DATA STRUCTURE COMPLEXITY METRICS

by

Ronald Sanger Curtis

October, 1994

A dissertation submitted to the

Faculty of the Graduate School of

State University of New York at Buffalo

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

Copyright by

Ronald Sanger Curtis

1994

Acknowledgements

Dr. Anthony Ralston has my gratitude for his guidance and patience during the extended work on this thesis. Also, the comments and encouragement of Dr. Patricia Eberlein and Dr. Patricia Van Verth, members of my committee, were appreciated. I also wish to thank Dr. Ronald Prather for his comments on the thesis.

I thank my family for everything from providing early encouragement for learning to putting up with reports over the years that the thesis was still "in the works".

Abstract

This thesis investigates the complexity of data structures in procedural programming languages. The complexity measure of interest in this thesis is the human perception of the complexity of a program. A set of axioms for data structure complexity are developed and used as the starting point for the definition of the Structural metric. An experiment was performed to help assign values to constants in the metric. Also a survey was performed to evaluate the metric. The results show that the Structural metric's complexity values are consistently higher than the values assigned by the survey participants and that the metric values have high correlation with the survey values.

Four data structure metrics based on the data structure graph are also defined. The Size metric counts the number of nodes in the graph representing user defined declarations. The Connection metric assigned weights to different types of arcs within the graph. The Layout metric also assigns weights to different types of arcs but also considers the distance between the heads and tails of arcs. The Volume metric computes the height of the graph and the width at the leaf level.

Examples from textbooks are used to compare the graph based metrics and the Structural metric. The comparisons show that the Structural metric gives consistently higher values than the graph-based metrics but there is a strong correlation between the two classes of metrics. Also, the examples show how the Layout and Connection metrics may be used to improve the ordering of the declarations.

All five metrics (the Structural metric and the four graph-based metrics) are compared

with a set of properties for software metrics that has been proposed in the literature. One property is not satisfied by any of the data structure metrics. One graph-based metric, Size, fails several other properties since the metric does not use context.

We also address the problem of using multiple metrics such as the five data structure metrics presented here. A visual approach is presented with examples on how it might be used.

Chapter 1

Introduction

This thesis deals with data structure program complexity. Specifically we investigate the complexity of data structures in imperative languages such as C or Modula-2. Axioms are presented that were used to develop a structural data structure metric. Several graph based metrics that capture different aspects of data structures are developed. In this section we put this in the context of program complexity research generally.

One way to classify complexity measures is based on whether they look at the development process or at a product of the process. One could record information about the design process, possibly by keeping complete histories of a project [Baker,86], and use this information either to assign a measure to the result or predict the complexity of future projects. Product complexity looks at a product of the development process and assigns it a complexity. For

example, a user's document is a product of the development process and one could define a complexity measure for a finished document. A program complexity measure is a function that takes a source program as input and computes one or more values for it that may be compared with similar values for other programs. Such a measure may be used to compare old and new versions of a program.

The complexity measure of interest in this thesis is the human perception of the complexity of a program. This measure attempts to capture the readability and style of the program. The efficiency, either theoretical or actual, of program execution is secondary to how programmers view the program. The goal is to find mechanisms that assign complexity values to programs that correspond to programmer perceptions of the difficulty of understanding a program. These complexity values can then be used, for example, to compare two or more programs solving the same problem or detecting parts of a program that have high complexity and may need to be redesigned.

There are several types of complexity studied in computer science that are not related to this research. First, we are not concerned with the complexity of a problem. Our research deals only with concrete programs developed to solve problems and not with a theoretical best solution. Second the order (time or space) of an algorithm is not the complexity issue here. We are mainly comparing different implementations of one algorithm. Third, we are not running a program and measuring the run time complexity or required data space. The run time speed or space requirements need not correlate with the readability of the source program. For example, most people would agree that the recursive algorithm for the Towers of Hanoi problem is more readable than the iterative version, although the latter

may be faster. Also, run time measurements would make the complexity dependent on the compiler, linker, libraries, and machine used. Although the complexity investigated here may reflect some aspects of the above, it is really attempting to measure a different type of complexity, namely the complexity as perceived by programmers reading the source code.

For this research, textual "sugar" that may or may not add to a person's perception of difficulty are ignored. Examples of these include comments, identifier names, and program indenting. The first two are eliminated because they require knowledge of the problem domain and are well beyond the scope of this research. Indenting is not considered because program formatters can be used to reformat program source code into a standard format.

Chapter 2

Overview of Complexity Metrics

This chapter presents the basic program measures that have been studied. Existing program complexity measures can be classified as count- (program size), control-, or data- based. Several basic measures can be combined into a composite measure, e.g. [Oviedo,83]. Data-based measures will be covered in the next chapter. For a more complete introduction see [Conte,86].

A simple and often used measure, which is very easy to implement, is Lines of Code. Longer programs are, in general, perceived as more complex than shorter programs. Since most of the information in a program is ignored when only counting the number of lines of source code, one would think that this measure would be poor. Surprisingly, experiments have not shown other metrics to be clearly superior. Nevertheless, the expectation is that, as

metrics using information in the code that is ignored by Lines of Code are refined, they should better approximate the complexity of programs.

One problem with Lines of Code as a measure is the abundance of definitions for "line of code". Are blank lines to be included? Are comment lines to be included? The lack of agreement results in different values.

A second problem is that Lines of Code does not take into account that different lines may require different levels of effort to read. For example, both code segment FindA and FindSmallestValue below have 13 lines. Yet, most readers would agree that FindSmallestValue requires more effort to understand.

The first segment's executable lines are exclusively sequential assignment statements. Each statement can be understood in sequence. Termination of the segment is never in question.

The second segment includes a loop and a selection statement. A loop requires that the reader think about the initial entry into the loop, subsequent passes through the loop, and loop termination. To do this, the reader must maintain knowledge of the loop and its context. A loop also requires that the reader consider the possibility the segment will not terminate.

In both segments, HIGH(G) returns the index to the last member of the array G. Indexing starts at 0 no matter how the indexing on the original array was defined. The definition of G in the parameter list (G:ARRAY OF INTEGER) hides the indexing specified for any actual array parameter. This is useful as it allows arrays with different indexing types but

the same base type, INTEGER in this case, to be passed into the procedure.

```
PROCEDURE FindA(G:ARRAY OF INTEGER):INTEGER;
  MidIndex, HighIndex, LowIndex: INTEGER;
  IntAvg, X, Y, Z: INTEGER;
  BEGIN
    HighIndex := HIGH(G);
    LowIndex := 0;
    MidIndex := (LowIndex + HighIndex)/2;
    X := G[LowIndex];
    Y := G[MidIndex];
    Z := G[HighIndex];
    IntAvg := ( X + Y + Z )/3;
    RETURN IntAvg;
  END FindA;

PROCEDURE FindSmallestValue(G:ARRAY OF INTEGER):INTEGER;
  Index, HighIndex: INTEGER;
  LowValue: INTEGER;
  BEGIN
    HighIndex := HIGH(G);
    LowValue := G[HighIndex];
    Index := 0;
```

```
    WHILE( Index < HighIndex )
      IF( G[Index] < LowValue ) LowValue := G[Index]; END;
      INC(Index);
    END;
    RETURN LowValue;
  END FindSmallest;
```

Similarly, no consideration is given to the differences even within a statement type. For example, as long as each is on a single line, both assignment statements below have Lines of Code equal to 1. Clearly, the second is more difficult to understand.

```
I := 0
Root := (-B + SQRT(B*B-4.0*A*C))/(2.0*A)
```

A measure similar to Lines of Code is program statement count. For some languages, Fortran IV for example, the two are almost identical. This measure tends to be less ambiguous as the language definition often explicitly states what is to be considered as a program statement. In free format Algol-based languages there is greater potential for differences between the two measures. For example, someone who prefers a purely vertical layout or who is paid by lines of code generated might format FindA above as:

```
PROCEDURE
  FindA(
    G:ARRAY OF INTEGER
```

```
     ):INTEGER;
MidIndex,
HighIndex,
LowIndex: INTEGER;
IntAvg,
X,
Y,
Z: INTEGER;
BEGIN
  HighIndex
    := HIGH(G);
  LowIndex
    := 0;
  MidIndex
    := (LowIndex + HighIndex)/2;
  X
    := G[LowIndex];
  Y
    := G[MidIndex];
  Z
    := G[HighIndex];
  IntAvg
    := ( X + Y + Z )/3;
```

```
    RETURN
      IntAvg;
  END FindA;
```

This would increase the Lines of Code from 13 to 29 while the statement count would remain at 13. It should be noted that program statement count also does not take into consideration individual statement complexity. In industry, programming standards and automatic formatters often control the layout, making the difference between Lines of Code and program statement count small.

Another count-based approach called Software Science was proposed by Halstead [Halstead,77]. Each syntactic token is considered to be an operator or operand. Four counts are generated for a program:

- η_1 = number of unique operators
- η_2 = number of unique operands
- N_1 = total operators
- N_2 = total operands

The *size* of a program is defined as $N = N_1 + N_2$.

A second size measure, *volume*, is defined. First, *Vocabulary* is defined as $\eta = \eta_1 + \eta_2$ and then volume is $V = N \times log_2\eta$. The Vocabulary gives the number of tokens required in a minimal language that would be capable of expressing the program. By applying the log

to the Vocabulary, the number of bits that would be required to express the operators and operands is determined. The resulting volume is a measure of the number of bits required to represent the program.

In the tables below are the operator and operand counts for the two assignment statements given above.

```
I := 0
```

Item	Count	Operator	Operand
I	1		X
:=	1	X	
0	1		X

The number of unique operators, η_1, is 1 for the assignment operator. There are two unique operands, I and 0, so η_2 is 2 so the Vocabulary, η, is 3. In this case, all operators and operands are unique so $N_1 = \eta_1 = 1$, $N_2 = \eta_2 = 2$, and $N = \eta = 3$. The volume is $V = N log_2 \eta = 3 log_2 3 \approx 4.7$.

```
Root = (-B + SQRT(B*B-4.0*A*C))/(2.0*A)
```

Item	Count	Operator	Operand
Root	1		X
:=	1	X	
()	3	X	
-(unary)	1	X	
B	3		X
+	1	X	
SQRT	1		X
	4	X	
-(binary)	1	X	
4.0	1		X
A	2		X
C	1		X
/	1	X	
2.0	1	X	

The number of unique operators η_1, is 8 while the number of unique operands η_2, is 6 giving a Vocabulary, η, of 14. The total number of operators, counting duplicates, is $N_1 = 13$ while the total number of operands occurrences is $N_2 = 9$, so $N = 22$. The volume is $V = Nlog_2\eta = 22log_214 \approx 83.6$.

There is evidence that Lines of Code, Size, and Volume are linearly related and each can be used as a measure of program size[Christensen,1981]. The Size of programs for many languages have been evaluated and an empirical formula $N \div c$ was determined that ap-

proximates the lines of code measure from the size measure, with c depending upon the language.

Software Science also defines Effort, and Programming Time. Effort is defined as $E = \frac{\eta_1 N_2 N log_2 \eta}{2 \times \eta_2}$. This is considered to be a measure of elementary mental discriminations. The definition of the Programming Time has raised many questions. It is defined as $T = \frac{E}{\beta}$ where β, the Stroud number, is suppose to be the number of elementary discriminations per second that a person can make. The controversy is about β. First, the Stroud number is used in Software Science although the domain is different than the original area (i.e. psychology) and no research exists to show the original work should carry over to software. Second, the Stroud number has not been widely accepted in psychology.

Another group of measures deals with control flow. The cyclomatic complexity measure [McCabe,76] is based on the control graph of a program. For a control graph of **e** edges, **n** nodes, and **p** connected components, the complexity is **e** - **n** + **p**. For structured programs, this reduces to the number of decisions in the program, that is the number of selection and loop statements plus 1. For example, FindA would have complexity 1 while FindSmallest-Value would have complexity 3.

A model developed at SUNY Buffalo [Oviedo,83][Oveido,80][VanVerth,85] includes a measure similar to McCabe's that is based on control flow. The complexity of a module is the number of edges in the flowgraph of the module.

The testing measure [Prather,84] is a control measure defined recursively on the statements of the language. The measure of a simple statement is defined as 1. The measure of

a sequence of statements is the sum of the measures of the individual statements. The measure of IF P THEN S1 ELSE S2 is two times the maximum of the measures of S1 and S2. The measure of WHILE P DO S is two times the measure of S. A set of axioms was defined and the measure was derived from them. This measure has been extended [Fenton, 1986] and experiments have been performed with this measure [Moore, 1989]. It has not been as extensively evaluated as the previous measures but is still an active research area. The axiomatic approach utilized later in this thesis applies a similar approach to data structure metrics.

All of the above measures give a nonnegative real number for the program complexity but this is not essential. In [Cantone,83], the characteristic polynomial with variable c is calculated as follows. The control graph is created. The complexity of a primitive node is the polynomial 1. The complexity of a sequence of statements is the product of the complexity of the structures and primitive nodes in the sequence. The complexity of a selective statement structure is given by the sum of the complexity of the structures and primitive nodes in the alternatives. An extra 1 is added for "if-then" (i.e. the else is forced to add 1 even when not present). Finally, for a cyclic structure, two cases are considered. For a simple counting loop, the complexity is the product of c, the polynomial variable, and the complexity of the structure in the loop. For a while loop, a 1 is added to the complexity of the loop before taking the product.

In Figure 2.1, the primitive nodes 2, 3, and 5 are each assigned complexity 1. Node 1 is a selection node so its complexity is the sum of nodes 2 and 3, or 2. The complexity of the inner loop consisting of nodes 1, 2, 3, and 4 is 2c. Since node 5 has complexity 1 and is in

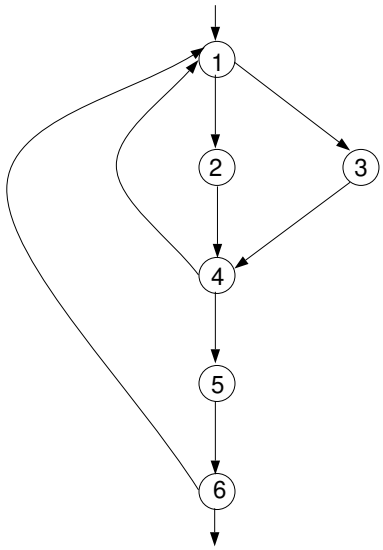


Figure 2.1: Graph for Characteristic Polynomial Example.

sequence with the inner loop, the complexity of everything inside the outside loop is also 2c. The outside loop contributes another c factor, giving a final complexity of $2c^2$.

Count-based and control-based metrics dominate research on program source complexity metric research. In the next chapter we will consider data-based metrics. They are of interest for two reasons. First, they capture an aspect of program complexity not addressed by control-based metrics. Second, in some design methods data structures exist before executable code so data-based metrics allow earlier estimation of complexity.

Chapter 3

Data Structure Metrics

This thesis deals with data structure program complexity as perceived by programmers. The purpose of this chapter is to explain why data complexity is of interest and to summarize existing data measures.

For several reasons, data-based measures have received less attention than count and control flow based measures. First, data-based measures cannot capture the full complexity of the program since there are programs which require little data yet are complex.

A second possible reason deals with the source of programs typically used in software measurement studies. Usually, the programs either come from student exercises or technical projects in industries such as aerospace. Programs in these areas tend to have simple data

structures even if their control structures are complex.

Data-based measures can be divided into data usage and data declaration measures. Data usage metrics investigate data accesses within the control flow of a program. Data declaration measures look at only the declarations of a program independent of any control flow information.

