=\text{EXP}\) where \(N\) is a name and \(\text{EXP}\) is an arbitrary expression. The scope of \(N\) is the expression \(\text{EXP}\), but this scope can be extended by the ":;\) operator (see below). An example is \(\text{SQUARE} = x:\text{INTEGER}\rightarrow x^2\). Another more complicated example is: \(\text{FACT} = 0\rightarrow 1|x:\text{INTEGER}\rightarrow x\cdot \text{FACT}(x-1),\) which denotes the factorial function.

The ":;\) or sub-expression operator, is used to define sub-expressions and extend the scope of sub-expression names. The form of this operator is \(\text{EXP1};\text{EXP2}\). The meaning of \(\text{EXP1};\text{EXP2}\) is identical to the meaning of \(\text{EXP2}\). The
usefulness of the sub-expression operator lies in the scope rules for sub-expression names. In an expression of the form \( N_1 \equiv E_1 \ldots ; \ldots ; N_K \equiv E_K \), the scope of each of the names \( N_1 \) through \( N_K \) is the entire expression. Given an expression of the form \( N \ EXP \) the scope of \( N \) is determined as follows:

1. If \( N \) is preceded by a ";" operator, then the scope of \( N \) extends to the left up to the first unmatched "(".
2. If \( N \) is the first symbol of the program or is preceded by a ";" or by a "(" then the scope of \( N \) extends to the right up to the first unmatched ")".
3. If neither 1 nor 2 above apply, then the scope of \( N \) terminates on the left with \( N \) and on the right with the first ";" operator or unmatched ")".

For example, consider the following expression which defines the function \( f(x) = 2x^3 + 3x^2 - 1 \).

\[
x: \text{INTEGER} \rightarrow \{ \text{SQ} = x^2; \ CUBE = \text{SQ} \times x; \ 2 \times \text{CUBE} + 3 \times \text{SQ} - 1 \}
\]

The scope of "SQ" and "CUBE" is defined by the parentheses. The scope of "x" is the entire expression. The following is an artificial example of mutual recursion:
Z: INTEGER → (F(x-1) + G(x-1))
G(x-1) = F(x-1) - G(x-1)
F(Z+1) * G(Z+2))

In this example the scope of "F" and "G" is defined by the outer parentheses, the scope of Z is the entire expression, and the scope of each "x" begins with its occurrence, and ends with the following ";" operator.
(This expression contains two distinct argument names, "x".)

The following is an example of a recursive function definition embedded within a larger expression. The function call POSMULTS(3) returns a function which maps every positive integer n onto 3*n. This example is also somewhat artificial.

POSMULTS ≡ x: INTEGER →
    n2 ≡ 0 + 0
    y: INTEGER → x + n2(y-1)

In this example the scope of "POSMULTS" is the entire expression, the scopes of n2, x, and y begin with their occurrences and extend to the end of the expression. It should be noted that given an expression N=EXP, N and EXP are semantically equivalent throughout the scope of N. Thus, N and EXP are interchangeable throughout the scope of N.

The "\" or restriction operator is used to restrict the size of a set, and can be used to build arbitrary
"conditionals" in conjunction with the "|" operator. The form of this operator is S\EXP where S denotes a set and EXP denotes a Boolean scalar. To be meaningful, EXP must contain at least one occurrence of an undefined name N. This name N must eventually be bound to S\EXP with a ":" operator as in the following: x:INTEGER\x>0. This example defines the positive integers. The use of this operator in defining conditionals is demonstrated by the following definition of the absolute value function:

\[ \text{ABS} = x: \text{INTEGER}\backslash x>0 \rightarrow x | x: \text{INTEGER} \rightarrow -x. \]

The ".." or range operator is essentially equivalent to the PASCAL sub-range operator. Its form is EXP1..EXP2 where EXP1 and EXP2 denote scalars of the same type.

An example is 3..7. This example is equivalent to x:INTEGER\x>3 and x<=7. This range operator is to be treated as a sugared form of the restriction operator. Because of this, a range of the form 3.7..4.8 is permissible, as the ranges of the form 5..3 and 2..2.

The "\circ" or composition operator is used to denote the composition of two functions. Its form is F\circ G where F and G denote functions. To be meaningful the intersection of the domain of F and the range of G must be non-null. However, a null intersection is treated as a legal definition of the undefined function rather than as an error.

The "#" or domain operator is a unary operator which may
be applied to a function or a set of functions all of which have the same domain. This operator has the form \#F where \( F \) is a function or a set of functions. The expression \#F denotes the domain (or common domain) of \( F \). Although this operator is potentially useful in defining functions, it is not readily implementable, and will not be discussed in depth here.

3. **Special parentheses**

Two special functions are denoted by parentheses. These are the vector function denoted by "( )" and the transpose function denoted by "[ ]". A vector is defined to be a function of the form \( 1..k \to \text{ANY} \). A vector is written as follows: \((E_1,...,E_K)\). This expression is equivalent to the expression \( 1\to E_1 | ... | K\to E_K \). This equivalence is extended to include function applications of the form \( F(E_1,...,E_K) \), thus, all FCL functions are functions of one argument. The vector function may be applied to sets to obtain cross products as in: \((\text{INTEGER, INTEGER})\). If single elements are mixed with sets in the vector specification, the single elements are treated as singleton sets as in \((2, \text{INTEGER})\). These last two examples have the respective equivalent forms: \( 1\to \text{INTEGER}|2\to \text{INTEGER} \) and \( 1+2|2\to \text{INTEGER} \).

Vector notation may be used to specify functions of several variables as in: \((x,y):(\text{INTEGER, INTEGER})\to x+y+1\). An
equivalent form of this function is: \[ V: (\text{INTEGER}, \text{INTEGER}) \rightarrow V(1) + V(2) + 1. \] The notation \((N_1, \ldots, N_K):\text{EXP}\) may be used whenever \(\text{EXP}\) denotes a set of \(K\) element vectors. The scope rules for the names \(N_1\) through \(N_K\) are identical to those for single names in expressions of the form \(x:\text{EXP}_1\) and \(x:\text{EXP}_1 \rightarrow \text{EXP}_2\). Vectors of names may be used in connection with the restriction operator as in the expression \((x, y): (\text{INTEGER}, \text{INTEGER}) \setminus x = y.\)

The transpose function may be applied to any function of the form \(S_1 \rightarrow S_2 \rightarrow S_3\) where \(S_1\), \(S_2\), and \(S_3\) are arbitrary sets. Given a function \(F\) of this form, \([(F)(x)](y) = (F(y))(x).\)

An expression of the form \([(E_1, \ldots, E_K)]\) may be written more compactly as \([E_1, \ldots, E_K]\).

As an example, consider the two definitions: \(\text{SQUARE} = x: \text{INTEGER} \rightarrow x \times x\) and \(\text{CUBE} = x: \text{INTEGER} \rightarrow x \times x \times x\). Then, \([\text{SQUARE}, \text{CUBE}]\) is a function which maps an integer \(i\) onto the vector \((i^2, i^3)\). To see how this works, recall that \([\text{SQUARE}, \text{CUBE}]\) is shorthand for \([1 \rightarrow \text{SQUARE} | 2 \rightarrow \text{CUBE}]\). The function \(1 \rightarrow \text{SQUARE} | 2 \rightarrow \text{CUBE}\) maps the integers 1 and 2 onto the functions \text{SQUARE} and \text{CUBE}, respectively. Thus, the expression \((1 \rightarrow \text{SQUARE} | 2 \rightarrow \text{CUBE})(1))(5)\) may be reduced to \(\text{SQUARE}(5)\) and subsequently, to 25. Since the transpose function reverses the order of the arguments, \([(\text{SQUARE}, \text{CUBE})(5)](1)\) may also be reduced to 25. Thus, \([\text{SQUARE}, \text{CUBE}])(5)\) is a function which maps 1 to 25 and 2 to 125 and is elsewhere undefined. In FCL
notation this function is written $1 \rightarrow 25 | 2 \rightarrow 125$, or equivalently $(25, 125)$.

4. The CURRY and UNCURRY operations

Let $F$ be the function $x : \text{INTEGER} \rightarrow y : \text{INTEGER} \rightarrow x + y + 1$. Ordinarily this function would be invoked using an expression of the form $F(3)(4)$. The implicit UNCURRY operation allows the programmer to rewrite this expression as $F(3, 4)$.

Now let $G$ be the function $(x, y) : (\text{INTEGER}, \text{INTEGER}) \rightarrow 2 \cdot x + y$. The implicit CURRY operation allows the programmer to write the following expressions: $G(3)$ and $G(,4)$. The first of these expressions denotes the function $y : \text{INTEGER} \rightarrow 2 \cdot 3 + y$ while the second denotes $x : \text{INTEGER} \rightarrow 2 \cdot x + 4$.

Both the implicit CURRY and implicit UNCURRY operations may be extended to functions of several arguments. The implicit curry and uncurry operations may be used together. Let $F$ be as defined above. Then $F(,7)$ is an allowable expression which denotes the function $x : \text{INTEGER} \rightarrow x + 7 + 1$.

5. Additional functions

Each of the functions denoted by operators, parens, and special notation may be written in prefix form. Figure 2.1 gives the alternative form for each of these functions.
<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>ALTERNATIVE FORM</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>PLUS(x,y)</td>
</tr>
<tr>
<td>-</td>
<td>MINUS(x,y)</td>
</tr>
<tr>
<td>*</td>
<td>TIMES(x,y)</td>
</tr>
<tr>
<td>/</td>
<td>QUOTIENT(x,y)</td>
</tr>
<tr>
<td>DIV</td>
<td>IQUOTIENT(x,y)</td>
</tr>
<tr>
<td>MOD</td>
<td>REMAINDER(x,y)</td>
</tr>
<tr>
<td>**</td>
<td>POWER(x,y)</td>
</tr>
<tr>
<td>UNARY+</td>
<td>IDENT(x)</td>
</tr>
<tr>
<td>UNARY-</td>
<td>NEGATE(x)</td>
</tr>
<tr>
<td>&lt;</td>
<td>LESS(x,y)</td>
</tr>
<tr>
<td>&gt;</td>
<td>GREATER(x,y)</td>
</tr>
<tr>
<td>&lt;=</td>
<td>GREATEREQ(x,y)</td>
</tr>
<tr>
<td>&gt;=</td>
<td>NEQ(x,y)</td>
</tr>
<tr>
<td>NOT</td>
<td>COMPLEMENT(x)</td>
</tr>
<tr>
<td>AND</td>
<td>BOTH(x,y)</td>
</tr>
<tr>
<td>OR</td>
<td>EITHER(x,y)</td>
</tr>
<tr>
<td>--</td>
<td>RANGE(x,y)</td>
</tr>
<tr>
<td>\</td>
<td>RS(x,y)</td>
</tr>
<tr>
<td>®</td>
<td>FN(x)(y)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>°</td>
<td>COMPOSE(x,y)</td>
</tr>
<tr>
<td>#</td>
<td>DOMAIN(x)</td>
</tr>
<tr>
<td>[ ]</td>
<td>TRANSPOSE(x)</td>
</tr>
<tr>
<td>UNCURRY</td>
<td>UNCURRY(x)</td>
</tr>
<tr>
<td>CURRY</td>
<td>FIX(f,n,v)</td>
</tr>
</tbody>
</table>

Figure 2.1. Alternative operator names
The ":", ";", "$\equiv$", and vector operators do not have alternative names since the first three are simply binding operations, and the vector operation is simply syntactic sugar.

FCL provides a number of other builtin function names. Among these are the trigonometric, exponential, and logarithmic functions, as well as the other arithmetic functions one would expect to find in a programming language.

Other less standard functions are INSERT, SUM, PROD, MAX, MIN, ALL, and ANY. INSERT is defined on all functions $F$ of the form $(S,S)\rightarrow S$ where $S$ is any set. INSERT($F$) in turn, denotes a function which is defined on all functions of the form $T\rightarrow S$ where $T$ is a set which is both finite and ordered. For example, assume the following definitions:

$F= (x,y) : (\text{INTEGER},\text{INTEGER}) \rightarrow x+y+1$ and

$G= (3,4,5,6)$

Then, INSERT($F$)($G$) is defined to be:

$F(F(F(3,4),5),6)=21$.

The functions SUM, PROD, MAX, MIN, ALL, and ANY are defined to be INSERT(PLUS), INSERT(TIMES), INSERT(HIGH), INSERT(LOW), INSERT(BOTH), and INSERT(EITHER), respectively, where HIGH is the function:

$(x,y) : (\text{SCALAR},\text{SCALAR}) \rightarrow x>y \rightarrow y>1$

$(x,y) : (\text{SCALAR},\text{SCALAR}) \rightarrow x>y \rightarrow x$ and LOW is defined analogously. In addition, these functions
are defined on functions whose domains are unordered, since the respective operations are associative and commutative.

Another useful function is the SIZE function. This function is defined on all functions whose domains are finite. SIZE(F) denotes an integer which is equal to the number of elements in the domain of F.

Two functions which are useful in practice are APPEND and DELETE. As explained in section C, FCL functions are used to simulate data-structures. APPEND and DELETE are used to simulate the analogous data-structure operations on such functions. APPEND has the form APPEND(F,S,V) where F is a function, and S and V are either scalars or functions. The expression APPEND(F,S,V) is precisely equivalent to the expression (S)+^(V)|^(F). APPEND effectively adds the value V to the data-structure (i.e., function) F with selector S.

DELETE has the form DELETE(F,S) where F is a function and S is a function or a scalar. The expression DELETE(F,S) is precisely equivalent to x:ANY\!/S+F(x). DELETE effectively removes the selector S from the data-structure (i.e., function) F. These two functions are provided for compactness, and are to be considered sugared forms of their equivalent expressions.

The last special FCL function which will be discussed here is the EXISTS function. Its form is EXISTS(F,S) where F is a function and S is a scalar or a function. EXISTS(F,S)
denotes the value TRUE if F(S) is defined and the value FALSE otherwise.

6. SETS, and SET functions

As stated above, the five basic sets are INTEGER, REAL, BOOLEAN, CHARACTER and STRING. In addition, the following pre-defined set names are provided: NUMBER (the union of INTEGER and REAL), SCALAR (the union of the five basic sets), FUNCTION (the set of all functions), TYPE (the set of all FCL sets), and ANY (the union of SCALAR and FUNCTION). The logical difficulties of the set "TYPE" are avoided in the semantics of FCL by treating FCL sets as functions (see Appendix C). The set ANY may be used to specify untyped arguments.

FCL provides a number of functions which are set valued. The functions denoted by \rightarrow, |,..., and \setminus have been discussed previously. The function SET may be used to create a finite set out of a number of objects. The expression SET(2, 3.5, "ABC") defines a set containing three objects. SET may be applied to any number of expressions of arbitrary type. The expression SET(E1,...,EK) is equivalent to the expression x:ANY\setminus{x=E1 or ... or x=EK.

Another useful set-valued function is UNION. Its form is UNION(S1,...,SK) where S1 through SK are set-valued functions. UNION(S1,...,SK) denotes the union of the sets
denoted by $S_1$ through $S_K$. For example the set $\text{NUMBER}$ may also be written as $\text{UNION}(\text{INTEGER}, \text{REAL})$.

It is possible to define set-valued functions, as in the following: $x: \text{INTEGER} \rightarrow l..x: \text{INTEGER}$. The use of set-valued functions has one special convention. To illustrate this consider the function: $\text{VECTOR} = (N,S):(\text{INTEGER}, \text{SET}) \rightarrow l..N \rightarrow S$. If the name $\text{VECTOR}$ is used as the type of a function argument it denotes the union of all sets in the range of the function $\text{VECTOR}$ rather than the function itself. This convention is given to make set valued functions the logical equivalent of parameterized types.

Although FCL has very powerful features for defining sets, the translation algorithm given in Chapter IV does not support all of them. It is necessary to ignore certain typing information at run time in order to keep the object program from becoming nothing but a mass of type checks. It is possible to make more complete use of typing information at compile time, but such use is not explored in this dissertation.

7. Syntactic sugar

In order to allow the programmer to use familiar programming constructs, two sugared constructs IF and WHILE are introduced. All FCL sugared constructs have the same format. The range of the sugared construct is defined by a
special set of parens consisting of the construct name and the construct name preceded by END. For the IF and WHILE constructs the parentheses are IF; ENDIF and WHILE, ENDWHILE. Inside the special parentheses there is a list of named sub-expressions separated by semicolons. Certain of these names are arbitrary, and others have special meanings for the particular construct. There may be a un-named sub-expression which always has a special meaning for the construct. The order of the sub-expressions within the sugared construct is irrelevant. Figure 2.2 gives the format of the IF construct.

IF <un-named EXP>;
    THEN = <arbitrary exp>;
    ELSE = <arbitrary exp>
ENDIF

Figure 2.2. The IF construct

As a concession to popular convention, the IF construct may be written as in Figure 2.3. In Figure 2.3, however, the order of the expressions is fixed.
IF <un-named exp>
    THEN <arbitrary exp>
    ELSE <arbitrary exp>
ENDIF

Figure 2.3. The alternative form of the IF constructs

An example of the IF construct is given in Figure 2.4.

IABS ≡ x : INTEGER→IF x>0;
    THEN ≡ x;
    ELSE ≡ -x
ENDIF

Figure 2.4. An example of the IF construct

Before an FCL program is completely parsed, the IF
construct will be desugared according to the rule given in
Figure 2.5.
Before desugaring:

IF <COND EXP>;
    THEN = <then EXP>;
    ELSE = <else EXP>

After desugaring:

(True + <then EXP> | False + <else EXP>) (<cond EXP>)

Figure 2.5. Desugaring of the IF construct

The WHILE construct is more complicated than the IF construct. Its format is given in Figure 2.6.

WHILE <COND EXP>;
    INIT = (<Named Exp List>);
    RESULT = <result Exp>;
    Named exp list

ENDWHILE

Figure 2.6. The WHILE construct.

An example of the WHILE construct is given in Figure 2.7.
FIB = x:INTEGER+WHILE  
  CNT<x;
  INIT (CNT≡1;
          PREV≡0;
          CURR≡1);
  RESULT ≡ CURR;
  CNT ≡ CNT+1;
  PREV ≡ CURR;
  CURR ≡ CURR+PREV
ENDWHILE

Figure 2.7. An example of the WHILE construct

The desugaring of the WHILE construct is given in Figure 2.8.

Before Desugaring:
X ≡ WHILE C
  UNIT ≡ (N_1≡E_1;...;N_i≡E_i;  
          M_j≡D_j;...;M_j≡D_j);
  RESULT R;
  L_1 ≡ B_1;...;L_k≡B_k;
  M_1 ≡ A_1;...;M_j≡A_j
ENDWHILE
After Desugaring:

\[ X \equiv (N_1 \equiv E_1; \ldots; N_i \equiv E_i; \]

\[ F \equiv \text{PARM:ANY}+ \]

\[ (L_1 \equiv B'_1; \ldots; L_k \equiv B'_k; \]

\[ \text{IF } C' \]

\[ \text{THEN } R' \]

\[ \text{ELSE} \]

\[ F("M_1" \rightarrow A'_1 | \ldots | "M_j" \rightarrow A'_j) \]

\[ \text{ENDIF}; \]

\[ F("M_1" \rightarrow D_1 \ldots "M_j" \rightarrow D_j) \]

Figure 2.8. Desugaring of the WHILE construct

In Figure 2.8 the names \( N_1 \) through \( N_i \) appear only in the INIT expression, the names \( M_1 \) through \( M_j \) appear both in the INIT expression and following the RESULT expression, and the names \( L_1 \) through \( L_k \) appear only after the RESULT expression. The expressions \( A'_1 \) through \( A'_j \), \( B'_1 \) through \( B'_k \), \( C' \) and \( R' \) are the expressions \( A_1 \) through \( A_j \), \( B_1 \) through \( B_k \), \( C_1 \) and \( R \) with all occurrences of \( M_1 \) through \( M_j \) replaced by \((\text{PARM}("M_1"))\) through \((\text{PARM}("M_j"))\), respectively. (Actually, only free occurrences of \( M_1 \) through \( M_j \) are replaced. See Chapter IV for a discussion of free variables.) The names \( F \) and \( \text{PARM} \) are new names which are generated in such a way that they do not occur in the original WHILE construct.

Figure 2.9 gives the desugared form of the construct given in Figure 2.7.
FIB ≡ \( (x: \text{INTEGER}) \rightarrow \)
\( (F \equiv \text{PARM}: \text{ANY}) \rightarrow \)
  IF (PARM("CNT") < x)
  THEN (PARM("CURR")
  ELSE F("CNT" \rightarrow (PARM("CNT") + 1 |
  "PREV" \rightarrow (PARM("CURR") |
  "CURR" \rightarrow (PARM("PREV") + (PARM("CURR")))
  ENDIF;
  F("CNT" \rightarrow 1 | "PREV" \rightarrow 0 | "CURR" \rightarrow 1))

Figure 2.9. Desugaring of the FIB function

The IF and WHILE constructs are provided for compati-

bility with existing data-flow languages [Ackerman and
Dennis 1978, Arvind, Gostelow and Plouffe 1978]. The WHILE
construct is provided with some hesitation because it is
felt that ordinary recursive notation is at least as compact
and as efficient as the WHILE construct. The only real
justification for providing such a construct is that it
allows the programmer to use a familiar construct for
repetitive programming.
C. Data Structures

One of the objectives of FCL is to provide a full range of structured data types, yet the previous section contains no discussion of physical memory structures. It would have been possible to include physical memory structures in the definition of FCL, but it was felt that this would complicate the language unnecessarily. Instead of treating a data structure as a physical object, the use and effect of data structures within a program was examined. In most programming languages, the elements of a data structure must be processed individually. Even operations which operate on the whole data-structure are typically defined in terms of their pointwise effect on the elements of the structure. There appear to be two operations which all data structures have in common. These are selection of an element, and creation of a new structure (this includes insertion of a new structure into a structure). These two operations are virtually identical to applying a function to an argument, and defining a new function. The main difference between functions and data structures (implementation details aside) appears to be that the "domain" of a data structure is explicitly finite. The equivalence of data-structures and functions is even more apparent when one considers physical memory structures which contain suspensions [Dennis and Weng
It was decided that FCL would have no data structures other than functions. Functions with explicitly finite domains will replace the structured data objects found in other languages. The purpose of this section is to show that this concept of data-structures is very powerful, and properly includes the concepts found in other languages.

First consider the list structure which is used by a number of functional languages. Figure 2.10 gives the definitions of the list structure, as well as the CONS, FIRST, and REST functions.

\begin{verbatim}
LIST ≡ UNION("NIL", (ANY,ANY));
CONS ≡ (x,y):(ANY, LIST)→(x,y);
FIRST ≡ x:LIST→x(1);
REST ≡ y:LIST→y(2)
\end{verbatim}

Figure 2.10. Definition of the list structure

Ordinary n-dimensional arrays are also quite easily defined. For example, the expression \( x:(1..5,1..5)→ x(1)+2\times x(2) \) defines a 5x5 matrix. The following function maps a positive integer \( I \) onto the \( IxI \) identity matrix:

\[
\text{MIDENT} \equiv I:INTEGER \backslash I>1→
\]

\[
(x,y):(1..I,1..I)→(\text{TRUE}→1|\text{FALSE}→0)(x=y).
\]

It is also very easy to define arrays of peculiar shapes.
The following expression defines a triangular array:

\[(x,y):(1..5,1..5) \rightarrow y = x + y + 1.\]

The following array is "L" shaped:

\[(x,y):(1..5,1..5) \rightarrow x = 5 \text{ or } y = 1 + 2x + y\]

It is equally easy to define sparse matrices as in the following:

\[(1,7) \rightarrow 3 | (14,72) \rightarrow 6 | (21,18) \rightarrow 5 | (1..100,1..100) \rightarrow 0.\]

It is possible to define arrays of nonuniform dimension as in: \((1,2) \rightarrow 7 | 2 \rightarrow 8 | (1,2,8) \rightarrow 17.\) (Implementation of such structures may be quite difficult.)

PASCAL-type records may be implemented as functions defined on a set of strings. An example of such a structure is:

"NAME" = "JOE JONES" |
"OCCUPATION" = "MACHINIST" |
"PAY-RATE" = 24.50

In order to make this type of structure more compatible with PASCAL-like languages, the following syntactic sugar is provided:

\(<NAME> \rightarrow <EXP>\) is desugared to "<NAME>" = <EXP>

and

\(<EXP> \cdot <NAME>\) is desugared to <EXP>("<NAME>").

Thus, the above expression may be rewritten as
NAME → "JOE JONES"
OCCUPATION → "MACHINIST"
PAY-RATE → 24.50

The WITH statement of PASCAL is not supported in FCL. This makes the notation for handling records somewhat more cumbersome, but since the component-names of an FCL record are unpredictable at compile-time, including such a feature would be extremely difficult.

FCL notation allows one to define structures which are neither records nor arrays as the following example shows: 1→3|2→7|COST → 5.60|3.8→"ABC".

Of course, any component of a data structure may itself be a data structure. (This is equivalent to the concept of higher-order functions.) The example of Figure 2.11 illustrates this. Structures may be nested to any depth.

The above discussion shows that treating data structures as functions gives the programmer immense power in defining structures. Another benefit is that the apply-to-all operation [Backus 1978] becomes superfluous. To see this, consider the functions:

VEC ≜ 1→3|2→5|3→9;

SQ ≜ x:INTEGER→x*x.

The programmer may apply the function SQ to every element of VEC simply by taking the composition of SQ and VEC: SQ o VEC.
EMPLOYEES ≡

1→(NAME → "JOE JONES" |
    TITLE → "BOTTLE WASHER" |
    CHILDREN 1→(NAME→"TOMMY" |
      AGE=10) |
    2→(NAME="ALICIA" |
      AGE=12)) |
SALARY = 23.50) |
2→(NAME → "BILL SMITH" |
    TITLE → "SWEeper UPPER" |
    CHILDREN = (1→(NAME=MARY |
      AGE=3) |
      2 (NAME=JOAN |
      AGE=8) |
      3 (NAME=JERRY |
      AGE=10)) |
SALARY = 15.56)

Figure 2.11. Data-structure example
To see this more clearly, rewrite SQ\circ VEC as SQ\circ (1\rightarrow 3|2\rightarrow 5|3\rightarrow 9). Applying the composition rule to this expression yields 
(1\rightarrow SQ(3)|2\rightarrow SQ(5)|3\rightarrow SQ(9)) which is equivalent to (9, 25, 81).
The composition operation is more powerful than ordinary apply-to-all operations because composition may be used with any function (i.e., data structure) not just with arrays.

D. The UNDEFINED Element

The object UNDEFINED is an implicit member of every set. This object is a function which returns itself when applied to any argument. When a function is applied to any element which is not in its domain, the resultant expression denotes the UNDEFINED object. UNDEFINED is a legal expression and may be used in more complicated expressions. With the exception of "*" and "|", the application of any built-in function to UNDEFINED denotes UNDEFINED. UNDEFINED is the identity element of the "|" operator. For any expression EXP, EXP\rightarrow UNDEFINED denotes UNDEFINED. UNDEFINED\rightarrow EXP denotes a function which is defined on the undefined element.

The undefined element of FCL is more than just an abstraction. It is intended that any implementation of FCL contain a physical representation for the undefined element. This representation will typically be one or more error codes. It is quite likely that a real implementation will require a whole set of error codes such as those provided by
the VAL language [Ackerman and Dennis 1978]. These include such things as OVERFLOW's, UNDERFLOW's, etc. In such a case, these error codes must be formally incorporated into FCL as distinct objects. Such implementations will still require a physical representation for the undefined element.

This atypical view of the undefined element is motivated by the fact that FCL functions may have explicitly finite domains. Consider the expression $(2 \rightarrow 6 | 3 \rightarrow 9)(5)$. It is immediately obvious that this expression denotes the undefined object, and it is reasonable to expect that an implementation of FCL could determine this fact. It is also reasonable to expect that an implementation could determine that EXISTS $(2 \rightarrow 6 | 3 \rightarrow 9, 5)$ denotes the value FALSE. The existence of multiple types in FCL also motivates this view, since it is reasonable to expect that an implementation would be able to determine that SIN("JONES") is not defined.

In FCL, nontermination and undefinedness are two distinct concepts as the following examples show: The expression $(F \equiv 0 \rightarrow 1 | :x :\text{INTEGER} \rightarrow x \cdot F(x-1))(-1)$ is undefined since the least-fixed-point solution of the equation defining $F$ is undefined for $-1$ [Burge 1975, Stoy 1977]. Since the translation algorithm of Chapter IV will produce a nonterminating recursion for this expression, it is also nonterminating. The expression $(2, 3)(3)$ is undefined, since the function
is defined only for the arguments 1 and 2. Nevertheless, the translation algorithm of Chapter IV will generate a graph which terminates by producing an error code. Therefore, this expression terminates, but is undefined. The expression 
\[(F \equiv 0 \Rightarrow 1 \mid x: \text{INTEGER} \Rightarrow F(x + 1) + 1 \mid 2 \Rightarrow 3)(2)\] is defined. The least-fixed-point of the expression defining F is undefined for the argument 2, thus, the semantics of the "\(|" operator dictate that the value of the entire expression is defined by the expression \((2 \Rightarrow 3)(2)\) which denotes the value 3. Nevertheless, the translation algorithm of Chapter IV will produce a nonterminating recursion for this expression. Thus, the expression is defined, but nonterminating.

Note that in the above discussion, termination is always defined in terms of a particular translation algorithm (that of Chapter IV). This illustrates the fact that "undefined-ness" is a property of the semantics of the language, while termination is a distinct property of the technique used to evaluate expressions. The fact that these examples could be constructed suggests that there exist evaluation techniques for which all three of the expressions given above terminate. (Such evaluation techniques are probably too complicated to be of practical value.) Since an expression's evaluation may terminate and produce the undefined element, expressions of the form UNDEFINED->EXP make sense. In fact,
the EXISTS function may be written as:

\[(f,x): (\text{FUNCTION, ANY}) \rightarrow (\text{UNDEFINED, FALSE} | \text{ANY, TRUE})(f(x)).\]

Certain other operations, such as tests for equality, are computationally feasible if applied to certain functions which have explicitly finite domains. It is important to note, however, that the evaluation of a function of the form \((1 \rightarrow 3 | 2 \rightarrow 7) \rightarrow 6 | x: \text{FUNCTION} x(2)\) is not possible in general since it is undecidable in general whether a function's domain is exactly the set \(\{1, 2\}\). Implementation of such functions may be facilitated by considering "weak" forms of various FCL operators which treat the declared domain of a function as its actual domain. (The fact that the declared domain may not equal the actual domain is what makes the above expression computationally infeasible.) In fact, an optional part of the translation algorithm of Chapter IV will substitute the weak form of the "|" operator for the strong form in some cases.

E. Examples of FCL Programs

This section gives some simple examples of FCL programs. More complicated examples are given in Appendix B. In these examples the following set-valued functions are considered to be previously defined:
VECTOR ≡ (S,N): (TYPE,INTEGER)→1..N→S

and

MATRIX ≡ (I,J):(INTEGER,INTEGER)→

(1..I,1..J)→NUMBER.

The following example defines the matrix multiplication operation:

MAT-MPY ≡ (A,B):(MATRIX,MATRIX)→

(I,J):(INTEGER,INTEGER)→

SUM(TIMES[A(I),B(J)])

This example illustrates the use of implicit CURRY, transposition and composition. Implicit CURRY is used to obtain the proper row and column of A and B. The result of these operations is a pair of vectors. The transpose operation creates a vector of pairs of numbers. The composition operation applies the multiplication operation to each of these pairs to obtain a vector of numbers. The sum operator "adds up" the vector of numbers to obtain the required element of the product matrix.

The next example demonstrates the use of the APPEND operation, the sub-expression operator, and the sugared IF construct. This example is the familiar (unoptimized) bubble sort algorithm.
BS = V:VECTOR(NUMBER) →

(BS2 = (N1,V1):(INTEGER,VECTOR(NUMBER)) →

IF N1>=SIZE(V1)

THEN V1

ELSE BS2(Nl+1,FLOAT(1,V1))

ENDIF;

FLOAT = (N2,V2):(INTEGER,VECTOR(NUMBER)) →

IF N2>=SIZE(V2)

THEN V2

ELSE FLOAT(N2+1,

IF V2(N2)<V2(N2+1)

THEN V2

ELSE

APPEND(APPEND(V2,N2,V2(N2+1)),

N2+1,V2(N2))

ENDIF)

ENDIF;

BS2(1,V))

Note the use of an IF construct as a parameter to the function FLOAT. As stated above, more extensive examples of FCL programs may be found in Appendix B. These examples are accompanied by equivalent algorithms written in five other languages which can be used for comparison purposes. Chapter III contains a quantitative comparison of these algorithms.
The FCL grammar of Appendix A, and the semantic issues discussed in Appendix C may also serve to make the structure of FCL more understandable.

F. Concluding Remarks

The examples given in this chapter and in Appendix B show that FCL is powerful enough to describe solutions to a wide class of problems. There are, however, a number of areas which require further research.

It may eventually become necessary to give the programmer some means to specify the physical realization of certain functions. A great deal of research is required in order to determine whether such features are necessary, and what form such features should take. Some of the choices which the programmer may choose to make are between a procedure and a physical memory structure, and between a completely evaluated memory structure and a memory structure containing unevaluated suspensions.

Another area which requires further research is the introduction of high-level features to take advantage of certain architecture-specific low-level features. An example of such a low-level feature is the data-flow stream [Arvind, Gostelow and Plouffe 1978, Dennis and Weng 1979]. The streams implemented by the Arvind, Gostelow and Plouffe
architecture are substantially different than those implemented by the Dennis and Weng architecture. These two architectures probably will require different high-level implementations of streams.

Other areas requiring more research are features to support modular data types and the feasibility of supporting data-processing applications.

Although FCL requires a great deal more research before it can be considered a serious alternative to existing "production" languages, its features are comparable to those of existing applicative languages, and it should form a solid basis for continuing research.
III. COMPARISON WITH OTHER LANGUAGES

Since, as yet, there is no working FCL compiler, it has not been possible to gain the large amount of experience necessary to evaluate the quality of FCL as a practical programming language. Nevertheless, it seems appropriate to do some comparative study to determine the strengths and weaknesses of FCL with respect to other high-level languages. One method of doing such a study would be to compare the individual features of FCL to similar features (or the lack thereof) in other languages. This, however, would not necessarily indicate how effective the unique features of FCL are in writing real programs.

Another method of comparison would be to attempt to rate FCL along with other languages on the accepted criteria of readability, maintainability, and so forth. However, without a substantial amount of practical experience with FCL, it is difficult to make any believable arguments with regard to these criteria.

In spite of these difficulties, it was desired to compare FCL code to that of certain other applicative languages. It was also desired to make the comparison quantitative, and as objective as possible. Programming time, as defined by software science [Halstead 1977], was selected as a quantitative measure which is fairly easy to compute. It is
recognized that Halstead's work is controversial, but since generally accepted quantitative criteria for comparing programming languages do not exist, any criterion selected would be subject to some controversy.

In order to apply Halstead's measures, ten FCL programs were written. These programs are matrix multiplication (MM), Euclidean distance between vectors (DIST), factorial (FACT), fibonacci sequence (FIB), binary tree reversal (REV), fourth-order Runge-Kutta (RK), binary search (BIN), Gaussian elimination (GAUSS), fast fourier transform (FFT), and the eight queens problem (Q8). These ten programs were also written in Backus FP systems [Backus 1978], VAL [Ackerman and Dennis 1978], ID [Arvind, Gostelow and Plouffe 1978], and the Friedman and Wise applicative language (FWAL) [Friedman and Wise 1978]. In order to compare FCL to a von Neumann language, these programs were also written in PASCAL [Wirth and Jensen 1974].

Certain nonexistent features were assumed to be present in the languages studied, in order to make the programs more equivalent. These features include the ability to define recursive functions in all languages, and inherent support for general vectors and arrays, complex numbers, and complex arithmetic. In all cases, it was assumed that all requisite type definitions had been made outside the program in
question. In the languages where it makes a difference (ID, VAL, and PASCAL) the algorithms were written as functions rather than as stand-alone programs. All of these conventions were used to insure that the respective programs were as equivalent as possible.