3.1 Data Usage Metrics

The *relative percentage usage pairs measure* [Basili,75] looks at the use of globals in different segments of the program. A usage pair (p,r) is an instance of a global variable r being used in segment p. Actual Usage Pairs is defined as the count of realized pairs. Potential Usage Pairs is the count of all (p,r) whenever p is in the scope of r, even if r is not used in p. The metric is Actual Usage Pairs/Potential Usage Pairs, which gives a measure of what portion of the global variables are actually referenced. Code accessing global variables is a prime source of bugs in programs. Since, potentially, the variable may be changed by any code in its scope, a programmer cannot look at one isolated piece of code to understand how the value of a global variable changes. The metric assigns low complexity to code that does not reference a large percentage of the global variables to which it has access. A complexity of 0 is assigned if no globals are referenced and the maximum of 1 is assigned if all globals are referenced.

The *data binding metric* [Basili,75][Stevens,74] considers the interactions between modules

through global variables. It considers all tuples (p,r,q) where segment p modifies global variable r and segment q accesses variable r. Actual and potential counts are again used and the ratio is computed. This metric provides information on the data dependency through globals between modules.

The *span metric* [Elshoff,76] is based on the amount of information about data a reader of program must remember. The span of an identifier is the range of source statements between two consecutive references to the identifier. The complexity is the maximum number of spans at any point in the program. Declarations are not counted since, to do so, would, for each variable, add a span from the declaration to its first usage. This would reduce the span metric to equal the number of variables since all variables would span the initial statement. In essence, the span metric measures the maximum number of variables a reader of the program must know about at any point in the program.

Data flow techniques similar to those used in compilers have also been used. The def-ref (definition and reference) relations are calculated [Aho,72]. Such techniques are used in the SUNY at Buffalo metric[Oviedo,80] [Oviedo,83].

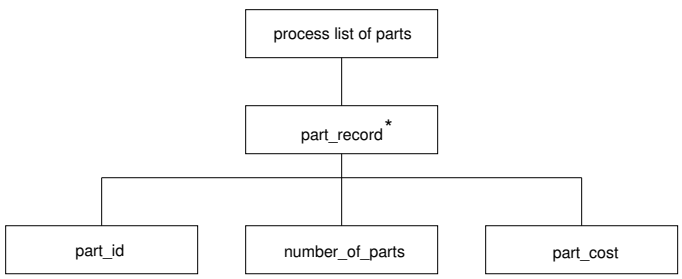
3.2 Data Declaration Metrics

Data declaration metrics consider only the declarations in a program. Studying the data declarations in isolation has several advantages.

First, in certain design methods the data structure aspects are known earlier than the control aspects. In fact, in some methods[Jackson,75] the basic control structure is derived from the data structures. As an example consider the problem of computing the parts cost of an item. Suppose the input is a file containing a list of parts used to build the item and each part is represented as the record below.

```
type part_record = record
                     part_id: integer;
                     number_of_parts: integer;
                     part_cost: real;
                   end
```

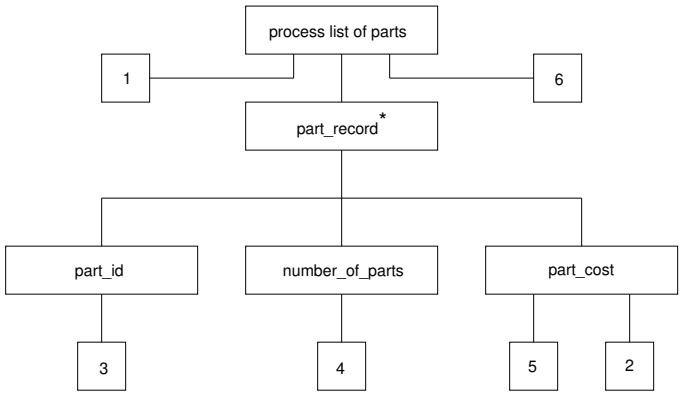
The structure diagramed is shown below. The asterisk superscript on part_record indicates 0 or more instances are possible.



The next step is to create a list of desired operations. In this case the list might include the items below.

```
1. cost := 0

2. cost := cost + part_cost * number_of_parts

3. read part_id

4. read number_of_parts

5. read part_cost

6. print cost
```

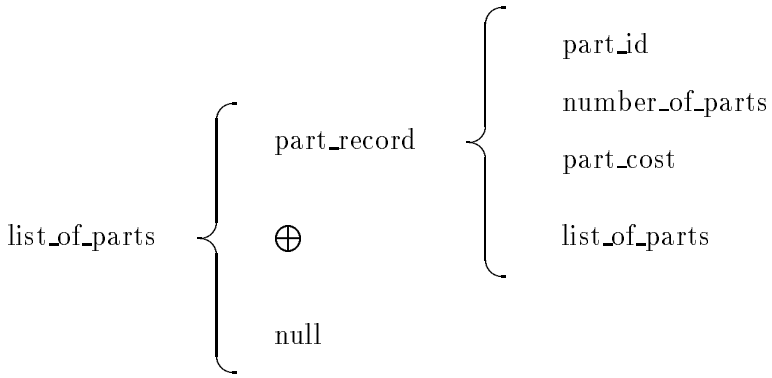
Each operation is then placed at an appropriate location on the diagram. In this simple case, the control structure can then be derived directly from the diagram.



```
procedure process_list_of_parts
  cost = 0
  while more_parts do
    process_part_record
  end
  print cost
end process_list_of_parts


procedure process_part_record
  read part_id
  read number_of_parts
  read part_cost
  cost = cost + part_cost * number_of_parts
end process_part_record
```

The Jackson method lends itself to record-oriented processing as is common in business data processing. It has also been used in designing real-time systems.

Similarly, design with Warnier-Orr diagrams [Warnier,1974] starts with the specification of data. Hierarchy structure is represented by an item followed by a bracket containing the subcomponents. Besides sequencing, alternation, and repetition supported by the Jackson method, Warnier-Orr diagrams also support recursion. A part_list data structure similar to the one discussed above but using recursion is shown below as a Warnier-Orr diagram. Note that the definition of list_of_parts contains a reference to a field of the same type.



A second reason for studying data declaration measures is that a metric based on program data structures might be applied to other stages of software development. For example, structure diagrams developed at the design stage can be treated as a data structure with repeated parts corresponding to arrays and subordinate parts as record fields. The structure diagram given earlier would correspond to the data declarations given below. The leaf nodes of the structure diagram are considered primitives.

```
TYPE ProcessListOfParts = ARRAY[0..N] OF PartRecordType;

TYPE Part_RecordType = RECORD
                         PartId: PRIMITIVE;
                         NumberOfParts: PRIMITIVE;
                         PartCost: PRIMITIVE;
                       END;
```

Third, a data structure metric captures an aspect of complexity that is orthogonal to most control flow metrics. Its use can be combined with them to provide a more complete view

of complexity.

A data structure measure could be developed giving the actual storage used by data in a program but this is probably not a good measure since not only will the programmer not remember all storage locations but also the difference in actual storage used by a 10 and a 1000 element array is not important to the reader of a program. Also, such a measure does not take into account the interrelationships among data structures.

Another approach which should be mentioned is to develop a graph for data types and variables. One measure [Tsai,86] creates the graph of data types and then partitions it into connected components. Loops in the typing structure and other aspects are counted and used in a polynomial in which the powers capture the loops (recursion) in the data declarations. With the measure suggested, a program using a single recursive structure, even a simple linked list, is more complex than a program using other structures, including arrays and records nested to arbitrary levels. The penalty for recursive structures seems too great.

The main data structure metric presented in this thesis was designed using the axioms presented in the next chapter. While recursive structures have a higher cost than simple structures, nevertheless, nested arrays and records may have greater complexity than a recursive structure.

Chapter 4

Data Structure Metric Axioms

This chapter presents a set of axioms for data structure metrics. Then, in the next chapter a metric will be presented based on these axioms. The axioms are based on a small set of general principles.

4.1 Axiom Description Information

In the descriptions below, certain standard notations are used. The table below explains the notation used in this and following chapters.

Symbol	Read As	Example
$\exists$	there exists	$\exists C \geq 0$
$\forall$	for all	$\forall\ integers\ I$
$\in$	element of	$x \in \{0, 5, 7, 9, 17, 21\}$
$\mid X \mid$	the complexity of X	$\mid CONST\ a\ =\ b \mid\ =\ 2$
$\mid X \mid_{\mu}$	the complexity of X by measure μ	$\mid CONST\ a\ =\ b \mid_{LOC}\ =\ 1$

The first three are standard notation in mathematics and logic. The notation used for complexity can be thought of as a generalization to the notation used to represent the absolute value of numeric values or the notation for magnitude of a vector. Complexity of a program or part of a program is a measure of its size or magnitude. The first complexity notation, without a subscript, assumes a complexity measure is understood by the context while the second allows explicit use of the name of the measure.

Modula-2 is used as the language for presenting the axioms and examples. However, no axiom is limited to Modula-2. The axioms are general and are the same for all imperative languages[1]. The set of simple types ($\{simpletypes\}$) used in the Simple Types section will be different for each language, but the axiom will still hold. For example, Modula-2 will include **INTEGER**, **REAL**, **CHAR**, and **BOOLEAN** in its set of simple types while C will include **int** and **float** but will not include any distinct simple types that correspond to **CHAR** and **BOOLEAN**.

[1] This restriction, to imperative languages, such as Modula-2 and C, simplifies identification of the data structures within a program. The standard ways imperative languages are used allows the syntax to help in identifying the data structures used. In other language models, it is often difficult to detect the data declarations used without simulating the execution of the program. For example, a Lisp program may create structures determined by input and not by source code. A secondary reason to use imperative languages is that most software metrics work has used such languages.

4.2 General Principles

The following general principles were used in developing the axioms.

GP 1 *A data structure type with subcomponents is at least as complex as the sum of the individual complexities of the subcomponents.*

This is similar to "a whole is greater or equal to the sum of its parts". For example, one would expect a record structure such as SampRec below to be at least as complex as the sum of the complexity of the A and B fields.

```
TYPE SampRec = RECORD
                 A: INTEGER;
                 B: INTEGER;
               END;
```

GP 2 *Each use of a type name adds little to the complexity.*

That is, a type name structures information so that a reader of a Program need not worry about the details. For example, the type name can be used to determine quickly if two structures are equivalent. Declaring X and Y individual as

```
VAR X = RECORD
          A: INTEGER;
```

```
          B: INTEGER;
        END;
```

⋮

```
VAR Y = RECORD
          A: INTEGER;
          B: INTEGER;
        END;
```

has a certain complexity. But defining SampRec as above and declaring X and Y using SampRec as

```
VAR X, Y: SampRec;
```

simplifies reading and should have a similar complexity. It is immediately known that X and Y have the same type and contain the same fields. One need not keep in mind the details of X and Y. They are automatically taken care of because the same type name is used.

GP 3 *Using one declaration or multiple adjacent declarations for items of the same type does not change the complexity.*

For example,

```
| VAR X: SampRec;  |     =     | VAR X, Y: SampRec; |
| VAR Y: SampRec;  |           |                    |
```

In such cases, it would be trivial to implement a program that would transform one declaration into multiple or multiple adjacent declarations into one. This principle is similar to our ignoring program layout issues as mentioned in Chapter 1.

GP 4 *When declarations are independent, their order doesn't matter.*

Two declarations are independent if neither declaration uses items defined by the other. For example, if the language supports it, independent TYPE and VAR declarations can be ordered in any way meaningful to the programmer. Thus,

```
| TYPE Line: ARRAY[1..80] |     =     | VAR X: INTEGER;         |
|              OF CHAR;   |           | TYPE Line: ARRAY[1..80] |
| VAR X: INTEGER;         |           |              OF CHAR;   |
```

However, if the two declarations have a dependency relationship, the complexity may depend on the ordering. For example, it is desirable that types be defined before they are used. This principle does not prevent a metric from assigning different complexities to different orderings of dependent declarations. The complexity ordering in the example below is allowed by this principle.

TYPE Line: ARRAY[1..80] OF CHAR; VAR X: Line;	<	VAR X: Line; TYPE Line: ARRAY[1..80] OF CHAR;

4.3 Simple Items

4.3.1 Constants

All constant declarations have the same measure. In many languages (Pascal, Modula 2), constants lack structure. C allows creation of structured constants but only following a structure declaration. Since the constant's structure is identical to that of the declaration, the constant's complexity due to structure is already handled in the complexity of the declaration.

For example, there is no difference in complexity between defining an integer constant and a real constant.

Axiom 1 $\exists C \geq 0$ *such that* $\forall$ a *and* b, $|\, CONST\ a\ =\ b\,| = C$

4.3.2 Simple Types

All simple types have the same non-zero complexity. Since simple types are atomic, the programmer reading a program will require the same amount of effort to understand any of them. Simple types correspond to simple statements in control flow metrics. For example, CHAR and INTEGER should have the same complexity.

Axiom 2 $\exists S > 0$ *such that* $\forall T \in \{simple\ types\}$, $\mid T \mid = S$

4.3.3 Range Types

A range specifies a range of values from an existing type. The least element in the range and the largest must be specified, either explicitly, as in Modula-2, or implicitly, as in C. Since a range requires specifying two bounds that the reader must remember, the complexity of a range declaration should be greater than 0. Furthermore, there is some minimal complexity of range declarations (R in the axiom below). Depending upon the language, the bounds may be literals, declared constants, expressions, or default values. Different combinations of defaults, literals, and constants in a range declaration may result in different complexity measures.

For example, in Modula-2 [1..8] should have the lowest range complexity while [A..B] should have the highest. The first case gives the minimal range so R = | [1..8] |

Axiom 3 $\exists R > 0$ *such that* $\forall$ *a and b,* $\mid a..b \mid \geq R$ *and* $\exists c, d$ *such that* $R = \mid c..d \mid$

4.4 Sequences of Declarations

4.4.1 Constant Declarations

Using a single CONST statement should be no more complex than using multiple CONST statements. This is an application of GP3.

Axiom 4 $| CONST\ a_1 = b_1;\ ...;\ a_n = b_n | \leq | CONST\ a_1 = b_1;\ ...\ ;\ CONST\ a_n = b_n |$

4.4.2 Type Declarations

Using a single TYPE statement should be no more complex than using multiple TYPE statements. This is an application of GP3.

Axiom 5 $| TYPE\ a_1 = T_1;\ ...\ ;\ a_n = T_n | \leq | TYPE\ a_1 = T_1;\ ...\ ;\ TYPE\ a_n = T_n |$

The complexity of a sequence of type declarations is the same as the sum of the complexity of the individual types.

Axiom 6 $| a_1 : T_1;\ ...\ ;\ a_n : T_n | = \sum_{i=1}^{n} | a_i : T_i |$

Although simple type redeclaration does not reduce complexity, declaring a new non-simple type reduces complexity when it is used enough times. Simple type redeclaration need not

reduce complexity. If AT is not a simple type, then declaring a type should make a program less complex when the type is used enough times.

For example, declaring a type Complex as shown below is useful if it is used "enough" times.

```
TYPE Complex = RECORD
                 RealPart, ImPart: REAL;
               END;
```

If only one variable will be of this type, the addition of the extra type name may result in a higher complexity than the same program with the variable defined as the record structure.

```
VAR X = RECORD
          RealPart, ImPart: REAL;
        END;
```

The point at which using a type name becomes worthwhile is left unspecificed, only that such a point should exist. Without this axiom, it would be possible to have a metric which always penalizes the use of explicit type declarations.

Axiom 7 $\exists\ n\ such\ that\ |\ VAR\ a_1\ :\ AT;\ ...\ ;\ VAR\ a_n\ :\ AT\ |\ >\ |\ \ TYPE\ \ T\ =\ AT;\ VAR\ a_1 : T;\ ...\ ;\ a_n : T\ |$

4.4.3 Variable Declarations

A single VAR section is no more complex than multiple VAR sections. That is, if one has a group of adjacent VAR declarations, they can be combined into one VAR without causing an increase in complexity.

For example, the single VAR on the left below should have a complexity no greater than the three VARs on the right hand side. Since the three separate declarations require more text to create exactly the same data structures, a metric should not assign the single VAR a higher complexity.

VAR X: SampRec;	$\leq$	VAR X: SampRec;
Y: SampRec		VAR Y: SampRec;
Z: SampRec		VAR Z: SampRec;

Axiom 8 $| VAR\ a_1 : T_1;\ ...\ ;\ VAR\ a_n : T_n | \quad \geq \quad | VAR\ a_1 : T_1;\ ...\ ;\ a_n : T_n |$

A sequence of separate VAR declarations for items of the same type is at least as complicated as a single declaration with the type appearing once. The left hand side includes everything on the right hand side plus duplicate occurrences of T. Since the two declarations are otherwise identical, the extra symbols could make the left side more complex but never less complex than the right side.

VAR X: SampRec; Y: SampRec Z: SampRec	$\geq$	VAR X, Y, Z: SampRec;

Axiom 9 $|\,VAR\ a_1 : T;\ ...;\ a_n : T\,| \quad \geq \quad |\,VAR\ a_1,\ ...\ ,\ a_n : T\,|$

The order of variable declarations does not change the complexity. Within a scope, variable declarations are independent of each other. That is, a variable declaration cannot depend on another variable within its scope.

For example, the complexity of declaring two variables of type SampRec is independent of the order of the declarations.

VAR X: SampRec; Y: SampRec	$=$	VAR Y: SampRec; X: SampRec;

Axiom 10 $|\,a_1 : T_1;\ ...\ ;\ a_n : T_n\,| \quad = \quad |\,Perm(a_1 : T_1;\ ,\ ...\ ,\ a_n : T_n;)\,|$

4.5 Record Axioms

A record declaration is at least as complex as the sum of the complexities of its fields, that is, GP 1 applies. For example, given type PersonRec below, and assuming each field declaration has complexity 2, we would expect the complexity of PersonRec to have a complexity of at least $2 + 2 + 2 = 6$.

```
TYPE PersonRec = RECORD
                   Id: IdType;
                   Name: NameType;
                   Job: JobType;
                 END
```

Axiom 11 $| \, RECORD \; a_1, \; a_2, \; ..., \; a_n \; END \, | \geq \; \sum_{i=1}^{n} | \, a_i \, |$

The order of fields in a record does not change the complexity. Any additional complexity would be negligible compared with the overall data structure complexity. For example, the declaration below should have the same complexity as the PersonRec declaration above.

```
TYPE PersonRec2 = RECORD
                    Job: JobType;
                    Name: NameType;
                    Id: IdType;
                  END
```

Axiom 12 $| \, RECORD \; a_1, \; ... \, , \; a_n \; END \, | = | \, RECORD \; Perm(a_1, \; ... \, , \; a_n) \; END \, |$

A case in a record has two sources of complexity, the tag field and the fields in the case. Certainly, the case must be at least as complex as the sum of the tag field complexity and the sum of the subfield complexities.

In PersonRec3, the reader of the program must remember the case tag and all the fields for all the possible values of the case tag, in this case, Hourly, Salary, and Volunteer. The case tag is a field within the record and should add at least as much complexity as a "normal" field.

```
TYPE PersonRec3 = RECORD
                    CASE EmpCode: EmpType OF
                      Hourly: (
                              HourlyRate: INTEGER;
                              MaxHours: INTEGER;
                              )
                      Salary: (
                              YearlySalary: INTEGER;
                              )
                      Volunteer: (
                                   HoursGiven: INTEGER;
                                 )
                  END
```

Axiom 13 $| CASE\ T\ OF\ f_1, f_2, ..., f_n | \geq | T | + \sum_{i=1}^{n} | f_i |$

4.6 Sequences of Ranges

The dimensions of an array are specified by a range or a sequence of ranges. A sequence of ranges is at least as complex as the sum of the individual complexities. This is another application of GP 1.

Axiom 14 $| [a_1..b_1,\ a_2..b_2,\ \ldots\ a_n..b_n] | \geq \sum_{i=1}^{n} | [a_i..b_i] |$

4.7 Arrays

An ARRAY declaration is more complex than simply adding the dimension complexity and the base complexity. The declaration of LineType below is as complex as the sum of the complexity of [1..80] and the complexity of CHAR.

```
TYPE LineType = ARRAY[1..80] OF CHAR;
```

Thus, if BaseType is not a circular reference to the ARRAY declaration,

Axiom 15 $| \mathit{ARRAY\ IndexType\ OF\ BaseType} | > | \mathit{IndexType} | + | \mathit{BaseType} |$

If BaseType is a circular reference to the ARRAY declaration, then the array must be more complex than its Index type. Note that the BaseType is not included. This is because it

may be a pointer type back to the ARRAY type which would result in a circular condition which could never be satisfied.

Axiom 16 | *ARRAY IndexType OF BaseType* | > | *IndexType* | *when BaseType is a circular reference to the ARRAY declaration.*

4.8 Sets

The set is more complex then its base type. Programmers (at least beginning ones) find sets more difficult than simple types. Since it is a composite type, a higher complexity would be expected. For example, a set of range of INTEGER should be more complex than INTEGER.

Axiom 17 | $SET\ OF\ X$ | > | X |

4.9 Pointers

A pointer is more complex than its base type. Most programmers would agree pointers are difficult. Again, a pointer is a composite type so an increase in complexity is to be expected.

Axiom 18 | $POINTER\ TO\ X$ | > | X |

4.10 Units

Many languages have a concept of Unit. We will consider procedures as well as modules, packages, objects, and tasks as units. The declaration complexity of such languages is the sum of the complexity of the individual units.

Axiom 19 $| U_1, U_2, ..., U_n | = \sum_{i=1}^{n} | U_i |$

Allowing constants and types within a unit to be visible to other units is allowed with no increase in complexity to encourage the use of units. Since constants and types are "read-only" in other units, there is never a problem with another unit changing their values.

Axiom 20 $\forall N \in \{constant\ or\ type\ name\}, | EXPORT\ N | = 0$

There should be a penalty for making a variable visible to other units. This is a major source of programming error. Since variables can be changed by other units, a reader must consider all other units that may change the variable which could be all units in the program. Making a variable available to other units this way is similar to, and in many cases identical to, using global variables.

Axiom 21 $\forall N \in \{variable\ name\}, | EXPORT\ N;\ VAR\ N : AT | > | VAR\ N : AT |$

Chapter 5

Structural Data Structure Metric

In this chapter, a data structure metric is defined based on the axioms in the previous chapter. This approach is similar to that used by Prather in the control structure area [Prather,] First, we discuss the problem introduced by redundant declarations in a language such as Modula-2 in which separate specification and implementation modules may contain some of the same declarations. Then, for each declaration construct in Modula-2 a formula is given to compute its complexity.

5.1 Collapsing Specification and Implementation Files

In Modula-2 and other languages that support information hiding in separately compiled units, data declarations appear in the specification module (the DEFINITION module in Modula-2) and in the implementation module (the IMPLEMENTATION module in Modula-2). If only specification modules exist, say at the design stage, there is no problem since only the specification declarations can be used. Later in development, both specification and implementation modules exist. At first, one might be tempted simply to concatenate the two. This would result in an inflated complexity estimate as the modules contain redundant declarations of two forms: type definitions and procedure headers. For example, the Modula-2 definition and implementation modules below contain duplicate procedure headers.

```
DEFINITION MODULE VectorLib;

  TYPE VectorType;

  PROCEDURE CreateVector( Data: ARRAY OF REAL; VAR V: VectorType );
  PROCEDURE DestroyVector( VAR V: VectorType );
  PROCEDURE VectorSum( V: VectorType): REAL;
  PROCEDURE ScalerAddition( Scaler: REAL; V: VectorType;
                            VAR VResult: VectorType );
  PROCEDURE VectorAddition( V1, V2: VectorType; VAR VResult: VectorType );
```

```
  PROCEDURE ScalarProduct( Scaler: REAL; V: VectorType;
                           VAR VResult: VectorType );
  PROCEDURE VectorProduct( V1, V2: VectorType; VAR VResult: VectorType );

END MODULE VectorLib;



IMPLEMENTATION MODULE VectorLib;

  TYPE VStore = RECORD
              VectorLen: INTEGER;
              VectorData: /* Defined as appropriate */
            END;
  TYPE VectorType = POINTER TO VStore;

  PROCEDURE CreateVector( Data: ARRAY OF REAL; VAR V: VectorType );
  PROCEDURE DestroyVector( VAR V: VectorType );
  PROCEDURE VectorProduct( V1, V2: VectorType; VAR VResult: VectorType );
  PROCEDURE ScalarProduct( Scaler: REAL; V: VectorType;
                           VAR VResult: VectorType );
  PROCEDURE VectorAddition( V1, V2: VectorType; VAR VResult: VectorType );
  PROCEDURE ScalerAddition( Scaler: REAL; V: VectorType;
                           VAR VResult: VectorType );
```

```
  PROCEDURE VectorSum( V: VectorType): REAL;

END MODULE VectorLib;
```

A type may be named in a specification module but not defined there with the type definition appearing in the implementation module. Types declared this way are called opaque or hidden types. **VectorType** in the sample Modula-2 declarations above is an explain of such a type. Since the type definition in the specification only gives its name, it should be thrown away when combining specification and implementation modules.

Procedure headers that appear in the specification module are repeated in the implementation module. Since the same information appears in both places, the only difference may be the order. When implementation modules exist, we will throw away the procedure headers in the specification module. This will also allow the procedure order in the implementation module to come into play.

5.2 Metric Definition

The metric is parameterized by the following set of constants:

Constant	Usage
C_a	ARRAY structure complexity factor
C_c	CASE complexity factor
C_e	EXPORT of variables complexity factor
C_p	POINTER complexity factor
C_P	PROCEDURE complexity factor
C_r	RECORD complexity factor
C_{rr}	recursive RECORD field complexity factor
C_s	SET complexity factor

In section 13 we suggest a set of values for these constants.

5.2.1 Simple Items

The aspect of complexity we are trying to measure is readability. All simple items built into the language and individual names should have the same complexity and it should be less than the complexity of composite items.

We normalize the metric described in this chapter by assigning the complexity 1 to all "simple items" that the programmer will identify as a single unit. For programmers, readability is, in large part, a function of the memory load placed on a reader. Each name requires that the reader remember 1 item, so, quite aside from normalization, this is a reasonable assignment.

For Modula-2 we define **simple item** to be a name, constant declaration, or a simple type.

Introducing or using a name requires 1 unit of effort, no matter what the named item's complexity. The extra load on the reader is to remember the new name. Similarly, each individual constant declared requires remembering one name. Finally, since simple types use minimal type names in declaration statements, each is assigned a complexity of 1 unit.

DSM 1 $\forall N \in \{\ simple\ items\}\ ,\ |\ N\ |\ =\ 1$

This definition satisfies two of the axioms of chapter 4. Considering only CONST declarations, DSM 1 satisfies Axiom 1 with $C\ =\ 1$. Similarly, restricting DSM 1 to simple types, it satisfies Axiom 2 with $S\ =\ 1$.

5.3 Range Types

The complexity of a range depends on the number of new items introduced. An existing declared constant is not a new item. The range entity itself is a new item and counts as 1.

DSM 2 *If a and b are declared constants,* $|\ a..b\ |\ =\ 1$ *. If one of a and b is a declared constant and the other is a literal,* $|\ a..b\ |\ =\ 2$*. If both a and b are literals,* $|\ a..b\ |\ =\ 3$*.*

5.4 Sequences of Declarations

Different shorthand notations for declarations within a class have the same complexity. Modula-2 allows a single CONST, TYPE, or VAR to start a list of declarations of the same class. This is mainly a shorthand for entering the statements. The same amount of information must be remembered by the reader of the code.

DSM 3 $| CONST\ a_1 = b_1\ ;\ ...\ ; CONST\ a_n = b_n | = | CONST\ a_1 = b_1\ ;\ ...\ ; a_n = b_n |$

This is consistent with Axiom 4.

DSM 4 $| TYPE\ a_1\ =\ T_1;\ ...\ ;\ TYPE\ a_n\ =\ T_n | = | TYPE\ a_1\ =\ T_1;\ ...;\ a_n\ =\ T_n |$

This is consistent with Axiom 5.

DSM 5 $| VAR\ a_1 : T_1;\ ...\ ;\ VAR\ a_n : T_n | = | VAR\ a_1 : T_1;\ ...\ ;\ a_n : T_n |$

This is consistent with Axiom 9.

The complexity of a sequence of declarations is the complexity of the sum of the individual declarations.

DSM 6 $| D_1;\ ...;\ D_n | = \sum_{i=1}^{n} | D_i |$

This definition can be combined with the three rules immediately before to convert declaration sequences of CONST, TYPE, and VAR into sums of the complexity of individual CONST, TYPE, and VAR declarations. That is, the complexity of the sample code:

```
CONST PI = 3.14;
      ALPHA = 0.263
      MaxLine = 80;
VAR ALine, BLine: INTEGER;
```

is the same as the complexity of the code below.

```
CONST PI = 3.14;
CONST ALPHA = 0.263
CONST MaxLine = 80;
VAR ALine: INTEGER;
VAR BLine: INTEGER;
```

The complexities of the statements are 1, 1, 1, 2, and 2 so by DSM 6 the total complexity is 1 + 1 + 1 + 2 + 2 = 7

Record fields and formal parameter lists are handled under DSM 6. Both have the syntax of a sequence of "name:type" declarations. Any additional complexity comes from their use and their enclosing construct.

5.5 TYPE and VAR Declarations

A type declaration adds a new type name and the type structure associated with it. The complexity of the TYPE is the sum of the complexity for the name (always 1 by the Names definition) and the complexity of the type structure given.

DSM 7 $| TYPE\ X = Y | = | X | + | Y | = 1+ | Y |$

If the type structure Y is just a TYPE name, the complexity becomes 2. Note that DSM 7 discourages renaming a simple type since this will increase the complexity.

For a variable declaration, the complexity comes from the new name and the type structure.

DSM 8 $| VAR\ A : T | = | A : T | = | A | + | T | = 1+ | T |$

Declaring a variable adds the complexity of a new name plus using an old name, if the type is built in or already declared and only its name is used. Otherwise, the complexity of the new type must also be included. Therefore, declaring a type will have lower complexity than repeating the type definition for each variable.

For example, the three declarations below create 3 variables and introduce 3 anonymous types that have identical structures.

```
VAR X: POINTER TO Integer;

VAR Y: POINTER TO Integer;
```

```
VAR Z: POINTER TO Integer;
```

The complexity is $C = \mid X \mid + \mid Y \mid + \mid Z \mid + 3 \times \mid POINTER\ TO\ Integer \mid$.

Since X, Y , and Z are names, each has complexity 1. The complexity of an Integer, a simple type, is 1 and the POINTER TO Integer has complexity C_p times the complexity of Integer. Hence, $C = 3 + 3 \times C_p$.

In the code below, the TYPE declaration introduces additional complexity, but the total complexity is reduced because the complexity of each individual variable declaration is reduced. The TYPE declaration creates a "chunk" for the reader of the code. This reduces the number of times the reader must read and understand the type details.

```
TYPE P = POINTER TO Integer;
VAR X: P;
VAR Y: P;
VAR Z: P;
```

The complexity is the sum of the TYPE declaration and the VAR declarations. The TYPE declaration complexity, CT, is computed by DSM 13, giving $CT = \mid P \mid + \mid POINTER\ TO\ Integer \mid = 1 + C_p$. Each variable declaration has the same complexity, 2, so the total complexity is $1 + C_p + 3 \times 2 = C_p + 7$. Unless $C_p \leq 2$, it will be less complex to define the named type P.