The following Halstead measures were calculated for each of the sixty programs:

\[ n_1 = \text{count of distinct operators} \]
\[ n_2 = \text{count of distinct operands} \]
\[ n = n_1 + n_2 \] (vocabulary)

\[ N_1 = \text{total number of operators} \]
\[ N_2 = \text{total number of operands} \]
\[ N = N_1 + N_2 \] (length)

\[ V = \log_2(n)*N \] (volume)

The Halstead formulas for calculating programming effort, \( E \), and programming time, \( T \), are: \( E = V^2/V^* \) and \( T = E/18 \). \( V^* \) is the minimum volume of an algorithm over all possible languages, and is constant for any particular algorithm. The constant 18 was derived empirically. Now, let \( P_1 \) and \( P_2 \) be two programs which implement the same algorithm, and let the respective volumes and programming times be \( V_1 \), \( V_2 \), \( T_1 \) and \( T_2 \). The above formulas imply that the ratio of programming times \( T_1/T_2 \) is equal to \( V_1^2/V_2^2 \), the ratio of the
squares of the volumes. Figure 3.1 gives the value of \( V_{FCL}^2/V_{other}^2 \) for all programs studied. The interested reader will find the programs and the details of the calculations in Appendix B.

<table>
<thead>
<tr>
<th>Program</th>
<th>FP</th>
<th>VAL</th>
<th>ID</th>
<th>PASCAL</th>
<th>FWAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM</td>
<td>.9411</td>
<td>.1480</td>
<td>.1505</td>
<td>.1458</td>
<td>.3715</td>
</tr>
<tr>
<td>DIST</td>
<td>.8846</td>
<td>.3227</td>
<td>.3492</td>
<td>.1727</td>
<td>.7034</td>
</tr>
<tr>
<td>FACT</td>
<td>.4148</td>
<td>.5879</td>
<td>.8006</td>
<td>.3659</td>
<td>.6362</td>
</tr>
<tr>
<td>FIB</td>
<td>.3369</td>
<td>.6292</td>
<td>.7972</td>
<td>.4369</td>
<td>.5102</td>
</tr>
<tr>
<td>REV</td>
<td>2.7100</td>
<td>.6078</td>
<td>1.0888</td>
<td>.1424</td>
<td>1.8916</td>
</tr>
<tr>
<td>RK</td>
<td>.6470</td>
<td>1.0291</td>
<td>1.5703</td>
<td>1.0598</td>
<td>.5428</td>
</tr>
<tr>
<td>BIN</td>
<td>.3497</td>
<td>.8195</td>
<td>1.1014</td>
<td>.6876</td>
<td>.3319</td>
</tr>
<tr>
<td>GAUSS</td>
<td>1.8817</td>
<td>.3357</td>
<td>.5673</td>
<td>.6205</td>
<td>.8653</td>
</tr>
<tr>
<td>FFT</td>
<td>.3219</td>
<td>.7927</td>
<td>.6874</td>
<td>.6920</td>
<td>.2159</td>
</tr>
<tr>
<td>Q8</td>
<td>.4841</td>
<td>.7001</td>
<td>.7859</td>
<td>.5148</td>
<td>.7454</td>
</tr>
</tbody>
</table>

Figure 3.1. \( T_{FCL}/T_{other} \) for all programs studied

It is not the purpose of this chapter to defend or dispute Halstead's work. For this reason, no order of merit will be given for the six languages studied. However, the results summarized in Figure 3.1 suggest that reasonably complicated programs are no more difficult to write in FCL than in any of the other languages studied. In this respect, FCL seems competitive with other applicative languages.
IV. THE TRANSLATION ALGORITHM

A. Introduction

The purpose of this chapter is to introduce a translation algorithm which will translate most FCL programs into a form suitable for generation of data-flow graphs. In order to avoid restricting the algorithm to a specific architecture, intermediate code will be produced, rather than the graph itself. The production of data-flow graphs from the intermediate code is straightforward.

The following restrictions are imposed on the source code:

1. Sets may not be processed at run-time. This implies, among other things, that programs may not be set-valued, that no program-argument may be set-valued, and that all set-valued sub-expressions must be evaluatable at compile time.

2. The # operator is not supported.

3. The functions SUM, PROD, SIZE, MAX, MIN, ANY, ALL and INSERT(f) may be applied only to vectors.

4. When two functions are combined using the extension operator ("|"), both functions must require the same number of arguments.

5. Function applications must be consistent with the form of the function definition.
6. The restriction operator ("\") may be used only in
the following form: \[ x : E_1 \backslash E_2 \rightarrow E_3 \text{ or } (x_1, \ldots, x_n) : E_1 \backslash E_2 \rightarrow E_3. \]

7. Within the above restrictions any argument specifi-
cation is permissible. However, each nonconstant
function argument will be treated as belonging to
one of the following classes: INTEGER, REAL, BOOLEAN,
CHARACTER, STRING, SCALAR, FUNCTION, and ANY.
Usually, the smallest class which includes the
original specification will be chosen.

Restriction 1 is imposed in order to avoid simulating
set operations at run-time. Restrictions 2 and 3 are imposed
in order to simplify the implementation of functions. Functions
will be implemented as data-flow procedures or as structured
data objects. It is not possible to extract domain informa-
tion from these objects. Restrictions 2 and 3 could be re-
moved by using run-time descriptors, but it is desired to
avoid this complication. Restrictions 4 and 5 are required
to facilitate the translation of function definitions. The
number of arguments required by a function is determined as
follows:

a. A function of the form \[ x : E_1 \rightarrow E_2 \] requires one argument.

b. A function of the form \[ (x_1, \ldots, x_n) : E_1 \rightarrow E_2 \] requires \( n \)
arguments.
c. A function of the form \((E_1, \ldots, E_n) \rightarrow E_{n+1}\) requires \(n\) arguments.

d. All other functions require one argument.

Restriction 4 forbids expressions similar to the following:

\((x, y) : (REAL, REAL) \mid x > y \rightarrow -x - y \mid Z : (REAL, REAL) \rightarrow Z(2) - Z(1)\). Restriction 5 forbids expressions similar to

\[ F \equiv (x, y) : (REAL, REAL) \rightarrow x + y; \]

\[ A \equiv (1 \mapsto 3 | 2 \mapsto 4); \]

\[ F(A). \]

Restriction 6 is imposed to simplify the abstraction of names. It forbids expressions of the form \((x : INTEGER \mid x > 0, y : INTEGER \mid y > 0)\). This expression has the equivalent legal form:

\[(x, y) : (INTEGER, INTEGER) \mid x > 0 \text{ and } y > 0. \]

Restriction 7 is imposed in order to avoid excessive run-time type checks. This restriction will cause the expression \(x : (REAL, REAL) \rightarrow x(1) + x(2) | y : (REAL, REAL, REAL) \rightarrow y(1) - y(2)\) to be translated incorrectly since both \(x\) and \(y\) will be treated as belonging to the same class: FUNCTION.

The remainder of this section is organized into six sections. Section B reviews existing work; section C describes the structure of the intermediate code, and discusses parsing requirements; section D introduces certain preliminary notions which are required for understanding the algorithm; section E presents the details of the translation process; section F
demonstrates how data-flow graphs are generated from the intermediate code. Section G discusses possible extensions and presents conclusions.

B. Review of Existing Work

One technique for compiling applicative languages which is currently receiving a great deal of attention is translation to pure combinatory code [Burge 1975, Curry and Feys 1958, Schoenfinkel 1924, Turner 1979a,b]. This approach has been used to create an efficient lazy evaluator for at least one applicative language [Turner 1979a,b], and has been shown to be highly compatible with the concept of data-driven evaluation [Sleep 1980]. The combinator approach removes the variables from expressions and introduces one or more occurrences of the functions I, K, S, B, and C. The definitions of these functions are given in Figure 4.1.

\begin{align*}
I & \equiv x: \text{ANY} \rightarrow x \\
K & \equiv x: \text{ANY} \rightarrow y: \text{ANY} \rightarrow x \\
S & \equiv f: \text{ANY} \rightarrow g: \text{ANY} \rightarrow x: \text{ANY} \rightarrow f(x)(g(x)) \\
B & \equiv f: \text{ANY} \rightarrow g: \text{ANY} \rightarrow x: \text{ANY} \rightarrow f(g(x)) \\
C & \equiv f: \text{ANY} \rightarrow x: \text{ANY} \rightarrow y: \text{ANY} \rightarrow f(y)(x)
\end{align*}

Figure 4.1. Definitions of I, K, S, B, and C
Note that the form of the definitions given in Figure 4.1 implies that these functions are curried. The combinator approach requires all functions (such as addition) to be curried, not just the combinators themselves. To gain an understanding of the combinator approach, consider the expression \((a+b)+(c+d)\). This expression has the obvious data-flow graph given in Figure 4.2.

Figure 4.2. Data-flow graph for \((a+b)+(c+d)\)

The combinator approach would first rewrite this expression as \(\text{plus} \ (\text{plus} \ a \ b) \ (\text{plus} \ c \ d)\) where \(\text{plus}\) is the curried form of the addition function, and juxtaposition implies left associative function application. Removal of the variables from this expression would produce: \(C(BC(B(BB)(B(BB)(B(B \text{plus}) \ plus))))\)). Although this expression could be translated into a data-flow graph as suggested by Sleep [Sleep 1980], the resultant graph would be bulky and would bear little resemblance to the graph of Figure 4.2. A method for reducing the bulk of pure combinatory expressions is given in [Turner
Applying this technique to the expression 
(a+b)+(c+d) gives C'(C'(B'(B' plus)))plus plus. The 
definitions of C' and B' are given in Figure 4.3.

\[
C' = k : \text{ANY} \to f : \text{ANY} \to x : \text{ANY} \to y : \text{ANY} \to k(f(y))(x)
\]

\[
B' = k : \text{ANY} \to f : \text{ANY} \to g : \text{ANY} \to x : \text{ANY} \to k(f)(g(x))
\]

Figure 4.3. Definitions of C' and B'

Although the new expression is more compact than the previous one, a strict translation to data-flow graphs would require the evaluation of four function applications in addition to the three plus operators. The algorithm presented in this chapter will solve this problem by "executing" the combinators at compile time to produce a data-flow graph similar to that of Figure 4.2. Even so, the combinatoric code given above is not sufficient for this purpose. First, it is not obvious how many arguments must be supplied to the expression C'(C'(B'(B' plus)))plus plus in order to evaluate it. Second, it is not obvious how the interconnections between the operators are to be made. Third, this expression represents a curried function and requires its arguments to be supplied serially.

A different combinator approach which does not have these difficulties is presented in [Abdali 1976]. This
approach uses combinators of the form $B_n^m I_n^m$ and $K_n$. The
definition of these combinators is given in Figure 4.4.

$$B_n^m f \cdot g_1 \ldots g_m x_1 \ldots x_n =$$
\[ f(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n)) \]

$$I_n^m x_1 \ldots x_n = x_m$$

$$K_n y_1 \ldots y_n = x$$

Figure 4.4. Definitions of $B_n^m$, $I_n^m$, and $K_n$.

The expression $(a+b)+(c+d)$ can be written as

$$B_4^2 \text{ plus } (B_4^2 \text{ plus } I_4^1 I_4^2)(B_4^2 \text{ plus } I_4^3 I_4^4)$$

using these combinators. (The algorithm presented in
[Abdali 1976] has been modified slightly to produce this
expression.) This expression can be used to generate code
using a stack algorithm similar to that used to evaluate
prefix arithmetic expressions. An expression of the form

$$B_n^m f E_1 \ldots E_m$$

means a node of type $f$ with $m$ input arcs is
to be generated. The value of the $i$th input arc will be
supplied by $E_i$. This expression requires $n$ arguments for
evaluation.

An expression of the form $I_n^m$ means that the $m$th argument's
arc is to be "hooked in" at this point. Figure 4.5 demon-
strates how the above expression is translated to a data-
flow graph.
Figure 4.5. Code generation for \((a+b)+(c+d)\)
Note that Figure 4.5d is identical to Figure 4.2. Because Abdali's combinators allow efficient data-flow graphs to be generated in a straightforward manner, they will be incorporated into the algorithm given in section E. However, none of the abstraction techniques discussed so far are powerful enough to translate FCL into combinatory code, as will be shown in section D.

C. Parsing FCL

FCL programs will be parsed using the grammar given in Appendix A. In addition, the parser will perform any necessary desugaring of IF and WHILE expressions. The result of parsing an FCL program is a structure called an op-list which is a linear form of the program's parse tree. An op-list has the form \((\text{OP} \ E_1 \ldots E_k)\) where \(\text{OP}\) is an operator and \(E_i\) is a name, a literal, or another op-list. The allowable operators include all of the builtin function names. For example, the expression \(a+b+c+d\) would be translated to the following op-list: \((+ (+a \ b) \ c) \ d\). Function calls such as \(F(x)\) (where \(F\) is not a builtin function name) produce op-lists of the form \((\text{APPLY} \ f \ x)\). The expressions \(E_1 \rightarrow E_2\), \(E_1 \mid E_2\), \(N \equiv E_2\), \(x : E_2\), \(E_1 \setminus E_2\), and \(E_1 ; E_2\) would produce op-lists of the following respective forms: \((\text{FN}^1 \ E_1 \ E_2)\), \((\text{EX}^1 E_1 E_2)\), \((\text{NM} \ N \ E_2)\), \((P \times E_2)\), \((R S \ E_1 E_2)\), and \((S C^1 E_1 E_2)\).

The parser is also expected to perform certain simple
transformations on the source code. Certain of these transfor­
mations involve the generation of new names. All parts of the compiler are assumed to be capable of generating an un­
limited number of unique names.

Function calls such as \( F(x_1, \ldots, x_n) \) are to be translated
as \((\text{APPLY } F \ x_1, \ldots, x_n)\) (or possibly \((F_1, \ldots, x_n)\) if \( F \) is a built­
in function). Expressions of the form \( E_1; \ldots; E_k \) are to be
translated as \((\text{SC}^{k-1} E_1 \ldots E_k)\) rather than as a nested series
of \( \text{SC}^1 \) op-lists. Similarly expressions of the form \( E_1| \ldots | E_k \)
are to be translated as \((\text{EX}^{k-1} E_1 \ldots E_k)\). A function of \( k \)
arguments is to be translated as \((P \ x_1 E_1) \ldots (P \ x_k E_k E_{k+1})\). The type of each argument of a multi­
argument function is determined from the form of the argu­
ment declaration. If the arguments are unnamed as in
\((2,3)+5\) then the compiler will generate a new unique name for
each argument. Thus the expression \((2,3)+5\) will, effectively,
be transformed to \((x,y):(2,3)+5\). Once all required names have
been generated, the parser builds an op-list of the form
\((P \ x \ E)\) where \( x \) is the argument name, and \( E \) is built according
to the following rules.

1. If no restriction operator ("\") is associated
   with \( x \), and \( x \) is a variable argument of class \( C \neq \)
   ANY, then \( E = (C \ x) \). If \( x \) is of class ANY, then
   \( E = \text{TRUE} \) (boolean literal).
2. If \( x \) is a constant argument (as in \((x,y) : (2,3) \mapsto 5\)) of value \( a \), and no restriction operator is associated with \( x \), then \( E = (\text{EQ} \ a \ x) \).

3. A restriction operator is associated with an argument \( x \) in two ways: by an expression \( x : E_1 \setminus E_2 \) or by an expression \((N_1, \ldots, N_k, x) : E_1 \setminus E_2\). In either case \( E_2 \) is parsed normally into an op-list \( E' \). If \( x \) is variable and of class \( C \neq \text{ANY} \) then \( E = (\text{RS}(C \ x) E') \) if \( x \) is of class \( \text{ANY} \) then \( E = E' \). If \( x \) is constant and of value \( a \) then \( E = (\text{AND}(\text{EQ} \ z \ x) E') \).

The purpose of these transformations is to convert set declarations into type tests.

Vector expressions are also subject to desugaring if they do not appear as a list of arguments in a function call. The vector \( V \equiv (E_1, \ldots, E_n) \) will be desugared into \( V \equiv (1 \mapsto E_1 | \ldots | n \mapsto E_n) \).

Figure 4.6 gives an example of the parsing process. The parsed code will be passed to phase 1 of the translation algorithm described in section E.
**Source code:**

\[(x,y) : (\text{INTEGER}, \text{INTEGER})\]

\[
A \equiv x+y;
B \equiv x-y;
(A,B,x)
\]

**Parsed code:**

\[
(FN^2(P \ x(\text{INTEGER} \ x))
(P \ y(\text{INTEGER} \ y))
(SC^2(NM \ A \ (+x \ y))
(NM \ B \ (-x \ y))
(EX^2(FN^1(P \ x(EQ \ 1 \ Z))A)
(FN^1(P \ w(EQ \ 2 \ w))B)
(FN^1(P \ t(EQ \ 3 \ t)x))))
\]

**Figure 4.6. Parsing the source code**
D. Preliminary Notions

As in virtually every other applicative language, the concept of free and bound variables plays an important role in the translation of FCL programs. In this and later sections, the term "variable" is used in a mathematical sense, and has nothing to do with alterable storage locations. FCL has two distinct types of variables: sub-expression names and argument names. An argument name x is defined by an expression of the form x:S, while a sub-expression name N is defined by an expression of the form N=E.

The rules for free and bound variables are given below. In this and later sections the term "token" will refer to a token as recognized by the parser. (In the following, x, x_1..., and y represent single tokens while E and E_1... represent arbitrary expressions.)

1. Built-in function names, built-in set names, operators and literals are constants, and do not represent either free or bound variables.
2. x is free in y if x=y and neither free nor bound in y if x≠y.
3. Let E=x:E_1. Then x is bound in E if x is free or bound in E_1 and is neither free nor bound in E otherwise. Let y≠x. Then y is bound in E iff y is bound in E_1, and y is free in E iff y is free in E_1.
4. Let $E = (x_1, \ldots, x_n) : E_1$ then $x_i$ is bound in $E$ if $x_i$ is free or bound in $E_1$ and is neither free nor bound in $E$ otherwise. Let $y \not\in x_i$ for all $1 \leq i \leq n$ then $y$ is bound in $E$ iff $y$ is bound in $E_1$ and $y$ is free in $E$ iff $y$ is free in $E_1$.

5. Let $E = x : E_1 \lor E_2$ then $x$ is bound in $E$ if $x$ is free or bound in $E_1$ or $E_2$, and is neither free nor bound in $E$ otherwise. Let $y \not\in x$. Then $y$ is bound in $E$ iff $y$ is bound in $E_1$, and $y$ is free in $E$ iff $y$ is free in $E_1$.

6. Let $E = (x_1, \ldots, x_n) : E_1 \lor E_2$ then $x_i$ is bound in $E$ if $x_i$ is free or bound in $E_1$ or $E_2$, and is neither free nor bound in $E$ otherwise. Let $y \not\in x_i$ for all $1 \leq i \leq n$ then $y$ is bound in $E$ iff $y$ is bound in $E_1$ or $E_2$, and $y$ is free in $E$ iff $y$ is free in $E_1$ or $E_2$.

7. Let $E = x : E_1$ then $x$ is bound in $E$ if $x$ is free or bound in $E_1$, and is neither free nor bound in $E$ otherwise. Let $y \not\in x$. Then $y$ is bound in $E$ iff $y$ is bound in $E_1$, and $y$ is free in $E$ iff $y$ is free in $E_1$.

8. Let $E = x_1 : E_1 ; \ldots ; x_n : E_n$. Then $x_i$ is bound in $E$ if $x_i$ is free or bound in any $E_j$, and is neither free nor bound in $E$ otherwise. Let $y \not\in x_i$ for all $1 \leq i \leq n$. Then $y$ is free in $E$ iff $y$ is free in some $E_j$ and $y$ is bound in $E$ iff $y$ is bound in some $E_j$. 
9. Let $E = E_1 \rightarrow E_2$ such that $E_1$ is not of the form $E_3 : E_4$. Then, $x$ is free in $E$ iff $x$ is free in $E_1$ or $E_2$, and $x$ is bound in $E$ iff $x$ is bound in $E_1$ or $E_2$.

10. Let $E = E_1 \ op \ E_2$ where $\ op$ is any binary operator other than ";", "\rightarrow", ":", or "\equiv". Then $x$ is free in $E$ iff $x$ is free in $E_1$ or $E_2$, and $x$ is bound in $E$ iff $x$ is bound in $E_1$ or $E_2$. (The comma is a binary operator for purposes of this rule.)

11. Let $E = (E_1)$ or \{$E_1$\}. Then $x$ is free in $E$ iff $x$ is free in $E_1$, and $x$ is bound in $E$ iff $x$ is bound in $E_1$.

12. Let $E = E_1 \ E_2$. Then, $x$ is free in $E$ iff $x$ is free in $E_1$ or $E_2$, and $x$ is bound in $E$ iff $x$ is bound in $E_1$ or $E_2$. (This rule includes the unary operators and function calls.)

If a variable $x$ appears in $E$, but is neither free nor bound in $E$, then $x$ is said to be superfluous in $E$. Figure 4.7 gives some examples of free and bound variables.

A necessary condition for a program to be semantically meaningful is that it contain no free variables.

FCL differs from many other applicative languages in that variable bindings can occur at virtually any point within the program. This complicates the abstraction of variables.
E = Status of variables

x:INTEGER x is superfluous in E

x:REAL→x×x x is bound in E

x:REAL→x+y y is free in E
x is bound in E

x+(x:REAL→x+2)(3) x is bound in E
x is free in E

Figure 4.7. Examples of free and bound variables

from a program. For example, all of the bracket-abstraction algorithms given in [Curry and Feys 1958] assume that every occurrence of the variable to be abstracted is a free occurrence. This assumption may be false for some FCL expressions as exemplified by the expression x+(x:REAL→x+2)(x) which contains both free and bound occurrences of the variable "x". The algorithm presented in the next section is designed to handle such expressions correctly.

Now consider the following expression:

x:INTEGER→(A≡x+2;

B≡y:INTEGER A+y+x;

B(x)+2*A)
The sub-expression B := y : INTEGER + A + y + x contains two free vari-
ables x and A. In a sense, these two variables represent
global references. If this sub-expression is to be imple-
mented as a data-flow procedure, then the global references
must be removed. Some data-flow architectures may allow
global references to procedure names, but the algorithm pre-
sented in the next section will not take this into account.

Global references within procedures may occur in three
ways. A free variable may eventually be bound as an argu-
ment of a higher-order function, as the name of a separate sub-
expression, or as the name of the procedure itself. The
translation algorithm has special provisions for each of
these cases.

The rules for free and bound variables in the parsed
code follow naturally from the rules given above.

E. Details of the Translation
Algorithm

1. The overall structure

The translation algorithm consists of five different
phases. Although it may be possible to combine two or more
of these phases into a single pass over the code, each phase
will be presented as a separate pass for reasons of clarity.
The five phases are:
1. Combining mutually recursive functions
2. Removal of recursion
3. Abstraction of sub-expression names
4. Uncurrying functions
5. Abstraction of argument names

Each of the phases will remove certain types of global references within functions. Phases 1 and 2 are optional for architectures which allow global references to function names. Phase 4 is optional for all architectures, but use of this phase will produce more compact code than could be generated without it.

2. Phase 1: Combining mutually recursive functions

Phase 1 is executed once for every op-list of the form \((\text{SC}^{n-1}E_1...E_n)\). Any of the \(E_i\) which are not of the form \((\text{NM }N_i D_i)\) are removed from consideration. This leaves \(k<n\) expressions of the form \((\text{NM }N_i D_i)\). If \(k<2\) then processing terminates for this op-list. Otherwise, the "occurs-free-in" relation is computed for the \(k\) remaining expressions. The "occurs-free-in" relation is a directed graph containing \(k\) nodes which are denoted \(Z_1,...,Z_k\). There is an arc from \(Z_i\) to \(Z_j\) iff \(N_i\) occurs free in \(D_j\).

Once the "occurs-free-in" relation has been computed, it is scanned for cycles. If the "occurs-free-in" relation contains a cycle of two or more nodes, then the set of
expressions \((NM N_1 D_1) \ldots (NM N_k D_k)\) is partitioned into one or more cycle-sets. If a node \(Z_i\) does not belong to a cycle containing two or more nodes, then the corresponding expression \((NM N_i D_i)\) is removed from consideration. If two nodes \(Z_i\) and \(Z_j\) belong to the same cycle, then the corresponding expressions \((NM N_i D_i)\) \((NM N_j D_j)\) are added to the same cycle-set. Once all cycle-sets have been computed, any two cycle-sets which share a common element are combined into a single cycle-set. This process continues until a class of pairwise disjoint sets has been obtained.

Each cycle-set consists of \(m \leq k\) expressions of the form \((NM N_i D_i)\). Assume that the order of these expressions \((NM N_1 D_1) \ldots (NM N_m D_m)\) has been preserved from the original op-list. The parsed form of the expression \(N'=(D'_1, \ldots, D'_m)\) is inserted ahead of \((NM N_1 D_1)\) in the original op-list (with adjustment of the superscript of the SC operator). The new name \(N'\) is generated in such a way as to guarantee that it does not occur anywhere in the original program. The expression \(D'_i\) is defined to be \(D_i\) with each free occurrence of \(N_j\) replaced by \((\text{APPLY } N'j)\) for all \(1 \leq j \leq m\). Then, each of the expressions \((NM N_i D_i)\) is replaced by \((NM N_i (\text{APPLY } N'i))\). Figure 4.8 gives an example of phase 1 processing in terms of the source code.
Before phase 1:

\[
\begin{align*}
  x &: \text{INTEGER}\rightarrow \\
  (A&=x+2; \\
  F&=0\rightarrow y:\text{INTEGER}\rightarrow F(y-1)+G(y-1)+A; \\
  B&=x*2; \\
  G&=0\rightarrow y:\text{INTEGER}\rightarrow F(y-1)-G(y-1)-B; \\
  F(x)\times G(x) 
\end{align*}
\]

After phase 1:

\[
\begin{align*}
  x &: \text{INTEGER}\rightarrow \\
  (A&=x+2; \\
  \text{NEW} &= (0\rightarrow y:\text{INTEGER}\rightarrow \text{NEW}(1)(y-1) \\
  &\quad + \text{NEW}(2)(y-1)*A, \\
  &\quad 0\rightarrow y:\text{INTEGER}\rightarrow \text{NEW}(1)(y-1) \\
  &\quad - \text{NEW}(2)(y-1)-B); \\
  F &= \text{NEW}(1); \\
  B &= x*2; \\
  G &= \text{NEW}(2); \\
  F(x)\times G(x) 
\end{align*}
\]

Figure 4.8. Phase 1 processing example
3. **Phase 2: Eliminating recursion**

Phase 2 operates on op-lists of the form \((NM \, N \, E)\) where \(N\) is free in \(E\). Each such op-list will be processed as follows. The new op-list \((NM \, N'(F^{1}(PN \, TRUE)E'))\) will be inserted ahead of the original op-list. If the original op-list appears in a SC op-list, the new op-list will be inserted into the existing SC op-list. Otherwise, a new SC\(^1\) op-list is created. The new name \(N'\) is created in such a way that it does not occur in the original program. The expression \(E'\) is defined to be the expression \(E\) with all free occurrences of \(N\) replaced by \((APPLY \, N \, N)\). The original op-list is replaced by \((NM \, N \, (APPLY \, N'N'))\). Figure 4.9 gives examples of phase 2 processing in terms of the source code.

It should be pointed out that the expressions selected for phase 1 and phase 2 processing need not denote functions. Figure 4.10 gives examples of arbitrary expressions selected for phase 1 and phase 2 processing.

If one attempts to hand-evaluate \(A\), \(B\), or \(C\) from the first example, or \(A\) from the second example of Figure 4.10, it will quickly become obvious that the evaluation will never terminate. Since each of these definitions is circular, the meanings of the expressions have not been altered. The most important benefit of allowing phases 1 and 2 to operate on arbitrary expressions is that
**Before phase 2:**

\[
x: \text{INTEGER} \Rightarrow
\]
\[
(A = x + 2;
\]
\[
\text{NEW} = (0 \rightarrow l | y: \text{INTEGER} \Rightarrow \text{NEW}(1)(y-1) +
\]
\[
0 \rightarrow l | y: \text{INTEGER} \Rightarrow \text{NEW}(1)(y-1) -
\]
\[
\text{NEW}(2)(y-1)+A,
\]
\[
F = \text{NEW}(1);
\]
\[
B = x \times 2;
\]
\[
G = \text{NEW}(2);
\]
\[
F(x) \times G(x);
\]

**After phase 2:**

\[
x: \text{INTEGER} \Rightarrow
\]
\[
(A = x + 2;
\]
\[
\text{NEW}_2 = \text{NEW}: \text{ANY} \Rightarrow (0 \rightarrow l | y: \text{INTEGER} \Rightarrow \text{NEW}(\text{NEW})(1)(y-1) +
\]
\[
\text{NEW}(\text{NEW})(2)(y-1)+A,
\]
\[
0 \rightarrow l | y: \text{INTEGER} \Rightarrow \text{NEW}(\text{NEW})(1)(y-1) -
\]
\[
\text{NEW}(\text{NEW})(2)(y-1)-B);
\]
\[
F = \text{NEW}(1);
\]
\[
B = x \times 2;
\]
\[
G = \text{NEW}(2);
\]
\[
F(x) \times G(x);
\]

**Before phase 2:**

\[
\text{FACT} 0 \rightarrow l | x: \text{INTEGER} \Rightarrow x \times \text{FACT}(x-1)
\]

**After phase 2:**

\[
\text{NEW} = \text{FACT}: \text{ANY} \Rightarrow (0 \rightarrow l | x: \text{INTEGER} \Rightarrow x \times \text{FACT}((\text{FACT})(x-1)));
\]
\[
\text{FACT} = \text{NEW}(\text{NEW})
\]

Figure 4.9. Examples of phase 2 processing
Before phase 1:
A=B+C;
B=C+2;
C=A+1

After phase 1:
NEW1=(NEW1(2)+NEW1(3),NEW1(3)+2,NEw1(1)+1);
A=NEW1(1);
B=NEW1(2);
C=NEW1(3)

After phase 2:
NEW2= NEW1:ANY+((NEW1(NEW1))(2)+(NEW1))(3),
(NEW1(NEW1))(3)+2(NEW1(NEW1))(1)+1);
NEW1= NEW2(NEW2);
A=NEW1(1);
B=NEW1(2);
C=NEW1(3)

Before phase 2:
A=A+1

After phase 2:
NEW=A:ANY+A(A)+1;
A=NEW(NEW)

Figure 4.10. Phases 1 and 2 processing for arbitrary expressions
expressions similar to the following will be handled correctly:
\[ P \equiv 0 + 1 \mid x : \text{INTEGER} \ x \ast G(x - 1); \ G \equiv F. \] A secondary benefit is that the code produced by phase 2 will be completely acyclic. This may be of some benefit on architectures which do not permit cyclic code.

4. Phase 3: Abstracting sub-expression names

The purpose of phase 3 is to remove sub-expression names from expressions. One method of doing so would be to convert sub-expression names to argument names as shown in Figure 4.11.

**Original expression:**

\[
A \equiv x + y;
B \equiv x \ast 2;
A + B + 1
\]

** Converted expression:**

\[
((A, B) : (\text{ANY, ANY}) \rightarrow A + B + 1)(x + y, x \ast 2)
\]

**Figure 4.11.** Conversion of subexpression names to argument names

The technique illustrated in Figure 4.11 is essentially that used by Turner in [Turner 1979a,b]. However, since data-flow computations cannot, in general, rely on the sharing properties of Turner graphs, the technique of conversion to argument names may cause some sub-expressions to be evaluated several times. For example, conversion to
argument names applied to the expression: \( A = 2 + 3; B = A + 1; A + B \), would cause the sub-expression \( A \) to be evaluated twice. A different technique which avoids this difficulty is presented here.

Consider an op-list of the form:

\[
(Sc^{k-1}(NM N_1 E_1) \ldots (NM N_k E_k)).
\]

This type of op-list is intended to be implemented as a set of independent computations as illustrated in Figure 4.12.

![Graph](image)

**Figure 4.12.** Implementation of a set of sub-expressions

In Figure 4.12, the nodes labeled \( \sigma_1 \ldots \sigma_n \) represent ID instructions. Uses of sub-expression names are intended to be replaced by an arc from the appropriate ID instruction. (Actually the ID instructions are assumed only for convenience and need not be physically present.)

The abstraction technique presented here will replace a
sub-expression name with a reference to the appropriate ID instruction. This algorithm is more than a simple name replacement because function definitions and nested SC op-lists must be handled appropriately.

Phase 3 consists of the function PH3 and an abstraction operation denoted by { }, which are defined below. In the following, x denotes a single token while E and D denote arbitrary expressions.

1. \[ \text{PH3}(\text{SC}^{k-1} \text{E}_1 \ldots \text{E}_k) = \]
   \[ (\text{IC}^{k-1}_1 \{\text{x}_1, \ldots, \text{x}_k\} \text{E}_1^{i} \]
   \[ \ldots \]
   \[ \{\text{x}_1, \ldots, \text{x}_k\} \text{E}_k^{i}) \]

2. \[ \text{PH3}(\text{OP} \text{E}_1 \ldots \text{E}_k) = \]
   \[ (\text{OP} \text{PH3(E}_1^{i}) \ldots \text{PH3(E}_k^{i})) \]
   (Except as in 1)

3. \[ \text{PH3(x)} = x \]

4. \[ \{\text{x}_1, \ldots, \text{x}_k\} (\text{FN}^i \text{E}_1^{i} \ldots \text{E}_{i+1}^{i}) = \]
   \[ (\text{APPLY} \{\text{x}_1, \ldots, \text{x}_{k-1}\} (\text{FN}^{i+1} (\text{P} \text{x}_k \text{TRUE}) \text{E}_1^{i+1} \ldots \text{E}_{i+1}^{i+1} \sigma_k)) \]
   if \text{x}_k \text{ is free in (} \text{FN}^i \text{E}_1^{i} \ldots \text{E}_{i+1}^{i} \]
   \[ \{\text{x}_1, \ldots, \text{x}_{k-1}\} (\text{FN}^i \text{E}_1^{i} \ldots \text{E}_{i+1}^{i}) \]
   otherwise.

5. \[ \{ \} (\text{FN}^i \text{E}_1^{i} \ldots \text{E}_{i+1}^{i}) = \]
   \[ (\text{FN}^i \text{PH3(E}_1^{i}) \ldots \text{PH3(E}_{i+1}^{i})) \]
6. \( \{X_1, \ldots, X_k\}(SC_i \ldots E_i) = \)
\[
(\text{IC}_k \ldots \{X_1', \ldots, X_k', X_{k+1}', \ldots, X_{k+l}'\}E_i'

\vdots

\{X_1', \ldots, X_k', X_{k+1}', \ldots, X_{k+l}'\}E_i')
\]

7. \( \{X_1, \ldots, X_k\}(\text{OP} \ E_1 \ldots E_i) = \)
\[
(\text{OP} \ \{X_1, \ldots, X_k\}E_1

\vdots

\{X_1, \ldots, X_k\}E_i)
\]
(Except as in 4, 5, and 6)

8. \( \{X_1, \ldots, X_n\} x = \sigma_i \)
   
   if \( x = X_i \).
   
   \( \{X_1', \ldots, X_n\} x = x \)

   otherwise.

In rule 1, if \( E_i \) is of the form \( (\text{NM} \ N_i D_i) \) then \( X_i = N_i \) and \( E'_i = D_i \). Otherwise \( X_i = \square \) and \( E'_i = E_i \). (The symbol \( \square \) is used as a place-holder.) In rule 6, if \( E_j \) is of the form \( (\text{NM} \ N_j D_j) \) then \( X_{k+j} = N_j \) and \( E'_j = D_j \). Otherwise \( X_{k+j} = \square \) and \( E'_j = E \). If \( X_n = X_{k+m} \) \( (1 \leq n < k) \) for any \( 1 \leq m < i \) then \( X'_n = \square \), otherwise \( X'_n = X_n \).

Figure 4.13 gives examples of phase 3 processing in terms of the source code.
Before phase 3:

\[
x : \text{INTEGER} \rightarrow (A \equiv x + 2; \\
F \equiv y : \text{INTEGER} \rightarrow A + y; \\
F(x) * A)
\]

After phase 3:

\[
x : \text{INTEGER} \rightarrow (x + 2; \\
((A, y) : (\text{ANY, INTEGER}) \rightarrow A + y) (\sigma_1); \\
\sigma_2 (x) * \sigma_1)
\]

Before phase 3:

\[
x : \text{INTEGER} \rightarrow (A \equiv x + 2; \\
B \equiv (C \equiv x * x; \\
C + C); \\
A * B)
\]

After phase 3:

\[
x : \text{INTEGER} \rightarrow (x + 2; \\
(x * x; \\
\sigma_4 + \sigma_4); \\
\sigma_1 * \sigma_2)
\]

Figure 4.13. Examples of phase 3 processing
Phase 3 processing has two benefits. First, it removes certain free variables from function definitions. Second, it removes the necessity to maintain environments for subexpression names, particularly in the case where the same name is reused in a nested set of sub-expressions.