5.6 Sequences of Ranges

Modula-2 allows two standard notations, shown in the example below, for sequences of ranges that are only small variations of syntax. For arrays with multiple dimensions, the indices can be specified with one set of braces or with a set per dimension. For example, either of the two declarations below can be used to define a matrix type having 10 rows indexed 1 through 10 and 17 columns indexed 4 through 20 of integers. The first notation was included as a shorthand notation for the second. The difference is two punctuation characters, too small to change the readability. Therefore, we equate their complexities.

```
TYPE Matrix1 = ARRAY[1..10,4..20] OF INTEGER;

TYPE Matrix2 = ARRAY[1..10],[4..20] OF INTEGER;
```

DSM 9 $| [a_1..b_1,\ \ldots\ , a_n\ ..\ b_n] | = | [a_1..b_1],\ \ldots\ , [a_n..b_n] |$

The complexity of a sequence of ranges is the sum of the complexity of the individual ranges. The braces add no complexity, and only act as separators for the different parts of an ARRAY declaration. For a single dimension, the complexity should be the same as the range alone. As dimensions are added, it becomes more difficult for the reader. But the complexity should be proportional to the number of dimensions. For Matrix1 and Matrix2 above, the complexity of the sequence of ranges would be $| 1..10 | + | 4..20 | = 3 + 3 = 6$.

DSM 10 $| [a_1..b_1],\ \ldots\ [a_n..b_n] | = \sum_{i=1}^{n} | a_i..b_i |$

This is consistent with Axiom 14.

5.7 Arrays

The complexity of the ranges in the dimensions and the BaseType are important to the complexity of an ARRAY declaration "ARRAY IndexType OF BaseType". Two cases exist for the BaseType.

In the most common case, the BaseType is not a POINTER back to the ARRAY declaration. In this case, both the complexity of the IndexType and BaseType may be directly computed.

DSM 11 $| \; ARRAY \; IndexType \; OF \; BaseType \; | = \; C_a \times (| \; IndexType \; | + | \; BaseType \; |)$

For $C_a > 1$, the metric satisfies Axiom 15.

If BaseType does point back to the ARRAY declaration, then a loop appears in the complexity computation. For example, ClassListType has elements of type POINTER TO ClassListType;

```
TYPE ClassListType = ARRAY[1..MaxClassList] OF POINTER TO ClassListType;
```

To avoid using the complexity of ClassListType to calculate the complexity of the BaseType, a constant is multiplied by the complexity of the IndexType. The constant should be large

as such a declaration will be more difficult for a reader to understand than the case where the BaseType is not a pointer back to the ARRAY declaration.

DSM 12 $| \, ARRAY \ IndexType \ OF \ BaseType \, | = \ C_p \times | \, IndexType \, |$

For $C_p > 1$, the metric satisfies Axiom 16.

5.8 Sets

The complexity of a set is a constant times the complexity of the base type of the set. For example, SET OF [1..4] requires the reader to remember the range and the structure introduced by the set.

DSM 13 $| \, SET \ OF \ B \, | = \ C_s \times | \, B \, |$

For $C_s > 1$, the metric satisfies Axiom 17.

Although a composite type, SET is fairly simple. In fact, it can be thought of as an ARRAY with one dimension, B, and implicit BaseType Boolean. That is, SET OF B is a form of shorthand notation for "ARRAY B OF Boolean;". Since the complexity of simple type Boolean is one, the complexity of the ARRAY version would be $C_a \times | \, B \, | \times | \, Boolean \, | = C_a \times | \, B \, |$. To ensure consistency between the ARRAY and SET complexities, we require that $C_a \ = \ C_s$.

5.9 Pointers

A pointer is more complex than its base type. We want the complexity of POINTER to increase quickly with the complexity of its base type. A pointer to a simple type is relatively easy to read. Programmer defined types require that the reader either remember the type from a previous declaration or look forward for the type. The constant C_p should be large compared with the constants previously defined.

DSM 14 | $POINTER\ TO\ X$ | = $C_p \times$ | X |

For $C_p > 1$, the metric satisfies Axiom 18.

5.10 Records

The programmer must understand each field of a record in order to understand the record. The complexity of the record must include the sum of the complexities of the fields within the record.

Let X be a declaration of the form:

```
record
```

$s_1, s_2, ..., s_l$, `(* fields with no pointers *)`

$f_1, f_2, ..., f_m$, `(* pointer fields with no circular references *)`

$p_1, p_2, ..., p_n$ `(* pointer fields with circular references *)`

`end (* node *)`

The record structure itself has a complexity greater than the complexity of its fields. We indicate this by multiplying the sum of the complexity of the fields by factor C_r.

DSM 15 $\mid X \mid = C_r \times (\sum_{i=1}^{l} \mid s_i \mid + \sum_{j=1}^{m} \mid f_j \mid + \sum_{k=1}^{n} \mid p_k \mid)$

For $C_r > 1$, the metric satisfies Axiom 11. Furthermore, since the record complexity is a sum of field complexities, the order of the fields within the record does not effect the record's complexity. Hence, the metric satisfies Axiom 12.

The complexity of the s_i and f_j fields requires only a straightforward application of other parts of the metric. The p_k fields cause a problem because they are of the form POINTER TO Y and they circularly reference the record type X. Since they point back to the current declaration, a simple application of the POINTER complexity computation is impossible. In order to calculate such a field's complexity we need the RECORD's complexity but to compute the RECORD's complexity we need to compute the field's complexity. For example, in the declarations below, computing the complexity of Node3Type requires computing the complexity of the field named N3. But N3 is a pointer to Node1Type so the complexity of Node1Type must be computed.

Node1Type contains a pointer field to Node2Type so the complexity of Node2Type must be computed. Node2Type contains a pointer to Node3Type so we must compute the complexity of Node3Type but that is what we were computing initially!

```
TYPE Node1Type = RECORD
                   Data1: CHAR;
                   N1: POINTER TO Node2Type;
                 END;
     Node2Type = RECORD
                   Data2: BOOLEAN;
                   N2: POINTER TO Node3Type;
                 END;
     Node3Type = RECORD
                   Data1: INTEGER;
                   N3: POINTER TO Node1Type;
                 END;
```

To avoid this circularity between the RECORD and the complexity of its fields, we compute an area **r** for such fields. The area involves the product of the a length and a breadth. For each $p_k,\ k = \{1, 2, \ ... \ n\}$, let area r_k be the number of circular reference paths without cycles times the maximal circular reference length. The maximal circular reference length is the number of links in the maximal circular reference path without cycles.

DSM 16 $\mid p_k \mid = \ C_{rr} \times r_k$

The complexity of a data structure using pointers depends on both the number of paths involved and the depth of the indirectness. The first factor, the number of paths, will result from the complex interrelation between types. In the extreme, this is the equivalent of

spaghetti code for declarations. This factor captures how bushy the declaration is. As an example, this factor will result in a simple linked list having a lower complexity than a binary tree structure. A complex data structure such as a compiler symbol table will have a greater number of different paths between items. The greater the number of paths, the more difficult it is for the reader to understand all the interactions between the declarations.

The second factor, the circular reference length, captures the difficulty of having to follow a chain of declarations to understand the current declaration. This factor captures how deep the structure is. It is easier for the reader if a pointer references the type directly. For example, a simple linked list of all the same type item is less complex than one that requires passing through several other types before returning to the original.

In the example given above, there is one circular path from Node1Type to Node2Type to Node3Type to Node1Type of length 3. Thus, the complexity of the N3 field is $C_{rr} \times 3$.

Although declarations of the form of X are the most common kind of record declarations used with pointers, we must also allow variant fields with CASE items. In this situation, the tag field T of the case is treated as a normal field of the record. All fields within the case have their complexity computed as normal fields and then multiplied by C_c.

DSM 17 $| \, CASE \; T \; OF \; f_1 \; ..., \; f_n \, | = | \, T \, | + C_c \times \{ \sum_{i=1}^{n} | \, f_i \, | \}$

For $C_c \geq 1$, the metric satisfies Axiom 13.

Consider the three sets of data declarations given below.

Example 1

```
     TYPE ListNode = RECORD
                       Data:INTEGER;
                       Next: POINTER TO ListNode
                     END; (* ListNode *)
```

Example 2

```
     TYPE BinaryTreeNode = RECORD
                             Data: INTEGER;
                             LeftTree,
                             RightTree:
                               POINTER TO BinaryTreeNode
                           END; (* BinaryTreeNode *)
```

Example 3

```
     TYPE SymbolTableNode = RECORD
                              Data: INTEGER;
                              SymbolType,
                              ArrayBaseType,
                              Next: POINTER TO SymbolTableNode;
```

```
                    ArrayRangeTypes,
                    RecordFields: POINTER TO SListNode
                END; (* SymbolTableNode *)
SListNode = RECORD
                ASymbol: SymbolTableNode;
                NextInList: POINTER TO SListNode
            END; (* SListNode *)
```

All the examples use Data: INTEGER to represent the data stored in the structure. Its complexity calculation is $|\ Data:\ INTEGER\ | = |\ Data\ | + |\ INTEGER\ | =\ 1 + 1 = 2$.

The first example is a simple linked list. The pointer field has only 1 path and has circular reference length of 1. The complexity of the ListNode declaration is

$$\begin{aligned} |\ ListNode\ | &= C_r \times (|\ Data:\ INTEGER\ | + |\ Next: POINTER\ \ TO\ ListNode\ |) \\ &= 1 + C_r \times (2 + C_{rr} \times 1) \\ &= 1 + 2 \times C_r + C_r \times C_{rr} \end{aligned}$$

The same record with all Integer fields would have a complexity of $1 + 4 \times C_r$.

The binary tree example has two pointer fields each of circular reference length 1. Since both subtree fields have the same complexity, the BinaryTreeNode declaration's complexity

is

$$
\begin{aligned}
|\, BinaryTreeNode \,| &= C_r \times (|\, Data : INTEGER \,| \\
&\quad + 2 \times |\, LeftTree : POINTER\ \ TO\ BinaryTreeNode \,|) \\
&= 1 + 2 \times C_r + 2 \times C_r \times (1 + C_{rr} \times 1) \\
&= 1 + 4 \times C_r + 2 \times C_r \times C_{rr}
\end{aligned}
$$

A similar record with all Integer fields would have complexity $1 + 6 \times C_r$.

The third example represents a simplified symbol table that might appear in a compiler. There are five pointer fields. Three fields directly point to the node type and so have a reference length of 1. The other two pointer fields do not directly reference the SymbolTableNode type. They reference it indirectly through SListNode. They each have a reference length of 2 and exactly 1 path each. Each of the pointers has only one path to reach SymbolTableNode. Therefore the sum of the references is 7. The resulting complexity for the SymbolTableNode declaration is

$$
\begin{aligned}
|\, SymbolTableNode \,| &= C_r \times (|\, Data :\ INTEGER \,| + C_{rr} \times 7) \\
&= 1 + 2 \times C_r + 7 \times C_r \times C_{rr}
\end{aligned}
$$

If all the fields of SymbolTableNode were of type Integer, its complexity would be $1 + 12 \times C_r$.

5.11 PROCEDURE Declarations

The complexity of a procedure declaration is proportional to the complexity of the formal parameters of the procedure, and the complexity of the return value type if present.

DSM 18 $| \ PROCEDURE \ ProcName \ (\ ParameterDeclarations \ ) : ReturnType \ | = C_P \times (| \ ParameterDeclarations \ | + | \ ReturnType \ |)$

If the procedure does not return a value, 0 is used for $| \ ReturnType \ |$. The complexity of the ParameterDeclarations is the sum of the complexity of the individual parameters.

DSM 19 $| \ ParameterDeclarations \ | = | \ P_1, P_2, ..., P_n \ | = \ \sum_{i=1}^{n} | \ P_i \ |$

Note that each parameter is of the form "VAR A:T" or "A:T". The complexity computations for these are given in the TYPE and VAR Declarations section.

5.12 Units

Creating units does not cost anything extra. The metric uses Axiom 19 for units.

DSM 20 $| \ U_1, \ U_2 \ , ..., \ U_n \ | = \ \sum_{i=1}^{n} | \ U_i \ |$

A penalty is imposed for exporting a variable from a unit. Once a variable is exported, any piece of code that imports it may change its value. This is similar to using global variables

or common areas. Such programming practices are a major source of programming error. So exporting a variable has a complexity defined by the following measure

DSM 21 $\forall N \in \{variable\ name\}, \mid EXPORT\ N;\ VAR\ N:\ AT \mid = C_e \times \mid VAR\ N:\ AT \mid$

where C_e will be large relative to other constants. Since C_e will be greater than 1, Axiom 21 is satisfied.

No extra complexity is added by exporting a constant name, type name, or procedure name. Since the reader only needs to remember the names, the complexity is the same as if they were defined within the current unit under consideration. Also, since the constant or type can not be changed, the reader does not have to worry about the values changing due to code in other locations. The metric is consistent with Axiom 20.

DSM 22 $\forall N \in \{\ constant\ name,\ typename,\ procedure\ name\}\ ,\ \mid EXPORT\ N \mid = 0$

5.13 Values of Metric Parameters

To utilize the metric, values must be assigned to the parameters. The parameter values listed in Table 5.1 are those used when evaluating the metric in Chapter 8. For simplicity, whole numbers are used. An experiment, reported in Chapter 7, was performed to provide insight on the choice of values for the constants. Due to the conflicting results obtained,

Constant	Value
C_a	2
C_c	2
C_e	8
C_p	4
C_P	2
C_r	2
C_{rr}	5
C_s	2

Table 5.1: Structural Metric Parameters

the constants given here are intuitive choices. As metric research matures and techniques for greater precision in experimental measurements become available, it may be possible to provide improved constants.

The structures associated with arrays, sets, variant fields, procedures, and records are relatively easy to understand so constants C_a, C_s, C_c, C_P and C_r are assigned the value 2. That is, a structure declaration doubles the complexity of the items within it. For example, a set declaration will be twice as complex as the set base type. As mentioned earlier, C_a and C_s must have the same value to keep the array and set metric definitions consistent.

The three remaining constants are for structures that appear to increase complexity more than simple structuring of data. Pointers and exporting of variables result in more complex programs.

Two constants deal with pointers - simple pointers and recursive pointers. Both add complexity due to the pointer structure. The recursive pointer is assigned a slightly higher value. Intuitively, the recursive pointer is more complex than the simple pointer. Looking at the complexity calculations for Example 1 (simple linked list), Example 2 (binary tree),

and Example 3 (symbol table) indicates that the value assigned recursive pointers should be high. For example, the ListNode complexity if all fields are simple variable and simple type declarations is $1 + 4 = 5$. The ListNode declaration with recursion has complexity $1 + 2 \times C_r + C_r \times C_{rr} = 1 + 4 + 2 \times C_{rr} = 5 + 2 \times C_{rr}$. We want an assignment to C_{rr} that will make the complexity of the recursion significant greater than the simpler record declaration. At the same time, we do not want to make the linked list too complex in comparison with arrays. The ARRAY declaration below has complexity 9. Assigning C_{rr} the value 5 results in a complexity of 15 for the ListNode declaration.

```
TYPE ListType = ARRAY[1..10] OF INTEGER;
```

Once 5 is selected for C_{rr}, C_p should be less than 5. Since the simple constructor parameters are 2, our choices for C_p are 3 and 4. The value 4 was selected to keep the complexity of all pointers substantially higher than the simple constructors..

The remaining constant for EXPORT of variables increases complexity more than other items since it opens up the locations in the code that can change the variable. Increasing the visibility of a variable results in greater complexity when reading the program. It is the complexity equivalent of a global variable. The largest value is assigned to the constant C_e to penalize code opening up the scope of variables.