5. **Phase 4: Uncurrying functions**

This phase is optional, since correct code will be generated even if phase 4 is not used. The primary reason for including this phase is the simplification of function definitions. Consider the expression \( F = x : \text{INTEGER} \rightarrow y : \text{INTEGER} \rightarrow x + y + 1 \). Phase 5 will remove the global reference to \( x \) in the expression \( x + y + 1 \) by converting the function definition to \( F = x : \text{INTEGER} \rightarrow ((x, y) : (\text{ANY}, \text{INTEGER}) \rightarrow x + y + 1)(x) \). Phase 4 will convert the function definition to the following equivalent form: \( F' = (x, y) : (\text{INTEGER}, \text{INTEGER}) \rightarrow x + y + 1 \). The translation of \( F'(n, m) \) will require fewer run-time instructions than the translation of \( F(n, m) \). Furthermore, the translation of \( F'(n) \) will be virtually identical to the translation of \( F(n) \).

Another problem which will be addressed by phase 4 is determining which alternation operators ("\( | \)"") can be implemented as "switch" operations. The conventional "switch" instruction is illustrated in Figure 4.14.
Figure 4.14. The conventional switch operator

The semantics of this operator dictate that when the data-token on the control arc $C$ is a boolean "true" then the data-token on the input arc $I$ is routed to the "T" output arc. Similarly, when the control token is "false" the input token is routed to the "F" output arc. One may design a more general switch operator whose control token may take values from an arbitrary finite set, say SET(1,2,3), as illustrated in Figure 4.15.

Figure 4.15. The generalized switch operator
The generalized switch operator operates in a manner analogous to the conventional switch operator with the additional requirement that if the value of the control token does not correspond to any output arc then the input token is absorbed and no output token is created. The generalized switch represents a template of conventional operators rather than a new operator. The concept of the generalized switch can be expanded to include type tests as well as tests for particular scalars, as illustrated in Figure 4.16.

Figure 4.16. Generalized switch with type tests
The generalized switch operator is the mechanism which will be used to route operands into expressions of the form $E_1^+E_2^+\ldots|E_{k-1}^+E_k^+$. For example the expression $1+x+2|2+x*3|3+x-4$ would require a generalized switch identical to that given in Figure 4.15 to route the value of $x$ to the proper expression. However, there are cases where the alternation operator cannot be implemented as a generalized switch as in the following $1\rightarrow 2|x|5\rightarrow 7$. Phase 4 must distinguish between these two cases.

The reductions performed by phase 4 are given below.

1. $(FN^nE_1\ldots E_n(EX^kD_1\ldots D_{k+1}))$  
   where $D_i$ is of the form $(FN^iC_1\ldots C_{i+1})$  
   is reduced to  
   $(EX^k(FN^nE_1\ldots E_nD_1)\ldots(FN^nE_1\ldots E_nD_{k+1}))$

2. $(EX^kE_1\ldots E_{i-1}(EX^jD_1\ldots D_{j+1})E_{i+1}\ldots E_{k+1})$  
   is reduced to  
   $(EX^{k+j}E_1\ldots E_{i-1}D_1\ldots D_{j+1}E_{i+1}\ldots E_{k+1})$

3. $(FN^nE_1\ldots E_nFN^iD_1\ldots D_{i+1})$  
   is reduced to  
   $(FN^{n+i}E_1\ldots E_nD_1\ldots D_{i+1})$

4. $(APPLY(APPLY E_1\ldots E_n)D_1\ldots D_k)$  
   is reduced to  
   $(APPLY E_1\ldots E_nD_1\ldots D_k)$

An example of phase 4 processing is given in Figure 4.17.
Original expression:
\[ a : \text{INTEGER} \to (b : \text{INTEGER} \to c : \text{INTEGER} \leftarrow a + b + c \mid \\
\quad e : \text{REAL} \to \\
\quad (f : \text{INTEGER} \to a \cdot e + f \mid h : \text{REAL} \to a \cdot e \cdot h) \]

Apply reduction 1:
\[ a : \text{INTEGER} \to b : \text{INTEGER} \to c : \text{INTEGER} \leftarrow a + b + c \mid \\
\quad a : \text{INTEGER} \to e : \text{REAL} \to \\
\quad (f : \text{INTEGER} \to a \cdot e + f \mid h : \text{REAL} \to a \cdot e \cdot h) \]

Apply reduction 1:
\[ a : \text{INTEGER} \to b : \text{INTEGER} \to c : \text{INTEGER} \leftarrow a + b + c \mid \\
\quad a : \text{INTEGER} \to (e : \text{REAL} \to f : \text{INTEGER} \to a \cdot e + f) \mid \\
\quad e : \text{REAL} \to h : \text{REAL} \to a \cdot e \cdot h) \]

Apply reduction 1:
\[ a : \text{INTEGER} \to b : \text{INTEGER} \to c : \text{INTEGER} \leftarrow a + b + c \mid \\
\quad (a : \text{INTEGER} \to e : \text{REAL} \to f : \text{INTEGER} \to a \cdot e + f) \mid \\
\quad a : \text{INTEGER} \to (e : \text{REAL} \to h : \text{REAL} \to a \cdot e \cdot h) \]

Apply reduction 2:
\[ a : \text{INTEGER} \to b : \text{INTEGER} \to c : \text{INTEGER} \leftarrow a + b + c \mid \\
\quad a : \text{INTEGER} \to e : \text{REAL} \to f : \text{INTEGER} \to a \cdot e + f \mid \\
\quad a : \text{INTEGER} \to e : \text{REAL} \to h : \text{REAL} \to a \cdot e \cdot h \]

Apply reduction 3 (6 times):
\[ (a, b, c) : (\text{INTEGER}, \text{INTEGER}, \text{INTEGER}) \to a + b + c \mid \\
\quad (a, e, f) : (\text{INTEGER}, \text{REAL}, \text{INTEGER}) \to a \cdot e + f \mid \\
\quad (a, e, h) : (\text{INTEGER}, \text{REAL}, \text{REAL}) \to a \cdot e \cdot h \]

Figure 4.17. Example of phase 4 processing
Once all reductions have been done, the following conversion will be made: Let $E$ be an op-list of the form $(E^k E_1 \ldots E_{k+1})$ where each $E_i$ is of the form $(F^m N^p D_j \ldots)$. Furthermore, let $n$ be the minimum over $E_1 \ldots E_{k+1}$ of the superscripts on the FN operators. Then $E$ is replaced by the op-list $(S^k E_1' \ldots E_{k+1}')$. Let $E_i = (F^m N^p D_j \ldots)$. If $m=n$ then $E_i' = (C^m D_j \ldots D_{m+1})$. Otherwise ($m>n$) $E_i' = (C^m D_j \ldots D_n (F^{m-n} D_{n+1} \ldots D_{m+1})$). This final conversion guarantees that each arm of the conditional represented by the SW operator requires an identical number of arguments.

Proper translation of an SW op-list requires a generalized merge as well as a generalized switch. The generalized merge corresponding to the switch of Figure 4.15 is illustrated in Figure 4.18.

Figure 4.18. The generalized merge operator
The operator of Figure 4.18 will select a data-token from the corresponding input arc if the control token's value is 1, 2, or 3. Otherwise, a token will be selected from the arc labeled "other". In most cases, this arc will provide an error code.

6. **Phase 5: Abstracting argument names**

The final phase of the translation algorithm removes all argument names from a program, and in the process, will remove all remaining free variables from function definitions. Phase 5 is based on Abdali's [Abdali 1976] abstraction technique, but is somewhat more complicated due to the free structure of FCL. Phase 5 consists of a function PH5 and an abstraction operation denoted by \[\] .

The rules for these are given below. In these rules, D and E (with or without subscripts) denotes arbitrary expressions; OP denotes ar arbitrary operator; X, Y, and N denote names; and x denotes a single arbitrary token.

1. \[\text{PH5}(P^{X_1 E_1} \ldots (P X_k E_k)E_{k+1}) = (K_0 (P^{X_1 \ldots X_k}E_1 \ldots [X_1 \ldots X_k]E_{k+1}))\]

2. \[\text{PH5}(C^{X_1 E_1} \ldots (P X_k E_k)E_{k+1}) = (C^{X_1 \ldots X_k}E_1 \ldots [X_1 \ldots X_k]E_{k+1})\]
3. \( \text{PH}^5(S_n^k E_1 \ldots E_{k+1}) \)
   \[ = (K_0(S_n^k \text{PH}^5(E_1) \ldots \text{PH}^5(E_{k+1}))) \]

4. Except as in 1, 2, and 3
   \( \text{PH}^5(\mathit{OP} E_1 \ldots E_k) = \)
   \[ (\exists^{k}_{0} \mathit{OP} \text{PH}^5(E_1) \ldots \text{PH}^5(E_k)) \]

5. \( \text{PH}^5(\sigma_i) = \sigma_i \)

6. Except as in 5
   \( \text{PH}^5(x) = (K_0 x) \)

7. Let \( D = (\text{FN}^n(\mathit{P} Y_1 E_1) \ldots (\mathit{P} Y_n E_n)E_{n+1}) \)
   Then if for all \( i \leq i \leq k \) \( X_i \) is not free in \( D \)
   then
   \[ [X_1, \ldots, X_k] D = \]
   \[ (K_n^{\text{FN}}[Y_1, \ldots, Y_n]E_1 \]
   \[ \ldots \]
   \[ [Y_1, \ldots, Y_n]E_{n+1}) \]
   otherwise, let \( \{N_1, \ldots, N_j\} \) be the set of those \( X_i \)
   which are free in \( D \).
   Then
   \[ [X_1, \ldots, X_k] D = \]
   \[ [X_1, \ldots, X_k] \{\text{APPLY}(\text{FN}^{n+j}(\mathit{P} N_1 \text{TRUE}) \]
   \[ \ldots \]
   \[ (\mathit{P} N_j \text{TRUE}) \]
   \[ (\mathit{P} Y_j E_1) \]
   \[ \ldots \]
   \[ (\mathit{P} Y_n E_n) \]
   \[ E_{n+1}) \]
   \[ N_1 \ldots N_j \]
8. Let $D = (\mathbf{Sw}_{m}^{n} E_{1} \ldots E_{n+1})$

If for all $1 \leq i \leq k$ $X_{i}$ is not free in

$$[X_{1}, \ldots, X_{k}]D = (K_{k}(\mathbf{Sw}_{m}^{n} \mathbf{PHS}(E_{1}) \ldots \mathbf{PHS}(E_{n+1})))$$

Otherwise, let $\{N_{1}, \ldots, N_{j}\}$ be the set of those $X_{i}$

which are free in $D$. Then

$$[X_{1}, \ldots, X_{k}]D =$$

$$[X_{1}, \ldots, X_{k}](\text{APPLY}(\mathbf{Sw}_{m+j}^{n}[N_{1}, \ldots, N_{j}]E_{1}$$

$$\ldots$$

$$[N_{1}, \ldots, N_{j}]E_{n+1})$$

$$N_{1} \ldots N_{j}$$

9. $[N_{1}, \ldots, N_{j}](\text{CH}^{n}(P Y_{1} E_{1}) \ldots (P Y_{n} E_{n})E_{n+1})$

$$= (\text{CH}^{n+j}(P N_{1} \text{ TRUE}) \ldots (P N_{j} \text{ TRUE})$$

$$(P Y_{1} E_{1}) \ldots (P Y_{n} E_{n})E_{n+1})$$

10. Except as in 7, 8, and 9

$$[X_{1}, \ldots, X_{k}](\text{OP} E_{1} \ldots E_{n})$$

$$= (E_{k} \text{ OP } [X_{1}, \ldots, X_{k}]E_{1}$$

$$\ldots$$

$$[X_{1}, \ldots, X_{k}]E_{n})$$

11. $[X_{1}, \ldots, X_{k}]\sigma_{i} = \sigma_{i}$

12. If $x \neq X_{i}$ for all $1 \leq i \leq k$ then

$$[X_{1}, \ldots, X_{k}]x = (K_{k}x)$$

otherwise, let $i$ be the largest integer

such that $X_{i} = x$. Then

$$[X_{1}, \ldots, X_{k}]x = i^{i}_{k}.$$
These rules complete the presentation of the translation algorithm. Figure 4.19 gives an example of phase 5 processing. More complicated examples can be found in the next section.

**Original expression:**

\[ x: \text{INTEGER} \rightarrow x + x - 1 \]

**Parsed form:**

\[ (\text{FN}^1 (P \times (\text{INTEGER} \times)) (- (+ x x)1)) \]

**Phase 5 result:**

\[ (K_0(\text{FN}^1(B_1^1 \times \text{INTEGER} I_1^1))

\[ (B_2^2 - (B_1^2 + I_1^1) I_1^1) (K_1 1))) \]

**Figure 4.19. Example of phase 5 processing**

The next section demonstrates how data-flow graphs are generated from the intermediate code generated by phases 1 through 5.
F. Code Generation

The two fundamental operators of the code generation phase are $B^n_m$ and $K_n$. An op-list of the form $(B^n_mE_1\ldots E_{n+1})$ is interpreted to mean: "Generate an instruction (or template) of type $E_1$ with $n$ input arcs, and let the value of input arc $i$ be supplied by the graph for $E_1$." An op-list $(K^n_mE)$ is interpreted to mean: "Generate a constant of type $E$." The expression $E$ may be a literal or an op-list of the form $(SW^n\ldots)$ or $(FN^n\ldots)$. In the first case, a constant-generating instruction whose value is $E$ is produced. In the second case a "PROCED" operator is generated, and the code generator is called recursively to generate the code for the procedure defined by $E$. The code generator has the option of executing operators of the form $K^n_0$ and $B^n_0$ itself rather than generating code for them. This section will, however, assume that code is generated these operators. Each op-list is assumed to have one or more output arcs whose destinations are implied by the position of the op-list in the code. Multiple output arcs may occur only for sub-expressions, and the destinations of these arcs are implied by the occurrence of $\sigma_i$ constants. Figure 4.20 gives examples of code generation. These examples omit the portion of the graphs which supply a function's arguments to the right-hand-side of its definition. This portion of the graph will be discussed later.
Original expression:

\[(x, y): (\text{INTEGER, INTEGER}) \rightarrow (x \times 3) + (y / 2)\]

Intermediate code:

\[
(K_0 (\text{FN}^2 (B_2^1 \text{ INTEGER } I_2^1) (B_2^1 \text{ INTEGER } I_2^2))
\]

\[
(B_2^2 + (B_2^2 \times I_2^1(K_2^3)) (B_2^2 / I_2^2(K_2^2)))
\]

Graph for RHS of function definition:

Original expression:

\[(x, y): (\text{REAL, REAL}) \rightarrow (A = x + 2; B = y - 3; C = A + b - y; C = 2 + x * A)\]

Intermediate code:

\[
(K_0 (\text{FN}^2 (B_2^1 \text{ REAL } I_2^1) (B_2^1 \text{ REAL } I_2^2))
\]

\[
(B_2^4 \text{ IC}^3 (B_2^2 + I_2^1(K_2^2))
\]

\[
(B_2^2 - I_2^2(K_2^2))
\]

\[
(B_2^2 - (B_2^2 + \sigma_1 \sigma_2) I_2^2)
\]

\[
(B_2^2 + (B_2^2 \times \sigma_3(K_2^2)) (B_2^2 I_2^1 \sigma_1))
\]

Figure 4.20. Graph generation for RHS of function definitions
Graph for RHS of function definition:

Figure 4.20 (Continued)
The left-hand-side of a function definition consists of a number of type-checks. These checks may include tests for specific values as well as for general types. In order for the full power of FCL to be realized, type tests for INTEGER, REAL, BOOLEAN, CHARACTER, STRING, and ATOM must be implementable in some form on the target architecture. This will permit the correct translation of expressions of the form: INTEGER\to3\mid REAL\to2. This section will assume the existence of type-checking instructions on the target machine.

There are two ways in which code could be generated for the left-hand-side of a function definition. These are illustrated in Figures 4.21 and 4.22. In both figures the intermediate code is assumed to be of the form \((\cap_{i=1}^{k} E_i\ldots E_k E_{k+1})\) where each \(E_i\ (1<i<k)\) represents a boolean valued function.

The graph given in Figure 4.21 provides more parallelism than that given in Figure 4.22. On the other hand, the graph of Figure 4.22 is likely to trap more errors than that of Figure 4.21. For example, consider the function \(F=x:REAL\to y:REAL\backslash x+y>1\to x*y\). In the function call \(F("ABC",.2)\), the graph of Figure 4.21 will allow the "+" operator to be applied to "ABC". The graph of Figure 4.22 will avoid this. In either case, one may treat the
Figure 4.21. Parallel implementation of function LHS
Figure 4.22. Conditional implementation of LHS of function
type-check graph as a generalized template of the form given in Figure 4.23.

The generalized type-check template can also be used to create procedures defined by SW operators. An example is given in Figure 4.24. It is assumed that the intermediate code is of the form

\[
(SW_n^k \ (CH^n_1 \ E(1,1) \cdots E(1,n) \ E(1,n+1)) \\
\vdots \\
(CH^n_1 \ E(k+1,1) \cdots E(k+1,n) \ E(k+1,n+1))
\]
Figure 4.24. Template for $S_{n}^{k}$ operator
Figure 4.24 shows the implementation of the generalized switch and merge operators pictured in Figures 4.15, 4.16, and 4.18. Figures 4.21, 4.22 and 4.24 do not show how procedure arguments are handled. This information has been omitted because handling of procedure arguments is highly dependent on the target architecture. For example, the basic data flow architecture described in [Dennis and Misunas 1975] would require all procedure arguments to be appended into a single structure, while the asynchronous argument-passing mechanism described in [Oldehoeft and Maurer 1981] would require each argument to be passed through a special argument-transmission instruction. It is assumed that the code generator will add whatever is needed for the architecture in question.

The FN and SW operators provide "environments" for \( I^m_n \) constants. A constant of the form \( I^m_n \) refers to the nth argument of the most recently encountered FN or SW operator. Similarly, IC operators provide "environments" for \( \sigma_n \) constants. However, the situation for \( \sigma_n \) constants is somewhat more complicated than for \( I^m_n \) constants. Figure 4.25 gives the graph for an op-list of the form

\[
(\text{B}^m_{k+1} \text{ IC}^k_n \ E_1 \ldots E_{k+1}).
\]

Note that in Figure 4.25 that each of \( E_i \) \( 1 \leq i \leq k \) may contain references to \( \sigma_1 \) through \( \sigma_{n-1} \) as well as to \( \sigma_n \) through
A $\sigma_i$ constant may refer back to any subexpression defined by any containing IC operator which has been encountered since the last FN or SW operator. Note that phases 1 and 2 of the translation algorithm guarantee that the graph for an IC operator will be acyclic. The nodes labeled $\sigma_n \ldots \sigma_{n+k}$ in Figure 4.25 may be physically represented as ID instructions, or may simply denote an element of a data-structure used by the code generator.

Although the templates for most op-lists are obvious,
some operators require more explanation. In order to handle higher-order functions, it is assumed that the target architecture provides an instruction FIX, which is pictured in Figure 4.26.

![Diagram of the FIX instruction]

Figure 4.26. The FIX instruction

The FIX instruction operates on three arguments: a procedure $P$, a positive integer $n$ and an arbitrary value $V$. FIX returns a procedure which is created by fixing the value of the $n$th argument of $P$ to the value $V$. It is assumed that if $n=1$ and $P$ requires only one argument, then FIX will return the result of applying $P$ to $V$. The FIX instruction will be used to implement certain special operators as well as higher-order functions. This instruction is similar to the compose operator proposed by Arvind and Gostelow [Arvind and Gostelow 1977]. To further
facilitate the handling of higher-order functions, a liberalized view of the APPLY instruction will be taken. The APPLY instruction is assumed to be capable of handling an arbitrary number of arguments. It is further assumed that if fewer arguments are supplied to the APPLY instruction than are required by its procedure argument, then the APPLY will return a new procedure by fixing the first n arguments of its procedure argument. This generalized form of the APPLY operator may be implemented using the FIX instruction. The generalized APPLY instruction eliminates the need for flow analysis to determine whether an op-list of the form \((B_n^k \text{ APPLY } E_1 \ldots E_k)\) should be implemented as an ordinary apply instruction on as a collection of FIX instructions.

The FIX operator is used to generate graphs for the following op-lists:

- \((B_n^1 \text{ TRANSPOSE } E)\),
- \((B_n^2 \text{ COMPOSE } E_1 E_2)\), and
- \((B_n^2 \text{ EX}^1 E_1 E_2)\) as illustrated in Figures 4.27, 4.28, and 4.29.
Figure 4.27. Graph for the TRANSPOSE operator
Figure 4.28. Graph for the COMPOSE operator
Figure 4.29. Graph for EX$^1$
It is obvious that operators of the form $E^x_n$ ($n>1$) may be implemented using several $E^1_x$ operators.

Another operator which requires more explanation is the RS operator.

Figure 4.30 gives the graph for an op-list of the form $(B^2_n \text{ RS } E_1 \ E_2)$.

Figure 4.30. Graph for the RS operator
Two other operators whose graphs are not obvious are INSERT (which includes SUM and PROD) and SIZE. Recall that this implementation allows these operators to be applied to vectors only. The implementation of these operators in terms of high-level code is given in Figures 4.31 and 4.32.

\[
\text{INSERT} \equiv F : \text{FUNCTION}\rightarrow \\
V : \text{VECTOR}\rightarrow \\
(G \equiv (V_1, N, OLD) : (\text{ANY}, \text{ANY}, \text{ANY}) \rightarrow \\
\text{IF EXISTS} (V_1, N) \text{ THEN} \\
G(V_1, N+1, F(OLD, V_1(N))) \\
\text{ELSE} \\
OLD \text{ ENDIF}; \\
G(V, 2, V(1))) \\
\]

Figure 4.31. Implementation of the INSERT operator
SIZE ≡ V:VECTOR

(SZ1≡(V,HI,LO):(ANY,INTEGER,INTEGER)+
 IF EXISTS (V,HI) THEN
 SZ1(V,2*HI,HI)
 ELSE
 SZ2(V,HI,LO)ENDIF;
 SZ2≡(V,HI,LO):(ANY,INTEGER,INTEGER)→
 IF LO>=HI-1 THEN LO
 ELSE
 IF EXISTS (V,LO+((HI-LO)DIV 2) THEN
 SZ2(V,HI,LO+((HI-LO)DIV 2))
 ELSE
 SZ2(V,LO+((HI-LO DIV 2),LO)ENDIF ENDIF;
 SZ1(V,1,0))

Figure 4.32. Implementation of the SIZE operator

Note that the time complexity of the algorithm for SIZE presented in Figure 4.32 is logarithmic in the size of the vector processed. The SIZE operator can be used to produce a logarithmic implementation of SUM and PROD. Figure 4.33 presents a logarithmic version of SUM. A similar implementation for PROD is possible.
\[ \text{SUM} \equiv V : \text{VECTOR} \to \\
(SZ1 \equiv (V, N, K) : (\text{ANY, INTEGER, INTEGER}) \to \\
\text{IF } K = 1 \text{ THEN } V(N) \text{ ELSE } \\
SZ1(V, N, K \div 2) + \\
SZ1(V, N + (K \div 2), K - (K \div 2)) \text{ENDIF;} \\
\text{IF EXISTS } (V, 1) \text{ THEN } \\
SZ1(V, 1, \text{SIZE}(V)) \text{ ELSE } \\
0 \text{ ENDIF}) \]

Figure 4.33. Logarithmic form of SUM.

Finally, Figure 4.34 gives an example of the entire translation process for the factorial function, and Figure 4.35 gives the resultant graph.

Original expression:
\[ \text{FACT} \equiv 0 + 1 \mid X : \text{INTEGER} \to x \cdot \text{FACT}(x - 1) \]

Parsed expression:
\[ (\text{NM FACT} (\text{EX}^1 (\text{PN}^1 (\text{P Z (EQ 0 Z))) 1)) \\
(\text{PN}^1 (\text{P} \times (\text{INTEGER} x))) \\
(* \times (\text{APPLY FACT} (- x 1)))))) \]
Phase 1 and 2 result:

\[(\text{SC}^1 (\text{NM NEW } (\text{FN}^1 (P \text{ FACT TRUE}))

\left(\text{EX}^1 (\text{FN}^1 (P \text{ Z (EQ O Z)})\ 1

\left(\text{FN}^1 (P \times (\text{INTEGER} \ x))

\left(\left(\left(* x (\text{APPLY } (\text{APPLY FACT FACT}) (-x \ 1))))))))))

\left(\text{NM FACT } (\text{APPLY NEW NEW}))\)

Phase 3 result:

\[(\text{IC}^1 (\text{FN}^1 (P \text{ FACT TRUE})

\left(\text{EX}^1 (\text{FN}^1 (P \text{ Z (EQ O Z)})\ 1

\left(\text{FN}^1 (P \times (\text{INTEGER} \ x))

\left(\left(\left(* x (\text{APPLY } (\text{APPLY FACT FACT}) (-x \ 1))))))))))

\left(\text{APPLY} \sigma_1 \sigma_1 )\)

Phase 4 result:

\[(\text{IC}^1 (\text{SW}_2^1 (\text{CH}^2 (P \text{ FACT TRUE}) (P \text{ Z (EQ O Z)})\ 1

\left(\text{CH}^2 (P \text{ FACT TRUE}) (P \times (\text{INTEGER} \ x))

\left(\left(\left(* x (\text{APPLY FACT FACT}) (-x \ 1))))))))

\left(\text{APPLY} \sigma_1 \sigma_1 )\)

Phase 5 result:

\[(\text{B}_0^2 \text{ IC}_1^1 (\text{K}_0^1 (\text{SW}_2^1 (\text{CH}^2 (K_2 \text{ TRUE}) (B_2^2 \text{ EQ O I}_2^2) (K_2)\ 1

\left(\text{CH}^2 (K_2 \text{ TRUE}) (B_2^1 \text{ INTEGER I}_2^2)

\left(\left(\left(B_2^2 \times I_2^2 (B_2^3 \text{ APPLY I}_2^1 I_2^1 (B_2^2 - I_2^2 (K_2)\ 1))))))

\left(\text{B}_0^2 \text{ APPLY} \sigma_1 \sigma_1 )\)

Figure 4.34. Derivation of the intermediate code for the factorial function
Figure 4.35. Graph for factorial function
G. Conclusion

Although the algorithm presented in this chapter bears little resemblance to normal flow analysis [Allan and Oldehoeft 1980], the two algorithms share a common underlying principle. The flow analysis presented in [Allan and Oldehoeft 1980] is essentially a technique for associating an occurrence of a data-name with the proper name-value binding. The two abstraction algorithms presented here (denoted by { } and [ ]) are techniques for performing the same function on FCL code. The "controlled" nature of variable bindings in FCL (as well as certain other applicative languages) allows the simpler abstraction algorithms to be used instead of normal flow analysis.

This algorithm allows compact code to be produced for a powerful subset of the FCL language. The combinator approach used here allows the algorithm to be expressed in compact algebraic terms without sacrificing compactness in the generated code.

One obvious feature of this algorithm is that all functions are implemented as data-flow procedures, even those functions which are clearly meant to model data structures such as vectors and arrays. One important
benefit of this technique is that the door is left open to realize some of the benefits of demand driven code. Implementing a data structure as a data flow procedure effectively makes the procedure equivalent to a set of suspensions. Unfortunately, each of these "suspensions" must be resolved every time they are accessed. It is probable that further research in physical memory structures and physical representations of procedures will allow this situation to be improved.

Another benefit of implementing data-structures as procedures is that infinite structures may be built. In this case, an infinite structure is simply a function with an infinite domain; thus, there is no difference between an infinite structure and an ordinary function. Because of this equivalence, there is little point in giving examples of infinite structures.

More research is needed to adapt this algorithm to certain specific architectures. The code discussed in section F is similar to that of the basic data-flow machine [Dennis and Misunas 1975], and does not take advantage of the stream features of more advanced architectures [Arvind, Gostelow and Plouffe 1978, Dennis and Weng 1979]. It is probable that some high-level representation of streams will have to be included in FCL in order to take advantage of
these features.

This chapter has shown that FCL translates very well into data-flow code. The algorithm presented here could be used to translate other functional languages as well. Thus, it has been shown that functional languages can be used effectively on existing data flow machines without sacrificing compactness in the generated code, and without the need for highly complicated translation algorithms.
V. THEORETICAL FOUNDATIONS FOR THE STUDY OF SHARED DATA

A. Introduction

The approach taken in this chapter is unconventional. It will, at first glance, appear to be overly detailed, but there are good reasons for including so much detail. In order to understand the motivation behind this chapter, it is first necessary to review some of the motivation behind applicative programming languages. It is well-known that applicative languages have their roots in the theory of formal mathematical systems [Curry and Feys 1958]. In fact, both Floyd-Hoare semantics [Hoare and Wirth 1973] and Scott-Strachey semantics [Stoy 1977] can be viewed as attempts to marry the concept of a programming language to that of a formal mathematical system. In this respect, one advantage of an applicative language is that the bond between the language and a formal system is already strong, and the semantics of the language are thereby simplified.

One aspect of a formal mathematical system is that a number of things which are normally taken for granted must be formally defined. Some examples are the integers and other data objects, and the rules for substitution of variable names. In fact, in order to discuss any concept in the context of a formal system, that concept must be formally
defined within the system. This last idea is readily apparent for ordinary applicative programs as exemplified by the programs of Appendix B. Note that the languages used in Appendix B provide formal definitions for functions, scalars, structures, and so forth.

It is the contention of this chapter that the concept of shared data requires the introduction of the same type of formalism that is required for "local" computations. That is, "sharing" cannot be discussed until it has been formally defined. Actually, the concept of shared data is a complex one, and requires a number of other definitions as well. The general requirements will be derived from existing literature as far as possible.

There is a certain amount of work being done in the area of shared data support [Arvind, Gostelow and Plouffe 1977, Bryant and Dennis 1979, Plas et al. 1976]. One of the aims of this chapter is to provide a theoretical framework which will allow this work to be incorporated into the theory of functional languages. This is done by providing formal definitions for the concepts of message passing and simultaneous independent processes. Since a great deal of operating systems theory depends on the concept of message passing [Hoare 1978, Lauer and Needham 1978], these definitions should also allow a great deal of this theory to be
lifted intact into the theory of functional languages.

The primary purpose of this chapter is to provide a self-contained solution to the shared data problem. Beyond accomplishing this goal, no results can be claimed. Thus, the primary justification for this work is that a certain continuity of thought between these ideas and the underlying concepts of functional languages has been maintained. The applicability of the ideas of this chapter to real-world problems has yet to be investigated. Nevertheless, there is some reason to hope that the concepts presented here will provide a basis for unifying much of the existing work in shared-data and operating-system support in an applicative environment.

The remainder of this chapter is divided into eleven sections. Section B reviews existing literature for the purpose of deriving requirements for a functional shared data system. Section C presents sugared syntax for the FCL data manager. The FCL data manager will provide the theoretical definition of a shared data-element. Section D presents tentative syntax for accessing shared data in an application program. Section E presents the formal definition of an FCL process, as well as the other necessary formalism. The FCL process is the foundation upon which the rest of the chapter is built. Section F presents sugared syntax for declaring processes. Section G uses the
concept of processes to define a general process communication system. This section presents the formal concepts of simultaneous independent processes, and process communication. Sections H through K demonstrate the usefulness of the concepts presented in sections E, F, and G for defining a shared-data system. The material presented in these sections is not a serious proposal for a real system.

Section H presents specifications for a process communication system which is designed to support shared-data access. Section I presents a desugared form of a simple data manager and data-flow templates for use in an application program. Section J presents a more complicated data manager which allows interleaved access to portions of a shared data base. Section K presents data flow templates for use in a program which accesses an interleaved data base. Section L presents conclusions.

B. Literature Review and Derivation of Requirements

Many shared data systems for von Neumann machines are based on the concept of controlled access to physically shared storage [Brinch Hansen 1973, Dijkstra 1971, Gray 1978, Habermann 1975, Hoare 1974, Reed and Kanodia 1979]. A direct implementation of any of these systems in FCL is impossible because FCL has no concept of storage, shared or
otherwise. (This characteristic of FCL is shared with most data flow architectures.) Thus, a primary requirement for a functional shared data system is that it not depend on physically shared storage.

A number of researchers have proposed mechanisms for control of shared data which are based on the idea of critical regions [Brinch Hansen 1973, Dijkstra 1971, Reed and Kanodia 1979]. Ignoring the fact that these mechanisms assume the use of physically shared storage, these mechanisms are still not suitable for implementation in an applicative language. Consider a critical region which is protected by $P$ and $V$ operations on a semaphore $S$ [Dijkstra 1971]. Implementation of the critical region would look as follows:

\[
P(S) \\
:\text{(critical region)} \\
:\text{V}(S)
\]

Control of the shared resource depends on the fact that the statements within the critical region are executed before the $S$ operation and after the $P$ operation. In FCL, as in other applicative languages, the only ordering which can be defined on the execution of expressions is as follows: if the value of sub-expression $X$ is required for the evaluation
of sub-expression Y, then the evaluation of Y cannot precede the evaluation of X. Since none of the values computed within the critical region would typically depend on the value of S, the required sequencing could only be done by artificially introducing the value of S into the computations within the critical region. Such an artifice would probably be too unwieldy to be practical. It should also be noted that in computer architectures which support concurrency at the instruction level, the concept of a critical region becomes difficult to define. Thus, another requirement for a functional shared data system is that the control of shared data must not depend on the sequential execution of program statements.

The Hoare monitor does not explicitly depend on the concept of critical regions, but the details of this construct, and the way it is normally used to control access to shared data depend on the sequential execution of program statements [Hoare 1974]. However, the concept of a collection of modules which control access to a shared resource is adaptable to data-driven environments. A construct similar to the Hoare monitor has been developed for use on a data-flow machine [Arvind, Gostelow and Plouffe 1977]. This construct is called the data-flow monitor and contains many ideas which will be useful in developing an
FCL solution to the shared data problem. The data-flow monitor has the general form of a loop as shown in Figure 5.1.

![Diagram](https://via.placeholder.com/150)

Figure 5.1. The data-flow monitor

The data-flow monitor can be thought of as an independent process which is driven by a stream of request tokens, which are sent from one or more other independent processes. In response to this stream of inputs, the data-flow monitor produces a stream of output tokens which are eventually routed back to the requesting processes. The fact that a monitor can be thought of as an independent process allows the values controlled by the monitor to be global in scope and not bound to a particular application program. The loop-arcs provide a means of maintaining values over a period of time. Global values are typically implemented as values which are passed around the loop-arcs of the monitor. Since the new value which appears on a
loop arc depends on the action of the data-flow monitor body, global values are updatable. Thus, the data-flow monitor provides a means for "programming-in" updatable storage. (Existing data-flow architectures do not have updatable storage in the von Neumann sense.) Functional languages, and in particular FCL, are similar to data-flow architectures in that there is no automatic means for maintaining updatable values over an extended period of time. Thus, it is necessary that a functional solution to the shared data problem provide some means for maintaining updatable values over an extended period of time.

The concept of communicating independent processes has been shown to be of value in a wide variety of environments [Dijkstra 1971, Hoare 1978, Lauer and Needham 1978]. Because of this, it is desirable to base the FCL model of shared data on this concept. Therefore, it is necessary to have some formal definition of processes in FCL. In addition, it is necessary to have a formal definition of process communication in order to keep the FCL solution to the shared data problem self-contained.

To summarize the above discussion, the FCL solution to the shared data problem must meet the following requirements.
1. Implementation of global values must not depend on physically shared storage.
2. Control of shared data must not depend on the sequential execution of programs.
3. There must be some means for maintaining updatable global values over an extended period of time.
4. There must be some means for defining processes.
5. There must be some formal definition of process communication.

To simplify the syntax of application programs, the following additional requirement is imposed:

6. The functional form of FCL programs must be preserved.

Some work has been done in the area of shared data support in functional languages. This work has largely been confined to the investigation of nondeterminate constructs such as the AMB operator of LISP [McCarthy et al. 1962], the OR operator of Henderson [Henderson 1980], and the FRONS operator of Friedman and Wise [Friedman and Wise 1980]. While nondeterminate constructs do provide a powerful means for describing solutions to certain programming problems, they do not, of themselves, constitute a self-contained solution to the shared data problem. In fact, the solution presented in this chapter does not depend
on any nondeterminate construct. The inherent nondeterminacy of a shared data system is modeled through an interaction between various layers of determinate processes rather than through the use of explicit nondeterminate constructs. It may still be desirable to use nondeterminate constructs to write programs, but the system presented here allows the programmer to choose between determinate and nondeterminate constructs, rather than forcing him to use nondeterminate ones.