Chapter 6

Graph Data Structure Metrics

In this chapter several data declaration metrics will be presented that are based on a directed graph, called the Data Graph, of the data declarations within a module. As with the structural data metric, the specification and implementation modules must be collapsed into one set of declarations.

The metrics described here should be applied on a per module basis. If the complexity of a set of modules is desired, the complexity of each module should be computed independently and then all the complexities should be added together to get the total complexity. For example, a data structure for a file system could involve one main module describing a general user interface to the file system. The module would use facilities described in other modules, such as low level input and output, to implement its structures on devices. In this

case, each supporting module's complexity can be computed along with the main file system module and the results summed to compute the complexity of the complete file system.

Unlike the structural metric previously defined, the metrics in this chapter are not directly based on the axioms. Instead, certain characteristics of the Data Graph are used to generate counts that reflect certain characteristics of the code. In the next chapter, they will be compared with the structural metric and we will consider the set of axioms satisfied by each.

Colors will be assigned to arcs of graphs in this chapter. The color map in Figure 6.1 shows the display representation for the colors used in this chapter. Blue arcs represent connections between declarations solely due to their sequencing. Green arcs represent references to items defined earlier in the same module. Yellow arcs represent references to items defined later in the same module. Brown arcs represent circular references (self-references). Red arcs represent connections to constants, types defined by the language, and types defined by the system environment. The gray arcs are the reverse of the blue arcs and the orange arcs are the reverse of green, yellow, and red arcs. Gray and orange arcs simplify the statement of some metrics by allowing traversals to be stated in terms of following arcs instead of traversing directed edges in reverse. Gray arcs will be used on forward references to "count" declarations between the reference to an item and its location. Orange arcs will be used to help compute a "height" for the graph.

First, a graph is created representing the data declarations. All shorthand declarations are split into multiple declarations. As shown below, one VAR declaration for three variables

Figure 6.1: Color Map.

is split into 3 VAR declarations.

```
VAR a,b,c: INTEGER;   becomes  VAR a: INTEGER;
                               VAR b: INTEGER;
                               VAR c: INTEGER;
```

To form the graph:

- create a node for each declaration of a CONST, TYPE, VAR, PROCEDURE,
- add a node for each unique simple type, system type, or imported name referenced in the declarations,
- add a node for each unique literal constant,
- for each declaration draw a blue arc from the node to the declaration that follows it in the text,
- draw a gray arc from the next node to the current node (reverse arc),

```
DEFINITION MODULE Sounds;
  TYPE WaveformProc = PROCEDURE( CARDINAL (* time *) );
  TYPE SoundSequence = POINTER TO SequenceNode;
  TYPE SequenceNode = RECORD
                        WF: WaveformProc;
                        StartTime, Duration: CARDINAL;
                      END;
  PROCEDURE PauseSound( time:CARDINAL );
  PROCEDURE GenerateTone( Tone: CARDINAL, Duration: CARDINAL );
  PROCEDURE PlaySequence( Sequence: SoundSequence );
END Sounds;
```

Figure 6.2: Example 1 Code.

- draw a green arc from the node to each earlier declaration it references,
- draw a brown arc from the node to itself if it references itself,
- draw a yellow arc from the node to each later declaration it references,
- draw a red arc from the node to each simple/system type or constant node that the node references,
- draw an orange arc in reverse direction for each arc except blue and gray arcs

We define four metrics on the Data Graph. Each measures a different aspect of data declaration complexity. The code shown below in Figure 6.2 will be used to illustrate the evaluation of each metric. The Data Graph appears in Figure 6.3.

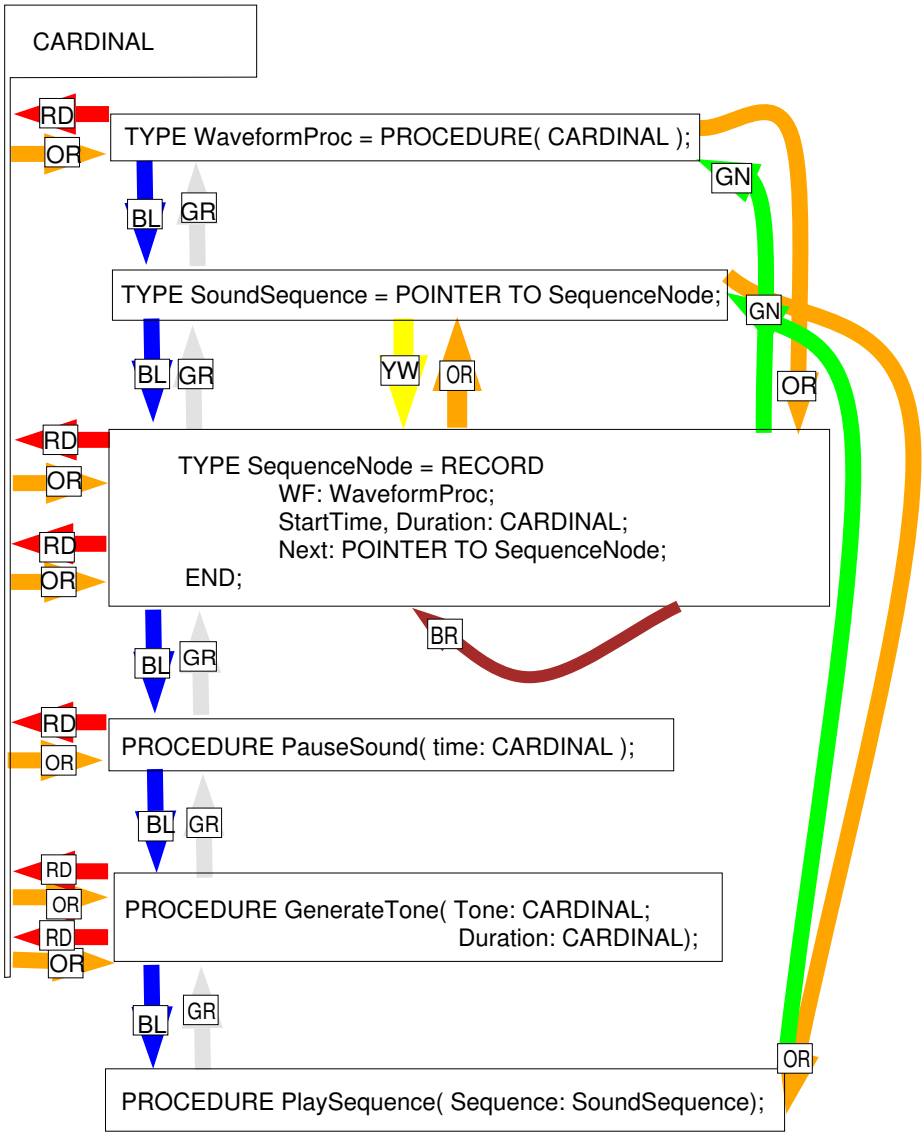


Figure 6.3: Example 1 Graph.

6.1 Simple Size Metric

This metric is similar to the statement count metric for control flow. Instead of statements, we count the number of declarations. Each declaration except the last is followed by another declaration so by counting the number of blue arcs and adding 1 we have the number of declarations.

```
simple_size_metric( graph )
  return number of blue arcs + 1
```

Note that we cannot use the number of nodes to compute the number of declarations. Each literal constant plus each simple or system type that is used is also counted in the number of nodes.

For example, the number of blue arcs is 5 in the Data Graph in Figure 6.3, so the value of this metric is 5+1 = 6.

6.2 Connection Metric

This metric computes the number of dependencies between declarations. The four weights SG (green), SB (brown), SY (yellow), and SR (red) are applied to the arcs.

```
connection_metric(graph,SG, SB, SY, SR)
```

```
complexity = 0
for each node
  for each arc
    if arc is blue,
      add 0 to complexity
    if arc is gray,
      add 0 to complexity
    if arc is green,
      add SG to complexity
    if arc is brown
      add SB to complexity
    if arc is yellow,
      add SY to complexity
    if arc is red,
      add SR to complexity
    if arc is orange,
      add 0 to complexity
return complexity
```

We will use the set of weights given in the table below.

SG	SR	SB	SY
1	1	2	4

These weights reflect an ordering of the complexity of understanding the different types of arcs. References within a declaration are considered simple if the references are to previously defined items. Such references should be assigned the lowest value which we choose to be 1.

An item can be predefined if it is a simple type of the language, is a system defined item, or is from another module specified by an IMPORT. In fact, one can think of the simple types as implicitly imported from a LanguageTypes module and the system items implicitly imported from a System module. In the case of references to predefined items, the arc will be red. Because such references are easily understood, SR is set to 1.

An item can also be predefined within the module. In that case, there will be a green arc from the declaration to the node defining the item (backward reference). Hence, SG is set to 1.

References to the current declaration (direct recursion) are more complicated so SB, the weight for a brown arc, should be greater than SR and SG. Since information for the current node is immediately available within the node, a weight of 2 is assigned.

Finally, references to items to be defined later in the module (forward references) are the most complicated. Since the arcs for such references are yellow, SY should be greater than all other weights. Forward references require that the reader jump forward in the text to understand the current declaration. This is harder than recalling a previously encountered item or looking at the current declaration. Therefore, SY must be substantially higher than SB. One common use of forward references is in declarations involving indirect recursion.

Since more context is required for indirect recursion than with direct recursion, it seems reasonable to make make SY twice the value of SB.

With these weights, the Connection complexity of example 1 can be computed. From the graph, we see there are 4 green arcs, 8 red arcs, 1 brown arc and 1 yellow arc so the metric value is $4 \times SG + 8 \times SR + 1 \times SB + 1 \times SY = 4 \times 1 + 8 \times 1 + 1 \times 2 + 1 \times 4 = 18$.

6.3 Layout Metric

The Layout metric captures complexity introduced due to the ordering of the declarations. It is easier to read a set of declarations if related items are grouped together so that the reader can focus on one area of the text when trying to understand the declarations.

```
layout_metric(graph,LG,LB,LY,LR)
  complexity = 0
  for each node
    for each arc add nc (node complexity) to complexity
      if arc is blue,
        nc is 0
      if arc is green,
        nc is LG*(path length from arc termination node to node
                  along blue arcs)
      if arc is brown
```

```
        nc is LB
      if arc is yellow,
        nc is LY*(path length from node to arc termination node
                  along gray arcs)
      if arc is red,
        nc is LR
      if arc is orange,
        nc is 0
      if arc is gray,
        nc is 0
  return complexity
```

The set of weights below are used with the Layout metric. Similar reasoning as was used for the Connection weights is applied here to determine the relative values of the weights.

LG	LR	LB	LY
1	1	2	4

In evaluating this metric on Example 1, we first compute the complexity due to the green arcs. There are four green arcs. The first, from the second node to the first, has a path of length 1 in the reverse direction along blue arcs so its complexity is $LG \times 1 = 1 \times 1 = 1$. The second green arc is from node 4 to node 2 and the reverse path has length 2 giving a complexity of $LG \times 2 = 1 \times 2 = 2$. The remaining two green arcs go from node 7 to node 2 and from node 8 to node 3. Both have a reverse path along blue arcs of length 5. Therefore,

each has complexity $LG \times 5 = 1 \times 5 = 5$. Therefore the total complexity due to green arcs is $1 + 2 + 5 + 5 = 13$.

There are 8 red arcs. Since LR is 1, each will add complexity of 1 and the total red arc complexity is 8.

There is one brown arc which will add complexity LB or 2. The one yellow arc has complexity 4.

Adding the complexities, we have $13 + 8 + 2 + 4 = 27$ for the Layout complexity.

6.4 Volume Metric

The Volume metric captures the complexity of the branching within the Data Graph. A height and a width at the leaf node level of the graph are computed and are multiplied together to give the Volume.

```
volume metric(graph)
  width = 0
  for each leaf node n
    width = width + number of orange arcs from n

  height = 0
  for each leaf node n
```

```
    for each orange arc initial_arc from n
      h = find_length_longest_path(initial_arc, graph)
      height = max(h,height)
  volume = width * height
  return volume

find_length_longest_path(initial_arc, graph)
  create empty queue
  set traversed mark in all arcs other than initial_arc to 0
  set traversal mark in initial_arc to 1
  set value in all nodes to 0
  height = 1 # count arc initial_arc as part of path length
  let n be the node at end of initial_arc
  set value of n to height
  place n into queue
  while queue isn't empty
    dequeue node into n1
    for each brown arc, increase value of n1 by 1
    for each orange arc a1 from n1
      if traversed mark in a1 is 1,
        do nothing
      else
        set traversed mark in a1 to 1
```

```
        let n2 be the destination node of a1
        if value of n2 is 0
          set value of n2 to value of n1 + 1
          height = max(height, value of n2)
          place n2 in queue
  return height
```

For example 1, there is one leaf node, the node for simple type CARDINAL. There are 6 orange arcs from the CARDINAL node so the width is 6. The longest path following orange and brown paths is from the CARDINAL node to the WaveformProc node to the SequenceNode node to the SoundSequence node to the PlaySequence node for a height of 4. Therefore, the Volume is width $\times$ height $= 6 \times 4 = 24$.

6.5 Examples

In this section we compute the metrics for several data declaration segments.

Example 2

```
CONST CharPerLine = 80;
TYPE LineType = ARRAY[1..CharPerLine] OF CHAR;
TYPE ComplexType = RECORD
                     R,I: REAL;
```

```
                    END;
VAR TmpLine: LineType;
VAR X, Y, Z: ComplexType;
```

First, the graph is drawn as shown in Figure 6.4. Counting the blue arcs and adding one gives a Size of 7.

Connection $= 5 \times SG + 4 \times SR = 9$

Layout $= 4 \times LR + 12 \times LG = 16$

Volume $= width \times height = 4 \times 3 = 12$

```
Example 3
CONST CharPerLine = 80;
TYPE LineType = ARRAY[1..CharPerLine] OF CHAR;
VAR TmpLine: LineType;
TYPE ComplexType = RECORD
                      R,I: REAL;
                    END;
VAR X, Y, Z: ComplexType;
```

Counting the blue arcs and adding one gives a Size of 7.

Connection $= 5 \times SG + 4 \times SR = 9$

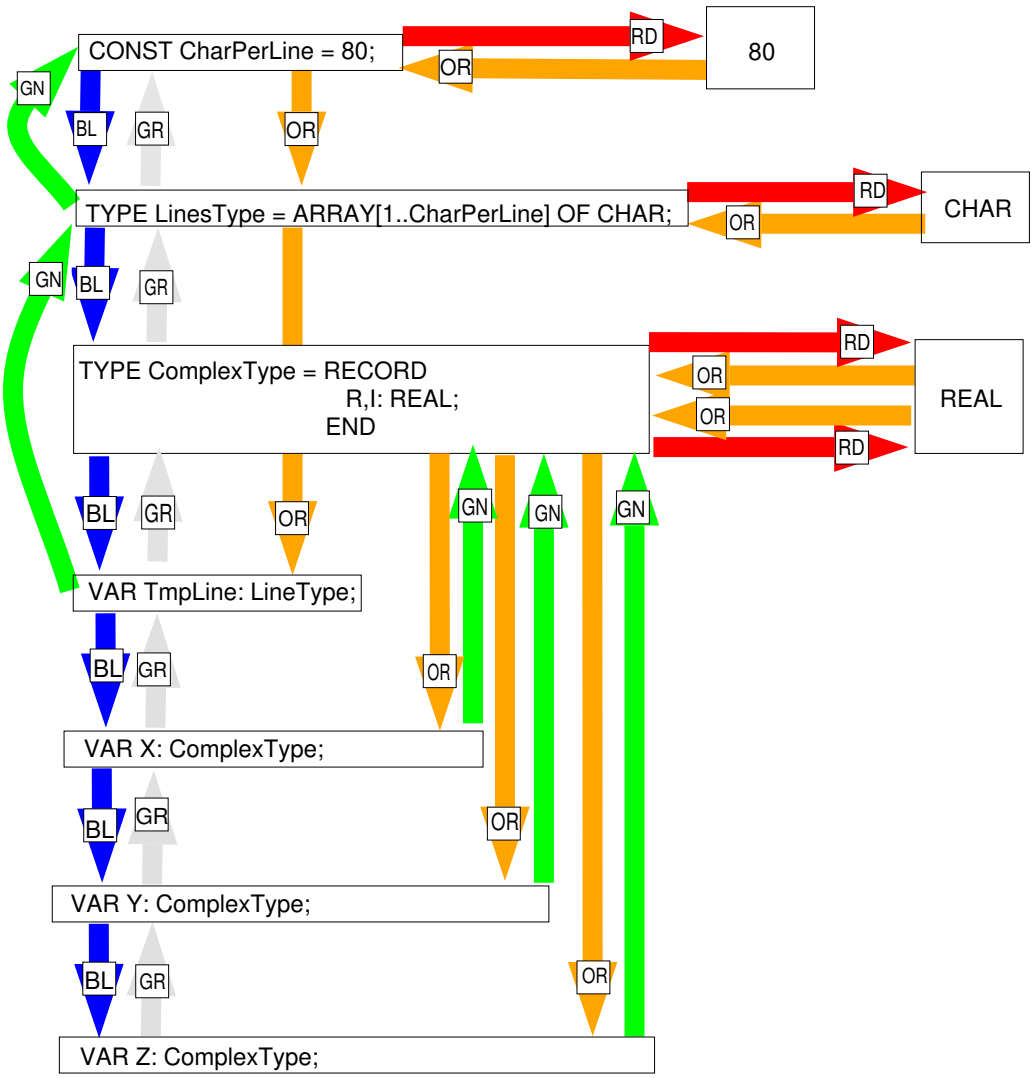


Figure 6.4: Example 2.

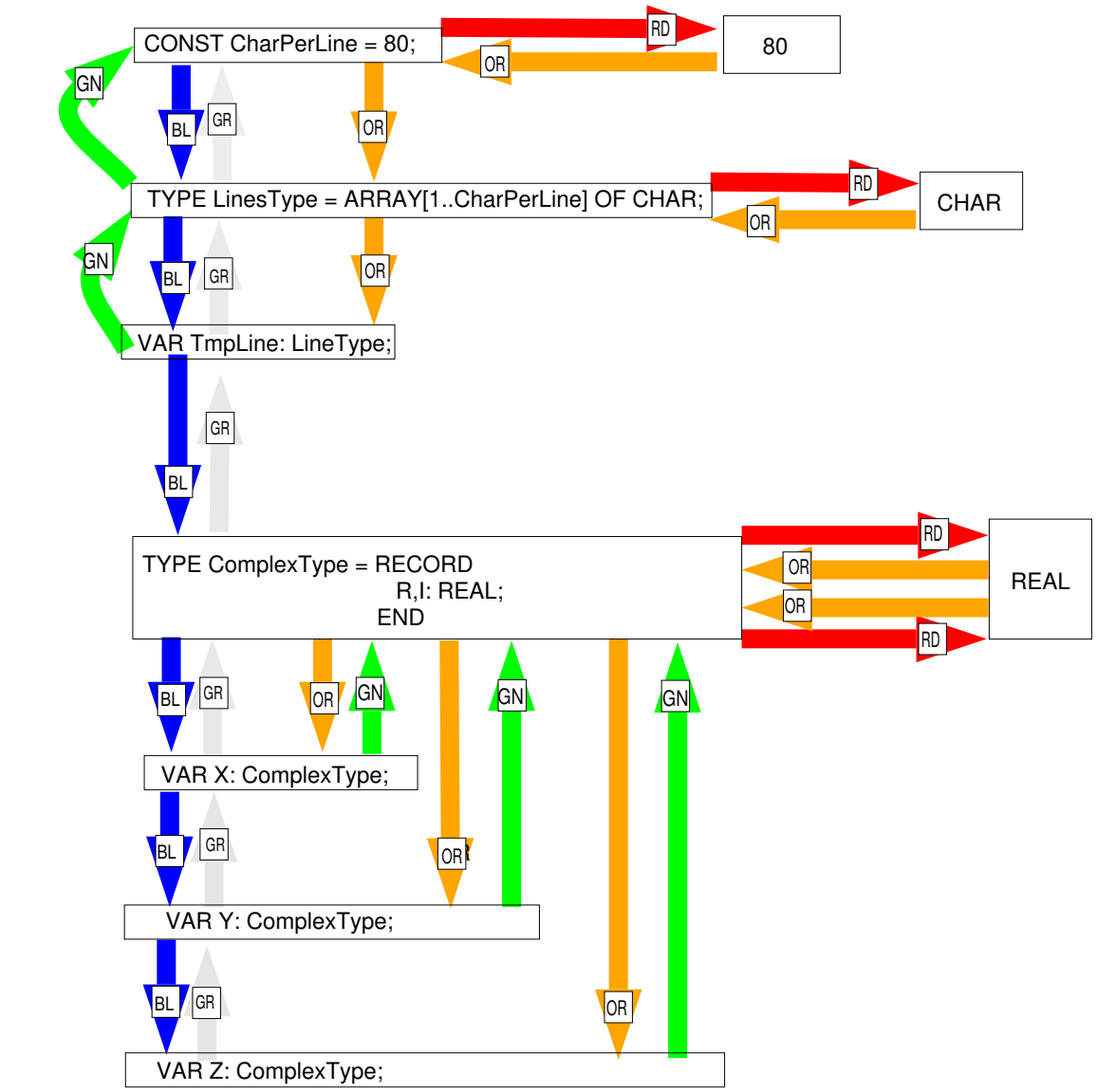


Figure 6.5: Example 3.

Layout $= 4 \times LR + 8 \times LG = 12$

Volume $= width \times height = 4 \times 3 = 12$

Note that the declarations here are the same as for Example 2 except for the ordering of statements. Three of the metrics are unchanged. The one metric that is changed, the Layout Metric, reasonably gives a lower complexity for this example than for Example 2 since in Example 3 related items are placed together. Of course, for languages that do not allow intermixing of different types of declarations, such as Pascal, the Layout Metric will have limited usefulness.

```
Example 4
TYPE ListNode = RECORD
                  Data:INTEGER;
                  Next: ListType
                END; (* ListNode *)
     ListType = POINTER TO ListNode;
VAR L: ListType;
```

Counting blue arcs and adding one gives a Size of 3.

Connection $= 2 \times SG + 1 \times SR + 1 \times LY = 7$

Layout $= 2 \times LG + 1 \times LY = 6$

Volume $= width \times height = 1 \times 3 = 3$

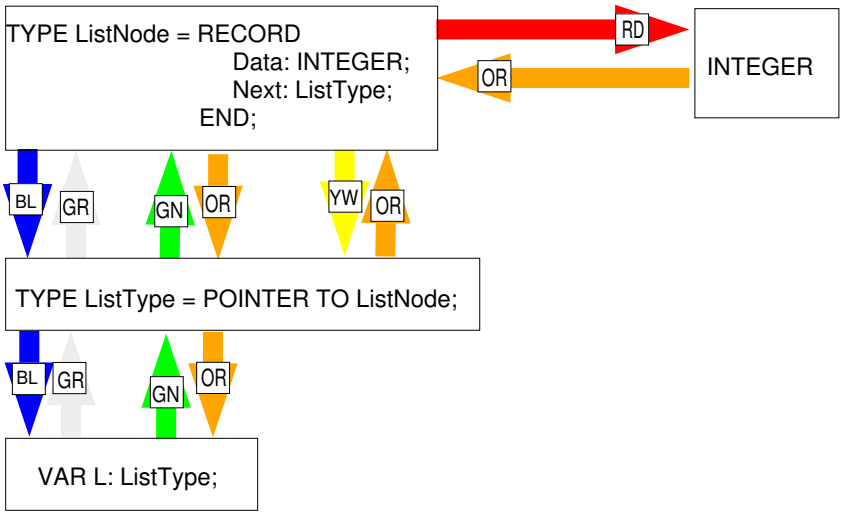


Figure 6.6: Example 4.

This is a simple linked list implementation. Both the Size and Volume metrics have low values. The Connection and Layout metrics are higher due to the forward reference in ListNode. Changing the order of ListNode and ListType doesn't lower the metrics since there will always be a forward reference in this set of declarations.

```
Example 5

TYPE BinaryTreeNode = RECORD
                        Data: INTEGER;
                        LeftTree,
                        RightTree: BinaryTreeType;
                      END; (* BinaryTreeNode *)
     BinaryTreeType = POINTER TO BinaryTreeNode;
VAR BT: BinaryTreeType;
```

Counting blue arcs and adding one gives a Size of 3.

Connection $= 2 \times SG + 1 \times SR + 2 \times LY = 11$

Layout $= 2 \times LG + 2 \times LY = 10$

Volume $= width \times height = 1 \times 3 = 3$

```
Example 6

TYPE BinaryTreeNode = RECORD
                        Data: INTEGER;
```

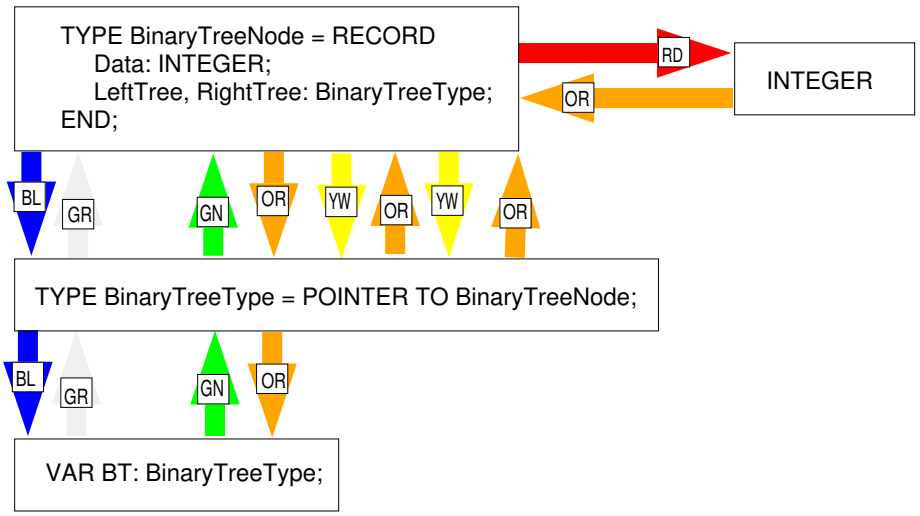


Figure 6.7: Example 5.

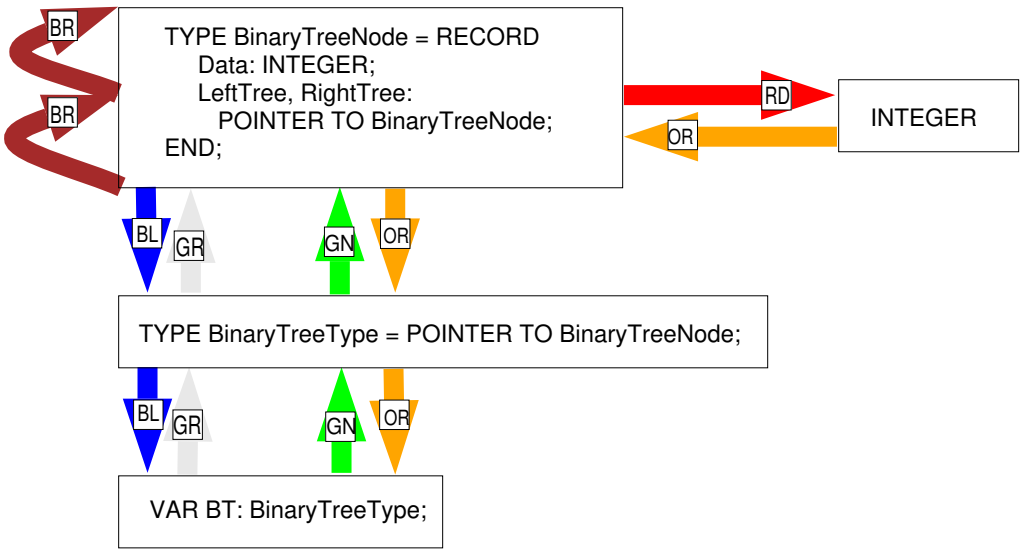


Figure 6.8: Example 6.

```
                    LeftTree,

                    RightTree:

                      POINTER TO BinaryTreeNode

                  END; (* BinaryTreeNode *)

    BinaryTreeType = POINTER TO BinaryTreeNode;

VAR BT: BinaryTreeType;
```

Counting blue arcs and adding one gives a Size of 3.

Connection $= 2 \times SG + 2 \times SB + 1 \times SR = 7$

Layout $= 2 \times LG + 2 \times LB = 6$

Volume $= width \times height = 1 \times 3 = 3$

The declarations here are almost the same as for the previous example. The only difference is in the declaration of the tree node. Example 5 uses the BinaryTreeType for the type of the left and right subtrees. Example 6 has the left and right subtree fields declared as pointers to the tree node. The Size and Volume metrics have the same value for the two examples. Note that the Connection and Layout Metrics assign the same rankings for the complexities of the two examples. A different assignment to the parameters for SY, SB, LY, and LB would change this ranking. For example, reversing the assignments so the yellow parameters are 2 (SY, LY) and the brown parameters are 4 (SB, LB) reverses the complexity rankings. Assigning values so that SY = SB and LY = LB will result in both metrics assigning the same complexity value to both examples. The observed rankings for the two metrics is a direct consequence of the decision to assign direct recursion (brown arcs) a lower complexity than indirect recursion (yellow arcs).

```
Example 7

CONST MaxSymbols = 1000;

TYPE SymbolTableNode = RECORD

                           Data: INTEGER;

                           SymbolType,

                           ArrayBaseType,
```

```
                        Next: POINTER TO SymbolTableNode;
                        ArrayRangeTypes,
                        RecordFields: POINTER TO SListNode
                    END; (* SymbolTableNode *)
    SListNode = RECORD
                    ASymbol: SymbolTableNode;
                    NextInList: POINTER TO SListNode
                END; (* SListNode *)
    STTYPE = ARRAY[1..MaxSymbols] OF SymbolTableNode;
VAR ST: STType;
```

Counting blue arcs and adding one gives a Size of 5.

Connection $= 3 \times SG + 3 \times SR + 4 \times SB + 2 \times SY = 22$

Layout $= 7 \times LG + 4 \times LB + 2 \times LY = 23$

Volume $= width \times height = 3 \times 3 = 24$

Anyone looking at this declaration sequence would find it more difficult to follow than the earlier examples. The Connection, Layout, and Volume metrics give it a higher complexity. Note that the height for the Volume metric includes counts for three direct recursive references and an indirect recursive reference of SymbolTableNode.