C. Data Manager Syntax

In order to provide motivation for the following sections, the high-level syntax for shared data access and definition will be presented before the details of process definition and communication.

A shared value is defined by declaring a data manager. A data manager is intended to be a separate process which will maintain a shared value and control access to it. The syntax of the data manager is given in Figure 5.2.

\[
\begin{align*}
\text{NAME} & \equiv \text{DATA_MANAGER} \\
\text{INIT} & = \text{<EXP>;} \\
\text{TYPE} & \\
\text{ENDDATA_MANAGER}
\end{align*}
\]

Figure 5.2. The syntax of the data manager
The data manager syntax given Figure 5.2 is a sugared construct which is usually written as a separate program. The element "<TYPE>" must denote a set, and is the type of the value controlled by the data-manager. The expression named "INIT" defines the initial value of the data-element controlled by the data-manager. The name of the data-manager is used by other programs to refer to the value controlled by it. Use of the data-manager name is explained fully in the next section. Figure 5.3 contains an example of a data manager which implements a shared integer.

\[
\text{SI} \equiv \text{DATA_MANAGER}
\]
\[
\text{INIT} = 0;
\]
\[
\text{INTEGER}
\]
\[
\text{ENDDATA_MANAGER}
\]

Figure 5.3. Example of a shared integer

The desugaring of the data-manager will be explained fully in sections I and J.

D. Application Program Syntax

The FCL application program syntax allows a program to read and update shared values without destroying the functional form of FCL programs. Shared values are read by including a declaration of the following form in the argument declaration of a program: \text{SHARED (<DATA MANAGER ID>)}.
This declaration is used in place of a type declaration. The following is an example of such a declaration:

\[ P \equiv A : \text{SHARED}("SI") \rightarrow A. \]

This program reads the value controlled by data manager "SI" and outputs it. The declaration of "SI" is given in Figure 5.3. In this example the type of "A" is implied by the type (INTEGER) of "SI".

The data manager id of the "SHARED" declaration has four forms which are given in Figure 5.4.

"<NAME>"

<NAME>

"<NAME>":<SET ID>

<NAME>;<SET ID>

Figure 5.4. The four forms of the data manager id

The element "<NAME>" must be a valid FCL name while the element "<SET ID>" must be a set-valued expression. When the element "<NAME>" is enclosed in quotes it must be the name of an independently defined data manager. When the element "<NAME>" is not enclosed in quotes, it represents a local name of type DATA-MANAGER. In this case the actual data manager name will be supplied at run-time. Figure 5.5 gives examples of each form of the data manager id.
A:SHARED("SI")\rightarrow A
Type of A is implied by declaration of "SI"

A:SHARED(SI)\rightarrow A
Type of A is ANY
Type of SI is DATA_MANAGER

A:SHARED("SI":INTEGER)\rightarrow A
Type of A is INTEGER
Value received from "SI" is type-checked

A:SHARED(SI:INTEGER)\rightarrow A
Type of A is INTEGER
Type of SI is DATA_MANAGER

Figure 5.5. Data manager id examples

In the second and fourth examples of Figure 5.5, the actual data-manager name will be supplied at run-time.

Shared values are updated by placing a sub-expression named SHARED in the application program. This name is a reserved word, and there may be at most one sub-expression named SHARED in the program. The form of the SHARED sub-expression must be <DM>\rightarrow<EXP>|...|<DM>\rightarrow<EXP>, where <DM> is either a data-manager name or a local name of type
DATA-MANAGER, and <EXP> is the new value of the shared data element controlled by the data-manager. The program of Figure 5.6 reads a shared integer, updates it, and outputs the new value.

\[
\text{UPDI} \equiv \text{I:SHARED(DMI:INTEGER)} + \\
(\text{NEW\_VAL}=I+5; \\
\text{SHAREDDMI} \rightarrow \text{NEW\_VAL}; \\
\text{NEW\_VAL})
\]

Figure 5.6. Example of shared value update

Note that in Figure 5.6, the output of the program is distinct from the updating of the shared value. Note also that the actual data manager name represented by DMI is to be supplied at run-time.

It is possible to update a shared value without first reading it. There are two ways of doing this. The first is to use a specific data manager name in the SHARED sub-expression while the second is to use a local name which is explicitly declared to be of type DATA-MANAGER. These options are illustrated in Figures 5.7 and 5.8, respectively.

\[
\text{INITSV} \equiv \text{IV:INTEGER} + \\
(\text{SHARED}="\text{INTDM}" \rightarrow \text{IV}; \\
"\text{INITIALIZATION DONE"})
\]

Figure 5.7. Initialization of a specific shared integer
Figure 5.8. Initialization of an arbitrary shared integer

The program of Figure 5.7 initializes the value of the data-manager named "INTDM" while the program of Figure 5.8 initializes the value of a run-time-supplied data manager. Figure 5.8 contains an example of an explicit data-manager name declaration. The form of such a declaration is DATA_MANAGER(<SET ID>) where <SET ID> is any set-valued expression. An explicit data-manager name declaration does not imply the reading of a value.

When data-manager names are supplied at run-time, the SHARED sub-expression must be checked to make sure there is no conflict between the new values assigned to a data-manager. To see why this is so, consider the following program:

\[
\text{UPDT} \equiv (A,B) : (\text{SHARED}(\text{DMA:INTEGER}), \text{SHARED}(\text{DMB:INTEGER})) \rightarrow (\text{SHARED} \equiv \text{DMA} \rightarrow A+5, \text{DMB} \rightarrow B-3; (A+5,B-3))
\]
If this program were invoked by the expression UPDT("HOURS", "HOURS"), a conflict would result in the new value for "HOURS". Since this sort of error cannot be detected at compile-time, there is a need for a run-time check for such conflicts.

Both application programs and data-managers run as independent processes. Shared values are read and updated through the use of process communication. Processes are defined in section E while process communication is explained in detail in section G.

At run-time, the application program consists of two layers: the computation layer and the communication layer. These layers are illustrated in Figure 5.9.

Figure 5.9. Layering of the application program
The communication layer accepts arguments from the external environment, performs whatever process communication is necessary, and passes arguments to the computation layer. The reverse procedure takes place when the computation layer produces its results. To illustrate, consider the following program:

\[
\text{EXAM} \equiv (A,B,C) : (\text{INTEGER}, \text{SHARED(DMA:INTEGER)}, \text{SHARED("DMB":INTEGER)}) \\
\rightarrow (A+B) \times C
\]

Within the computation layer, the program EXAM appears as a function requiring three integer arguments. However, within the communication layer the program appears as a function requiring one integer and one data-manager name argument (note that "DMB" is enclosed in quotes). The communication layer is used to resolve the discrepancy. The external environment will invoke EXAM with an expression such as EXAM (6,"INTDM") where INTDM is the name of an integer-valued data manager. The communication layer will pass the value "6" to the computation layer, and do the necessary process communication to obtain the current values of "INTDM" and "DMB". These values will also be passed to the communication layer. The net effect is that the computation layer "sees" three integer arguments.

When the computation layer completes execution it
produces a structure of the following type:
"SHARED"->FUNCTION | "OUTPUT"->ANY. The "SHARED" component of this structure contains the updated values for shared data elements while the "OUTPUT" component contains the ordinary output of the program. The value of the "OUTPUT" component will be passed to the external environment, while the "SHARED" component will be broken down and the new values will be sent to the proper data-managers. As an example, consider the following program:

\[
\text{UPDT} \equiv (A,B): (\text{SHARED}(\text{DMA: INTEGER}), \text{INTEGER})
\]

\[
\rightarrow (\text{SHARED DMA} \rightarrow A + B;
\]

\[
(A,B,A+B)).
\]

Suppose that the external environment invokes UPDT with the following expression: \text{UPDT("HOURS",8)}, and suppose the current value of the data-manager, HOURS, is 32. Then, the output of the computation layer would be:
"SHARED"->"DMA"->40 | "OUTPUT"->(32,8,40). The value 40 would be returned to the data-manager, HOURS, and the value (32, 8,40) would be passed to the external environment. The mechanism for making all this happen properly is discussed in section I.
E. FCL Processes and Timing

As stated above, the data-manager is a sugared form of an FCL process. The purpose of this section is to give a precise definition for the term "FCL process".

One accepted definition of a process is: "A process is a triple \((S,f,x)\) where \(S\) is a state space, \(f\) is an action function in that space, and \(x\) is a subset of \(S\) which defines the initial states of the process [Horning and Randall 1973]." This definition will be adopted with two modifications. The first is the inclusion of a process-communication mechanism, while the second is stylistic and is adopted to permit timing definitions to be made. When it is not clear from the context, these definitions will be distinguished by calling the object defined above an HR-process, and the new object an FCL-process.

In order to motivate the differences between the two definitions, it is necessary to give a formal introduction to the concept of timing. When timing is addressed in an applicative language, a typical approach is to define timing in terms of an interpreter for the language in question [Friedman and Wise 1980]. Since the FCL solution to the shared data problem is intended to be self-contained, a formal discussion of those characteristics of the interpreter which give rise to timing is required. In the
physical world, time can be defined to be a measure of events [Einstein 1955]. This definition has been adapted to provide a definition of timing for distributed processes [Lamport 1978]. Applying this adapted definition to the HR-process, one may define an event to be one application of the action function to the state variables of the process. Successive applications of the action function give rise to a sequence of states of the collection of state variables. The time between two events A and B can then be defined in terms of the number of intermediate versions of the state variables which exist between the version created by event A and the version created by event B.

This view of events can be expressed formally in FCL. A sequence is defined to be a function from the positive integers (x:INTEGER\(\times\)>0) to some set S. The set of sequences may be defined as follows: SEQ\(\equiv\)x:INTEGER\(\times\)>0\(\times\)ANY.

The formal definition of a sequence may then be used to introduce a formal definition of timing. Let S be a sequence and n be a positive integer. The pair \((S,n)\) is called an \underline{instant} in S, the triple \((S(n),S(n+1),n)\) is called an \underline{event} in S, and the pair \((S(n),n)\) is called a \underline{state} in S. An instant \((S,n)\) in S can be thought of as a reading of the clock of S. There are obvious mappings from
this set of "clock-readings" to the set of states in $S$ and
the set of events in $S$. The event $(S(n), S(n+1), n)$ is said
to happen at time $n$, while the state $(S(n), n)$ is said to
exist at time $n$. The time between two instants $(S, n)$ and
$(S, m)$ is defined to be equal to $m - n$. The event $(S(n),
S(n+1), n)$ is said to occur between instant $(S, i)$ and
instant $(S, j)$ iff $i \leq n < j$ or $j < n < i$. The absolute value of
the time between two instants is equal to the number
of distinct events which occur between them. Now, suppose
that the sequence $S$ is defined recursively as follows:
$S = 1\forall x : \text{INTEGER} \land x > 1 \rightarrow \text{F}(S(x-1))$. The events of $S$ are said
to be generated by the function $F$, since every event in $S$
may be written as $(S(n), F(S(n)), n)$. In turn, the instants
and states of $S$ are said to be generated by the events of
$S$. These definitions constitute a formal definition of
timing which will be used in connection with the FCL-
process.

As stated above, an event within an HR-process can be
considered to be an application of the action function to
the state variables of the process. This type of event
can be called a state transition. Since the FCL-process
will include the concept of process communication, it is
also necessary to consider input and output events. To
simplify matters, output events will be modeled as state
transitions which are visible outside the process. Input events differ from output events and state transitions in that input events are generated from outside the process while state transitions and output events are generated by an action function. To resolve this difference, input events and state transitions will be modeled by two different sequences: the input sequence and the state sequence.

A process specification is defined to be a 4-tuple \((TS, TI, ISV, ST)\) where \(TS\) is the set of potential state values, \(TI\) is the set of potential input values, \(ISV\) is the initial state value, and \(ST\) is the action or state transition function. \(ISV\) must be an element of the set \(TI\), and \(ST\) must be of the type \((TI,TS)\to TS\). The elements \(TS\), \(ISV\), and \(ST\) correspond, respectively, to the elements \(S\), \(x\), and \(f\) of the KR-process. The element \(TI\) has been added.

Given the process specification: \((TS, TI, ISV, ST)\), an FCL-process is defined by the following expression:

\[
P \equiv IS:(x:INTEGER \ x>0\to TI) \\
\to (SS\equiv1\to ISV|y:INTEGER \ y>1\to ST(IS(y-1), \ SS(y-1)))
\]

The form of an FCL-process is a function from a sequence of inputs to a sequence of states. Let \(P\) be the process defined by the process definition \((TS, TI, ISV, ST)\). Then
TS is called the state-type of P, TI is called the input-type of P, ISV is called the initial-state-value of P, and ST is called the state-transition-function of P. If P is a process, and IS is a sequence of elements of the input-type of P, then the pair (P,IS) is called an instance of the process P. Given an instance of P, (P,IS), P(IS) is a sequence of elements of the state-type of P. The definitions of instant, event, and state may be applied to P(IS) and define timing within the instance of P.

Examples of FCL-processes and their use are given in the next section. The FCL-process is the basic building block of the FCL solution to the shared data problems.

F. Process Syntax

The syntax for declaring FCL-processes is a sugared form of the process specification. Its general form is given in Figure 5.10.

```
PROCESS
  INPUT=<input-type specification>;
  STATE=<state-type specification>;
  INIT=<Initial-state-value specification>;
  OUTPUT=<output specification>;
  <state-transition-function specification>
ENDPROCESS
```

Figure 5.10. FCL-process declaration
The input-type and state-type specifications must be set-valued expressions. The state-type specification must denote a set of functions. The initial-state-value specification must denote an element of the state-type. The output specification, which is not part of the formal process specification, is used to define what part of the process-state is visible outside the process. This specification must be a literal which denotes an element which is contained in the domain at every element of the state-type. If the output specification is omitted, the entire process-state is visible outside the process. The use of this specification is demonstrated below. The state transition function specification is subject to extensive desugaring which allows a simplified notation to be used. Figure 5.11 gives an abstract process definition. The state transition function specification of this abstract definition is desugared in Figure 5.12.

```plaintext
PROCESS
    INPUT=TI;
    STATE=TS;
    INIT=PSV;
    OUTPUT="X";
    STEXP
ENDPROCESS
```

Figure 5.11. Abstract process definition
Note that in Figure 5.11, TS must denote a set of functions.

\[
(\text{INPUT, STATE}): (\text{TI, TS}) \rightarrow \\
\begin{array}{l}
\text{IF INPUT=} \text{NULL THEN} \\
"X" \rightarrow \text{NULL} | \text{STATE} \\
\text{ELSE} \\
\text{STEXP} | "X" \rightarrow \text{NULL} | \text{STATE} \\
\text{ENDIF}
\end{array}
\]

Figure 5.12. The desugaring of the state-transition-function definition

The meaning of the name NULL used in Figure 5.12 will be explained below. The first line of Figure 5.12 implies that the keywords STATE and INPUT may be used within the expression STEXP to refer to the previous state and previous input, respectively. The last line of Figure 5.12 implies that the value of the "X" element of the state is set to NULL if it is not specified by the expression STEXP. This line also implies that any other elements of the state whose values are not specified by STEXP will retain their old values. The second and third lines of Figure 5.12 imply that if the input to a process is NULL then the output will be NULL and the state will remain unchanged. The reason for this part of the desugared expression will be explained below.

The process syntax is illustrated in Figure 5.13. This
example implements an updatable one-word memory.

\[
\text{ONE-WD-MEM} = \\
\text{PROCESS} \\
\text{INPUT} \equiv \langle \text{SET ("R", "W") | NV+INTEGER} \rangle; \\
\text{STATE} \equiv \langle \text{VAL+INTEGER | OUT+INTEGER} \rangle; \\
\text{INIT} \equiv \langle \text{VAL+0 | OUT+NULL} \rangle \\
\text{OUTPUT} \equiv \langle \text{OUT} \rangle; \\
\text{IF INPUT.T} = \langle \text{"R"} \rangle \text{ THEN} \\
\quad \text{OUT+STATE.VAL} \\
\text{ELSE} \\
\quad \text{VAL+INPUT.NV} \\
\text{ENDIF} \\
\text{ENDPROCESS}
\]

Figure 5.13. Example of a process

The "VAL" component of the state of the process of Figure 5.13 contains the value of a one word memory, while the "OUT" component represents the output port of the memory. When the "T" component of the input is equal to "R" the value of the memory is presented to the output port and the internal value is retained. When the "T" component is "W" then the old value of the memory is replaced, and the value NULL is presented to the output.

The value NULL is an implicit member of every set which represents the absence of an input or an output.
Its main use is in the definition of sub-processes. The data-flow structure of a main process with two sub-processes is given in Figure 5.14.

![Diagram of sub-process structure]

**Figure 5.14.** Sub-process structure

In some cases it may be desirable to send a token to only one sub-process. Since the FCL code for part 1 of the main process of Figure 5.14 must be functional, it must supply an input value for both processes every time it receives an
input. By supplying the value NULL, the high-level code can specify that no input is to be supplied to a particular sub-process. The desugaring of the state-transition function of a process guarantees that the action of a sub-process upon receipt of a NULL input conforms to the expected action in a data-flow graph when no input is received (i.e., no action).

Sub-processes are specified through the use of the SUBPROCESS function. Figure 5.15 demonstrates the use of the SUBPROCESS function. This example implements a two-word random access memory.

The first argument of the SUBPROCESS function must identify a process and the second identifies the value of the input part of the sub-process. The SUBPROCESS function is normally specified as follows:

<NAME> ≜ SUBPROCESS(<PROCESS ID>,<EXP>). This expression must be specified as a sub-expression of the state-transition-function specification.

A process containing sub-processes is desugared into a single process as illustrated in Figure 5.16.
TWO_WD_MEM=

(WORD=PROCESS
 INPUT=T⇒SET("R","W") | NV⇒INTEGER;
 STATE=VAL⇒INTEGER | OUT⇒INTEGER;
 INIT=VAL⇒0 | OUT⇒NULL;
 OUTPUT=\"OUT\";
 IF INPUT.T="R" THEN
  OUT⇒STATE.VAL
 ELSE
  VAL⇒INPUT.NV
 ENDIF
ENDPROCESS

PROCESS
 INPUT=ADR⇒1..2 | TYPE⇒SET("R","W") | NV⇒INTEGER;
 STATE=OUT⇒INTEGER;
 INIT=OUT⇒NULL;
 OUTPUT="OUT";
 WORD1=SUBPROCESS(WORD,SPIN.12);
 WORD2=SUBPROCESS(WORD,SPIN.12);
 SPIN=IF INPUT.ADR=1 THEN
   I1⇒(T⇒INPUT.TYPE | NV⇒INPUT.NV)
   I2⇒NULL
 ELSE
   I1⇒NULL |
   I2⇒(T⇒INPUT.TYPE | NV⇒INPUT.NV)
 ENDIF;
 OUT⇒IF INPUT.ADR=1 THEN
   WORD1
 ELSE
   WORD2
 ENDIF

ENDPROCESS

Figure 5.15. Example of a sub-process
Before desugaring:

\begin{verbatim}
P=PROCESS
  INPUT=TIP;
  STATE=TSP;
  INIT=ISVP;
  OUTPUT="OUTP";
  STP
ENDPROCESS
Q=PROCESS
  INPUT=TIQ;
  STATE=TSQ;
  INIT=ISVQ;
  OUTPUT=OUTQ;
  SP=SUBPROCESS(P,SPINEXP);
  STQ
ENDPROCESS
\end{verbatim}

After desugaring:

\begin{verbatim}
Q=PROCESS
  INPUT=TIQ;
  STATE=TSQ|"@SP"->TSP;
  INIT=ISVQ|"@SP"->ISVP;
  OUTPUT="OUTQ"
  SP=STP*(SPINEXP,STATE("@SP"))
  STQ|"@SP SP
ENDPROCESS
\end{verbatim}

Figure 5.16. The desugared form of a sub-process
In Figure 5.16, "@SP" represents the string "SP" with a prefix added to insure the resultant string is not in the domain of any element of TSQ. STP* represents the desugared form of the expression STP, and STQ* represents the expression STQ with each free occurrence of SP replaced by SP("OUTP"). The resultant single process is then subject to the standard process-desugaring described above.

A sub-process specification may be parameterized. This allows the programmer to specify a parameterized collection of subprocesses with similar characteristics. Figure 5.17 gives an example of a parameterized sub-process and its desugared form.

It is also possible to define a function which returns a process as its value. This, effectively, allows the programmer to define a parameterized family of processes. The process BIG_MEM of Figure 5.17 can be converted to a parameterized family of processes by changing the first line to VAR_MEM=SIZE:INTEGER\SIZE>1^(BIG_MEM=, replacing the occurrences of 1000 with SIZE on lines 17 and 20, and adding an additional closing paren at the end. Thus, converted, the function VAR_MEM could be used to define a memory of any size.

The examples given above were presented for illustrative
Before desugaring:

BIG_MEM =

(WORD = PROCESS
   INPUT = \( T \rightarrow \text{SET}\) ("R", "W") | \( \text{NV} \rightarrow \text{INTEGER}\);
   STATE = \( \text{VAL} \rightarrow \text{INTEGER}\) | \( \text{OUT} \rightarrow \text{INTEGER}\);
   INIT = \( \text{VAL} \rightarrow 0 | \text{OUT} \rightarrow \text{NULL}\);
   OUTPUT = "OUT";
   IF INPUT.T = "R" THEN
      OUT = STATE.VAL
   ELSE
      VAL = INPUT.NV
   ENDIF
ENDPROCESS;

PROCESS
   INPUT = ADR \rightarrow \text{INTEGER}
      T \rightarrow \text{SET}\) ("R", "W") |\n      \( \text{NV} \rightarrow \text{INTEGER}\);
   STATE = \( \text{OUT} \rightarrow \text{INTEGER}\);
   INIT = \( \text{OUT} \rightarrow \text{NULL}\);
   OUTPUT = "OUT";
   REGS = x: (1..1000) \rightarrow \text{SUBPROCESS} (\text{WORD}, \text{SPIN}(x));
   SPIN = INPUT.ADR \rightarrow (T \rightarrow \text{INPUT.T} | \( \text{NV} \rightarrow \text{INPUT.NV}\) |\
   \(1..1000) \rightarrow \text{NULL};
   OUT = REGS (INPUT.ADR)
ENDPROCESS

Figure 5.17. Example of a parameterized sub-process
After desugaring:

PROCESS

INPUT=ADR->INTEGER | T->SET("R","W") | NV->INTEGER;
STATE=OUT->INTEGER |
"@REGS"->1..1000->(VAL->INTEGER)
OUT->INTEGER);
INIT=OUT->NULL |
"@REGS"->1..1000->(VAL=0|OUT=NULL);
OUTPUT="OUT";
REGS=
 (INPUT,STATE): (T->SET("R","W") | NV->INTEGER,
 VAL->INTEGER | OUT->INTEGER)

IF INPUT=NULL THEN OUT=STATE
ELSE IF INPUT.T="R" THEN OUT=STATE.VAL
ELSE VAL=INPUT.NV|OUT=STATE
ENDIF
ENDDIF
ENDPROCESS

Figure 5.17 (Continued)
purposes only. The actual long-term maintenance of updatable shared values will be discussed in the next five sections.

G. A General Process Communication System

The previous section presented a primitive model of process communication, however, the primitive form of communication provided by the sub-process construction does not provide the dynamic behavior needed to model a dynamic shared data system. Some method is needed to initiate and terminate processes, and provide dynamic communication between them.

The FCL process will be used as the basic mechanism for describing general process communication. The communication system itself will be defined as an FCL-process with several subprocesses. The general form of a process communication system is given in Figure 5.18.

In Figure 5.18, the elements labeled "P1" through "Pn", and "communication system" are sub-processes of the process-communication system. The sub-processes P1 through Pn are called execution processes and are responsible for executing programs while the communication system process is a distinguished process which is responsible for maintaining communications between two or more running programs, and between programs and the outside world. The
Figure 5.18. Process communication system
communication system is also responsible for creating new processes.

Figure 5.19 defines a family of process-communication systems. In this figure, VECTOR is the function: VECTOR→(S,N):(TYPE,INTEGER)→(1..N→S). The meanings of the parameters of PCS are given in Figure 5.20.

The definition of PCS implies the following restrictions on the parameters EXEC-CODE and COMMUNICATION-SYST. COMMUNICATION-SYST must be of the form

\[(VECTOR(OS),VECTOR(INPUT-PORT,NIP),
\]
\[\text{COMM-SYST-INTERNAL-STATE})\rightarrow
\]
\[\text{(OUTPUT→VECTOR(OUTPUT-PORT,0) |}
\]
\[\text{PROCESS-COMM→VECTOR(PROCESS-INPUT) |}
\]
\[\text{INT-STATE→COMM-SYST-INTERNAL-STATF).}
\]

EXEC-CODE must be of the form

\[\text{UNION((PROCESS-INPUT,PROCESS-STATE),}
\]
\[1→\text{PROCESS-INPUT,2→PROCESS-STATE)→}
\]
\[\text{PROCESS-STATE.}
\]

The ordinary input to EXEC-CODE is a vector of the form (PROCESS-INPUT, PROCESS-STATE). An input of the form 1→PROCESS-INPUT signifies the creation of a new process, while an input of the form 2→PROCESS-STATE signifies the continuing execution of a process with no new input.

A NULL input to a process communication system will not
PCS ≡ (INPUT_PORT, NIP, PROCESS_STATE, PROCESS_INPUT, OUTPUT_PORT, NOP, COMM_SYST_INTERNAL_STATE, EXEC_CODE, COMMUNICATION.SYST):
(TYPE, INTEGER, OUT^TYPE FUNCTION, TYPE, TYPE, INTEGER, TYPE, FUNCTION, FUNCTION) →

PROCESS

INPUT ≡ VECTOR(INPUT_PORT, NIP);
STATE ≡ PV VECTOR(PROCESS_STATE) |
PIV VECTOR(PROCESS_INPUT) |
OV VECTOR(OUTPUT_PORT, NOP) |
CIS COMM_SYST_INTERNAL_STATE;
INIT ≡ PV NULL | PIV NULL | OV NULL | CIS NULL;
OUTPUT ≡ "OV";
(NEW_COMM = COMMUNICATION_SYST(STATE.PV("OUT"),
INPUT, STATE.CIS);
PV EXEC_CODE[STATE.PIV, STATE.PV] |
OV NEW_COMM.OUTPUT |
PIV NEW_COMM.PROCESS_COMM |
CIS NEW_COMM.INT_STATE)
ENDPROCESS

Figure 5.19. Definition of a family of process communication systems
INPUT-PORT a set which defines the type of data received from the system input ports

NIP an integer which specifies the number of input ports

PROCESS-STATE a set which specifies the type of the internal state of the execution processes

PROCESS-INPUT a set which specifies the type of inputs received by execution processes

OUTPUT-PORT a set which defines the type of data passed to output ports

NOP an integer which specifies the number of output ports

COMM-SYST-INTERNAL-STATE a set which specifies the type of the internal state of the communication system

EXEC-CODE a function which defines the execution of programs

COMMUNICATION-SYST A function which defines process communication

Figure 5.20. Process communication system parameters
necessarily result in no change to its internal state. This is reasonable, since programs typically take several time units to execute without outside interaction. A NULL input to a process communication system will be a vector of the form 1..NIP->NULL. This object will not compare equal to NULL, so the standard desugaring of process definitions will not interfere with the expected behavior of the process communication system.

H. Specifications for a Shared-data Process Communication System

The process communication system presented in the previous section is too general to be of use in describing a shared data system. This section will suggest some specific forms for the parameters of a process-communication-system to make it suitable for handling the shared-data constructs presented in sections B and C. These specifications are made primarily for illustrative purposes.

INPUT_PORT and OUTPUT_PORT are defined to be ANY. NIP and NOP are left arbitrary. PROCESS_INPUT is of the form VECTOR(MSG), where MSG is the set FROM^ (PID^ ANY | MID^ ANY) | TO^ (PID^ ANY | MID^ ANY) | TYPE=STRING | DATA=ANY. PROCESS_STATE is of the type PROC_INFO^ ANY | INT_CODE=ANY | OUT^ VECTOR(MSG), where MSG is as above. COMM_SYST_INTERNAL_STATE is of the type INPUT_MAP^ ANY | OUTPUT_MAP^ ANY | PROC_NAME=
MAP=ANY|OTHER=ANY. Specifications for EXEC_CODE and COMMUNICATION_SYST will be given below.

The basic form of process communication is the message whose form has been described above to be:

\[
\text{FROM} = \text{\{PID=ANY|MID=ANY\}} | \\
\text{TO} = \text{\{PID=ANY|MID=ANY\}} | \\
\text{TYPE=STRING} | \\
\text{DATA=ANY}
\]

The component PID of the TO and FROM components of a message is a process name. This may be in the form of a high level name (in which case it will be a string) or in the form of a process number (in which case it will be an integer). The process number indicates the position of a processes internal state within the process state vector. The MID component of the TO and FROM components is a process-specific value which identifies the message. This component will be discussed in greater depth below. The TYPE component of a message identifies the message's function. This component may have one of the values "INPUT", "OUTPUT", "REPLY", "COMM", "STATUS", or "COMMAND". These types of messages will be discussed in connection with the specifications for the EXEC_CODE and COMMUNICATION_SYST parameters. The DATA component of a message is dependent on the process, and will be discussed below.
As stated above, the process state has the form 
PROC_INFO→ANY|INT_CODE→ANY|OUT→VECTOR(MSG). The component PROC_INFO is used to maintain process-status information which will include the type of program being executed (data-manager or application program), status of communications in progress, and other information. The component INT_CODE contains the code of the program currently being executed. The OUT component is used to pass messages into the communication system.

COMM_SYST_INTERNAL_STATE has the form INPUT_MAP→ANY|OUTPUT_MAP→ANY|PROC_NAME_MAP→ANY|OTHER→ANY. The components INPUT_MAP and OUTPUT_MAP maintain lists of permanent associations between processes and input and output ports. The component PROC_NAME_MAP maintains a list of associations between high-level process names and process numbers. The OTHER component contains messages in progress.

The primary function of the EXEC_CODE function is to execute data-flow programs. This will be accomplished through the use of two functions named EXEC_ONE_STEP and EXEC_COMPLETE. Both of these functions will be implemented in the data flow machine hardware. Details of these functions will not be presented. EXEC_CODE will not be allowed to look at or modify the state component INT_CODE except through the use of these functions, with the exception that EXEC_CODE is allowed to place data on input arcs and inspect
output-arcs for data. EXEC_CODE will maintain a list of output arcs for a program in progress. When a token appears on one of these arcs, EXEC_CODE will convert it to an appropriate message and pass it to the communication system. EXEC_CODE will also accept messages and after appropriate processing, will pass them to the appropriate input arc of the program.

If the program being executed is a data manager, or a high-level process, then messages of the types "COMM", "INPUT", "REPLY", and "OUTPUT" will be passed intact to the program. Similarly, the output of a data manager or high-level process will be passed intact to the communication system.

If the program being executed is an application program, messages of the type "OUTPUT", "INPUT", and "COMM" will be treated as argument specifications while messages of type "REPLY" will be treated as responses from data managers. In all four cases, the DATA component of a message will be stripped out and passed to the program.

Messages of the types "COMMAND" and "STATUS" are used for system communication and do not contain program data.

When the COMMUNICATION_SYST function receives a message from a process, it decodes the TO component and either sends the whole message to the appropriate process, or sends the data component to the appropriate output port.
COMMUNICATION SYST will also create messages out of data received from input ports. The components INPUT_MAP, OUTPUT_MAP, PROC_NAME_MAP are used in this process.

It is assumed that certain distinguished processes are used to perform operating system type functions, such as maintaining program libraries, and allocating resources. The following example illustrates process communication. This example involves communication with a data manager named "MONQ". Data manager communication will be discussed in greater depth in the next section. Suppose P is defined as follows:

\[ P \equiv (I,J) : (\text{INTEGER,SHARED(DM:INTEGER)}) \rightarrow I+J \]

Suppose further that P is included in the system program library. It is desired to execute P with the parameters \( P(5,"\text{MONQ}" \) and return the output to port 7. Suppose the "JCL" for such a request is:

```
EXEC P(5,"MONQ") TO PORT (7)
```

This JCL is received through input port 3. The following message traffic might take place:

```
1: TO=\(\text{PID=\"JCL DECODER\"} \mid \text{FROM=\(\text{PID=\"COMM-SYST\", MID=PORT(3)} \mid \text{TYPE=\"INPUT\"} \mid \text{DATA=\"EXEC P(5,"MONQ") TO PORT(7)"}
```
2:  TO→PID→"PROCESS-ALLOCATOR" | 
    FROM→PID→"JCL DECODER" | 
    TYPE→"COMM" | 
    DATA→(PGM→"P" | 
      PARMS→(5, "MONQ") | 
      OUTPUT→PORT(7) | 
      FROM→PORT(3))

3:  TO→PID→6 | 
    FROM→PID→"PROCESS-ALLOCATOR" | 
    TYPE→"COMMAND" | 
    DATA→(ACTION→"START PGM" | 
      TEXT→(PGM→"P" | 
        PARMS→(5, "MONQ") | 
        OUTPUT→PORT(7) | 
        FROM→PORT(3)))

4:  TO→PID→"PROGRAM-LIBRARY" | 
    FROM→(PID→6 | MID→PGMREQ) | 
    TYPE→"COMM" | 
    DATA→(REQ→"READ" | 
      ID→"P")

5:  TO→(PID→6 | MID→PGMREQ) | 
    FROM→PID→"PROGRAM-LIBRARY" | 
    TYPE→"REPLY" 
    DATA→<Program text of P>
6:  TO\(\to\)PID\(\to\)"MONO" I
    FROM\(\to\)PID\(\to\)6 | MID\(\to\)<Arc-id> I
    TYPE\(\to\)"COMM" I
    DATA\(\to\)(T\(\to\)"READ-SHARED" | N\(\to\)UNDEFINED)

7. TO\(\to\)(PID\(\to\)6 | MID\(\to\)<arc-id>) | FROM\(\to\)PID\(\to\)"MONO" |
   TYPE\(\to\)"REPLY" |
   DATA\(\to\)8

8:  TO\(\to\)(PID\(\to\)"COMM-SYST" | MID\(\to\)PORT(7)) | FROM\(\to\)PID\(\to\)6 |
     TYPE\(\to\)"OUTPUT" |
     DATA\(\to\)13

9:  TO\(\to\)PID\(\to\)"PROCESS-ALLOCATOR" |
     FROM\(\to\)PID\(\to\)6 |
     TYPE\(\to\)"STATUS" |
     DATA\(\to\)"IDLE"

This example is not meant to be a serious proposal for operating system structure, but is merely intended to illustrate the various types of messages which could be passed through the message communication system.