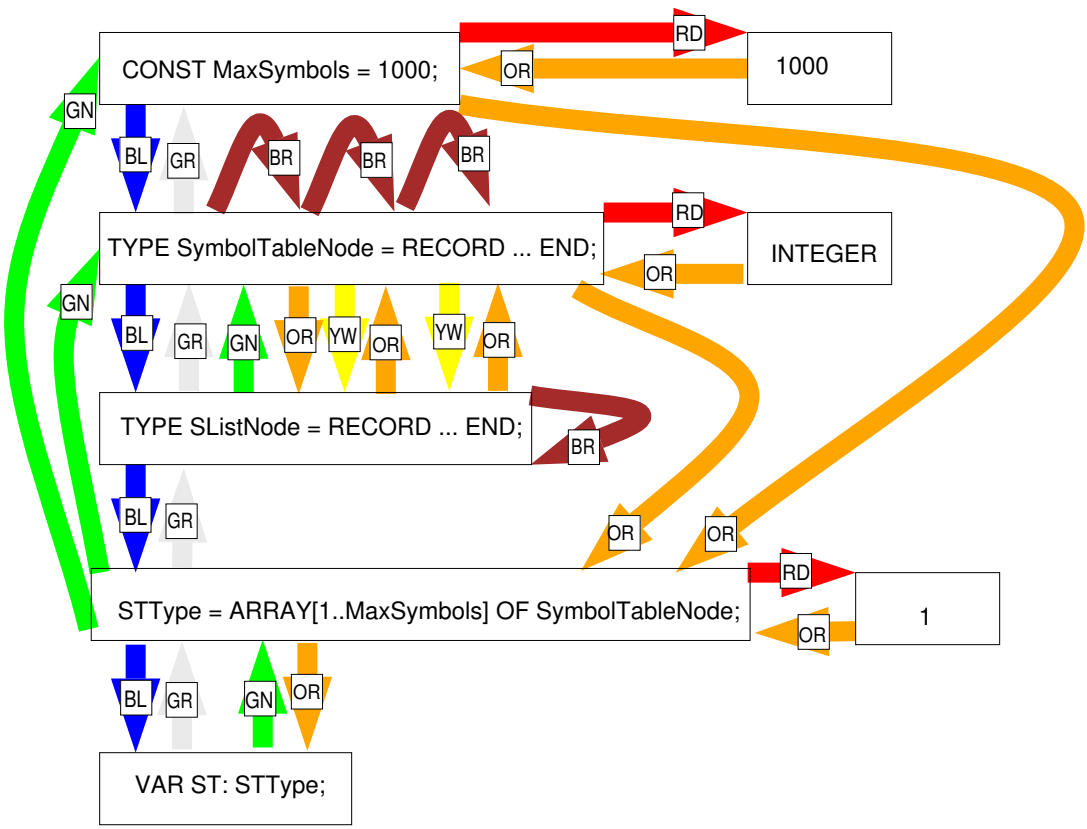


Figure 6.9: Example 7.

Chapter 7

Experimental Results

During the course of this research, we collected data to be used with the structural data structure metric through two activities. The goal of the first, an experiment, was to assign values to the constants appearing in the metric.

The goal of the second, a survey, was to collect human assessment of the readability of a set of data structures which could be used to determine if the values returned from the metric were reasonable.

7.1 Experiment Description and Results

The purpose of the experiment was to determine values or ranges of values for the eight constants that are used as multipliers in the structural data structure metric. Items were selected that allowed computation of each constant except C_P for procedures and C_e for exports. Both were excluded as they required looking at larger code units and would have extended the time required for the experiment. We wanted to keep the experiment short so that fatigue would not be an issue.

The message below was posted in the SUNY Buffalo computer science student news groups (graduate and undergraduate) and mailed to the computer science faculty mailing list.

```
Subject: Volunteers Wanted: LESS THAN HALF AN HOUR OF YOUR TIME.

     Software Metrics Experiment

I am looking for faculty, staff, or upper level students who can program
for participation in an experiment.  Locally, a metric was developed that
attempts to capture the data complexity of a program.  The complexity we
are trying to capture is the complexity perceived by readers of the code.
The metric involves a small number of constants that need to be estimated.

The experiment will take less than half an hour.  It involves reading 12
sample data structures and giving us your evaluation of each data
```

structure.

The code is in Modula-2. Anyone with experience in Pascal or other languages that provide pointers should be able to complete the experiment.

If you can volunteer the time to participate in this experiment, please respond via email to curtis. I will contact you as soon as possible to establish a convenient time. Since many people are busy with end of semester tasks, it is expected that experiments will start on May 16th and will continue into the summer as needed. Due to a previous commitment, I will be out of computer contact from May 9th through the 13th.

Help us understand the perception of program complexity.

Your participation in this experiment is greatly appreciated.

Thirteen people responded but one person was unable to schedule an appointment. The twelve subjects scheduled separate times to meet with the researcher at his office. Each subject was presented the experiment sheet shown in Appendix 1. The subject read the instructions and performed the warmup exercise with the researcher present and was allowed to ask questions. All participants appeared to understand the directions and were able to perform as expected on the sample problem. Subjects were alone in the office for the remainder of the experiment. Although the researcher scheduled appointments at 1 hour intervals, no subject required a half hour to complete the experiment. So the subjects were not under time pressure during the experiment.

Each item in the experiment presented the data structures, a short comment describing the constant of interest, information on the complexity of other items that would be used in the complexity evaluation, and a request that the subject assign a complexity value. For example, the first sample exercise was:

```
TYPE

  MySet = SET OF SmallCard;
```

Set complexity is SC $\times$ complexity of the base type. The complexity of SmallCard is 1. What complexity do you assign to MySet? __

The raw data from the data structure evaluation portion of the experiment appears in Table 7.1.

Person	1	2	3	4	5	6	7	8	9	10	11	12
Item												
1	4	5	2	5	3	6	25	4	25	5	7	28
2	4	5	3	5	5	8	30	4	45	10	14	65
3	3	4	2	4	3	6	20	2	44	4	6	200
4	4	5	2	5	4	6	25	4	28	5	7	30
5	6	8	2	8	4	16	48	6	68	6	12	108
6	6	8	2	8	4	16	288	6	980	12	24	374
7	3	5	2	5	3	4	48	4	80	8	12	30
8	2	6	3	6	3	8	10	2	40	9	12	50
9	4	6	4	7	4	8	38	4	37	6	8	40
10	5	6	4	6	8	24	36	5	66	12	21	108
11	4	4	2	4	3	5	24	4	24	4	6	28
12	3	6	2	6	4	9	36	3	45	20	30	80

Table 7.1: Experiment Results

The statistics for all items appear in Table 7.2. Included are the minimum value, maximum

value, median, mode, mean, and standard deviation. Also, the 95% confidence interval for the mean is presented. That is, if the experiment was repeated many times, we would expect the mean to be in the confidence interval 95% of the time. The end points are computed from the sample mean, $\overline{X}$, and sample standard deviation, $\overline{\sigma}$, by $\overline{X} \pm 2.145 \times (\overline{\sigma}/\sqrt{n})$ [Schulman, 1992]. Since negative complexity values are meaningless, any negative endpoints are replaced by 0.

Item	Min	Max	Median	Mode	Mean	Mean Interval	Standard Deviation
1	2	28	5	5	9.9	3.8 - 16.0	9.8
2	3	65	8	5	16.5	4.2 - 28.8	19.9
3	2	200	4	4	24.8	10.1 - 59.8	56.5
4	2	30	5	4	10.4	3.9 - 16.9	10.5
5	2	108	8	6	24.3	3.8 - 44.9	33.2
6	2	980	12	6	144.0	0.0 - 324.6	291.6
7	2	80	5	3	17.0	2.0 - 32.0	24.2
8	2	50	8	2	12.6	2.9 - 22.3	15.6
9	4	40	7	4	13.8	4.6 - 23.0	14.9
10	4	108	12	5	25.1	5.7 - 32.1	31.8
11	2	28	4	4	9.3	2.5 - 13.9	8.4
12	2	80	9	3	20.3	5.6 - 35.1	23.8

Table 7.2: Experiment Statistics

Whenever subjective values are studied, a few extreme values can greatly affect statistical values, especially when the sample size is small. For example, some athletic events use a team of judges to provide subjective scores of each participant's performance. Often the highest and lowest scores for a participant are dropped and the average of the remaining scores is used as the final score. For this experiment we also provide trimmed statistics by dropping one tenth of the values (1 in this case) from each end of the scores. The trimmed statistics appear in Table 7.3 and will be used exclusively for computing values for the structural metric constants that follow in this chapter.

Item	Min	Max	Median	Mode	Mean	Mean Interval	Standard Deviation
1	3	25	5	5	8.9	3.1 - 14.7	8.6
2	4	45	8	5	13	3.7 - 22.3	13.8
3	2	44	4	4	9.6	0.7 - 18.5	13.2
4	4	28	5	4	9.3	3.1 - 15.5	9.1
5	4	68	8	6	18.2	3.4 - 33.0	21.8
6	4	374	12	6	74.6	0.0 - 167.4	136.8
7	3	48	5	3	12.2	2.0 - 22.4	15.0
8	2	40	8	3	9.9	2.4 - 17.4	11.1
9	4	38	7	4	12.2	3.1 - 21.3	13.4
10	5	66	12	5	18.9	5.7 - 32.1	19.5
11	3	24	4	4	8.2	2.5 - 13.9	8.4
12	3	45	9	3	16.2	5.6 - 26.8	15.6

Table 7.3: Experiment Statistics with Trim

7.1.1 Item 1

```
TYPE

  AType = ARRAY[1..MaxA] OF INTEGER;
```

This item was included to estimate the ARRAY structure complexity factor, C_a. In the experiment, the index type complexity was given as 2 and the base type complexity as 1. The participants were told that the complexity should be a constant times the sum of the two given values. That is, they were asked to supply their X for $X = C_a \times (\ index\ type\ complexity + \ base\ type\ complexity) = C_a \times 3$, from which we can estimate a possible value for C_a. As stated earlier, we will use only the trim statistics when computing values and ranges for constants. Both the median and mode values yield 1.6 for C_a while the mean yields 3. The mean interval end points yield 1 and 4.9. In chapter 5 a value of 2 was proposed for C_a which is in reasonable agreement with the results of the

experiment.

7.1.2 Item 2

```
TYPE

  BType = RECORD

            C: CHAR;

            D: CHAR;

          END;
```

This item was included to estimate the RECORD structure complexity factor, C_r. The sum of the complexity of the two fields within the record was given as 4 and that the complexity should be some constant times the sum of the complexity of the fields. That is, there were asked to provide X for $X = C_r \times 4$. Substituting observed statistics for X, the median yields a value of 2 for C_r while the mode yields 1.3 and the median yields 3.3. The median interval yields end point values of 0.9 and 5.5. The value of 2 given for C_r is consistent with these values.

7.1.3 Item 3

```
TYPE

  FType = POINTER TO INTEGER;
```

This item was included to estimate the POINTER structure complexity factor, C_p. Participants were given that the complexity of INTEGER was 1 and that they should provide X for $X = C_p \times 1$. In this case, the pointer complexity C_p is the same as the value given since the INTEGER portion has complexity 1 by the metric. The value of 4 assigned in chapter 5 equals the median and mode values and is within the mean interval.

7.1.4 Item 4

```
TYPE

  HType = RECORD

            CCode: CARDINAL;

            ICode: INTEGER;

          END;
```

This item, like Item 2, was included to estimate the RECORD structure complexity factor, C_r. The complexity of the items inside the record is 4 so C_r can be estimated by dividing observed values by 4. The median, mode, and median yield 1.2, 1, and 2.3, respectively. Item 2 also was designed to yield a value for C_r. The two declarations are almost identical, a record with two fields each of a simple type. Since item 4 uses two data types, one would expect that if one of the two were to be considered more complexity it would be this one. Yet Table 7.1 shows eight participants gave item 2 higher complexity values and none gave it a lower value. The presence of a pointer in item 3 seems to have resulted in some participants adjusting downward their perception of record complexity.

7.1.5 Item 5

```
TYPE

  SetType = SET OF [1..MaxSetType];
```

The equation for the set complexity is $X = C_s \times \mid 1..MaxSetType \mid = 2 \times C_s$. Estimating C_s from the observed values yields 4, 3, and 9.1 for median, mode, and mean respectively. End points estimated from the mean interval are 1.7 and 16.6. A value of 2 for C_s, as proposed in chapter 5, is within the range estimated from the mean interval but is a little low compared with the other estimated values. This may reflect limited experience with the set construct by the participants.

7.1.6 Item 6

```
TYPE

  JType = POINTER TO JRec;

  JRec = RECORD

           CCode: CARDINAL;

           ICode: CARDINAL;

         END;
```

The equation for the complexity of JType is $X = C_p \times \mid JRec \mid = C_p \times 8$. This gives 1.5, 0.8, and 9.3 for median, mode, and mean and 83.7 as the upper interval endpoint. Since the

lower mean endpoint is negative, we will take 0 as the lowest possible value. The median and mode values appear too low for pointers and the mean too high. The proposed value for C_p is 4 seems a reasonable compromise.

7.1.7 Item 7

```
TYPE
  LClass = (Hourly, Salaried, Commission, Consultant);
  WType = RECORD
            FName, LName, MI: String;
            Address: String;
            IdCode: CARDINAL;
            Category: LClass;
            TelephoneNumber: String;
            Wage: CARDINAL;
            Salary: CARDINAL;
            BaseSalary: CARDINAL;
            CommissionRate: CARDINAL;
            Fee: CARDINAL;
          END;
```

The results on this item are extremely questionable. Participants were given that the complexity of the fields within the record structure were 24, yet only 3 gave complexities greater

than 24. It is axiomatic that the complexity of a structure containing other structures must be more complex than the inner items. Again, the results here are inconsistent with item 2 and item 4 which also address attaining a value for C_r.

7.1.8 Item 8

```
TYPE
  GType = ARRAY[1..MaxG] OF BOOLEAN;
```

The constant involved in this item is C_a. Given index type complexity as 2 and base type complexity of 1, the participants were requested to supply X for $X = C_a \times 3$. Substituting the median for X gives 2.6, the mode gives 1, and the mean gives 3.3 . The Mean interval end points yield 0.8 and 5.8 when used in the equation. These values are consistent with the results in item 1 and with the value of 2 assigned to C_r in chapter 5.

7.1.9 Item 9

```
TYPE
  LClass = (Hourly, Salaried, Commission, Consultant);
  WType = RECORD
            FName, LName, MI: String;
            Address: String;
```

```
    IdCode: CARDINAL;
    TelephoneNumber: String;
    CASE Category: JobClass OF
      Hourly: Wage: CARDINAL;
    | Salaried: Salary: CARDINAL;
    | Commission: BaseSalary: CARDINAL;
                  CommissionRate: CARDINAL:
    | Consultant: Fee: CARDINAL;
    END; (* CASE *)
  END;
```

The constant C_c was the target for this item. We requested that the participants assign a complexity to WType. They were given that the complexity of fields outside the CASE (NormalFields below) had total complexity 14 and the complexity of the fields within the CASE (CASEFields below) had total complexity 10. The complexity for WType is given by $X = C_r \times (NormalFields + C_c \times CASEFields)$. Substituting 2 for C_r, the equation becomes $X = 2 \times (14 + C_c \times 10)$ which simplifies to $C_c = X/20 - 1.4$.

Only 3 participants gave complexities greater than 10 so all statistics result in values well below 1 and most give negative values. This item was unsuccessful in providing any information on the CASE complexity factor.

7.1.10 Item 10

```
TYPE
  FType = POINTER TO FElement;
  FElement = RECORD
               Data: REAL;
               Another: FType;
             END;
```

The constant of interest is C_{rr} for recursive pointer fields. Participants were asked to give a complexity for FElement. The complexity of Data: REAL was given as 2 and the complexity of the recursive field as 2. The complexity equation is $X = C_r \times (2 + C_{rr} \times 2)$. Using 2 for C_r, the record construct constant, we can simplify the equation to $X = 4 + C_{rr} \times 4$. Solving for desired constant yields $C_{rr} = X/4 - 1$. Substituting median, mode, and mean yields 2, 0, and 3.8 respectively. The mean interval endpoints yield 0 and 7. In chapter 5 C_{rr} was assigned the value 5 which is in the interval but is higher than the values obtained from the statistics.