The details of the structure of messages 6 and 7 will be given in the next section.
The purpose of this section is to provide mechanisms for guaranteeing data consistency. The definition of consistency used in this section is due to Grey [Grey 1978] and can be roughly described as an absence of destructive interaction between programs which access shared data. It has been shown that a two-phase lock protocol is sufficient to guarantee this type of consistency [Grey 1978]. This section will demonstrate a two-phase lock protocol for shared data access in FCL. This two-phase lock protocol arises from the interaction between the data manager, and the communication layer of an application program.

\begin{verbatim}
DATAMANAGER :: (T,IV):(SET,ANY)→
PROCESS
  MSG=TO→ANY|
  FROM→ANY|
  TYPE→STRING;
IMSG=MSGPLITDATA→(REQ→SET("READ-SHARED",
                          "READ-EXCLUSIVE",
                          "RS-DONE",
                          "RX-DONE");
\end{verbatim}

Figure 5.21. The DATA-MANAGER function
OMSG\equiv MSG \mid DATA\equiv T;
INPUT\equiv IMSG;
STATE\equiv VAL\equiv T|
SACT\equiv INTEGER|
XCOUNT\equiv INTEGER|
Q\equiv (SIZE\equiv INTEGER|
B\equiv INTEGER|
E\equiv INTEGER|
LIST\equiv VECTOR(IMSG))|
OUT\equiv VECTOR(OMSG); .
INIT\equiv VAL\equiv IV|
SACT\equiv 0|
XCOUNT\equiv 0|
Q\equiv (SIZE\equiv 0|
B\equiv 0|
E\equiv 0|
LIST\equiv UNDEFINED)|
OUT\equiv NULL;
IF INPUT\cdot DATA\cdot REQ=\"READ-SHARED\" THEN
IF STATE\cdot XCOUNT>0 THEN
Q\equiv (SIZE\equiv STATE\cdot Q\cdot SIZE+1|
E\equiv STATE\cdot Q\cdot E+1|
LIST\equiv (STATE\cdot Q\cdot E+1\cdot INPUT\mid STATE\cdot Q\cdot LIST)|
STATE\cdot Q)
Figure 5.21 (Continued)
ELSE

    OUT\arrow{1}{FROM=INPUT.TO}|
    TO\arrow{INPUT.FROM}|
    TYPE\"REPLY"|
    DATA\STATE.VALUE|\n
    SACT\STATE.SACT+1 ENDIF

ELSE IF INPUT.DATA.REQ="READ-EXCLUSIVE" THEN

    IF STATE.Q.SIZE\neq0 THEN

        XCOUNT\STATE.XCOUNT+1|\n
        Q\arrow{(SIZE=STATE.Q.SIZE+1|\n
                E\STATE.Q.E+1|\n
                LIST\STATE.Q.E+1\INPUT|STATE.Q.LIST|\n
                STATE.Q)\n
    ELSE

        XCOUNT\STATE.XCOUNT+1|\n
        OUT\arrow{1}{FROM=INPUT.TO}|
        TO\arrow{INPUT.FROM}|
        TYPE\"REPLY"|
        DATA\STATE.VALUE) ENDIF

ELSE IF INPUT.DATA.REQ="RS-DONE" THEN

    RS\STATE.SACT-1|\n
    (IF STATE.SACT=1 THEN

        IF STATE.XCOUNT>0 THEN

            Figure 5.21 (Continued)
\[ Q = ( \text{SIZE} \rightarrow \text{STATE}.Q.\text{SIZE}-1) | \]
\[ B = \text{STATE}.Q.B+1 | \]
\[ \text{STATE}.Q( \] \]
\[ \text{OUT} \rightarrow \]
\[ 1 \rightarrow ( \text{TO} \rightarrow \text{STATE}.Q.\text{LIST}(\text{STATE}.Q.B).\text{FROM} | \]
\[ \text{FROM} \rightarrow \text{STATE}.Q.\text{LIST}(\text{STATE}.Q.B).\text{TO} | \]
\[ \text{TYPE} \rightarrow "\text{REPLY}" | \]
\[ \text{DATA} \rightarrow \text{STATE}.\text{VAL}) | \]
\[ \text{ELSE} \]
\[ \text{STATE ENDIF} \]
\[ \text{ELSE} \]
\[ \text{STATE) ENDIF} \]
\[ \text{ELSE IF INPUT.DATA.REQ="RX-DONE" THEN} \]
\[ \text{XCOUNT} \rightarrow \text{STATE}.XCOUNT-1 | \]
\[ \text{VAL} \rightarrow \text{INPUT.NV} | \]
\[ \text{IF STATE.Q.SIZE}=0 \text{ THEN} \]
\[ \text{STATE} \]
\[ \text{ELSE} \]
\[ (\text{IF STATE.Q.LIST(STATE.Q.B).DATA.REQ= "READ-EXCLUSIVE" THEN} \]

Figure 5.21 (Continued)
OUT→1
(TO→STATE.Q.LIST STATE.Q.B).FROM|
FROM→STATE.Q.LIST(STATE.Q.B).TO|
TYPE→"REPLY"|
DATA→INPUT.NV)
Q→(SIZE→STATE.Q.SIZE-1|
B→STATE.Q.B+1|
STATE.Q)
ELSE
(ER=WILE(IC=E AND STATE.Q.LIST(I).DATA.REQ=
"READ-SHARED");
INIT=I→STATE.Q.B;
RESULT=I;
I→I+1
ENDWHILE;
LR=ER-STATE.Q.B;
BR=STATE.O.B-1;
OUT→x:(1..LR)→(TO→STATE.Q.LIST(BR+x).FROM|
FROM→STATE.Q.LIST(BR+x).TO|
TYPE→"REPLY"|
DATA→INPUT.NV)
Q→(SIZE→STATE.Q.SIZE-LR|
B→STATE.Q.B+LR|
STATE.Q))ENDIF ENDIF ELSE
STATE ENDIF ENDIF ENDIF ENDIF ENDPROCESS
Figure 5.21 (Continued)
The processes produced by the DATA MANAGER function are called simple data-managers because they do not provide for the interleaving of data base transactions. The problem of interleaving will be addressed in the next two sections. The simple data manager schedules the data it controls as a single unit. The desugaring of the data manager is a first in-first out solution to the readers and writers problem. Other scheduling algorithms may be written explicitly using the PROCESS construct. This data manager is able to process four different types of requests: READ-SHARED, READ-EXCLUSIVE, RS-DONE, and RX-DONE.

A READ-EXCLUSIVE request asks the data manager for a value, and prevents any other use of the data until the data-manager receives an RX-DONE request. A READ-SHARED request prevents any READ-EXCLUSIVE request from being granted until an RS-DONE request is received. Granting a request may sometimes be called setting a shared or exclusive lock. Similarly, processing an RS-DONE or RX-DONE request will sometimes be called releasing a lock. The DATA-MANAGER function will be used to desugar the data manager construct. Consider the following data-manager:
SI = DATAMANAGER
INIT=0;
INTEGER
ENDATA_MANAGER

This construct would be desugared as follows:

SI = DATA_MANAGER(INTEGER,0).

At compile time, an analysis of the data-manager names of an application program is done to determine what types of requests must be issued by the communication layer of the program. Figure 5.22 gives a decision table which is based on the results of the analysis.

<table>
<thead>
<tr>
<th>Declaration type</th>
<th>Does name appear in SHARED COMPONENT?</th>
<th>ACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit</td>
<td>Yes</td>
<td>1</td>
</tr>
<tr>
<td>Explicit</td>
<td>No</td>
<td>2</td>
</tr>
<tr>
<td>Implicit</td>
<td>Yes</td>
<td>3</td>
</tr>
<tr>
<td>Implicit</td>
<td>No</td>
<td>4</td>
</tr>
</tbody>
</table>

**Actions:**

1. Send a Dummy "READ-EXCLUSIVE" request before the program begins, but throw the value away. Send an "RX-DONE" when the SHARED component is completed.

2. Gate the name out of existence; it will never be used.

Figure 5.22. Actions performed by an application program
3. Send a "READ-EXCLUSIVE" request before the program begins. Send an "RX-DONE" when the SHARED component is completed.

4. Send a "READ-SHARED" request before the program begins. Send an "RS-DONE" when the SHARED component is complete.

Figure 5.22 (Continued)

The data flow templates for the above actions are given in Figure 5.23.

The structures which are passed to EXEC-CODE for transmittal to a data manager all have the basic form:

TO^<D.M. Name>|<other values>

EXEC_CODE will remove the "TO" component, construct a message which is acceptable to the communication system, and append the remainder of the structure into the message with the selector "DATA". EXEC_CODE maintains a list of REQUEST-REPLY arc correspondences. The FROM component of EXEC_CODE's output message will be constructed thus: FROM^<PID^<Proc. ID>|MID^<arc id>) where <arc id> identifies the REPLY arc which is to receive the data component of the REPLY message. Data manager processing will preserve the content of the MID component, so returned values may be routed to the correct arc.

Note the presence of nodes labeled "build ARG structure" in Figure 5.23. This is designed to enforce the two-phase
Figure 5.23. Data flow templates for data-manager communication
Action 2:

D.M. name

Action 3:

D.M. name  

D.M name  

append "TO"

"read-exclusive"  

Append "REQ"

To EXEC-CODE

From EXEC-CODE

Build Arg structure

Build "SHARED" component

Build main expression

D.M. name  

select

"RX-DONE"

To EXEC-CODE

Figure 5.23 (Continued)
Figure 5.23 (Continued)
lock protocol. All shared values are built into a structure, and then selected out as needed. If this were not done the following situation could arise:

Program P reads A and B, and updates B. The values of both A and B are needed to construct the main expression of P, but only the value of B is needed to construct the SHARED component. If the values of A and B were not built into a single structure, the "RX-DONE" message could be processed before the "READ-SHARED" request for A is granted. This is a clear violation of the two-phase lock protocol, and the mutual consistency of the values of A and B cannot be guaranteed. In the simple monitor, it is not necessary to build the entire SHARED structure before sending any "RX-DONE" or "RS-DONE" messages, but this is done to maintain compatibility with the interleaving data manager presented in the next section.

At this point it is possible to show that all of the requirements given in section A have been met:

1. The solution must not depend on physically shared storage.

Section H makes it clear that the solution proposed in this chapter depends on message passing and not on shared storage.
2. It must not depend on the sequential execution of programs.

   Section D makes it clear that the high-level syntax does not depend on sequential program execution. The mechanisms proposed in this section make it clear that the low-level implementation does not depend on sequential execution.

3. It must provide a means of maintaining updatable values over a period of time.

   The data manager syntax of section C provides such a mechanism, as does the more detailed syntax of processes presented in section F.

4. It must provide some means for defining processes.

   Sections E and F provide the details of FCL processes.

5. It must provide mechanisms for process communication.

   Details of process communication are given in sections G, H, and I.

6. The functionality of FCL programs must not be destroyed.

   Section D demonstrates that application program syntax is functional. Section E demonstrates that FCL-processes are functional. This section demonstrates that the low level implementation of application programs is functional even to the extent of maintaining a primitive form of data consistency.
J. Specifications for an Interleaved Data Manager

The data manager described in the last section is sufficient if the shared values are scalars or relatively small structures. However, if a large data-base is under the control of a single data manager, the simple data-manager will force a high degree of serialization on data-base transactions which may not be acceptable [Grey 1978]. For example, consider an airline reservation system. It is reasonable to expect that two transactions wishing to update the status of a single flight will be serialized. But it is unreasonable to force the serialization of two transactions which update the status of two different flights. The simple data manager will, in fact, cause just such a serialization. One solution would be to implement the data base using several simple data-managers. However, if a transaction is allowed to select the components of the database it wishes to update, this solution is unacceptable. Because the transaction could potentially update any component of the database, all components must be made available to the program. This would require a change to the program every time a new component (with its own data manager) was added to the data base.

To address this problem, this section will give
specifications for a data manager which allows interleaved updates on a single data base. The data base will appear as a single structure to the program. The program is allowed to examine this structure and make changes to it. At the implementation level, the program processes a structure which contains only the information required by the program, and creates a new local copy of the data base which also contains only part of the information contained in the actual data base.

At the high level, the program is not able to distinguish an interleaved data base from a noninterleaved data base. At the low level interleaving is accomplished through the use of pseudo-structures. Figure 5.24 gives the format of a pseudo-structure.

```
Figure 5.24. Pseudo-structure format
```

```
   ...(other selectors)
   $
   "MNAME"   "ID"
```

The pseudo structure is an ordinary data flow structure value which is distinguished from other structured values by the presence of the selector $. The symbol $ represents a special value which will not be used as a selector in ordinary structures. The substructure selected by $ identifies the pseudo-value. The "MNAME" component of this substructure identifies the data manager which supplied the pseudo-structure. The "ID" component identifies the level of the pseudo-structure (the function of this component will be explained below).

The interleaved data manager will return a pseudo-structure rather than the value of the entire data base. Initially the pseudo-structure will contain only the $ component.

The program may create a local copy of the data base by appending new values to the pseudo structure. The program may create several distinct local copies of the data base, but the high level syntax permits the program to return at most one of these local copies to the data manager. Appending new values into a pseudo-structure does not cause the actual data base to change. Changes to the data base are made only when a local copy of the data base is returned to the data manager. This is done only at program termination.

At the low level, the operations pseudo-select and
pseudo-append will replace the select and append operations for pseudo-structures. Graphs for these operations are given in section J.

The interleaved data manager is able to process four types of requests; which are given in Figure 5.25.

<table>
<thead>
<tr>
<th>Request-type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ-EXCLUSIVE</td>
<td>read a component and prevent any action on this component by any other program until the requesting program completes</td>
</tr>
<tr>
<td>READ-SHARED</td>
<td>read a component and do not grant any READ-EXCLUSIVE or INSERT request by any other program until the requesting program completes</td>
</tr>
<tr>
<td>INSERT</td>
<td>prevent any action on this component by any other program until the requesting program completes</td>
</tr>
<tr>
<td>COMPLETE</td>
<td>perform any required updates, and grant any requests blocked by this program</td>
</tr>
</tbody>
</table>

Figure 5.25. Interleaved data manager requests

These requests will be issued by both the communication layer of an application program, and the pseudo-select and pseudo-append operations (see section K). The logic of the processing of each type of request is given in Figure 5.26.
"READ-EXCLUSIVE"

If element-to-be-returned-is-pseudo-structure
then
construct-pseudo-structure
construct-reply
else if process-has-already-been-granted-read-exclusive-or-insert-on-this-element
then
select-element
construct-reply
else if process-has-already-been-granted-read-shared-on-this-element
then
if ok-to-grant-exclusive
then
if ok-to-grant-exclusive-now
then
select-element
construct-reply
else
queue-convert-request
else
construct-error-reply
else if no-request-is-active and no-exclusive-waiting
then
select-element
construct-reply
else queue-this-request

a. Logic for READ-EXCLUSIVE request

Figure 5.26. Interleaved data manager logic
"READ-SHARED"

if element-to-be-returned-is-pseudo-structure
then
    construct-pseudo-structure

    construct-reply

else if process-has-already-been-granted-read-exclusive-insert-or-read-shared-on-this-element
then
    select-element

    construct-reply

else if no-exclusive-request-for-this-element-active-or waiting
then
    select-element

    construct-reply

else queue-this-request

b. Logic for READ-SHARED processing

Figure 5.26 (Continued)
"INSERT"

If inserted-element-is-a-pseudo-structure then
construct-ACK-reply
else if process-has-already-been-granted-read-exclusive-or-insert-on-this-element
then
construct-ACK-reply
else if process-has-already-been-granted-read-shared-on-this-element
then
if ok-to-grant-exclusive
then if ok-to-grant-exclusive-now
then
construct-ACK-reply
else
queue-convert-request
else
construct-error-reply
else if no-request-is-active and no-exclusive-waiting
then construct-ACK-reply
else queue-this-request

C. Logic for INSERT request

"COMPLETE"
MERGE-DATA-VALUES
Release-blocked-requests

d. Logic for COMPLETE request
The interleaved data manager requires two data structures in addition to the data base structure itself. These are called the queue structure and the lock structure, respectively. The queue structure contains information about active requests and a queue of waiting requests for each element. This structure is used to determine whether or not it is permissible to grant a request. For each process with an active request, the lock structure contains a list of which elements of the data base have been referenced by the process, and whether or not the element was referenced by an INSERT or READ-EXCLUSIVE request.

The lock structure is updated by the actions "construct-reply" and "construct-ack" reply of Figure 5.26. The action "release-blocked-requests" of Figure 5.26d uses the lock structure as shown in Figure 5.27. This action will remove the process's list of elements from the lock structure.

As will be shown in the next section, a process may request the same element several times. If the process issues a read-shared request which is granted and later issues a read-exclusive request (or insert request) for the same element, the new request is referred to as a conversion request since the process's control of the element must be converted from shared to exclusive. A conversion request may be granted immediately if no other process has
Release-blocked-requests:
  Select-element-list-from-lock-structure using-process-id
Create-new-lock-structure
For each element in element list do begin
  if there-is-a-request-waiting-for-this-element-for-this-process
    then
      dequeue-request
  else if request-was-shared and no-other-shared-requests-are-active
    then
      construct-required-replies
  else if request-was-exclusive
    then
      construct-required-replies
  else
    no-action-for-this-element
end

Construct-required-replies:
  if queue-is-empty
    then
      remove-substructure-from-queue-structure
  if head-of-queue-is-exclusive
    then
      construct-one-reply
  if head-of-queue-is-shared
    then
      repeat
        construct-one-reply
      until end-of-queue or exclusive-request-found
  if head-of-queue-is-convert
    change-lock-structure
    construct-one-reply

Figure 5.27. Logic for releasing blocked requests
shared control of the element. Otherwise, the conversion request must be queued. If a conversion request must be queued, and there is a conversion request from another process which is already queued then queuing the new request will lead to deadlock. Therefore, the second conversion request must be denied. This is the meaning of the action "construct-error-reply" of Figure 5.26. A diagram of the interleaved data manager's internal state is given in Figure 5.28.

Figure 5.28. Interleaved data manager's internal state

Figure 6.28. Interleaved data-manager's internal state
The interleaved data manager is constructed so that multi-level interleaving is permitted. To demonstrate the meaning of multi-level interleaving consider a data base with components A, B, and C. Suppose that A is a structure with components X, Y, and Z. Multi-level interleaving allows interleaving between X, Y, and Z as well as between A, B, and C. This multi-level interleaving is accomplished by allowing the data manager to return another pseudo-structure instead of the actual value of the component A. The "ID" component of the pseudo-structure will identify the structure A and the level of A within the data base. It is assumed that the compiler will be aware of which elements will be returned as pseudo-structures, and which will be returned as actual values. This allows the compiler to generate the proper select and append operations without inserting tests to determine the nature of a data manager supplied token.

If a component of the data base is supplied as a pseudo-structure, then that component of the data base must be updated by appending values into the pseudo-structure and then appending the result into the pseudo-structure which represents the data base. This guarantees that the data manager is informed about all activity on the component. This condition is enforced by the action "merge-data-values" of Figure 5.26d. The logic of this action is given in Figure 5.29.
Merge-data-values (Data base, Pseudo-structure)
if test-format (pseudo-structure, "DB") = bad
then
    construct-error-message
    leave-data-base-intact
else
    construct-ok-message
    Insert-new-values (data-base, pseudo-structure)

Figure 5.29a. The structure of merge-data-values

Test-format (pseudo-structure, Level)
if this-level-should-be-pseudo-structure and
    this-level-is-pseudo-structure
then
    apply-test-format-to-all-non-$-components
    if all-results-are-good
        then return-good
        else return-bad
else if this-level-should-be-ordinary-structure and
    this-level-is-ordinary-structure then
    apply-test-format-to-all-non-$-components
    if all-results-are-good
        then return-good
        else return-bad
else if this-level-should-be-scalar and this-level-is-scalar
    then
        return-good
    else return-bad

Figure 5.29b. The structure of test-format
Insert-new-values (Data base, Pseudo-structure)

for each non-$-$-selector S in pseudo-structure do

if Pseudo-structure(S) is a pseudo-structure

then

append Insert-new-values (Data-base(S), pseudo-structure (S))

into Data-base with selector S

else

append Pseudo-structure (S) into Data-base with selector S

Figure 5.29c. The structure of insert-new-values

The interleaved data-manager does not provide for locks of varying granularity [Grey 1978]. It appears that this type of locking mechanism will require some sort of direct specification by the programmer as to where locks are set. This aspect of the problem definitely requires more investigation.

K. Application Program Graphs for Interleaved Data Manager

Because of the restrictions implied by the "test-format" action of Figure 5.29b, data manager names for interleaved data managers may not appear in an explicit declaration. The type of request (i.e., READ-SHARED, READ-EXCLUSIVE or INSERT) will depend on the location of the issuing pseudo-
append or pseudo-append or pseudo-select operation, and not on the use of the data manager name.

The template given in Figure 5.30 replaces both action 3 and action 4 of Figure 5.23.

There are two versions of the pseudo-select and pseudo-append operations. The versions given in Figure 5.31 are used to build the new copy of the data base, while the versions given in Figure 5.32 are used every place else.

These operations may cause excessive message traffic between the program and the data manager. A sophisticated optimizer could probably be used to reduce this traffic but this may require some additional serialization between pseudo-select and pseudo-append operations.

The interleaved system presented in the last two sections is probably too primitive to be used in a real system. It does, however, demonstrate the feasibility of interleaved data base updates in a functional environment.

L. Conclusion

The primary contribution of this chapter is the concept of the FCL-process, and the timing concepts defined in section E. Although little use of the timing definitions has been made, they form an implicit part of the FCL process. The remainder of this chapter demonstrates the usefulness of these concepts for defining solutions to time dependent
Figure 5.30. Interleaved data-manager communication layer template
Figure 5.31a. Pseudo-select for building new data-base
Figure 5.31b. Pseudo-append for constructing new database
Figure 5.32a. General-use pseudo-select
Figure 5.32b. General-use pseudo-append
problems, particularly the problem of access to shared data. Although data flow templates were given for various constructs, the semantics of these constructs do not depend on any computer architecture. The data flow templates serve only to make the concepts presented here more concrete.

Parts of this chapter contain considerably more detail than is usually present in a discussion of a solution to the shared data problem. For example, it is usually considered unnecessary to discuss program execution to the depth it was presented in section H. Nevertheless, such a discussion was deemed necessary for two reasons. First, the FCL solution to the shared data problem would not be self-contained if program execution and in particular, program execution-time were ignored. Second, this discussion shows how a formal definition of program execution-time might be made. As stated in the introduction, one objective of this chapter is to introduce formal definitions of certain concepts which have been treated informally up to this time.

This chapter shows how formality can be introduced at every level of a computer system, from the hardware, to the operating system, to the execution of high-level programs. One major area for further research is the filling in of details in those areas where this chapter
presents only a basic outline. Two obvious areas which require more detail are the formal specification of hardware characteristics and the formal specification of operating system features.

Although this chapter discusses every level of a computing system, it is strongly emphasized that a modular development of computing systems is advocated. The insistence on a self-contained solution to the shared data problem has necessitated an introduction of formality at levels which formal definitions are not usually made. However, once the formal specifications of the hardware level (say) have been made, it should be possible to ignore much of the detail of this level when making the formal specifications for the next level.

The process-communication system presented in this chapter is intended to serve as a model of a local computing system as opposed to a distributed system. The problem with extending the process-communication-system to distributed systems is that proper modeling of timing within the system becomes considerably more complicated, due to the possible varying execution rates of the individual computing systems. It is probable that a multi-process approach is best for distributed systems [Lamport 1978] as opposed to the single-process approach used here.

One interesting aspect of the work presented here is
that it is process-oriented. Since it has been shown that all operating system features may be implemented as message passing between processes [Lauer and Needham 1978], this suggests that a great deal of existing work in operating systems theory can be lifted intact into the theory of functional languages. This should greatly accelerate the development of computing systems based on functional semantics.
VI. CONCLUSION

It has been shown that it is possible to combine a number of useful features of existing functional languages into a compact, effective language for data flow programming. This language can be translated into data-driven code in a simple fashion in such a way that the door is left open to realize certain benefits of demand-driven code. It has also been shown that FCL code is at least as compact as that of other applicative languages, and that FCL can be used to provide a theoretical basis for the study of shared data.

There are several areas where further research is required. First, if functional languages are ever to be considered serious alternatives to existing languages, some form of modular data-types must be introduced. This is equivalent to programmer-defined scalars in FCL. Although, it would have been relatively easy to introduce such features into the syntax of FCL, it is advisable to proceed carefully in order to make sure that any new feature introduced fits into the existing framework properly. Other useful features which have not been introduced here are string handling features, and features for handling data processing problems.

In the area of comparison of languages, the main
benefit to be realized from such work is the redesign of languages to make them more usable. To this end, specific objectives must be established and solid, empirically-based measures must be selected (or developed) to measure how well the objectives have been met. Measures or techniques which single out specific features as good or bad should be given special attention.

The most obvious need in the area of translation is a working compiler. Ideally the target machine should be a simulator rather than an actual piece of hardware. This will allow investigation of hardware features for more efficient execution of FCL programs, as well as the ability to perform comparative measurements between FCL object code and the object code of other languages.

The area of shared data support requires the most research. Support for streams would enable the FCL process and the FCL data manager to be translated in a reasonable fashion. More research is needed to determine the place of the FCL process-communication-system in real systems. The formal definitions introduced in Chapter V have the potential for providing some sort of unifying link between various different time-dependent (or state-dependent) systems. This potential needs further exploration to determine whether it has any substance.

Finally, it has been shown that FCL, and functional
languages in general, are highly compatible with data-driven computation, and it appears that this will be a fruitful area of research in the future.
VII. REFERENCES


VIII. ACKNOWLEDGMENTS

I would like to thank my major professor, Arthur Oldehoeft, for his kind support throughout the course of this work. Without his guidance, this research could never have been completed. I would also like to thank the faculty members who have served on my committee: Dennis Kafura, Terry Smay, Roy Keller, Alan Selman, Robert Stewart, and R. Krishnaswamy.

My deepest thanks must go to my wife Laura, for without her encouragement and trust I would never had the courage to face graduate studies.

I also wish to thank my parents, Fred and June Maurer for giving me a deep appreciation for the value of learning. My thanks also go out to my brothers Max, Matt, Marc, and Mitch and to my sister, Mary Ellen, for their moral support during my studies.

I would also like to thank my former employers Jerry Lewis and Nancy Norman for their help and encouragement in my decision to return to school.

There are many other persons, too numerous to name, who have made it possible for me to achieve this goal. I extend my thanks to all of them.

This work was supported in part by the National Science Foundation, and the Iowa State University Graduate College.
IX. APPENDIX A: THE FCL GRAMMAR

EXP ::= EXP1 EXP1LIST
EXP1LIST ::= ; EXP1 EXP1LIST
EXP1LIST ::= ε
EXP1 ::= NAME = EXP2
EXP1 ::= EXP2
EXP2 ::= EXP3 EXP3LIST
EXP3LIST ::= | EXP3 EXP3LIST
EXP3LIST ::= ε
EXP3 ::= EXP4LIST EXP4
EXP4LIST ::= EXP4LIST EXP4+
EXP4LIST ::= ε
EXP4 ::= NAME : EXP5
EXP4 ::= (NAMELIST):EXP5
EXP4 ::= EXP5
EXP5 ::= EXP6 EXP6LIST
EXP6LIST ::= \ EXP6 EXP6LIST
EXP6LIST ::= ε
EXP6 ::= EXP7..EXP7
EXP6 ::= EXP7
EXP7 ::= EXP8 EXP8LIST
EXP8LIST ::= OR EXP8 EXP8LIST
EXP8LIST ::= ε
EXP8 ::= EXP9 EXP9LIST
EXP9LIST ::= AND EXP9 EXP9LIST
EXP9LIST ::= ε
EXP9 ::= NOT EXP10
EXP9 ::= EXP10
EXP10 ::= EXP11 COMPOP EXP11
EXP10 ::= EXP11
EXP11 ::= EXP12 EXP12LIST
EXP12LIST ::= ADDOP EXP12 EXP12LIST
EXP12LIST ::= ε
EXP12 ::= EXP13 EXP13LIST
EXP13LIST ::= MULTOP EXP13 EXP13LIST
EXP13LIST ::= ε
EXP13 ::= EXP14LIST EXP14
EXP14LIST ::= EXP14LIST EXP14 **
EXP14LIST ::= ε
EXP14 ::= +EXP15
EXP14 ::= -EXP15
EXP14 ::= EXP15
EXP15 ::= EXP16 EXP16LIST
EXP16LIST ::= EXP16 EXP16LIST
EXP16LIST ::= ε
EXP16 ::= #EXP17
EXP16 ::= EXP17
EXPL17 ::= LITERAL
EXPL17 ::= NAME
EXPL17 ::= FUNCAPPLY
EXPL17 ::= (EXPLIST)
EXPL17 ::= [EXPLIST]
EXPL17 ::= WHILECONST
EXPL17 ::= IFCONST
EXPL17 ::= RECDEF
EXPL17 ::= RECCALL

EXPLIST ::= EXP
EXPLIST ::= EXP, EXPLIST

FUNCAPPLY ::= BUILTIN (EXPLIST)
FUNCAPPLY ::= NAME(EXPLIST)
FUNCAPPLY ::= (EXPLIST)(EXPLIST)
FUNCAPPLY ::= [EXPLIST](EXPLIST)

COMPOP ::= <
COMPOP ::= >
COMPOP ::= =
COMPOP ::= >=
COMPOP ::= <=
COMPOP ::= ≠

ADDOP ::= +
ADDOP ::= -

MULTOP ::= *
MULTOP ::= /
MULTOP ::= DIV
MULTOP ::= MOD

LITERAL ::= INTLIT
LITERAL ::= REALIT
LITERAL ::= BOOLIT
LITERAL ::= CHARLIT
LITERAL ::= STRINGLIT

INTLIT ::= DIGIT DIGITLIST
DIGITLIST ::= DIGIT DIGITLIST
DIGITLIST ::= ε

REALIT ::= INTLIT.INTLIT
REALIT ::= INTLIT.
REALIT ::= .INTLIT

BOOLIT ::= TRUE
BOOLIT ::= FALSE
CHARLIT ::= 'CHAR'

STRINGLIT ::= "CHARLIST"
CHARLIST ::= CHAR CHARLIST
CHARLIST ::= ε

NAME ::= LETTER
NAME ::= LETTER NLIST LETDIG
NAME ::= LETDIG NLIST LETTER
NAME ::= LETDIG NLIST LETTER NLIST LETDIG
NAME ::= LETDIG NLIST _ NLIST LETDIG
NLIST ::= LDU NLIST
NLIST ::= ε

LDU ::= LETDIG
LDU ::= _
LETDIG ::= LETTER
LETDIG ::= DIGIT

DIGIT ::= 0
::=
DIGIT ::= 9

LETTER ::= A
::=
LETTER ::= Z
LETTER ::= a
::=
LETTER ::= z

CHAR ::= LETTER
CHAR ::= DIGIT
CHAR ::= SPECIAL
SPECIAL ::= {implementation dependent}

NAMELIST ::= NAME, NAMELIST
NAMELIST ::= NAME

WHILE CONST ::= WHILE EXP;
  INIT = (NEXPLIST);
  RESULT = EXP;
  NEXPLIST
ENDWHILE
NEXPLIST ::= NEXP; NEXPLIST
NEXPLIST ::= NEXP
NEXP ::= NAME = EXP2

IFCONST ::= IF EXP;
THEN = EXP;
ELSE = EXP
ENDIF

IFCONST ::= IF EXP
THEN EXP
ELSE EXP
ENDIF

BUILTIN ::= NAME {implementation dependent}

RECDEF ::= NAME = EXP

RECCALL ::= NAME.NAME
RECCAL ::= (EXP).NAME
APPENDIX B: EXAMPLES AND DETAILS OF HALSTEAD-MEASURE CALCULATIONS

MATRIX MULTIPLY FCL

\[ \text{MM} \equiv (A,B) : (\text{MATRIX}, \text{MATRIX}) \]
\[ \rightarrow (I,J) : (\text{INTEGER}, \text{INTEGER}) \]
\[ \rightarrow \text{SUM}(\text{TIMES}^o[A(I), B(J)]) \]

MATRIX MULTIPLY FCL - ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>\equiv</td>
<td>1</td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>( )</td>
<td>7</td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>,</td>
<td>6</td>
<td>MATRIX</td>
<td>2</td>
</tr>
<tr>
<td>:</td>
<td>2</td>
<td>INTEGER</td>
<td>2</td>
</tr>
<tr>
<td>+</td>
<td>2</td>
<td>I</td>
<td>2</td>
</tr>
<tr>
<td>SUM</td>
<td>1</td>
<td>J</td>
<td>2</td>
</tr>
<tr>
<td>TIMES</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[ ]</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ n_1 = 9 \quad N_1 = 22 \]
\[ n_2 = 6 \quad N_2 = 12 \]
\[ n = 14 \quad N = 34 \]

\[ V = \log 14 \cdot 34 = 129.450 \quad V^2 = 16767.3 \]
MATRIX MULTIPLY FP

DEF IP = (/+)*x*TRANS
DEF MM = (α*IP)*((a*dist)*distr °[l,TRANS°2])

MATRIX MULTIPLY FP ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEF</td>
<td>2</td>
<td>IP</td>
<td>1</td>
</tr>
<tr>
<td>≡</td>
<td>2</td>
<td>MM</td>
<td>1</td>
</tr>
<tr>
<td>( )</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>/</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>+</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>°</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>α</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRANS</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IP</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>distl</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>distr</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[ ]</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \eta_1 = 14 \quad \eta_2 = 4 \quad \eta = 18 \quad N_1 = 28 \quad N_2 = 4 \quad N = 32 \]

\[ V = \log 18 . \ 32 = 133.438 \quad V^2 = 17805.7 \]
MATRIX MULTIPLY VAL

function MM(A,B:Array[real]; N,M,L:lint;
   returns Array [real])

   forall I in [1,N]
   Z:array[real] := forall J in [1,M]
      Y:real := forall K in [1,L]
         X:real := A(I,K)*B(K,J);
         eval plus X
   end
   Construct Y
   end

   Construct Z
   end

end
## MATRIX MULTIPLY VAL ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>function</td>
<td>1</td>
<td>MM</td>
<td>1</td>
</tr>
<tr>
<td>( )</td>
<td>3</td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>:</td>
<td>5</td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>,</td>
<td>8</td>
<td>real</td>
<td>5</td>
</tr>
<tr>
<td>Array</td>
<td>3</td>
<td>N</td>
<td>2</td>
</tr>
<tr>
<td>[ ]</td>
<td>3</td>
<td>M</td>
<td>2</td>
</tr>
<tr>
<td>;</td>
<td>3</td>
<td>L</td>
<td>2</td>
</tr>
<tr>
<td>returns</td>
<td>1</td>
<td>int</td>
<td>1</td>
</tr>
<tr>
<td>forall</td>
<td>3</td>
<td>I</td>
<td>2</td>
</tr>
<tr>
<td>:=</td>
<td>3</td>
<td>l</td>
<td>3</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
<td>J</td>
<td>2</td>
</tr>
<tr>
<td>eval plus</td>
<td>1</td>
<td>K</td>
<td>3</td>
</tr>
<tr>
<td>construct</td>
<td>2</td>
<td>X</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Y</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Z</td>
<td>2</td>
</tr>
</tbody>
</table>

\[ \eta_1 = 13 \quad \eta_2 = 15 \quad \eta = 28 \]

\[ N_1 = 37 \quad N_2 = 33 \quad N = 70 \]

\[ V = \log 28 \cdot 70 = 336.515 \quad V^2 = 113242 \]
MATRIX MULTIPLY ID

Procedure mm(a,b,c,m,n)

(initial c<-A)
    for i from 1 to l do
        new c[i]<-

(initial d<-A)
    for j from 1 to n do
        new d[j]<-

(initial s+0)
    for k from 1 to m do
        new s+s+a[i,k]*b[k,j]
    return s)

return d)
return c)
### MATRIX MULTIPLY ID ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Procedure</td>
<td>1</td>
<td>mm</td>
<td>1</td>
</tr>
<tr>
<td>( )</td>
<td>4</td>
<td>a</td>
<td>2</td>
</tr>
<tr>
<td>,</td>
<td>6</td>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>initial</td>
<td>3</td>
<td>&amp;</td>
<td>2</td>
</tr>
<tr>
<td>+</td>
<td>6</td>
<td>m</td>
<td>2</td>
</tr>
<tr>
<td>for...</td>
<td>3</td>
<td>n</td>
<td>2</td>
</tr>
<tr>
<td>new</td>
<td>3</td>
<td>c</td>
<td>3</td>
</tr>
<tr>
<td>[ ]</td>
<td>4</td>
<td>i</td>
<td>3</td>
</tr>
<tr>
<td>+</td>
<td>1</td>
<td>d</td>
<td>3</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>return</td>
<td>3</td>
<td>l</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>j</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>s</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>k</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ \eta_1 = 11 \quad N_1 = 35 \]
\[ \eta_2 = 15 \quad N_2 = 36 \]
\[ \eta = 26 \quad N = 71 \]

\[ V = \log 26 \cdot 71 = 333.731 \quad V^2 = 111376 \]
MATRIX MULTIPLY PASCAL

FUNCTION MM(A,B: MATRIX; N, M, L: INTEGER)
  : MATRIX;
VAR I, K, J: INTEGER;
BEGIN
  FOR I := 1 TO N DO
    FOR J := 1 TO M DO BEGIN
      MM[I, J] := 0;
      FOR K := 1 TO L DO
    END
END
# MATRIX MULTIPLY PASCAL ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUNCTION</td>
<td>1</td>
<td>MM</td>
<td>4</td>
</tr>
<tr>
<td>( )</td>
<td>1</td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>,</td>
<td>10</td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>:</td>
<td>4</td>
<td>N</td>
<td>2</td>
</tr>
<tr>
<td>;</td>
<td>4</td>
<td>M</td>
<td>2</td>
</tr>
<tr>
<td>VAR</td>
<td>1</td>
<td>L</td>
<td>2</td>
</tr>
<tr>
<td>BEGIN</td>
<td>2</td>
<td>MATRIX</td>
<td>2</td>
</tr>
<tr>
<td>FOR</td>
<td>3</td>
<td>INTEGER</td>
<td>2</td>
</tr>
<tr>
<td>:=</td>
<td>2</td>
<td>I</td>
<td>6</td>
</tr>
<tr>
<td>[ ]</td>
<td>5</td>
<td>J</td>
<td>6</td>
</tr>
<tr>
<td>+</td>
<td>1</td>
<td>K</td>
<td>4</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
\eta_1 = 12 \quad N_1 = 35 \\
\eta_2 = 13 \quad N_2 = 38 \\
\eta = 25 \quad N = 73 \\
V = \log 25 \cdot 73 = 339.002 \quad V^2 = 114922
\]
MATRIX MULTIPLY FWAL

dotproduct: \langle V_1 V_2 \rangle \equiv \text{SUM}:<\text{product* caffe}<V_1 V_2>;

row:<\text{vec transp}> \equiv <\text{dotproduct* caffe}>:<\langle VEC>\text{ transp}>;

mm:<ml m2> \equiv <\text{row* caffe}>:<ml \text{ transpose:<m2> caffe}>
## MATRIX MULTIPLY FWAL ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>dotproduct</td>
<td>1</td>
<td>dotproduct</td>
<td>1</td>
</tr>
<tr>
<td>:</td>
<td>8</td>
<td>V1</td>
<td>2</td>
</tr>
<tr>
<td>&lt; &gt;</td>
<td>12</td>
<td>V2</td>
<td>2</td>
</tr>
<tr>
<td>=</td>
<td>3</td>
<td>row</td>
<td>1</td>
</tr>
<tr>
<td>SUM</td>
<td>1</td>
<td>vec</td>
<td>2</td>
</tr>
<tr>
<td>*</td>
<td>5</td>
<td>transp</td>
<td>2</td>
</tr>
<tr>
<td>;</td>
<td>2</td>
<td>mm</td>
<td>1</td>
</tr>
<tr>
<td>row</td>
<td>1</td>
<td>ml</td>
<td>2</td>
</tr>
<tr>
<td>transpose</td>
<td>1</td>
<td>m2</td>
<td>2</td>
</tr>
<tr>
<td>product</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \eta_1 = 10 \quad N_1 = 35 \]
\[ \eta_2 = 9 \quad N_2 = 15 \]
\[ n = 19 \quad N = 50 \]

\[ V = \log 19 \cdot 50 = 212.396 \quad V^2 = 45112.1 \]
VECTOR DISTANCE FCL

\[ \text{DIST} \equiv (A, B) : (\text{VECTOR}, \text{VECTOR}) \rightarrow \quad \sqrt{\text{SUM}(\text{POWER}(, 2) \circ \text{MINUS} \circ [A, B])} \]

VECTOR DISTANCE FCL ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>\equiv</td>
<td>1</td>
<td>DIST</td>
<td>1</td>
</tr>
<tr>
<td>( )</td>
<td>5</td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>,</td>
<td>4</td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>+</td>
<td>1</td>
<td>VECTOR</td>
<td>2</td>
</tr>
<tr>
<td>SQRT</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>SUM</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>POWER</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>°</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MINUS</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[ ]</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ n_1 = 10 \quad N_1 = 18 \]
\[ n_1 = 5 \quad N_2 = 8 \]
\[ \eta = 15 \quad N = 26 \]

\[ V = \log 15 \cdot 26 = 101.579 \quad V^2 = 10318.3 \]
VECTOR DISTANCE FP

DEF DIST ≡ SQRT○(/+)○(αX)○TRANS○[DIFF,DIFF]

DEF DIFF ≡ (α-)○TRANS

VECTOR DISTANCE FP ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEF</td>
<td>2</td>
<td>DIST</td>
<td>1</td>
</tr>
<tr>
<td>≡</td>
<td>2</td>
<td>DIFF</td>
<td>1</td>
</tr>
<tr>
<td>SQRT</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>○</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( )</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>α</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRANS</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[ ]</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIFF</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

η₁ = 14 \quad N₁ = 25
η₂ = 2 \quad N₂ = 2
n = 16 \quad N = 27

V = \log 16 \cdot 27 = 108.000 \quad V^2 = 11664.0
VECTOR DISTANCE VAL

FUNCTION DIST(A,B:array[real];
    size:int; returns real)

SQRT(forall I in [1,SIZE]
    X:real := (A(I)-B(I))**2
    eval plus X
    end)
end
VECTOR DISTANCE VAL ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUNCTION</td>
<td>1</td>
<td>DIST</td>
<td>1</td>
</tr>
<tr>
<td>( )</td>
<td>5</td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>,</td>
<td>1</td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>:</td>
<td>3</td>
<td>real</td>
<td>3</td>
</tr>
<tr>
<td>array</td>
<td>1</td>
<td>size</td>
<td>2</td>
</tr>
<tr>
<td>[ ]</td>
<td>1</td>
<td>int</td>
<td>1</td>
</tr>
<tr>
<td>;</td>
<td>2</td>
<td>I</td>
<td>3</td>
</tr>
<tr>
<td>returns</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SQRT</td>
<td>1</td>
<td>X</td>
<td>2</td>
</tr>
<tr>
<td>forall</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>:=</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>**</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval plus</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\eta_1 = 14 \quad N_1 = 21 \\
\eta_2 = 10 \quad N_2 = 18 \\
\eta = 24 \quad N = 39 \\
V = \log 24 \cdot 39 = 178.814 \quad V^2 = 31974.4
\]
VECTOR DISTANCE ID

Y+procedure x(a,b,s)

(initial t+0

for i from 1 to S do

new t+=t+(a[i]-b[i])^2

return SQrt(t))
VECTOR DISTANCE ID ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>3</td>
<td>y</td>
<td>1</td>
</tr>
<tr>
<td>procedure</td>
<td>1</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>( )</td>
<td>4</td>
<td>a</td>
<td>2</td>
</tr>
<tr>
<td>,</td>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>initial</td>
<td>1</td>
<td>s</td>
<td>2</td>
</tr>
<tr>
<td>for</td>
<td>1</td>
<td>t</td>
<td>4</td>
</tr>
<tr>
<td>new</td>
<td>1</td>
<td>i</td>
<td>3</td>
</tr>
<tr>
<td>+</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>[ ]</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>†</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>return</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sqrt</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ n_1 = 13 \quad N_1 = 20 \]
\[ n_2 = 10 \quad N_2 = 18 \]
\[ n = 23 \quad N = 38 \]

\[ V = \log 23 \cdot 38 = 171.895 \quad v^2 = 29547.9 \]
VECTOR DISTANCE PASCAL

FUNCTION DIST(A,B:VECTOR;SIZE:INTEGER)
  :REAL;

VAR T:REAL;
  I:INTEGER;

BEGIN
  T := 0;
  FOR I := 1 TO Size DO
    T := T+(A[I]-B[I])**2;
  DIST := SQRT(T)

END
VECTOR DISTANCE PASCAL ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUNCTION</td>
<td>1</td>
<td>DIST</td>
<td>1</td>
</tr>
<tr>
<td>( )</td>
<td>3</td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>,</td>
<td>1</td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>:</td>
<td>5</td>
<td>VECTOR</td>
<td>1</td>
</tr>
<tr>
<td>;</td>
<td>6</td>
<td>SIZE</td>
<td>2</td>
</tr>
<tr>
<td>VAR</td>
<td>1</td>
<td>INTEGER</td>
<td>2</td>
</tr>
<tr>
<td>BEGIN</td>
<td>1</td>
<td>REAL</td>
<td>2</td>
</tr>
<tr>
<td>:=</td>
<td>3</td>
<td>T</td>
<td>5</td>
</tr>
<tr>
<td>FOR</td>
<td>1</td>
<td>I</td>
<td>4</td>
</tr>
<tr>
<td>+</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>**</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>SQRT</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[ ]</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( n_1 = 14 \) \quad N_1 = 28

\( n_2 = 12 \) \quad N_2 = 24

\( n = 26 \) \quad N = 52

\( V = \log 26 \cdot 52 = 244.423 \quad \chi^2 = 59742.6 \)
VECTOR DISTANCE FWAL

DIST:\langle V1 \ V2 \rangle \triangleq \text{sqrt:sum:<square*>:<minus*>:<V1 \ V2>;}.

square \ X \triangleq \text{product:<}X \ X>.

VECTOR DISTANCE FWAL ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>;</td>
<td>7</td>
<td>Dist</td>
<td>1</td>
</tr>
<tr>
<td>&lt; &gt;</td>
<td>5</td>
<td>V1</td>
<td>2</td>
</tr>
<tr>
<td>\equiv</td>
<td>2</td>
<td>V2</td>
<td>2</td>
</tr>
<tr>
<td>sqrt</td>
<td>1</td>
<td>x</td>
<td>3</td>
</tr>
<tr>
<td>sum</td>
<td>1</td>
<td>square</td>
<td>1</td>
</tr>
<tr>
<td>square</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>minus</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>product</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>;</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\eta_1 = 10 \quad N_1 = 22
\eta_2 = 5 \quad N_2 = 9
\eta = 15 \quad N = 31
V = \log 15 \cdot 31 = 121.114 \quad V^2 = 14668.6
FACTORIAL FCL

\[ \text{FACT} \equiv \begin{array}{l}
0 
\rightarrow \begin{array}{l}
x 
: \text{INTEGER} 
\rightarrow x \cdot \text{FACT}(x-1)
\end{array}
\end{array} \]

FACTORIAL FCL ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>1</td>
<td>FACT</td>
<td>1</td>
</tr>
<tr>
<td>+</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>:</td>
<td>1</td>
<td>x</td>
<td>3</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
<td>INTEGER</td>
<td>1</td>
</tr>
<tr>
<td>()</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FACT</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \eta_1 = 8 \quad N_1 = 9 \]

\[ \eta_2 = 5 \quad N_2 = 8 \]

\[ \eta = 13 \quad N = 17 \]

\[ V = \log_{10} 13 \cdot 17 = 62.9075 \quad V^2 = 3957.35 \]
**FACTORIAL FP**

Def fact ≡ eq[id,0]→1;

\[ x \cdot [id, fact \cdot - \cdot [id, 1]] \]

**FACTORIAL FP ANALYSIS**

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Def</td>
<td>1</td>
<td>fact</td>
<td>1</td>
</tr>
<tr>
<td>≡</td>
<td>1</td>
<td>id</td>
<td>3</td>
</tr>
<tr>
<td>eq</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>°</td>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>[ ]</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>,</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>→</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>;</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fact</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

\[ \eta_1 = 11 \quad N_1 = 18 \]
\[ \eta_2 = 4 \quad N_2 = 7 \]
\[ \eta = 15 \quad N = 25 \]

\[ V = \log 15 \cdot 25 = 97.6723 \quad V^2 = 9539.88 \]
FACTORIAL VAL

Function fact (N:int; returns int)

    if N = 0 then 1
    else fact (N-1)*N

end
end

FACTORIAL VAL ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>function</td>
<td>1</td>
<td>fact</td>
<td>1</td>
</tr>
<tr>
<td>( )</td>
<td>2</td>
<td>N</td>
<td>4</td>
</tr>
<tr>
<td>;</td>
<td>1</td>
<td>int</td>
<td>2</td>
</tr>
<tr>
<td>;</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>returns</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>if</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fact</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\eta_1 = 10 \quad N_1 = 11 \\
\eta_2 = 5 \quad N_2 = 10 \\
\eta = 15 \quad N = 21 \\
V = \log 15 \cdot 21 = 82.0447 \quad V^2 = 6731.33
\]
FACTORIAL ID

Y<procedure fact(n)

(if n=0 then 1 else n*fact(n-1))

FACTORIAL ID ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>1</td>
<td>Y</td>
<td>1</td>
</tr>
<tr>
<td>Procedure</td>
<td>1</td>
<td>fact</td>
<td>1</td>
</tr>
<tr>
<td>(</td>
<td>3</td>
<td>n</td>
<td>4</td>
</tr>
<tr>
<td>if</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>=</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
<td>fact</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \eta_1 = 8 \quad N_1 = 10 \]
\[ \eta_2 = 5 \quad N_2 = 9 \]
\[ \eta = 13 \quad N = 19 \]

\[ V = \log 13 \cdot 19 = 70.3084 \quad V^2 = 4943.27 \]
FACTORIAL PASCAL

FUNCTION FACT(X:INTEGER):INTEGER;
BEGIN
  IF X=0 THEN
    FACT := 1
  ELSE
    FACT := X*FACT(X-1)
  END
END
FACTORIAL PASCAL ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUNCTION</td>
<td>1</td>
<td>FACT</td>
<td>3</td>
</tr>
<tr>
<td>( )</td>
<td>2</td>
<td>X</td>
<td>4</td>
</tr>
<tr>
<td>;</td>
<td>2</td>
<td>INTEGER</td>
<td>2</td>
</tr>
<tr>
<td>;</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Begin</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>IF</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>:=</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FACT</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \eta_1 = 11 \quad N_1 = 14 \]
\[ \eta_2 = 5 \quad N_2 = 12 \]
\[ \eta = 16 \quad N = 26 \]

\[ V = \log 16 \cdot 26 = 104.000 \quad V^2 = 10816.0 \]
FACTORIAL FWAL

fact:n \equiv \text{if } eq:<n \ o> \ \text{then } 1 \ \text{else} \\
\quad \quad \text{product:<n fact:minus:<n 1>}}

FACTORIAL FWAL ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
<td>5</td>
<td>fact</td>
<td>1</td>
</tr>
<tr>
<td>\equiv</td>
<td>1</td>
<td>n</td>
<td>4</td>
</tr>
<tr>
<td>if</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>eq</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>&lt; &gt;</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Product</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fact</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>minus</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\eta_1 = 8 \quad \quad N_1 = 14 \\
\eta_2 = 4 \quad \quad N_2 = 8 \\
\eta = 12 \quad \quad N = 22 \\

V = \log 12 \cdot 22 = 78.8692 \quad \quad V^2 = 6220.35
FIBONACCI SEQUENCE FCL

FIB \equiv 0 \rightarrow 0 \rightarrow 1 \rightarrow 1 \rightarrow X : \text{INTEGER} \rightarrow \text{FIB}(X-1) + \text{FIB}(X-2)

FIBONACCI SEQUENCE FCL ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>1</td>
<td>FIB</td>
<td>1</td>
</tr>
<tr>
<td>+</td>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>:</td>
<td>1</td>
<td>X</td>
<td>3</td>
</tr>
<tr>
<td>FIB</td>
<td>2</td>
<td>INTEGER</td>
<td>1</td>
</tr>
<tr>
<td>( )</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \eta_1 = 8 \quad N_1 = 14 \]
\[ \eta_2 = 6 \quad N_2 = 11 \]
\[ \eta = 14 \quad N = 25 \]
\[ v = \log 14 \cdot 25 = 95.1839 \quad v^2 = 9059.97 \]
FIBONACCI SEQUENCE FP

DEF FIB ≡ eqichte {id,0} + 0;
   eqichte {id,1} + 1;
   +o [{FIBichte {id,1}, FIBichte {id,2}}]

FIBONACCI SEQUENCE FP ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEF</td>
<td>1</td>
<td>id</td>
<td>4</td>
</tr>
<tr>
<td>≡</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>eq</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>o</td>
<td>7</td>
<td>FIB</td>
<td>1</td>
</tr>
<tr>
<td>[ ]</td>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>+</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>;</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>,</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIB</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \eta_1 = 11 \quad N_1 = 30 \]
\[ \eta_2 = 5 \quad N_2 = 11 \]
\[ \eta = 16 \quad N = 41 \]
\[ V = \log{16} \cdot 41 = 164.000 \quad V^2 = 26896.0 \]
FIBONACCI SEQUENCE VAL

Function FIB(N:INT ; RETURNS INT)

IF N=0 THEN 0

ELSE IF N=1 THEN 1

    ELSE FIB(N-1) + FIB(N-2)

    END

END

END
FIBONACCI SEQUENCE VAL ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>function</td>
<td>1</td>
<td>FIB</td>
<td></td>
</tr>
<tr>
<td>( )</td>
<td>3</td>
<td>N</td>
<td>5</td>
</tr>
<tr>
<td>:</td>
<td>1</td>
<td>INT</td>
<td>2</td>
</tr>
<tr>
<td>;</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>RETURNS</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>IF</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>=</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIB</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\eta_1 = 10 \quad N_1 = 16 \\
\eta_2 = 6 \quad N_2 = 14 \\
\eta = 16 \quad N = 30 \\
V = \log 16 \cdot 30 = 120.000 \quad V^2 = 14400.0
\]
FIBONACCI SEQUENCE ID

Y Procedure fib(n)

(if n = 0 then 0
else if n = 1 then 1
else fib(n-1) + fib(n-2))

FIBONACCI SEQUENCE ID ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>1</td>
<td>y</td>
<td>1</td>
</tr>
<tr>
<td>Procedure</td>
<td>1</td>
<td>fib</td>
<td>1</td>
</tr>
<tr>
<td>( )</td>
<td>4</td>
<td>n</td>
<td>5</td>
</tr>
<tr>
<td>if</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>=</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>fib</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \eta_1 = 8 \quad N_1 = 15 \]
\[ \eta_2 = 6 \quad N_2 = 13 \]
\[ n = 14 \quad N = 28 \]

\[ V = \log 14 \cdot 28 = 106.606 \quad V^2 = 11364.8 \]
FIBONACCI SEQUENCE PASCAL

FUNCTION FIB(N:INTEGER):INTEGER
BEGIN
  IF N=0 THEN
    FIB := 0
  ELSE IF N=1 THEN
    FIB := 1
  ELSE
    FIB := FIB(N-1) + FIB(N-2)
END
FIBONACCI SEQUENCE PASCAL ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUNCTION</td>
<td>1FIB</td>
<td>FIB</td>
<td>4</td>
</tr>
<tr>
<td>( )</td>
<td>3</td>
<td>N</td>
<td>5</td>
</tr>
<tr>
<td>:</td>
<td>2</td>
<td>INTEGER</td>
<td>2</td>
</tr>
<tr>
<td>BEGIN</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>IF</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>=</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>:=</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIB</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \eta_1 = 10 \quad N_1 = 19 \]
\[ \eta_2 = 6 \quad N_2 = 17 \]
\[ \eta = 16 \quad N = 36 \]

\[ V = \log 16 \cdot 36 = 144.000 \quad V^2 = 20736.0 \]
FIBONACCI SEQUENCE FWAL

fib:n ≡ if eq:<n 0> then 0
    else if eq:<n 1> then 1
    else
        sum: fib:minus:<n 1>
        fib: minus:<n 2>>

FIBONACCI SEQUENCE FWAL ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
<td>8</td>
<td>fib</td>
<td>1</td>
</tr>
<tr>
<td>≡</td>
<td>1</td>
<td>n</td>
<td>5</td>
</tr>
<tr>
<td>if</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>eq</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>&lt; &gt;</td>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>elseif</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sum</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fib</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>minus</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ n_1 = 9 \quad N_1 = 23 \]
\[ n_2 = 5 \quad N_2 = 12 \]
\[ n = 14 \quad N = 35 \]

\[ V = \log 14 \cdot 35 = 133.257 \quad V^2 = 17757.4 \]
TREE REVERSAL FCL

\[ \text{REV} \equiv x: \text{INTEGER} \rightarrow \text{x|} \]

\[ t: \text{BINTREE} \rightarrow (\text{REV}(t(2)), \text{REV}(t(1))) \]

TREE REVERSAL FCL ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>\equiv</td>
<td>1</td>
<td>REV</td>
<td>1</td>
</tr>
<tr>
<td>:</td>
<td>2</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>+</td>
<td>2</td>
<td>INTEGER</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>t</td>
</tr>
<tr>
<td>( )</td>
<td>5</td>
<td>BINTREE</td>
<td>1</td>
</tr>
<tr>
<td>REV</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>,</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ n_1 = 7 \quad N_1 = 14 \]
\[ n_2 = 7 \quad N_2 = 10 \]
\[ n = 14 \quad N = 24 \]
\[ v = \log 14 \cdot 24 = 91.3765 \quad v^2 = 8349.66 \]
TREE REVERSAL FP

DEF REV ≡ atom+id;

[REV °2, REV °1]

TREE REVERSAL FP ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEF</td>
<td>1</td>
<td>id</td>
<td>1</td>
</tr>
<tr>
<td>≡</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>atom</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>°</td>
<td>1</td>
<td>REV</td>
<td>1</td>
</tr>
<tr>
<td>;</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[ ]</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>,</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REV</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>°</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\eta_1 = 9 \quad N_1 = 11 \\
\eta_2 = 4 \quad N_2 = 4 \\
\eta = 13 \quad N = 15 \\
V = \log 13 \cdot 15 = 55.5066 \quad V^2 = 3080.98
\]
TREE REVERSAL VAL

function rev(t:TREE; returns TREE)

   If UNDEF (t(1)) then t
   else [1:rev(t(2)), rev(t(l))]
   end
end

TREE REVERSAL VAL ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>function</td>
<td>1</td>
<td>rev</td>
<td>1</td>
</tr>
<tr>
<td>( )</td>
<td>7</td>
<td>t</td>
<td>5</td>
</tr>
<tr>
<td>:</td>
<td>2</td>
<td>TREE</td>
<td>2</td>
</tr>
<tr>
<td>;</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>returns</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>If</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNDEF</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[ ]</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>,</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rev</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ n_1 = 10 \quad N_1 = 18 \]
\[ n_2 = 5 \quad N_2 = 12 \]
\[ n = 15 \quad N = 30 \]

\[ V = \log \frac{15}{30} = 117.207 \quad V^2 = 13737.5 \]
TREE REVERSAL ID

Y<-procedure rev(t)

(if leaf(t) then t

    else <rev(t[2]), rev(t[1])>)

TREE REVERSAL ID ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;-</td>
<td>1</td>
<td>Y</td>
<td>1</td>
</tr>
<tr>
<td>Procedure</td>
<td>1</td>
<td>rev</td>
<td>1</td>
</tr>
<tr>
<td>( )</td>
<td>4</td>
<td>t</td>
<td>5</td>
</tr>
<tr>
<td>if</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>leaf</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>&lt; &gt;</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rev</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[ ]</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>,</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \eta_1 = 9 \quad N_1 = 14 \]
\[ \eta_2 = 5 \quad N_2 = 9 \]
\[ \eta = 14 \quad N = 23 \]

\[ V = \log 14 \cdot 23 = 87.5692 \quad V^2 = 7668.36 \]
TREE REVERSAL PASCAL

PROCEDURE REV(T: NODE);
VAR TEMP: NODE;
BEGIN
IF T ≠ NULL THEN BEGIN
    REV(T↑.LEFT);
    REV(T↑.RIGHT);
    TEMP := T↑.LEFT;
    T↑.LEFT := T↑.RIGHT;
    T↑.RIGHT := TEMP
END
END
## TREE REVERSAL PASCAL ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCEDURE</td>
<td>1</td>
<td>REV</td>
<td>1</td>
</tr>
<tr>
<td>( )</td>
<td>3</td>
<td>T</td>
<td>8</td>
</tr>
<tr>
<td>:</td>
<td>2</td>
<td>NODE</td>
<td>2</td>
</tr>
<tr>
<td>†</td>
<td>8</td>
<td>TEMP</td>
<td>3</td>
</tr>
<tr>
<td>;</td>
<td>6</td>
<td>NULL</td>
<td>1</td>
</tr>
<tr>
<td>VAR</td>
<td>1</td>
<td>RIGHT</td>
<td>3</td>
</tr>
<tr>
<td>BEGIN</td>
<td>2</td>
<td>LEFT</td>
<td>3</td>
</tr>
<tr>
<td>IF</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>≠</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REV</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>:=</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \eta_1 = 12 \quad N_1 = 36 \]
\[ \eta = 7 \quad N_2 = 21 \]
\[ \eta = 19 \quad N = 57 \]

\[ V = \log(19 \cdot 57) = 242.132 \quad V^2 = 58627.9 \]
TREE REVERSAL FWAL

rev\(:t\) \(=\) if atom\(:t\) then \(t\)

else

\(<\text{rev}:2:t\ \text{rev}:\text{first}:t\>

TREE REVERSAL FWAL ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>(:)</td>
<td>6</td>
</tr>
<tr>
<td>(\equiv)</td>
<td>1</td>
</tr>
<tr>
<td>if</td>
<td>1</td>
</tr>
<tr>
<td>atom</td>
<td>1</td>
</tr>
<tr>
<td>(&lt;&gt;)</td>
<td>1</td>
</tr>
<tr>
<td>\text{rev}</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>\text{First}</td>
<td>1</td>
</tr>
</tbody>
</table>

\(\eta_1 = 8\) \quad \(N_1 = 14\)
\(\eta_2 = 2\) \quad \(N_2 = 6\)
\(\eta = 10\) \quad \(N = 20\)

\(V = \log_{10} 20 = 66.4386\) \quad \(V^2 = 4414.09\)
4th Order Runge-Kutta FCL

RK (IX, IY, F, H, N):

(REAL, REAL, FUNCTION, REAL, INTEGER)

WHILE I<N;

INIT=(I=0;
    Y=IY;
    X=IX)

RESULT=Y;

K1=H*F(X, Y);
K2=H*F(X+H/2, Y+K1/2);
K3=H*F(X+H/2, Y+K2/2);
K4=H*F(X+H, Y+K3);
Y=Y+(1/6)*(K1+2*K2+2*K3+K4);
X=X+H;
I=I+1

ENDWHILE
Runge-Kutta FCL ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>13</td>
<td>IX</td>
<td>2</td>
</tr>
<tr>
<td>( )</td>
<td>9</td>
<td>IY</td>
<td>2</td>
</tr>
<tr>
<td>,</td>
<td>12</td>
<td>F</td>
<td>1</td>
</tr>
<tr>
<td>:</td>
<td>1</td>
<td>H</td>
<td>9</td>
</tr>
<tr>
<td>+</td>
<td>1</td>
<td>N</td>
<td>2</td>
</tr>
<tr>
<td>WHILE</td>
<td>1</td>
<td>RK</td>
<td>1</td>
</tr>
<tr>
<td>&lt;</td>
<td>1</td>
<td>REAL</td>
<td>3</td>
</tr>
<tr>
<td>;</td>
<td>10</td>
<td>FUNCTION</td>
<td>1</td>
</tr>
<tr>
<td>*</td>
<td>7</td>
<td>INTEGER</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>4</td>
<td>I</td>
<td>4</td>
</tr>
<tr>
<td>+</td>
<td>12</td>
<td>O</td>
<td>1</td>
</tr>
<tr>
<td>/</td>
<td>5</td>
<td>Y</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INIT</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RESULT</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>K1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>K2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>K3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>K4</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ n_1 = 12 \quad N_1 = 76 \]
\[ n_2 = 22 \quad N_2 = 64 \]
\[ n = 34 \quad N = 140 \]

\[ V = \log 34 \cdot 140 = 712.245 \quad V^2 = 507292 \]
Runge-Kutta FP

Def RK ≡ eqo[4,0]+id;

RK o

[+ o [1, 3],
  +° [2, * o [¥ [1, 6], + o [+ o [K1, * o [2, K2]],
                 + o[* o [2, K3], K4]]],
  3, -° [4, 1]]

DEF K1 ≡ * o [f o [1, 2], 3]

KEF K2 ≡ * o [3, f o [+ o [1, + o [3, 2]],
                  + o [2, ¥ o [K1, 2]]]]

DEF K3 ≡ * o [3, f o [+ o [1, + o [3, 2]],
                  + o [2, ¥ o [K2, 2]]]]

DEF K4 ≡ * o [3, f o [+ o [1, 3], + o [2, K3]]]
Runge-Kutta FP ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Def</td>
<td>5</td>
<td>RK</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>eq</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>29</td>
<td>id</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>RK</td>
<td>1</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>+</td>
<td>11</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>*</td>
<td>7</td>
<td>K1</td>
<td>1</td>
</tr>
<tr>
<td>÷</td>
<td>5</td>
<td>K2</td>
<td>1</td>
</tr>
<tr>
<td>K1</td>
<td>2</td>
<td>K3</td>
<td>1</td>
</tr>
<tr>
<td>K2</td>
<td>2</td>
<td>K4</td>
<td>1</td>
</tr>
<tr>
<td>K3</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>K4</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ n_1 = 18 \quad N_1 = 140 \]
\[ n_2 = 14 \quad N_2 = 37 \]
\[ n_3 = 32 \quad N = 177 \]
\[ V = \log 32 \cdot 177 = 885.000 \quad V^2 = 783225 \]
Runge-Kutta VAL

function RK(IX,IY,H:Real;N:int;returns real)
    for Y:real := IY;
        X:real := IX;
        I:int := 1 STEP 1
    let K1:real := H*F(X,Y);
        K2:real := H*F(X+H/2,Y+K1/2);
        K3:real := H*F(X+H/2,Y+K2/2);
        K4:real := H*F(X+H,Y+K3)
    do if I<=N then
        iter X := X+H;
        Y := Y+(1/6)*(K1+2*K2+2*K3+K4)
    else y
    end
end
end
Runge-Kutta VAL ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>function</td>
<td>1</td>
<td>RK</td>
<td>1</td>
</tr>
<tr>
<td>( )</td>
<td>7</td>
<td>IX</td>
<td>2</td>
</tr>
<tr>
<td>,</td>
<td>6</td>
<td>IY</td>
<td>2</td>
</tr>
<tr>
<td>;</td>
<td>9</td>
<td>N</td>
<td>9</td>
</tr>
<tr>
<td>;</td>
<td>8</td>
<td>real</td>
<td>8</td>
</tr>
<tr>
<td>$\text{return}$</td>
<td>1</td>
<td>N</td>
<td>2</td>
</tr>
<tr>
<td>for</td>
<td>1</td>
<td>int</td>
<td>2</td>
</tr>
<tr>
<td>let</td>
<td>1</td>
<td>Y</td>
<td>8</td>
</tr>
<tr>
<td>:=</td>
<td>9</td>
<td>X</td>
<td>7</td>
</tr>
<tr>
<td>STEP</td>
<td>1</td>
<td>I</td>
<td>2</td>
</tr>
<tr>
<td>*</td>
<td>7</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>F</td>
<td>4</td>
<td>Kl</td>
<td>3</td>
</tr>
<tr>
<td>+</td>
<td>11</td>
<td>K2</td>
<td>3</td>
</tr>
<tr>
<td>/</td>
<td>5</td>
<td>K3</td>
<td>3</td>
</tr>
<tr>
<td>if</td>
<td>1</td>
<td>K4</td>
<td>2</td>
</tr>
<tr>
<td>&lt;=</td>
<td>1</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>iter</td>
<td>1</td>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ \eta_1 = 17 \quad \eta_2 = 17 \quad \eta = 34 \]
\[ N_1 = 74 \quad N_2 = 64 \quad N = 138 \]
\[ V = \log 34 \cdot 138 = 702.070 \quad V^2 = 492902 \]
Runge-Kutta ID

P+Procedure (IX, IY, H, N, F)

(initial X=IX;
    Y=IY
    for I from 1 to N do
        K1=H*F(X, Y);
        K2=H*F(X+H/2, Y+K1/2);
        K3=H*F(X+H/2, Y+K2/2);
        K4=H*F(X+H, Y+K3);
        new X=X+H;
        new Y=Y+(1/6)*(K1+2*K2+2*K3+K4)
    return Y)
Runge-Kutta ID ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>9</td>
<td>P</td>
<td>1</td>
</tr>
<tr>
<td>Procedure</td>
<td>1</td>
<td>IX</td>
<td>2</td>
</tr>
<tr>
<td>( )</td>
<td>8</td>
<td>IY</td>
<td>2</td>
</tr>
<tr>
<td>;</td>
<td>8</td>
<td>H</td>
<td>9</td>
</tr>
<tr>
<td>initial</td>
<td>1</td>
<td>N</td>
<td>2</td>
</tr>
<tr>
<td>;</td>
<td>6</td>
<td>F</td>
<td>1</td>
</tr>
<tr>
<td>for</td>
<td>1</td>
<td>X</td>
<td>7</td>
</tr>
<tr>
<td>*</td>
<td>7</td>
<td>Y</td>
<td>8</td>
</tr>
<tr>
<td>F</td>
<td>4</td>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>+</td>
<td>11</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>/</td>
<td>5</td>
<td>K1</td>
<td>3</td>
</tr>
<tr>
<td>new</td>
<td>2</td>
<td>K2</td>
<td>3</td>
</tr>
<tr>
<td>return</td>
<td>1</td>
<td>K3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>K4</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
\eta_1 = 13 \quad N_1 = 64 \\
\eta_2 = 16 \quad N_2 = 53 \\
\eta = 29 \quad N = 117 \\
V = \log 29 \cdot 117 = 568.384 \quad V^2 = 323060
\]
Runge-Kutta PASCAL

FUNCTION RK(IX,IY,H:REAL;N:INTEGER)
  :REAL;
VAR X,Y,K1,K2,K3,K4:REAL;
  I:INTEGER;
BEGIN
  Y := IY;
  X := IX;
  FOR I := 1 to N DO BEGIN
    K1 := H*F(X,Y);
    K2 := H*F(X+H/2,Y+K1/2);
    K3 := H*F(X+H/2,Y+K2/2);
    K4 := H*F(X+H,Y+K3);
    X := X+H;
    Y := Y+(1/6)*(K1+2*K2+2*K3+K4)
  END;
  RK := Y
END
RUNGE-KUTTA PASCAL ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUNCTION</td>
<td>1</td>
<td>RK</td>
<td>2</td>
</tr>
<tr>
<td>( )</td>
<td>7</td>
<td>IX</td>
<td>2</td>
</tr>
<tr>
<td>,</td>
<td>11</td>
<td>IY</td>
<td>2</td>
</tr>
<tr>
<td>:</td>
<td>5</td>
<td>H</td>
<td>9</td>
</tr>
<tr>
<td>;</td>
<td>12</td>
<td>REAL</td>
<td>3</td>
</tr>
<tr>
<td>VAR</td>
<td>1</td>
<td>N</td>
<td>2</td>
</tr>
<tr>
<td>BEGIN</td>
<td>2</td>
<td>INTEGER</td>
<td>2</td>
</tr>
<tr>
<td>:=</td>
<td>9</td>
<td>X</td>
<td>8</td>
</tr>
<tr>
<td>FOR</td>
<td>1</td>
<td>Y</td>
<td>9</td>
</tr>
<tr>
<td>*</td>
<td>7</td>
<td>K1</td>
<td>4</td>
</tr>
<tr>
<td>F</td>
<td>4</td>
<td>K2</td>
<td>4</td>
</tr>
<tr>
<td>+</td>
<td>11</td>
<td>K3</td>
<td>4</td>
</tr>
<tr>
<td>/</td>
<td>5</td>
<td>K4</td>
<td>3</td>
</tr>
</tbody>
</table>

\[ n_1 = 13 \quad N_1 = 76 \]
\[ n_2 = 17 \quad N_2 = 65 \]
\[ n = 30 \quad N = 141 \]

\[ V = \log 30 \cdot 141 = 691.872 \quad V^2 = 478687 \]
Runge-Kutta FWAL

K1:<x y h> ≡ product:<h F:<x y>>;

K2:<x y h> ≡
product: <h F:<sum:<x quotient:<h 2>>
sum:<y quotient:<K1:<x y h>2>>>;

K3:<x y h> ≡
product:<h F:<sum:<x quotient:<h 2>
sum;<y quotient:<K2:<x y h>2>>>;

K4:<x y h> ≡
product:<h F:<sum:<x h> sum:<y K3:<x y h>>>;

RK:<x y h n> ≡ if equal:<n o> then y
else
    RK:<sum:<x h>
sum:<y product:<quotient:<l 6>
SUM:<K1:<x y h>
product:<K2:<x y h>2>
product:<K3:<x y h>2>
    K4:<x y h>>>>
    h
    minus:<n 1>>
Runge-Kutta FWAL ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
<td>40</td>
<td>Kl</td>
<td>1</td>
</tr>
<tr>
<td>&lt; &gt;</td>
<td>40</td>
<td>X</td>
<td>17</td>
</tr>
<tr>
<td>≡</td>
<td>5</td>
<td>Y</td>
<td>18</td>
</tr>
<tr>
<td>product</td>
<td>7</td>
<td>h</td>
<td>21</td>
</tr>
<tr>
<td>F</td>
<td>4</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>;</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>sum</td>
<td>9</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>quotient</td>
<td>5</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>Kl</td>
<td>2</td>
<td>K2</td>
<td>1</td>
</tr>
<tr>
<td>K2</td>
<td>2</td>
<td>K3</td>
<td>1</td>
</tr>
<tr>
<td>K3</td>
<td>2</td>
<td>K4</td>
<td>1</td>
</tr>
<tr>
<td>K4</td>
<td>1</td>
<td>RK</td>
<td>1</td>
</tr>
<tr>
<td>if</td>
<td>1</td>
<td>n</td>
<td>3</td>
</tr>
<tr>
<td>equal</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RK</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>minus</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \eta_1 = 16 \quad N_1 = 125 \]
\[ \eta_2 = 13 \quad N_2 = 74 \]
\[ \eta = 29 \quad N = 199 \]

\[ v = \log 29 \cdot 199 = 966.738 \quad v^2 = 934582 \]
BINARY SEARCH FCL

BIN ≡ (A, VAL, SIZE, START): (VECTOR, REAL, INTEGER, INTEGER) →

IF SIZE = 1 THEN
    IF A(START) = VAL THEN START
    ELSE -1
ENDIF
ELSE
    (Q = SIZE DIV 2;
    T = START + Q - 1;
    IF A(T) > VAL THEN
        BIN(A, VAL, Q, START)
    ELSE
        IF A(T) < VAL THEN
            BIN(A, VAL, SIZE - Q, START + Q)
        ELSE T
        ENDIF
    ENDIF
ENDIF
ENDIF
BINARY SEARCH FCL ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>≡</td>
<td>3</td>
<td>BIN</td>
<td>1</td>
</tr>
<tr>
<td>( )</td>
<td>8</td>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>,</td>
<td>12</td>
<td>VAL</td>
<td>6</td>
</tr>
<tr>
<td>:</td>
<td>1</td>
<td>SIZE</td>
<td>4</td>
</tr>
<tr>
<td>+</td>
<td>1</td>
<td>START</td>
<td>6</td>
</tr>
<tr>
<td>DIV</td>
<td>1</td>
<td>VECTOR</td>
<td>1</td>
</tr>
<tr>
<td>;</td>
<td>2</td>
<td>REAL</td>
<td>1</td>
</tr>
<tr>
<td>+</td>
<td>2</td>
<td>INTEGER</td>
<td>2</td>
</tr>
<tr>
<td>-</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>A</td>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>&lt;</td>
<td>1</td>
<td>Q</td>
<td>5</td>
</tr>
<tr>
<td>&gt;</td>
<td>1</td>
<td>T</td>
<td>4</td>
</tr>
<tr>
<td>BIN</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IF</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\eta_1 &= 15 \quad N_1 = 48 \\
\eta_2 &= 12 \quad N_2 = 35 \\
\eta &= 27 \quad N = 83 \\
V &= \log 27 \cdot 83 = 394.656 \quad V^2 = 155753
\end{align*}
\]
BINARY SEARCH FP

Def $ BIN \equiv eq^o[3,\bar{I}] +$

\[ (eq^o[select^o[1,4],1\circ1] + 4; \]

\[ -\bar{I}; \]

\[ gt^o[select^o[1,\bar{I}],2] + \]

\[ BIN^o[1,2,Q,4]; \]

\[ lt^o[select^o[1,T],2] + \]

\[ BIN^o[1,2,-o[3,Q],+o[4,Q]]; \]

$ T$

Def $ Q \equiv Div^o[3,\bar{2}]$

Def $ T \equiv -o[+o[Q,4],\bar{I}]$

Def $ select \equiv eq^o[2,\bar{I}] + 1\circ1;$

\[ select^o[t1\circ1,-o[2,\bar{I}]] \]
### BINARY SEARCH FP ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Def</td>
<td>4</td>
<td>BIN</td>
<td>1</td>
</tr>
<tr>
<td>≡</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>eq</td>
<td>3</td>
<td>T̄</td>
<td>4</td>
</tr>
<tr>
<td>°</td>
<td>20</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>[ ]</td>
<td>17</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>+</td>
<td>5</td>
<td>–T̄</td>
<td>1</td>
</tr>
<tr>
<td>( )</td>
<td>1</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>select</td>
<td>4</td>
<td>2̄</td>
<td>1</td>
</tr>
<tr>
<td>;</td>
<td>5</td>
<td>Q</td>
<td>1</td>
</tr>
<tr>
<td>gt</td>
<td>1</td>
<td>T</td>
<td>1</td>
</tr>
<tr>
<td>BIN</td>
<td>2</td>
<td>select</td>
<td>1</td>
</tr>
<tr>
<td>Q</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lt</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Div</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>–</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tl</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>η1</td>
<td>19</td>
<td>N1</td>
<td>102</td>
</tr>
<tr>
<td>η2</td>
<td>11</td>
<td>N2</td>
<td>34</td>
</tr>
<tr>
<td>η</td>
<td>30</td>
<td>N</td>
<td>136</td>
</tr>
<tr>
<td>V</td>
<td>[\log 30 \cdot 136 = 667.337]</td>
<td>V² = 445338</td>
<td></td>
</tr>
</tbody>
</table>
BINARY SEARCH VAL

Function BIN(A:ARRAY[real]; VAL:REAL;
    SIZE,START:INT;
    RETURNS INT)
    if SIZE=1 then
        if A(START) = VAL then start
            else -1
        end
    else
        Begin
            Q:int := SIZE/2;
            T:int := Q+START-1
            RESULT IF A(T)>VAL THEN
                BIN(A,VAL,Q,START)
            ELSEIF A(T)<VAL THEN
                BIN(A,VAL,SIZE-Q,START+Q)
            ELSE
                T
            end
        end
    end
end
### BINARY SEARCH VAL ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>function</td>
<td>1</td>
<td>Bin</td>
<td>1</td>
</tr>
<tr>
<td>( )</td>
<td>6</td>
<td>A</td>
<td>6</td>
</tr>
<tr>
<td>:</td>
<td>5</td>
<td>real</td>
<td>2</td>
</tr>
<tr>
<td>Array</td>
<td>1</td>
<td>Val</td>
<td>6</td>
</tr>
<tr>
<td>[ ]</td>
<td>1</td>
<td>size</td>
<td>4</td>
</tr>
<tr>
<td>;</td>
<td>4</td>
<td>start</td>
<td>6</td>
</tr>
<tr>
<td>returns</td>
<td>1</td>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>if</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>=</td>
<td>2</td>
<td>Q</td>
<td>5</td>
</tr>
<tr>
<td>-</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Begin</td>
<td>1</td>
<td>T</td>
<td>4</td>
</tr>
<tr>
<td>:=</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bin</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ElseIf</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Result</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \eta_1 = 20 \quad N_1 = 46 \]
\[ \eta_2 = 11 \quad N_2 = 42 \]
\[ \eta = 31 \quad N = 88 \]

\[ V = \log 31 \cdot 88 = 435.969 \quad V^2 = 190069 \]
BINARY SEARCH ID

Y+Procedure BIN(A,val,size,start)

(if size=1 then
   if A[start] = val then start
   else -1

else
   (q+size div 2;
    t=q+start-1;
    r+if A[t]>Val then
       BIN(A,val,q,start)
    else if A[t]<val then
       BIN(A,val,size-q,Start+q)
    else t
    return r)
### BINARY SEARCH ID ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>4</td>
<td>Y</td>
<td>1</td>
</tr>
<tr>
<td>Procedure</td>
<td>1</td>
<td>Bin</td>
<td>1</td>
</tr>
<tr>
<td>( )</td>
<td>5</td>
<td>A</td>
<td>6</td>
</tr>
<tr>
<td>,</td>
<td>9</td>
<td>val</td>
<td>6</td>
</tr>
<tr>
<td>if</td>
<td>4</td>
<td>size</td>
<td>4</td>
</tr>
<tr>
<td>=</td>
<td>2</td>
<td>start</td>
<td>6</td>
</tr>
<tr>
<td>[ ]</td>
<td>3</td>
<td>l</td>
<td>3</td>
</tr>
<tr>
<td>=</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>div</td>
<td>1</td>
<td>q</td>
<td>5</td>
</tr>
<tr>
<td>+</td>
<td>2</td>
<td>t</td>
<td>4</td>
</tr>
<tr>
<td>&gt;</td>
<td>1</td>
<td>r</td>
<td>2</td>
</tr>
<tr>
<td>BIN</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>return</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\eta_1 &= 15 \quad N_1 = 41 \\
\eta_2 &= 11 \quad N_2 = 39 \\
\eta &= 26 \quad N = 80 \\
V &= \log 26 \cdot 80 = 376.035 \quad V^2 = 141402
\end{align*}
\]
BINARY SEARCH PASCAL

FUNCTION BIN(A:VECTOR;VAL:REAL;
    SIZE,START:INTEGER):INTEGER;

VAR Q,T:INTEGER;

BEGIN
    IF SIZE>1 THEN BEGIN
        Q := SIZE DIV 2;
        T := Q+START -1;
        IF A[T]>VAL THEN
            BIN := BIN(A,VAL,Q,START)
        ELSE IF A[T]<VAL THEN
            BIN := BIN(A,VAL,SIZE-Q,START+Q)
        ELSE
            BIN := T
    END
    ELSE
        IF VAL = A[START] THEN
            BIN := START
        ELSE
            BIN := -1
    END
# Binary Search Pascal Analysis

<table>
<thead>
<tr>
<th>Operators</th>
<th>Usage</th>
<th>Operands</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>1</td>
<td>Bin</td>
<td>6</td>
</tr>
<tr>
<td>( )</td>
<td>3</td>
<td>A</td>
<td>6</td>
</tr>
<tr>
<td>;</td>
<td>5</td>
<td>Vector</td>
<td>1</td>
</tr>
<tr>
<td>;</td>
<td>6</td>
<td>Val</td>
<td>6</td>
</tr>
<tr>
<td>Var</td>
<td>1</td>
<td>Real</td>
<td>1</td>
</tr>
<tr>
<td>Begin</td>
<td>2</td>
<td>Size</td>
<td>4</td>
</tr>
<tr>
<td>If</td>
<td>4</td>
<td>Start</td>
<td>6</td>
</tr>
<tr>
<td>&gt;</td>
<td>2</td>
<td>Integer</td>
<td>3</td>
</tr>
<tr>
<td>:=</td>
<td>7</td>
<td>Q</td>
<td>6</td>
</tr>
<tr>
<td>Div</td>
<td>1</td>
<td>T</td>
<td>5</td>
</tr>
<tr>
<td>+</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>-</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>[ ]</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bin</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>,</td>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \eta_1 = 16 \quad N_1 = 51 \]
\[ \eta_2 = 12 \quad N_2 = 48 \]
\[ \eta = 28 \quad N = 99 \]

\[ V = \log 28 \cdot 99 = 475.928 \quad V^2 = 226507 \]
BINARY SEARCH FWAL

BIN:<ls val sz strt>:=
   if equal:<sz l> then
      if equal:<val select:<ls strt>> then strt
      else -1
   else if greater:<val select:<ls T:<sz strt>>> then BIN:<ls val half:sz strt>
   elseif less:<val select:<ls T:<sz strt>>> then BIN:<ls val minus:<sz half:sz>
      sum:<strt half:sz>
   else T:<sz strt>;
   half:x div:<x 2>;
   T:<x y> := sum:<half:x y>;
   select:<ls n> := if equal:<n l> then first:ls
      else
         select:<rest:ls minus:<n l>>
BINARY SEARCH FWAL ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
<td>29</td>
<td>9s</td>
<td>9</td>
</tr>
<tr>
<td>&lt; &gt;</td>
<td>22</td>
<td>val</td>
<td>6</td>
</tr>
<tr>
<td>=</td>
<td>4</td>
<td>sz</td>
<td>9</td>
</tr>
<tr>
<td>if</td>
<td>3</td>
<td>strat</td>
<td>7</td>
</tr>
<tr>
<td>equal</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>select</td>
<td>4</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>elseif</td>
<td>2</td>
<td>BIN</td>
<td>1</td>
</tr>
<tr>
<td>greater</td>
<td>1</td>
<td>half</td>
<td>1</td>
</tr>
<tr>
<td>T</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>BIN</td>
<td>2</td>
<td>x</td>
<td>4</td>
</tr>
<tr>
<td>half</td>
<td>4</td>
<td>T</td>
<td>1</td>
</tr>
<tr>
<td>minus</td>
<td>2</td>
<td>y</td>
<td>2</td>
</tr>
<tr>
<td>sum</td>
<td>2</td>
<td>select</td>
<td>1</td>
</tr>
<tr>
<td>div</td>
<td>1</td>
<td>n</td>
<td>3</td>
</tr>
<tr>
<td>first</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rest</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>;</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>less</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \eta_1 = 18 \quad N_1 = 88 \]
\[ \eta_2 = 14 \quad N_2 = 49 \]
\[ \eta = 32 \quad N = 137 \]

\[ V = \log 32 \cdot 137 = 658.000 \quad \chi^2 = 469225 \]
GAUSSIAN ELIMINATION PCL

GAUSS=M: MATRIX+ 
(K=SIZE(M(,1)); 
TRIANG=(M,N): (MATRIX, INTEGER)→ 
IF N>K THEN M 
ELSE 
(I=N+1; 
TRIANG(ELIM(PIV(M,I,1+1),1),1) 
ENDIF;

PIV=(M,N,L): (MATRIX, INTEGER, INTEGER)→ 
IF M(N,N)≠0 OR L>K THEN M 
ELSE 
PIV((I,J): (N,INTEGER)→M(L,J) | 
(I,J): (L,INTEGER)→M(N,J) |M,L+1) 
ENDIF;

ELIME=(M,N): (MATRIX, INTEGER)→ 
(I,J): (INTEGER, INTEGER) \ I>N AND J>N→ 

BAKSOLEM: MATRIX* 
(K→M(K,K+1)/M(K,K) | 
X: INTEGER→ M(X,K+1) = 
(F K+1→0 | Z: INTEGER+F(Z+1) 
+M(X,Z) *BAKSOLEM(Z+1))(X+1)) 
/M(X,X)); 
BAKSOL(TRIANG(M,1))
GAUSSIAN ELIMINATION FCL ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>8</td>
<td>GAUSS</td>
<td>1</td>
</tr>
<tr>
<td>:</td>
<td>10</td>
<td>M</td>
<td>11</td>
</tr>
<tr>
<td>+</td>
<td>12</td>
<td>MATRIX</td>
<td>5</td>
</tr>
<tr>
<td>( )</td>
<td>40</td>
<td>K</td>
<td>10</td>
</tr>
<tr>
<td>SIZE</td>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>M</td>
<td>13</td>
<td>TRIANG</td>
<td>1</td>
</tr>
<tr>
<td>,</td>
<td>33</td>
<td>N</td>
<td>15</td>
</tr>
<tr>
<td>IF</td>
<td>2</td>
<td>INTEGER</td>
<td>10</td>
</tr>
<tr>
<td>&gt;</td>
<td>4</td>
<td>I</td>
<td>11</td>
</tr>
<tr>
<td>;</td>
<td>6</td>
<td>O</td>
<td>1</td>
</tr>
<tr>
<td>+</td>
<td>9</td>
<td>L</td>
<td>6</td>
</tr>
<tr>
<td>TRIANG</td>
<td>2</td>
<td>J</td>
<td>8</td>
</tr>
<tr>
<td>ELIM</td>
<td>1</td>
<td>X</td>
<td>6</td>
</tr>
<tr>
<td>PIV</td>
<td>2</td>
<td>Z</td>
<td>4</td>
</tr>
<tr>
<td>≠</td>
<td>1</td>
<td>PIV</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>ELIM</td>
</tr>
<tr>
<td>\</td>
<td>1</td>
<td>BAKSOL</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BAKSOL</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

η₁ = 22    N₁ = 160
η₂ = 17    N₂ = 102
η = 39     N = 262

v = log 39 . 262 = 1384.78   v² = 1917620
GAUSSIAN ELIMINATION FP
Def GAUSS=BACKSOLVE*TRIANG
Def TRIANG=eq*[length,l]→id;
apndl*[1,TRIANG*ELIM*PIV]
Def PIV eq*[length,l]→id;
eq*[l*1,0]+apndr*[PIV*t1,1];
id
Def ELIMl=(a+)⋅trans o
 [(aX)⋅distl*[neg⋅[1,2],3],4]
Def ELIM=(αELIMl*[l^1,2^1,l^1,l^2])⋅distl*[1,t1]
Def=BACKSOLVE SOLVEALL*[tlr,⋅*[2^1,l^1,l^1]]
Def SOLVEALL=null⋅l+2;
  SOLVEALL*[tlr^1,SOLVEl*[l^1,2]]
Def SOLVEl=apndl*[⋅[−*[l^1],
  (+)⋅(aX)⋅trans*[tl*l^1,2]],l^1],2]
GAUSSIAN ELIMINATION FP ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Def</td>
<td>8</td>
<td>t</td>
<td>2</td>
</tr>
<tr>
<td>≡</td>
<td>8</td>
<td>1</td>
<td>17</td>
</tr>
<tr>
<td>BACKSOLVE</td>
<td>1</td>
<td>o</td>
<td>1</td>
</tr>
<tr>
<td>°</td>
<td>42</td>
<td>id</td>
<td>3</td>
</tr>
<tr>
<td>TRIANG</td>
<td>2</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>eq</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>length</td>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>,</td>
<td>20</td>
<td>lr</td>
<td>4</td>
</tr>
<tr>
<td>+</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>;</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>apnd1</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ELIM</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PIV</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>apndr</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tl</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>trans</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>distl</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>α</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>neg</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>÷</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ELIM1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SOLVEALL</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>null</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tlr</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[ ]</td>
<td>18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SOLVEL</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>()</td>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
\( \eta_1 = 31 \quad N_1 = 154 \)
\( \eta_2 = 8 \quad N_2 = 37 \)
\( \eta = 39 \quad N = 191 \)
\[ V = \log 39 \cdot 191 = 1009.51 \quad v^2 = 1019110 \]
GAUSSIAN ELIMINATION VAL PAGE 1

FUNCTION GAUSS(A:array[real]; returns array[real])
BEGIN
    B:array[real] :=
    FOR I:INT := 1 STEP 1;
        C:array[real] := A
    DO IF I>=SIZE(A) THEN C
    ELSE
        ITER I := I+1;
        C := BEGIN
            D:array[real] :=
                IF C(I,I)^0 THEN C
                ELSE BEGIN
                    J:INT :=
                        FOR K:INT := I+1 STEP 1
                            DO IF K = SIZE(A) OR
                                C(K,I)^ 0 THEN K
                            ELSE ITER
                        END
                END
            E:array[real] :=
                IF C(I,J)=0 THEN empty[real]
                ELSE
                    FOR L:INT := I STEP 1;
                        F:array[real] := C
                    DO IF L>SIZE(A)+1 THEN F
                    ELSE iter
                        F := F[I,K:F(J,L);J,K:F(I,L)]
                    END
                END
            END
        END
    END;
RESULT E
END
GAUSSIAN ELIMINATION VAL PAGE 2

G: array[real] :=
FORALL M IN [1, SIZE(A)]
H: array[real] :=
FORALL N IN [1, SIZE(A)+1]
X: real :=
IF M<=I THEN D(M,N)
ELSE
(D(M,N) - (D(M,I)/D(I,I))*D(I,N))
END
CONSTRUCT X
END
CONSTRUCT H
END
RESULT G
END

END

END:

AA: array[real] :=
IF B(SIZE(A), SIZE(A)) = 0 THEN empty[real]
ELSE
B[SIZE(A), SIZE(A)+1: B(SIZE(A), SIZE(A)+1)/
B(SIZE(A), SIZE(A))]
END;
GAUSSIAN ELIMINATION VAL PAGE 3

AB: array[real] :=
    FOR II: INT := SIZE(A)-1 STEP -1;
    AC: array[real] := AA
    DO IF II=0 THEN AC
    ELSE ITER
        AC := BEGIN
            Szreal := FORALL IJ IN [II+1, SIZE(A)]
                EVAL PLUS
                    AC(II, IJ)*AC(IJ, SIZE(A)+1)
            END
            RESULT AC[II, SIZE(A)+1:
            (AC(II, SIZE(A)+1)-S/AC(II, II))
        END
    END
END

AD: array[real] := FORALL IK IN [1, SIZE(A)]
    CONSTRUCT AB(IK, SIZE(A)+1)
END
RESULT AD
END
END
<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUNCTION</td>
<td>1</td>
<td>GAUSS</td>
<td>1</td>
</tr>
<tr>
<td>(</td>
<td>42</td>
<td>A</td>
<td>22</td>
</tr>
<tr>
<td>:</td>
<td>24</td>
<td>real</td>
<td>17</td>
</tr>
<tr>
<td>array</td>
<td>13</td>
<td>B</td>
<td>5</td>
</tr>
<tr>
<td>[ ]</td>
<td>18</td>
<td>I</td>
<td>17</td>
</tr>
<tr>
<td>;</td>
<td>10</td>
<td>int</td>
<td>5</td>
</tr>
<tr>
<td>returns</td>
<td>1</td>
<td>l</td>
<td>20</td>
</tr>
<tr>
<td>BEGIN</td>
<td>4</td>
<td>C</td>
<td>7</td>
</tr>
<tr>
<td>::=</td>
<td>22</td>
<td>D</td>
<td>6</td>
</tr>
<tr>
<td>FOR</td>
<td>4</td>
<td>O</td>
<td>5</td>
</tr>
<tr>
<td>STEP</td>
<td>4</td>
<td>J</td>
<td>4</td>
</tr>
<tr>
<td>IF</td>
<td>8</td>
<td>K</td>
<td>4</td>
</tr>
<tr>
<td>&gt;=</td>
<td>1</td>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>SIZE</td>
<td>20</td>
<td>L</td>
<td>4</td>
</tr>
<tr>
<td>ITER</td>
<td>4</td>
<td>F</td>
<td>6</td>
</tr>
<tr>
<td>+</td>
<td>10</td>
<td>G</td>
<td>2</td>
</tr>
<tr>
<td>,</td>
<td>21</td>
<td>M</td>
<td>5</td>
</tr>
<tr>
<td>≠</td>
<td>2</td>
<td>H</td>
<td>2</td>
</tr>
<tr>
<td>=</td>
<td>4</td>
<td>N</td>
<td>4</td>
</tr>
<tr>
<td>empty</td>
<td>2</td>
<td>X</td>
<td>2</td>
</tr>
<tr>
<td>&gt;</td>
<td>1</td>
<td>AA</td>
<td>2</td>
</tr>
<tr>
<td>RESULT</td>
<td>4</td>
<td>AB</td>
<td>2</td>
</tr>
<tr>
<td>FORALL</td>
<td>4</td>
<td>II</td>
<td>8</td>
</tr>
<tr>
<td>&lt;=</td>
<td>1</td>
<td>AC</td>
<td>8</td>
</tr>
<tr>
<td>-</td>
<td>4</td>
<td>IJ</td>
<td>3</td>
</tr>
<tr>
<td>/</td>
<td>3</td>
<td>AD</td>
<td>2</td>
</tr>
<tr>
<td>*</td>
<td>2</td>
<td>IK</td>
<td>2</td>
</tr>
<tr>
<td>CONSTRUCT</td>
<td>3</td>
<td>S</td>
<td>2</td>
</tr>
<tr>
<td>EVAL PLUS</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
\[ \eta_1 = 30 \quad N_1 = 239 \]
\[ \eta_2 = 28 \quad N_2 = 169 \]
\[ \eta = 58 \quad N = 408 \]

\[ V = \log 58 \cdot 408 = 2390.06 \quad V^2 = 5712390 \]
GAUSSIAN ELIMINATION ID

gauss->procedure (A, size)
(B, E0+)
(initial C=A;
i+1;
E1=FALSE
While i<size and not E1 do
new C, E1+
(D, E2+ if C[i,i]≠0 then C, false
else
(j+(initial K=i+1
while K<=size and C[K,i]=0 do
new K=K+1
return K);
E3=C[i,j]=0;
E+if E3 then ∧ else
(initial F=C
for l from i to size +1 do
new F=F+[i,l]F[j,l]+[j,l]F[i,l]
return F)
return F)
return E, E3);
G+(initial H=D
for m from i+1 to size do
new H+(initial AA+H
for n from i+1 to size +1 do
new AA[m,n]=D[m,n]
-(D[m,i]/D[i,i])*D[i,n]
return AA))
return H)
return G, E2);
new i=i+1;
return C, E1);
AB+if EO or B[size, size] = 0 then \Lambda
else
(\text{initial } \text{AC}+\Lambda+[\text{size}](B[size, size]/B[size/size])
for ii from 1 to size - 1 do
ij+size-ii,

S+(\text{initial } t 0
for ik from ij+1 to size do
new t+t+B[ij, ik]*AC[ik]
return t)
new AC[ii] (B[ii, size+1]-S)/B[ii, ii]
return AC)
return AB)
GAUSSIAN ELIMINATION ID ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>27</td>
<td>gauss</td>
<td>1</td>
</tr>
<tr>
<td>procedure</td>
<td>1</td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>size</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>26</td>
<td>B</td>
<td>7</td>
</tr>
<tr>
<td>initial</td>
<td>7</td>
<td>EO</td>
<td>2</td>
</tr>
<tr>
<td>;</td>
<td>9</td>
<td>C</td>
<td>8</td>
</tr>
<tr>
<td>while</td>
<td>2</td>
<td>i</td>
<td>18</td>
</tr>
<tr>
<td>&lt;</td>
<td>1</td>
<td>l</td>
<td>12</td>
</tr>
<tr>
<td>not</td>
<td>1</td>
<td>El</td>
<td>3</td>
</tr>
<tr>
<td>new</td>
<td>8</td>
<td>false</td>
<td>2</td>
</tr>
<tr>
<td>[ ]</td>
<td>21</td>
<td>D</td>
<td>6</td>
</tr>
<tr>
<td>≠</td>
<td>1</td>
<td>E2</td>
<td>2</td>
</tr>
<tr>
<td>+</td>
<td>13</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>&lt;=</td>
<td>1</td>
<td>j</td>
<td>4</td>
</tr>
<tr>
<td>=</td>
<td>3</td>
<td>K</td>
<td>6</td>
</tr>
<tr>
<td>return</td>
<td>10</td>
<td>E3</td>
<td>2</td>
</tr>
<tr>
<td>if</td>
<td>3</td>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>for</td>
<td>5</td>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>-</td>
<td>4</td>
<td>F</td>
<td>6</td>
</tr>
<tr>
<td>/</td>
<td>3</td>
<td>\</td>
<td>5</td>
</tr>
<tr>
<td>*</td>
<td>2</td>
<td>G</td>
<td>2</td>
</tr>
<tr>
<td>And</td>
<td>2</td>
<td>H</td>
<td>4</td>
</tr>
<tr>
<td>or</td>
<td>1</td>
<td>m</td>
<td>4</td>
</tr>
<tr>
<td>A</td>
<td>3</td>
<td>n</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AB</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AC</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ii</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ij</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>t</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ik</td>
<td>3</td>
</tr>
</tbody>
</table>
\[ \eta_1 = 32 \quad N_1 = 164 \]
\[ \eta_2 = 32 \quad N_2 = 154 \]
\[ \eta = 55 \quad N = 318 \]

\[ V = \log 55 \cdot 318 = 1838.47 \quad V^2 = 3379970 \]
GAUSSIAN ELIMINATION PASCAL
FUNCTION GAUSS(A:MATRIX;N:INTEGER):VECTOR;
VAR I,J,K:INTEGER;
   FOUND,ERROR:BOOLEAN;
   S,Q:REAL;
BEGIN
   ERROR := FALSE;
   I := 1;
   WHILE (I<N) AND NOT ERROR DO
   BEGIN
      IF A[I,I]=0 THEN DO
      BEGIN
         J := I;
         FOUND := FALSE;
         WHILE (J<N) AND NOT FOUND DO
         BEGIN
            J := J+1;
            IF A[J,I]<>0 THEN
               FOUND := TRUE
         END;
         ERROR := NOT FOUND;
      END;
      IF FOUND THEN
      BEGIN
         FOR K := 1 TO N+1 DO
         BEGIN
            TEMP := A[I,K];
         END
      END;
   END;
IF NOT ERROR THEN
FOR J := I+1 TO N DO
BEGIN
  Q := -A[J,I]/A[I,I];
  A[J,I] := 0;
  FOR K := I+1 TO N+1 DO
END; I := I+1
END;
ERROR := ERROR OR (A[N,N]=0);
IF NOT ERROR THEN
BEGIN
  FOR I := N-1 DOWNTO 1 DO
  BEGIN
    S := 0;
    FOR J := N DOWNTO N-I+1 DO
      S := S+A[I,J]*A[J,N+1];
  END;
  FOR I := 1 TO N DO
    VAUSS[I] := A[I,N+1]
END
ELSE GAUSS := NULL
END
### Gaussian Elimination Pascal Analysis

<table>
<thead>
<tr>
<th>Operators</th>
<th>Usage</th>
<th>Operands</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>1</td>
<td>Gauss</td>
<td>1</td>
</tr>
<tr>
<td>( )</td>
<td>5</td>
<td>A</td>
<td>23</td>
</tr>
<tr>
<td>:</td>
<td>6</td>
<td>Matrix</td>
<td>1</td>
</tr>
<tr>
<td>;</td>
<td>24</td>
<td>N</td>
<td>22</td>
</tr>
<tr>
<td>VAR</td>
<td>1</td>
<td>Integer</td>
<td>2</td>
</tr>
<tr>
<td>,</td>
<td>26</td>
<td>Vector</td>
<td>1</td>
</tr>
<tr>
<td>BEGIN</td>
<td>8</td>
<td>I</td>
<td>27</td>
</tr>
<tr>
<td>:=</td>
<td>27</td>
<td>J</td>
<td>16</td>
</tr>
<tr>
<td>&lt;</td>
<td>2</td>
<td>K</td>
<td>9</td>
</tr>
<tr>
<td>AND</td>
<td>2</td>
<td>Found</td>
<td>6</td>
</tr>
<tr>
<td>NOT</td>
<td>5</td>
<td>Error</td>
<td>8</td>
</tr>
<tr>
<td>WHILE</td>
<td>2</td>
<td>Boolean</td>
<td>1</td>
</tr>
<tr>
<td>IF-THEN</td>
<td>4</td>
<td>S</td>
<td>5</td>
</tr>
<tr>
<td>[ ]</td>
<td>23</td>
<td>Q</td>
<td>3</td>
</tr>
<tr>
<td>=</td>
<td>1</td>
<td>Real</td>
<td>1</td>
</tr>
<tr>
<td>+</td>
<td>15</td>
<td>1</td>
<td>17</td>
</tr>
<tr>
<td>≠</td>
<td>1</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>FOR-TO</td>
<td>4</td>
<td>FALSE</td>
<td>2</td>
</tr>
<tr>
<td>IF-THEN-ELSE</td>
<td>1</td>
<td>TRUE</td>
<td>1</td>
</tr>
<tr>
<td>FOR-DOWNTO</td>
<td>2</td>
<td>NULL</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\eta_1 = 24 \quad N_1 = 170 \\
\eta_2 = 20 \quad N_2 = 152 \\
\eta = 44 \quad N = 322 \\
V = \log 44 \cdot 322 = 1757.94 \quad V^2 = 3090350 \\
\]
GAUSSIAN ELIMINATION FWAL

\[
\text{GAUSS} : M \equiv \text{Backsolve} : \text{reverse} : \text{triang} : M
\]

\[
\text{TRIANG} : M \equiv \begin{cases} 
\text{if equal} : \text{length} : M = 1 & \text{then } M \\
\text{else} & \\
\text{cons} : \begin{cases} 
\text{first} : M & \\
\text{TRIANG} : \text{ELIM} : \text{PIV} : M
\end{cases}
\end{cases}
\]

\[
\text{PIV} : M \equiv \begin{cases} 
\text{if equal} : \text{length} : M = 1 & \text{then } M \\
\text{elseif equal} : \text{first} : \text{first} : M = 0 & \text{then} \\
\text{consr} : \begin{cases} 
\text{piv} : \text{rest} : M & \\
\text{first} : M
\end{cases}
\end{cases}
\]

\[
\text{consr} : \begin{cases} 
\text{ls e} & \equiv \begin{cases} 
\text{if null} : \text{ls} & \text{then } e \\
\text{else} & \\
\text{cons} : \begin{cases} 
\text{first} : \text{ls} & \\
\text{consr} : \begin{cases} 
\text{rest} : \text{ls e}
\end{cases}
\end{cases}
\end{cases}
\end{cases}
\]

\[
\text{ELIM} : M \equiv \begin{cases} 
\text{ELIM} : \begin{cases} 
\text{first} : M & \\
\text{rest} : M
\end{cases}
\end{cases}
\]

\[
\text{ELIM} : \begin{cases} 
\text{la lb}
\end{cases}
\equiv \begin{cases} 
\text{sum} & : \\
\text{product} & : \begin{cases} 
\text{neg} : \text{quotient} : \begin{cases} 
\text{first} : \text{lb} & \\
\text{first} : \text{la}
\end{cases}
\end{cases}
\end{cases}
\]

\[
\text{rest} : \text{lb}
\]

\[
\text{reverse} : \text{ls} & \equiv \begin{cases} 
\text{if null} : \text{ls} & \text{then } < > \\
\text{else} & \\
\text{consr} : \begin{cases} 
\text{reverse} : \text{rest} : \text{ls first} : \text{ls}
\end{cases}
\end{cases}
\]

\[
\text{Backsolve} : M \equiv \text{solveall} : \begin{cases} 
\text{rest} : M
\end{cases}
\]

\[
\text{solveall} : \begin{cases} 
\text{M ps}
\end{cases} & \equiv \begin{cases} 
\text{if null} : \text{AA} & \text{then } \text{ps} \\
\text{else} & \\
\text{solveall} : \begin{cases} 
\text{rest} : M
\end{cases}
\end{cases}
\]

\[
\text{Solve} : \begin{cases} 
\text{V ps}
\end{cases} & \equiv \begin{cases} 
\text{quotient} : \begin{cases} 
\text{minus} : \begin{cases} 
\text{first} : \text{reverse} : \text{V}
\end{cases}
\end{cases}
\end{cases}
\]

\[
\text{sum} : \begin{cases} 
\text{product} & : \begin{cases} 
\text{reverse} : \text{rest} & : \begin{cases} 
\text{reverse} : \text{rest} : \text{V}
\end{cases}
\end{cases}
\end{cases}
\]

\[
\text{ps}
\]

\[
\text{first} : \text{V}
\]
<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
<td>74</td>
<td>gauss</td>
<td>1</td>
</tr>
<tr>
<td>=</td>
<td>10</td>
<td>M</td>
<td>25</td>
</tr>
<tr>
<td>Backsolve</td>
<td>1</td>
<td>triang</td>
<td>1</td>
</tr>
<tr>
<td>reverse</td>
<td>5</td>
<td>l</td>
<td>2</td>
</tr>
<tr>
<td>triang</td>
<td>3</td>
<td>PIV</td>
<td>1</td>
</tr>
<tr>
<td>if</td>
<td>5</td>
<td>o</td>
<td>1</td>
</tr>
<tr>
<td>equal</td>
<td>3</td>
<td>consr</td>
<td>1</td>
</tr>
<tr>
<td>length</td>
<td>2</td>
<td>ls</td>
<td>8</td>
</tr>
<tr>
<td>cons</td>
<td>3</td>
<td>e</td>
<td>2</td>
</tr>
<tr>
<td>first</td>
<td>15</td>
<td>elim</td>
<td>1</td>
</tr>
<tr>
<td>Elim</td>
<td>1</td>
<td>eliml</td>
<td>1</td>
</tr>
<tr>
<td>Piv</td>
<td>2</td>
<td>la</td>
<td>3</td>
</tr>
<tr>
<td>consr</td>
<td>4</td>
<td>lb</td>
<td>3</td>
</tr>
<tr>
<td>rest</td>
<td>10</td>
<td>reverse</td>
<td>1</td>
</tr>
<tr>
<td>null</td>
<td>3</td>
<td>Backsolve</td>
<td>1</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>33</td>
<td>ps</td>
<td>6</td>
</tr>
<tr>
<td>;</td>
<td>9</td>
<td>V</td>
<td>4</td>
</tr>
<tr>
<td>eliml</td>
<td>1</td>
<td>Solveall</td>
<td>1</td>
</tr>
<tr>
<td>*</td>
<td>6</td>
<td>Solve1</td>
<td>1</td>
</tr>
<tr>
<td>sum</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>product</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>neg</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>quotient</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>solveall</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>solve1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>minus</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>elseif</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
\[ \eta_1 = 28 \quad N_1 = 204 \]
\[ \eta_2 = 19 \quad N_2 = 64 \]
\[ \eta = 47 \quad N = 268 \]

\[ V = \log 47 \cdot 268 = 1488.63 \quad v^2 = 2216020 \]
FAST FOURIER FCL

\[ FFT \equiv (N,A,W):(INTEGER,VECTOR,COMPLEX) \rightarrow \]

\[ IF \ N=1 \ THEN \ l^A(l) \]

\[ ELSE \]

\[ K \equiv N \ DIV \ 2; \]

\[ B \equiv FFT(K,A \circ (x:\INTEGER \rightarrow 2x),w \ast w); \]

\[ C \equiv FFT(K,A \circ x:\INTEGER \rightarrow 2x-1),w \ast w); \]

\[ x:(1..K) \rightarrow B(x) + w \ast (x-1) \ast C(x) | \]

\[ x:(K+1..N) \rightarrow B(x-K) - w \ast (x-1-K) \ast C(x-K) \]

\[ ENDIF \]
### FAST FOURIER FCL ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>$=$</td>
<td>4</td>
<td>FFT</td>
<td>1</td>
</tr>
<tr>
<td>( )</td>
<td>16</td>
<td>N</td>
<td>3</td>
</tr>
<tr>
<td>,</td>
<td>8</td>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>:</td>
<td>5</td>
<td>W</td>
<td>7</td>
</tr>
<tr>
<td>$\rightarrow$</td>
<td>6</td>
<td>INTEGER</td>
<td>3</td>
</tr>
<tr>
<td>IF</td>
<td>1</td>
<td>VECTOR</td>
<td>1</td>
</tr>
<tr>
<td>$=$</td>
<td>1</td>
<td>COMPLEX</td>
<td>1</td>
</tr>
<tr>
<td>PIV</td>
<td>1</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>;</td>
<td>2</td>
<td>K</td>
<td>8</td>
</tr>
<tr>
<td>FFT</td>
<td>2</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>$^*$</td>
<td>2</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>$\cdot$</td>
<td>6</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>$-$</td>
<td>7</td>
<td>x</td>
<td>12</td>
</tr>
<tr>
<td>$..$</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$**$</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$+$</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$!$</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$n_1 = 20 \quad N_1 = 73$

$n_2 = 13 \quad N_2 = 52$

$n = 33 \quad N = 125$

$V = \log 33 \cdot 125 = 630.549 \quad V^2 = 397649$
FAST FOURIER PFP

Def FFT ≡ eq°[1, 1]→2;

concat ° [(\alpha+)°trans, (\alpha-)°trans]
°[1, (\alpha\times)°trans°[2, 3]]
°[1,2, Plis°[3, o[4, 1], [1]]]
°[(\alpha\FFT)°[[DIV°[2, 2], 1°1, X°[3, 3]],
[DIV°[2, 2], 1°2, X°[3, 3]]], 1, 3]
°[split°[\phi, \phi, 2], 1, 3]

Def split ≡ null °3 + tlr;

split°[apndro[1, 3°1], apndro[2, 3°2], t1°t1]

Def plis ≡ eq°[2, 0]→3;

plis°[1, o[2, 1], apndro[3, X°[1r°3, 1]]]

Def concat ≡ null ° 2→1;

concat°[apndro[1, 2°1], +1°2]
### FAST FOURIER FP ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Def</td>
<td>4</td>
<td>FFT</td>
<td>1</td>
</tr>
<tr>
<td>≡</td>
<td>4</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>eq</td>
<td>2</td>
<td>l</td>
<td>4</td>
</tr>
<tr>
<td>*</td>
<td>38</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>[ ]</td>
<td>27</td>
<td>3</td>
<td>14</td>
</tr>
<tr>
<td>,</td>
<td>36</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>+</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>;</td>
<td>4</td>
<td>φ</td>
<td>2</td>
</tr>
<tr>
<td>concat</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>( )</td>
<td>4</td>
<td>lr</td>
<td>1</td>
</tr>
<tr>
<td>α</td>
<td>4</td>
<td>split</td>
<td>1</td>
</tr>
<tr>
<td>+</td>
<td>1</td>
<td>plis</td>
<td>1</td>
</tr>
<tr>
<td>trans</td>
<td>3</td>
<td>concat</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Plis</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FFT</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>div</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>split</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>null</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tlr</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>apndr</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tl</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\eta_1 = 23 \quad N_1 = 157 \\
\eta_2 = 13 \quad N_2 = 58 \\
\eta = 36 \quad N = 215 \\
V = \log 36 \cdot 215 = 1111.53 \quad V^2 = 1235500
\]
FAST FOURIER VAL

Function FFT(N:int;
    A:array[complex];
    W:complex;
    returns array [complex])

IF N=1 then A else
BEGIN
    Nl:int := N/2;
    B,C:array[complex] :=
    FORALL I in [1,Nl]
        construct A(2*I),A(2*I-1)
    end;
    D,E:array[complex] :=
    FFT(Nl,B,W*W),FFT(Nl,C,W*W);
    F,G:array[complex] :=
    FORALL J in [1,Nl]
        CONSTRUCT D(J)+W**(J-1)*E(J),
                       D(J)-W**(J-1)*E(J)
    end
    result F||G
end
end
### Fast Fourier Val Analysis

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>1</td>
<td>FFT</td>
<td>1</td>
</tr>
<tr>
<td>( )</td>
<td>11</td>
<td>N</td>
<td>3</td>
</tr>
<tr>
<td>;</td>
<td>7</td>
<td>int</td>
<td>2</td>
</tr>
<tr>
<td>;</td>
<td>6</td>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td>array</td>
<td>5</td>
<td>complex</td>
<td>6</td>
</tr>
<tr>
<td>[ ]</td>
<td>5</td>
<td>W</td>
<td>7</td>
</tr>
<tr>
<td>returns</td>
<td>1</td>
<td>Nl</td>
<td>5</td>
</tr>
<tr>
<td>if</td>
<td>1</td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>=</td>
<td>1</td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>begin</td>
<td>1</td>
<td>I</td>
<td>3</td>
</tr>
<tr>
<td>:=</td>
<td>4</td>
<td>D</td>
<td>3</td>
</tr>
<tr>
<td>/</td>
<td>1</td>
<td>E</td>
<td>3</td>
</tr>
<tr>
<td>Forall</td>
<td>2</td>
<td>J</td>
<td>7</td>
</tr>
<tr>
<td>Construct</td>
<td>2</td>
<td>F</td>
<td>2</td>
</tr>
<tr>
<td>*</td>
<td>6</td>
<td>G</td>
<td>2</td>
</tr>
<tr>
<td>-</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>+</td>
<td>1</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>**</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>,</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>FFT</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>result</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ n_1 = 22 \quad N_1 = 75 \]
\[ n_2 = 17 \quad N_2 = 59 \]
\[ n = 39 \quad N = 134 \]
\[ V = \log 39 \cdot 134 = 708.244 \quad V^2 = 501610 \]
FAST FOURIER ID

Y-t-procedure FFT(n,a,w)
  if n=1 then a else
  (nl+n div 2;
  b,c+(initial d+\Lambda;
    e+\Lambda
    for i from 1 to nl do
      new d[i]+a[2*i];
      new e[i]+a[2*i-l]
    return d,e);
  f+FFT(nl,b,w*w);
  g+FFT(nl,c,w*w);
  n+(initial aa+\Lambda
    for j from 1 to nl do
      new aa+aa+[j](f[j]+w+(j-l)*g[j])
      +[j+nl](f[j]-w+(j-l)*g[j])
    return aa)
  return h)
### FAST FOURIER ID ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>12</td>
<td>Y</td>
<td>1</td>
</tr>
<tr>
<td>Procedure</td>
<td>1</td>
<td>FFT</td>
<td>1</td>
</tr>
<tr>
<td>( )</td>
<td>10</td>
<td>n</td>
<td>3</td>
</tr>
<tr>
<td>,</td>
<td>8</td>
<td>a</td>
<td>4</td>
</tr>
<tr>
<td>if</td>
<td>1</td>
<td>w</td>
<td>7</td>
</tr>
<tr>
<td>=</td>
<td>1</td>
<td>nl</td>
<td>6</td>
</tr>
<tr>
<td>div</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>;</td>
<td>6</td>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>initial</td>
<td>2</td>
<td>c</td>
<td>2</td>
</tr>
<tr>
<td>for</td>
<td>2</td>
<td>i</td>
<td>3</td>
</tr>
<tr>
<td>new</td>
<td>3</td>
<td>l</td>
<td>6</td>
</tr>
<tr>
<td>[ ]</td>
<td>10</td>
<td>f</td>
<td>3</td>
</tr>
<tr>
<td>*</td>
<td>6</td>
<td>g</td>
<td>4</td>
</tr>
<tr>
<td>FFT</td>
<td>2</td>
<td>aa</td>
<td>3</td>
</tr>
<tr>
<td>+</td>
<td>4</td>
<td>j</td>
<td>9</td>
</tr>
<tr>
<td>return</td>
<td>3</td>
<td>d</td>
<td>3</td>
</tr>
</tbody>
</table>

\[ \eta_1 = 18 \quad N_1 = 78 \]
\[ \eta_2 = 19 \quad N_2 = 68 \]
\[ \eta = 37 \quad N = 146 \]

\[ V = \log 37 \cdot 146 = 760.580 \quad V^2 = 578482 \]
FAST FOURIER PASCAL

FUNCTION FFT(N:INTEGER; A:VECTOR;
               W:COMPLEX):VECTOR;

VAR B,C,D,E:VECTOR;
    N1,I:INTEGER;
    WP:COMPLEX;

BEGIN
  ELSE BEGIN
    N1 := NDIV 2;
    FOR I := 1 TO n1 DO
      BEGIN
      END;
    D := FFT(N1,B,W*W);
    E := FFT(N1,C,W*W);
    WP := 1/W,
    FOR I := 1 to N1 DO
      BEGIN
        WP := W*WP;
        FFT[I] := D[I] + WP*E[I];
        FFT[I+N1] := D[I] - WP*E[i]
      END
  END
END
## FAST FOURIER PASCAL ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUNCTION</td>
<td>1</td>
<td>FFT</td>
<td>1</td>
</tr>
<tr>
<td>( )</td>
<td>3</td>
<td>N</td>
<td>3</td>
</tr>
<tr>
<td>:</td>
<td>7</td>
<td>INTEGER</td>
<td>2</td>
</tr>
<tr>
<td>;</td>
<td>14</td>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td>VAR</td>
<td>1</td>
<td>VECTOR</td>
<td>2</td>
</tr>
<tr>
<td>,</td>
<td>8</td>
<td>W</td>
<td>7</td>
</tr>
<tr>
<td>BEGIN</td>
<td>4</td>
<td>COMPLEX</td>
<td>2</td>
</tr>
<tr>
<td>IF</td>
<td>1</td>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>=</td>
<td>1</td>
<td>C</td>
<td>3</td>
</tr>
<tr>
<td>[ ]</td>
<td>12</td>
<td>D</td>
<td>4</td>
</tr>
<tr>
<td>:=</td>
<td>10</td>
<td>E</td>
<td>4</td>
</tr>
<tr>
<td>DIV</td>
<td>1</td>
<td>N1</td>
<td>7</td>
</tr>
<tr>
<td>FOR</td>
<td>2</td>
<td>I</td>
<td>13</td>
</tr>
<tr>
<td>*</td>
<td>6</td>
<td>WP</td>
<td>6</td>
</tr>
<tr>
<td>-</td>
<td>2</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>FFT</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>/</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ n_1 = 18 \quad N_1 = 78 \]
\[ n_2 = 16 \quad N_2 = 71 \]
\[ n = 34 \quad N = 149 \]

\[ V = \log 34 \cdot 149 = 758.032 \quad V^2 = 574613 \]
FAST FOURIER FWAL

\[
\text{split}: <A \ B \ C> \equiv \text{if null}: A \ \text{then} \ <B \ C> \\
\quad \text{else} \\
\quad \text{split}: <\text{rest}: \text{rest}: A> \\
\quad \text{cons}: <\text{first}: A \ B> \\
\quad \text{cons}: <2: A \ C>>;
\]

\[
\text{Y}: <N \ A \ W> \equiv \text{CONS}: <\text{Plis}: <\text{div}: <N \ 2\text{W}>1>> \\
\quad <\text{FFT}*>: <<\text{div}: <N 2>>*> \\
\quad \text{split}: <A >> >>>> \\
\quad \text{<product>: <W W>*>>>;
\]

\[
\text{Plis}: <N \ W \ LS> \equiv \text{if equal}: <N 1> \ \text{then} \ LS \\
\quad \text{else} \\
\quad \text{Plis}: <\text{minus}: <N 1 \ W> \\
\quad \text{cons}: <l<product*>: <<w*> <s>>>>;
\]

\[
\text{FFT}: <N \ A \ W> \equiv \text{if equal}: <N 1> \ \text{then} \ A \\
\quad \text{else} \\
\quad \text{concat}: <<\text{SUM}*>: <2: Y: <N A W> \\
\quad \quad <\text{product}*>: <first: Y: <N A W> \\
\quad \quad \quad 3: Y: <N A W>>> \\
\quad <\text{minus}*>: <2: Y: <N A W> \\
\quad \quad <\text{product}*>: <first: Y: <N A W> \\
\quad \quad \quad 3: Y: <N A W>>> ;
\]

\[
\text{concat}: <L A \ LB> \equiv \text{if null}: LB \ \text{then} \ LA \\
\quad \text{else} \\
\quad \text{concat}: <\text{consr}: <L A \ first: LB> \ \text{rest}: LB>>;
\]

\[
\text{consr}: <L S \ E> \equiv \text{if null}: LS \ \text{then} \ <E> \\
\quad \text{else} \\
\quad \text{cons}: <\text{first}: LS \ \text{consr}: <\text{rest}: LS \ E>.
\]
FAST FOURIER FWAL ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
<td>54</td>
<td>split</td>
<td>1</td>
</tr>
<tr>
<td>&lt; &gt;</td>
<td>51</td>
<td>A</td>
<td>15</td>
</tr>
<tr>
<td>≡</td>
<td>6</td>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>if</td>
<td>5</td>
<td>C</td>
<td>3</td>
</tr>
<tr>
<td>null</td>
<td>3</td>
<td>N</td>
<td>14</td>
</tr>
<tr>
<td>split</td>
<td>2</td>
<td>W</td>
<td>14</td>
</tr>
<tr>
<td>rest</td>
<td>4</td>
<td>Y</td>
<td>1</td>
</tr>
<tr>
<td>cons</td>
<td>5</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>first</td>
<td>5</td>
<td>LS</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>Plis</td>
<td>1</td>
</tr>
<tr>
<td>Plis</td>
<td>2</td>
<td>LA</td>
<td>3</td>
</tr>
<tr>
<td>div</td>
<td>2</td>
<td>LB</td>
<td>4</td>
</tr>
<tr>
<td>FFT</td>
<td>1</td>
<td>FFT</td>
<td>1</td>
</tr>
<tr>
<td>*</td>
<td>9</td>
<td>concat</td>
<td>1</td>
</tr>
<tr>
<td>product</td>
<td>4</td>
<td>consr</td>
<td>1</td>
</tr>
<tr>
<td>;</td>
<td>5</td>
<td>E</td>
<td>3</td>
</tr>
<tr>
<td>equal</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>minus</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>concat</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sum</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>consr</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \eta_1 = 23 \quad E_1 = 178 \]

\[ \eta_2 = 17 \quad E_2 = 77 \]

\[ \eta = 40 \quad E = 255 \]

\[ V = \log 40 \cdot 255 = 1357.09 \quad V^2 = 1841690 \]
EIGHT QUEENS FCL

Q8 ≡ (SRCH≡(V,N):(VECTOR,INTEGER)⁺

WHILE I<=8 AND UNDEF(X);

INIT≡(I=1;X=UNDEFINED);

RESULT≡X;

I≡I+1;

X≡IF OK (N+I|V) THEN

IF N=8 THEN N+I|V
ELSE SRCH(N+I|V,N+1)
ENDIF

ELSE UNDEFINED
ENDIF

ENDWHILE;

OK W:VECTOR⁺

(S≡SIZE(W);

A≡DELETE(W,S);

IF S=1 THEN TRUE
ELSE

NOT (INSERT (OR) (equal(W(S))+A)

OR INSERT(OR) (equal(W(S)+S)⁺

(X2:INTEGER+A(X2)+X2))

OR INSERT(OR) (equal(W(S)-S)⁺

(X3:INTEGER+A(X3)-X3)))

ENDIF;

SRCH(UNDEFINED,1))
### EIGHT QUEENS FCL ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>11</td>
<td>Q8</td>
<td>1</td>
</tr>
<tr>
<td>( )</td>
<td>28</td>
<td>SRCH</td>
<td>1</td>
</tr>
<tr>
<td>,</td>
<td>4</td>
<td>V</td>
<td>4</td>
</tr>
<tr>
<td>:</td>
<td>4</td>
<td>N</td>
<td>5</td>
</tr>
<tr>
<td>+</td>
<td>7</td>
<td>VECTOR</td>
<td>2</td>
</tr>
<tr>
<td>WHILE</td>
<td>1</td>
<td>INTEGER</td>
<td>3</td>
</tr>
<tr>
<td>&lt;=</td>
<td>1</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>KND</td>
<td>1</td>
<td>I</td>
<td>7</td>
</tr>
<tr>
<td>UNDEF</td>
<td>1</td>
<td>INIT</td>
<td>1</td>
</tr>
<tr>
<td>;</td>
<td>8</td>
<td>X</td>
<td>4</td>
</tr>
<tr>
<td>+</td>
<td>1</td>
<td>l</td>
<td>5</td>
</tr>
<tr>
<td>IF</td>
<td>3</td>
<td>UNDEFINED</td>
<td>3</td>
</tr>
<tr>
<td>OK</td>
<td>1</td>
<td>RESULT</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>OK</td>
</tr>
<tr>
<td>=</td>
<td>2</td>
<td>W</td>
<td>3</td>
</tr>
<tr>
<td>SRCH</td>
<td>2</td>
<td>S</td>
<td>8</td>
</tr>
<tr>
<td>SIZE</td>
<td>1</td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>DELETE</td>
<td>1</td>
<td>OR</td>
<td>3</td>
</tr>
<tr>
<td>NOT</td>
<td>1</td>
<td>X2</td>
<td>3</td>
</tr>
<tr>
<td>OR</td>
<td>2</td>
<td>X3</td>
<td>3</td>
</tr>
<tr>
<td>INSERT</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>equal</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>°</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
\[ n_1 = 26 \quad N_1 = 102 \]
\[ n_2 = 20 \quad N_2 = 62 \]
\[ n = 46 \quad N = 164 \]

\[ V = \log 46 \cdot 164 = 905.864 \quad V^2 = 820590 \]
EIGHT QUEENS FP

Def Q8 ≡ SRCH°[I, I, φ]

Def SRCH ≡ gt°[1, φ] + 3;
Λ°[gt°[2, φ], eq°[1, I]] → φ;
gt°[2, φ] +
SRCH°[−0[1, I], I, tlr o 3];
OK°apndr°[3, 2] +
SRCH°[+0[1, I], I, apndr°[3, 2]]; SRCH°[1, +0[2, I], 3]

Def OK ≡ eq°[length, I] + true;

−0/V) ° [(/V) ° (aeq) ° distr°[tlr, lr],
(/V) ° (aeq) ° distr°[(a+) ° trans°[tlr, iv° length], +0[lr, length]],
(/V) ° (aeq) ° distr°[(a-) ° trans°[tlr, iv° length], −0[lr, length]]]

Def iv ≡ atom°iv°[−0[id, I], φ];
eq°[1, φ] + 2;
iv°[−0[1, I], apndl°[1, 2]]
EIGHT QUEENS FP ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Def</td>
<td>4</td>
<td>Q8</td>
<td>1</td>
</tr>
<tr>
<td>≡</td>
<td>4</td>
<td>r</td>
<td>11</td>
</tr>
<tr>
<td>SRCH</td>
<td>4</td>
<td>φ</td>
<td>3</td>
</tr>
<tr>
<td>°</td>
<td>42</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>[ ]</td>
<td>29</td>
<td>s</td>
<td>3</td>
</tr>
<tr>
<td>,</td>
<td>34</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>gt</td>
<td>3</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>+</td>
<td>7</td>
<td>true</td>
<td>1</td>
</tr>
<tr>
<td>;</td>
<td>7</td>
<td>lr</td>
<td>3</td>
</tr>
<tr>
<td>Λ</td>
<td>1</td>
<td>o</td>
<td>1</td>
</tr>
<tr>
<td>eq</td>
<td>6</td>
<td>id</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>5</td>
<td>SRCH</td>
<td>1</td>
</tr>
<tr>
<td>tlr</td>
<td>4</td>
<td>OK</td>
<td>1</td>
</tr>
<tr>
<td>OK</td>
<td>1</td>
<td>iv</td>
<td>1</td>
</tr>
<tr>
<td>apndr</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>length</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>α</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>distr</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>trans</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iv</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>atom</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>apndl</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( )</td>
<td>9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
\[\eta_1 = 27 \quad N_1 = 196\]
\[\eta_2 = 14 \quad N_2 = 47\]
\[\eta = 41 \quad N = 243\]

\[V = \log 41 \cdot 243 = 1301.89 \quad \eta^2 = 1694920\]
EIGHT QUEENS VAL

Q8:array[int] :=
  begin
    function try (i:int; x:array[int];
        a,b,c:array[bool];
    returns array[int],bool)
    for j:int := 1 step 1;
        q:bool := false;
        xl:array[int] := x
    do if q or (j>8) then xl,q
    else iter
        xl,q := if a(j) and b(i+j) and c(j-j) then
            if i=8 then xl[8:j],true
            else begin
                x2:array[int], q2:bool :=
                    try(i+1,xl[i:j],a[j],false,
                        b[i+j],false,c[i-j],false)
                    result if q2 then x2,q2
                    else xl,q
            end
        end
    end
  end
  end
A:array[int],P:bool :=
  try(1,empty[int],array(1,8,true),
      array(2,16,true),array(-7,7,true))
  RESULT A
end
**EIGHT QUEENS VAL ANALYSIS**

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
<td>16</td>
<td>Q8</td>
<td>1</td>
</tr>
<tr>
<td>array</td>
<td>10</td>
<td>int</td>
<td>9</td>
</tr>
<tr>
<td>[ ]</td>
<td>13</td>
<td>try</td>
<td>1</td>
</tr>
<tr>
<td>:=</td>
<td>7</td>
<td>i</td>
<td>8</td>
</tr>
<tr>
<td>begin</td>
<td>2</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>function</td>
<td>1</td>
<td>a</td>
<td>3</td>
</tr>
<tr>
<td>;</td>
<td>5</td>
<td>b</td>
<td>3</td>
</tr>
<tr>
<td>,</td>
<td>24</td>
<td>C</td>
<td>3</td>
</tr>
<tr>
<td>returns</td>
<td>1</td>
<td>bool</td>
<td>5</td>
</tr>
<tr>
<td>for</td>
<td>1</td>
<td>j</td>
<td>10</td>
</tr>
<tr>
<td>step</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>OR</td>
<td>1</td>
<td>xl</td>
<td>6</td>
</tr>
<tr>
<td>( )</td>
<td>10</td>
<td>q</td>
<td>5</td>
</tr>
<tr>
<td>&gt;</td>
<td>1</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>iter</td>
<td>1</td>
<td>true</td>
<td>4</td>
</tr>
<tr>
<td>and</td>
<td>2</td>
<td>x2</td>
<td>2</td>
</tr>
<tr>
<td>+</td>
<td>3</td>
<td>q2</td>
<td>3</td>
</tr>
<tr>
<td>-</td>
<td>3</td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>=</td>
<td>1</td>
<td>P</td>
<td>1</td>
</tr>
<tr>
<td>try</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>result</td>
<td>2</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>if</td>
<td>3</td>
<td>false</td>
<td>4</td>
</tr>
<tr>
<td>empty</td>
<td>1</td>
<td>7</td>
<td>2</td>
</tr>
</tbody>
</table>

\[ n_1 = 23 \quad N_1 = 111 \]
\[ n_2 = 23 \quad N_2 = 85 \]
\[ n = 46 \quad N = 196 \]

\[ V = \log 46 . 196 = 1082.62 \quad V^2 = 1172070 \]
EIGHT QUEENS ID

Q8+(Y+procedure try (i,a,b,c,x)
  (initial x1+x; q=false
   for j := 1 to 8 while q do
     if a[j] and b[i+j] and c[i-j] then
       if i=8 then
         new x1[8]=j;
         new q+true
       else
         new x1,q+
         try(i+1,a+[j]false,b+[j+i]false,
            c+[i-j]false,x1+[i]j)
       return if q then x1,q
       else x,q);
   sa, sb, sc+(initial ta=A;
               tb=A;
               tc=A;
   for k := 1 to 15 do
     new ta[k]=true;
     new tb[k+1]=true;
     new tc[k-8]=true
   return ta, tb, tc);
   soln, cond+y(1, sa, sb, sc, A)
  return if cond then soln
  else A)}
### EIGHT QUEENS ID ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>15</td>
<td>Q8</td>
<td>1</td>
</tr>
<tr>
<td>( )</td>
<td>6</td>
<td>y</td>
<td>1</td>
</tr>
<tr>
<td>Procedure</td>
<td>1</td>
<td>try</td>
<td>1</td>
</tr>
<tr>
<td>,</td>
<td>19</td>
<td>i</td>
<td>8</td>
</tr>
<tr>
<td>initial</td>
<td>2</td>
<td>a</td>
<td>3</td>
</tr>
<tr>
<td>;</td>
<td>8</td>
<td>b</td>
<td>3</td>
</tr>
<tr>
<td>for</td>
<td>2</td>
<td>c</td>
<td>3</td>
</tr>
<tr>
<td>while</td>
<td>1</td>
<td>x</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>xl</td>
<td>5</td>
</tr>
<tr>
<td>if-then</td>
<td>1</td>
<td>q</td>
<td>7</td>
</tr>
<tr>
<td>if-then-else</td>
<td>3</td>
<td>false</td>
<td>4</td>
</tr>
<tr>
<td>new</td>
<td>6</td>
<td>j</td>
<td>9</td>
</tr>
<tr>
<td>[ ]</td>
<td>11</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>and</td>
<td>2</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>=</td>
<td>1</td>
<td>true</td>
<td>4</td>
</tr>
<tr>
<td>+</td>
<td>8</td>
<td>sa</td>
<td>2</td>
</tr>
<tr>
<td>-</td>
<td>3</td>
<td>sb</td>
<td>2</td>
</tr>
<tr>
<td>return</td>
<td>3</td>
<td>sc</td>
<td>2</td>
</tr>
<tr>
<td>Y</td>
<td>1</td>
<td>ta</td>
<td>3</td>
</tr>
<tr>
<td>try</td>
<td>1</td>
<td>tb</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tc</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>K</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>l15</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>soln</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cond</td>
<td>2</td>
</tr>
</tbody>
</table>
\[ \begin{align*}
\eta_1 &= 20 \quad N_1 = 95 \\
\eta_2 &= 26 \quad N_2 = 90 \\
\eta &= 46 \quad N = 185
\end{align*} \]

\[ v = \log 46 \cdot 185 = 1021.86 \quad v^2 = 1044200 \]
EIGHT QUEENS PASCAL

FUNCTION Q8( ) : array[1..8] of integer;
VAR i:integer;
   a:array[1..8] of boolean;
   b:array[2..16] of boolean;
   c:array[-7..7] of boolean;
   x:array[1..8] of integer;
   q:boolean;
Procedure try (i:integer; var q:boolean);
   VAR j:integer;
begin
   j := 0;
   repeat
      j := j+1;
      q := false;
      if a[j] and b[i+j] and c[i-j] then
         begin
            x[i] := j;
            a[j] := false;b[i+j] := false;c[i-j] := false;
            if i<8 then
               begin
                  try (i+1,q);
                  if not q then
                     begin
                        a[j] := true;b[i+j] := true;c[i-j] := true
                     end
               end
            else q := true
         end
until q or (j=8)
end;
begin
  for i := 1 to 8 do a[i] := true;
  for i := 2 to 16 do b[i] := true;
  for i := -7 to 7 do c[i] := true;
  try (l,q);
  for i := 1 to 8 do
    if q then Q8[l] := X[i]
    else Q8[i] := 0
end
EIGHT QUEENS PASCAL ANALYSIS

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>USAGE</th>
<th>OPERANDS</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>function</td>
<td>1</td>
<td>Q8</td>
<td>3</td>
</tr>
<tr>
<td>( )</td>
<td>5</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>array</td>
<td>5</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>;</td>
<td>24</td>
<td>integer</td>
<td>5</td>
</tr>
<tr>
<td>VAR</td>
<td>3</td>
<td>i</td>
<td>21</td>
</tr>
<tr>
<td>:</td>
<td>10</td>
<td>a</td>
<td>5</td>
</tr>
<tr>
<td>Procedure</td>
<td>1</td>
<td>boolean</td>
<td>5</td>
</tr>
<tr>
<td>begin</td>
<td>5</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>:=</td>
<td>17</td>
<td>c</td>
<td>5</td>
</tr>
<tr>
<td>repeat</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>+</td>
<td>5</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>[ ]</td>
<td>16</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>and</td>
<td>2</td>
<td>X</td>
<td>3</td>
</tr>
<tr>
<td>-</td>
<td>5</td>
<td>q</td>
<td>9</td>
</tr>
<tr>
<td>&lt;</td>
<td>1</td>
<td>j</td>
<td>15</td>
</tr>
<tr>
<td>not</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>for</td>
<td>4</td>
<td>try</td>
<td>1</td>
</tr>
<tr>
<td>try</td>
<td>2</td>
<td>true</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>false</td>
<td>7</td>
</tr>
<tr>
<td>if-then</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if-then-else</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\eta_1 = 22 \quad N_1 = 115 \\
\eta_2 = 19 \quad N_2 = 113 \\
\eta = 41 \quad N = 228 \\
v = \log 41 \cdot 228 = 1262.52 \quad v^2 = 1593960
\]
EIGHT QUEENS FWAL
Q8 ≡ SRCH:<l 1>>;
SRCH:<I J PS> ≡
  if greater:<I 8> then PS
  elseif
    And:<greater:<J 8> equal:<I 1>> then <>
    elseif
      greater:<J 8> then
        SRCH:<minus:<I 1> sum:<first:PS 1>
        rest:PS>
    elseif
      OK:<cons:<J PS>I> then
        SRCH:<SUM:<I l> 1 cons:<J PS>>
      else
        SRCH:<I SUM:<J l>PS>;
IV:N ≡ if equal:<N 0> then <>
  else
    cons:<N IV:minus:<N l>>;
OK:<LS N> ≡
  if equal:<N 1> then true
  else
    NOT: OR:<
    OR:<equal*>:<minus:<first:LS N>*>
      <minus*>:<rest:LS rest:IV:N>>
    OR:<equal*>:<SUM:<first:LS N>*>
      <SUM*>:<rest:LS rest:IV:N>>
    OR:<equal*>:<first:LS* rest:LS> >
### Eight Queens FWAL Analysis

<table>
<thead>
<tr>
<th>Operators</th>
<th>Usage</th>
<th>Operands</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>4</td>
<td>Q8</td>
<td>1</td>
</tr>
<tr>
<td>SRCH</td>
<td>4</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>:</td>
<td>47</td>
<td>SRCH</td>
<td>1</td>
</tr>
<tr>
<td>&lt; &gt;</td>
<td>41</td>
<td>I</td>
<td>7</td>
</tr>
<tr>
<td>;</td>
<td>3</td>
<td>J</td>
<td>6</td>
</tr>
<tr>
<td>if</td>
<td>3</td>
<td>PS</td>
<td>6</td>
</tr>
<tr>
<td>greater</td>
<td>3</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>elseif</td>
<td>3</td>
<td>N</td>
<td>10</td>
</tr>
<tr>
<td>and</td>
<td>1</td>
<td>IV</td>
<td>1</td>
</tr>
<tr>
<td>equal</td>
<td>6</td>
<td>O</td>
<td>1</td>
</tr>
<tr>
<td>minus</td>
<td>4</td>
<td>LS</td>
<td>7</td>
</tr>
<tr>
<td>sum</td>
<td>5</td>
<td>true</td>
<td>1</td>
</tr>
<tr>
<td>rest</td>
<td>6</td>
<td>OK</td>
<td>1</td>
</tr>
<tr>
<td>first</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OK</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cons</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IV</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>not</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>or</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \eta_1 = 20 \quad N_1 = 153 \]
\[ \eta_2 = 13 \quad N_2 = 55 \]
\[ \eta = 33 \quad N = 208 \]
\[ V = \log 33 \cdot 208 = 1049.23 \quad V^2 = 1100880 \]
XI. APPENDIX C: SEMANTIC ISSUES

Throughout this dissertation, various references have been made to the FCL semantic domain and to expressions which are semantically equivalent. This appendix will put these concepts on firmer ground.

Let EXP denote the set of FCL expressions, and SMD denote the semantic domain of FCL. The meaning of each expression is given by a function $\mu : (\text{EXP} \times \text{SMD})$. The selection of the set SMD and the function $\mu$ will define the semantics of FCL. The function $\mu$ must be many-one if the semantics are to be useful. For instance, it is reasonable to expect that $\mu(2+2) = \mu(4)$. The semantics of FCL must meet the following specifications:

1. SMD must contain unique elements which correspond to the elements of the five basic sets. For example, SMD might contain the integers and rationals which represent the members of INTEGER and REAL, respectively. (STRING may be represented as a set of functions.)

2. SMD must contain the "characteristic" functions of the five basic sets. These functions are defined on the entire semantic domain and map each element onto $\mu(\text{TRUE})$ or $\mu(\text{FALSE})$. 
3. SMD must contain a unique element x such that 
   \( \mu(\text{UNDEFINED}) = x \).

4. SMD must contain a set of functions \( G : (\text{SMD} \rightarrow \text{SMD}) \).

5. SMD must contain a set of functions such that for every built-in function \( f \), \( \mu(f) \in G \). Furthermore, 
   \( \mu(I), \mu(K), \text{ and } \mu(S) \in G \) where I, K, and S are as defined in Chapter IV.

6. SMD must be closed under application.

7. The function \( \mu \) must be homomorphic with respect to application, i.e., 
   \( \mu(f(x)) = \mu(f)(\mu(x)) \).

A complete specification of the semantics would include many additional requirements. Among these would be specifications regarding each of the built-in functions, as well as specifications which prevent degenerate solutions to the semantic equations. Since most of these specifications are intuitively obvious, they are omitted from this informal discussion. The semantics of the "\(-\)" operator require more explanation. Given an expression of the form \( E_1 \rightarrow E_2 \), both \( \mu(E_1) \) and \( \mu(E_2) \) are functions. If 
   \( \mu(E_1)(x) = \mu(\text{TRUE}) \) then 
   \( \mu(E_1 \rightarrow E_2)(x) = \mu(E_2)(x) \) otherwise 
   \( \mu(E_1 \rightarrow E_2)(x) = \mu(\text{UNDEFINED}) \). The semantic interpretation of an FCL set is as a "characteristic function" for the set. The semantics of the "\(-\)" operator derive from this fact.
The constructs of Chapter V require specification of additional semantic functions. The semantic function $\mu$ allows one to determine whether two expressions are semantically equivalent but it does not give a systematic method for converting an expression to another semantically equivalent expression. One method of providing such a method is to define one or more functions $g: (\text{EXP} \rightarrow \text{EXP})$ such that $\mu(g(x)) = \mu(x)$. This approach is fairly typical (viz. the reductions of the lambda calculus [Curry 1958]) but is insufficient for the purposes of Chapter V. Chapter V demands the following more complicated approach: Let $\pi: (\text{EXP} \rightarrow \text{SMD})$ be a 1-1 function called a representation function. (The mapping from FCL expressions to strings is a trivial example of such a function.) Let $C: (\pi(\text{EXP}) \rightarrow \pi(\text{EXP}))$ be a function such that $\mu(\pi^{-1}(x)) = \mu(\pi^{-1}(C(x)))$. If $\pi$ and $C$ are defined properly then $C$ will be definable in FCL. As pointed out in Chapter V, this allows a formal definition of program execution time to be made.

This appendix is not intended to provide a complete derivation of FCL semantics. Such a derivation would make a substantial research project in itself. This appendix should, however, give more substance to the term "semantically equivalent" as used throughout this dissertation.