7.1.11 Item 11

```
TYPE
  EType = POINTER TP ERec;
  ERec = RECORD
```

```
            ECode: CARDINAL;
            Ptr1, Ptr2: EType;
          END;
```

This item also attempts to estimate C_{rr}. The values are estimated by dividing observations by 8 and subtracting 0.5. We get 0, 0, and 0.5 respectively for the statistics and 0 to 1 for the interval. Participants consistently assigned item 11 a lower complexity than item 10 although the two have identical structures except that item 11 has two instances of the recursive field. The structure given in item 11 is that of a tree or queue while item 10 is a simple linked list. It is possible that the appearance of recursion in item 10 primed participants for the appearance of recursion in this item.

7.1.12 Item 12

```
TYPE
  AType = POINTER TO AElement;
  AElement = RECORD
               AData: CARDINAL;
               PtrA1: BType;
               PtrA2: CType;
             END;
  BType = POINTER TO BElement;
  BElement = RECORD
```

```
              BData: REAL;
              PtrB: CType;
            END;
  CType = POINTER TO CElement;
  CElement = RECORD
              CData: String;
              PtrC: DType;
            END;
  DType = POINTER TO DElement;
  DElement = RECORD
              DData: INTEGER;
              Ptr1: AType;
              Ptr2: BType;
              Ptr3: CType;
            END;
```

This item also sought a value for C_{rr}. The X value sought was $X = C_r \times (2 + C_{rr} \times (16+6+4))$. The 16, 6, and 4 were given as the complexity of the recursive fields. Using a value of 2 for C_r, the equation simplifies to $X = 42 \times C + 4$. Solving for C_{rr} yields $C_{rr} = (X - 4)/42$. Using any of the trim statistics of Table 7.3 results in 0 or negative values.

In spite of the intricate dependencies between structures, 7 people assigned single digit complexity to this item. Two participants assigned values indicating it was less complex than item 11 and one gave it an equal complexity. It may be that, as participants were

exposed to more pointer related items, they became more accustomed to them and so assigned a lower complexity. Another possibility is that the description of the constant we were interested in was too detailed. It might have been better on this item to point out the recursive fields but not to explicitly give information on how the metric evaluates such fields. This item was not useful in assigning a value to C_{rr}.

To summarize, the experiment provides partial support for the assignment of values given in chapter 5 for C_a, C_r, C_p and C_s. Item 10 gives partial support for C_{rr}. The experiment was unsuccessful in providing support for the value of C_c.

7.2 Survey Description and Results

The survey in Appendix 2 was distributed in news groups comp.software-eng and comp. object on the Internet, with a request that results be returned by email. At the same time, hardcopy versions of the survey were distributed to all faculty and graduate student mailboxes in the Computer Science Department.

The raw data for the survey appears in Table 7.4 and Table 7.5. The first table list seventeen responses received over the Internet while the second table list nine hardcopy responses received from within the department.

The survey started by giving the complexity of two items to establish a scale. The two items were the complexity of a constant declaration with a complexity of 1 and the TYPE

Per.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Item																	
1	7	7	11	7	7	7	10	6	10	8	10	12	6	10	8	8	7
2	9	3	5	4	7	4	5	6	5	8	3	8	4	5	5	7	3
3	4	5	5	3	5	3	3	2	3	5	3	1	2	3	1	4	5
4	3	5	5	5	5	5	5	5	4	5	5	5	5	5	5	4	4
5	7	5	5	7	5	5	5	5	5	5	6	7	5	6	5	5	6
6	17	5	10	7	13	5	22	8	10	20	10	9	11	8	6	8	6
7	38	13	28	11	20	9	26	16	25	30	15	15	20	16	14	12	15
8	19	6	14	8	19	7	33	10	12	25	15	12	14	12	10	16	10
9	7	5	5	7	5	5	5	5	6	5	6	8	5	6	5	5	6
10	3	3	2	2	5	2	2	1	2	1	1	4	3	2	1	2	2
11	3	5	5	5	5	5	5	5	4	5	5	5	6	5	5	4	5
12	12	7	20	9	24	9	10	10	10	16	20	12	10	15	10	10	15
13	33	15	69	15	30	20	56	15	20	23	10	15	18	15	14	10	15
14	19	6	14	8	19	8	33	10	12	25	15	12	14	12	10	167	10
15	39	11	75	18	36	24	67	18	24	25	15	20	22	20	20	15	20
16	12	7	20	9	24	12	10	10	10	16	21	12	10	16	10	10	12
17	77	20	87	21	42	28	84	30	32	30	23	25	30	25	28	20	30
18	19	7	14	8	19	8	33	10	12	16	15	10	15	16	10	14	10
19	105	50	296	27	54	35	132	30	38	35	30	30	30	30	35	24	35
20	120	18	288	27	55	30	106	25	25	32	26	22	30	29	23	24	25
21	103	60	493	36	73	45	181	35	40	42	35	30	38	40	42	35	38
22	96	50	368	27	55	38	106	28	30	38	30	26	35	35	30	28	29
23	48	8	36	21	43	22	68	20	22	27	20	18	20	25	18	18	22
24	150	65	1616	36	73	60	181	40	45	50	45	36	46	54	60	45	46
25	120	25	304	27	55	28	106	28	25	33	30	22	30	29	25	24	25

Table 7.4: Network Survey Results

Person	1	2	3	4	5	6	7	8	9
Item									
1	8	7	6	7	5	1	6	2	6
2	6	4	5	4	4	1	4	2	3
3	5	5	6	7	6	1	4	3	3
4	5	5	4	5	4	1	3	2	4
5	3	5	5	5	4	1	3	2	4
6	7	7	8	10	9	1	8	4	4
7	9	8	12	15	13	1	14	5	10
8	8	7	8	11	11	1	12	4	5
9	6	5	6	8	4	1	6	2	5
10	3	3	6	3	3	1	4	2	3
11	5	6	6	7	4	1	6	2	5
12	10	8	10	8	10	1	8	3	8
13	12	8	16	20	18	1	19	5	15
14	10	7	8	11	9	1	12	3	5
15	12	8	15	20	20	1	16	4	18
16	13	6	10	11	10	1	8	3	10
17	13	8	16	25	25	1	16	5	18
18	10	6	9	11	9	1	8	4	6
19	13	8	16	30	28	1	24	5	18
20	17	9	24	36	35	1	60	4	24
21	15	9	20	28	30	1	40	5	24
22	15	9	24	32	29	1	44	4	24
23	14	9	24	22	25	1	20	4	24
24	17	9	20	38	35	1	54	4	24
25	15	9	20	32	30	1	40	4	24

Table 7.5: Local Survey Results

Line = ARRAY[1..MaxLine] OF CHAR with a complexity of 5.

Each item on the survey presented a set of data structures such as

```
TYPE P = RECORD
           A, B: INTEGER;
         END;
TYPE M = ARRAY[1..N] OF P;
```

and a space for the person to enter the complexity.

Table 7.6 presents the statistical values of all of the survey data in Table 7.4 and Table 7.5. The statistics are the minimum, maximum, median, mode, mean and standard deviation and the 95% confidence interval for the mean.

Table 7.7 contains the trim statistics for the same data. For the trimmed statistics, the two highest and two lowest values for each item were removed.

For each item, the declarations are given along with the computation of the structural metric. In particular, it is important that the metric value fall within the mean interval. If any falls outside the interval, then the metric does not reflect the participants' perception of complexity of the data declaration.

Item	Min	Max	Median	Mode	Mean	Mean Interval	Standard Deviation
1	1	12	7	7	7.3	6.2 - 8.3	2.4
2	1	9	5	4	4.8	4.0 - 5.6	1.9
3	1	7	4	3	3.7	3.1 - 4.4	1.6
4	1	5	5	5	4.4	3.9 - 4.8	1.1
5	1	7	5	5	4.9	4.3 - 5.4	1.4
6	1	22	8	8	9.0	7.0 - 11.0	4.7
7	1	38	15	15	15.8	12.3 - 19.2	8.2
8	1	33	11	12	11.9	9.1 - 14.7	6.7
9	1	8	5	5	5.4	4.7 - 6.0	1.5
10	1	6	2	2	2.5	2.0 - 3.1	1.2
11	1	7	5	5	4.8	4.3 - 5.3	1.2
12	1	24	10	10	11.0	8.9 - 13.1	5.0
13	1	69	15	15	19.5	13.4 - 25.6	14.4
14	1	33	11	10	11.9	9.1 - 14.7	6.7
15	1	75	20	20	22.4	15.5 - 29.3	16.5
16	1	24	10	10	11.3	9.2 - 13.4	5.0
17	1	87	25	25	29.2	20.1 - 38.3	21.7
18	1	33	10	10	11.5	8.9 - 14.2	6.2
19	1	296	30	30	44.6	20.0 - 69.1	58.3
20	1	288	26	24	42.9	19.0 - 66.7	56.7
21	1	493	38	35	59.2	19.1 - 99.2	95.2
22	1	368	30	30	47.4	18.2 - 76.5	69.4
23	1	68	22	20	23.0	17.3 - 28.8	13.7
24	1	1616	45	45	109.6	0.0 - 239.9	309.7
25	1	304	28	25	42.7	17.8 - 67.7	59.3

Table 7.6: Survey Statistics

Item	Min	Max	Median	Mode	Mean	Mean Interval	Standard Deviation
1	5	10	7	7	7.4	6.7 - 8.1	1.5
2	3	8	5	4	4.7	4.1 - 5.4	1.4
3	1	6	4	3	3.7	3.1 - 4.3	1.3
4	3	5	5	5	4.6	4.2 - 4.9	0.7
5	3	7	5	5	5.0	4.5 - 5.4	0.9
6	4	17	8	8	8.5	7.1 - 9.8	2.9
7	8	28	15	15	15.3	12.8 - 17.8	5.5
8	5	19	11	12	11.2	9.4 - 13.0	3.9
9	4	7	5	5	5.5	5.1 - 5.8	0.7
10	1	4	2	2	2.4	2.0 - 2.8	0.9
11	3	6	5	5	4.9	4.6 - 5.2	0.7
12	7	20	10	10	10.8	9.3 - 12.2	3.2
13	8	33	15	15	17.1	14.4 - 19.8	5.9
14	5	19	11	10	11.2	9.5 - 13.0	3.8
15	8	39	20	20	19.8	16.6 - 23.1	7.1
16	6	20	10	10	11.1	9.7 - 12.5	3.1
17	8	77	25	25	26.5	20.3 - 32.6	13.5
18	6	19	10	10	11.1	9.4 - 12.7	3.6
19	8	105	30	30	33.0	24.2 - 41.8	19.2
20	9	106	26	24	31.9	22.8 - 41.0	20.0
21	9	103	38	35	39.0	30.0 - 48.0	19.7
22	9	96	30	30	34.2	26.4 - 42.0	17.1
23	8	43	22	20	21.7	18.3 - 25.2	7.5
24	9	150	45	45	47.6	34.9 - 60.3	27.8
25	9	106	28	25	31.0	22.4 - 39.7	18.9

Table 7.7: Survey Statistics with Trim

7.2.1 Item 1

```
TYPE DA = ARRAY[1..M][1..N] OF INTEGER;
```

Structural Data Structure Metric

$$
\begin{aligned}
|\ Item\ 1\ | &= |\ DA\ | + C_a \times (|\ IndexType\ | + |\ BaseType\ |) \\
&= 1 + 2 \times (|\ [1..M]\ | + |\ [1..N]\ | + |\ INTEGER\ |) \\
&= 1 + 2 \times (1 + 1 + 1) \\
&= 1 + 2 \times 3 = 7
\end{aligned}
$$

The metric value is within the mean interval. Furthermore it is equal to the median, mode, and the integer truncation of the mean.

7.2.2 Item 2

```
TYPE R = RECORD
           A: INTEGER;
           B: INTEGER;
         END;
```

Structural Data Structure Metric

$$
\begin{aligned}
|\ Item\ 2\ | &= |\ R\ | + C_r \times (|\ A:\ INTEGER\ | + |\ B:\ INTEGER\ |) \\
&= 1 + 2 \times (2 + 2) = 9
\end{aligned}
$$

The metric value is high compared with the survey results. In fact, it is equal to the highest value assigned by participants.

7.2.3 Item 3

```
TYPE S = SET OF [1..20];
```

Structural Data Structure Metric

$$
\begin{aligned}
|\ Item\ 3\ | &= |\ S\ | + C_s \times |\ [1..20]\ | \\
&= 1 + 2 \times 3 = 7
\end{aligned}
$$

The metric value is higher than the survey values. In the experiment, participants gave higher values to the SET construct over the ARRAY. Here, we see participants assigned lower values to the SET compared to the ARRAY in item 4.

7.2.4 Item 4

```
TYPE T = ARRAY[1..2] OF INTEGER;
```

Structural Data Structure Metric

$$
\begin{aligned}
|\ Item\ 4\ | &= |\ T\ | + C_a \times (|\ [1..2]\ | + |\ INTEGER\ |) \\
&= 1 + 2 \times (3 + 1) = 9
\end{aligned}
$$

The value of the metric is high compared with the survey values. If the complexity of the index type is taken as 1, the metric would give 5 and agree with the statistics.

7.2.5 Item 5

```
TYPE A = ARRAY[N..M] OF INTEGER;
```

Structural Data Structure Metric

$$
\begin{aligned}
|\ Item\ 5\ | &= |\ A\ | + C_a \times (|\ [N..M]\ | + |\ INTEGER\ |) \\
&= 1 + 2 \times (1 + 1) = 5
\end{aligned}
$$

The metric value agrees with the survey values. From this item and item 4 it appears that literals and declared constants should have the same complexity and that a range should always have the value 1.

7.2.6 Item 6

```
TYPE RR = RECORD
            A, B: INTEGER;
            Next: POINTER TO RR;
          END;
```

Structural Data Structure Metric

$$
\begin{aligned}
|\, Item\ 6\,| &= |\, RR\,| + C_r \times (|\, A:\ INTEGER\,| + |\, B:\ INTEGER\,| + |\, Next\,| \\
&\quad + |\, POINTER\ \ TO\ \ RR\,|) \\
&= 1 + 2 \times (2 + 2 + 1 + 5) = 21
\end{aligned}
$$

The metric is high compared with the survey values. The participants assigned complexity values approximately equal to a 2 dimensional array. The metric value is approximately 2.5 times the survey values.

7.2.7 Item 7

```
TYPE R1 = RECORD
            A: INTEGER;
            NextR2: POINTER TO R2;
          END;

TYPE R2 = RECORD
            A: INTEGER;
            NextR1: POINTER TO R1;
          END;
```

Structural Data Structure Metric

$$
\begin{aligned}
|\ Item\ 7\ | &= |\ R1\ | + C_r \times (|\ A:\ INTEGER\ | + |\ NextR2\ | + |\ POINTER\ \ TO\ \ R2\ |) \\
&\quad + |\ R2\ | + C_r \times (|\ A:\ INTEGER\ | + |\ NextR1\ | \\
&\quad + |\ POINTER\ \ TOR1\ |) \\
&= 1 + 2 \times (2 + 1 + 5 \times 2) + 1 + 2 \times (2 + 1 + 5 \times 2) \\
&= 1 + 2 \times 13 + 1 + 2 \times 13 = 54
\end{aligned}
$$

The survey values are approximately double the direct recursive structure in item 6. The metric value is 3 times the observed values.

7.2.8 Item 8

```
TYPE L = RECORD
           A, B: INTEGER;
           P1, P2: POINTER TO L;
         END;
```

Structural Data Structure Metric

$$
\begin{aligned}
|\ Item\ 8\ | &= |\ L\ | + C_r \times (|\ A:\ INTEGER\ | + |\ B:\ INTEGER\ | + |\ P1\ | + |\ P2\ | \\
&\quad + 2 \times |\ POINTER\ \ TO\ \ L\ |) \\
&= 1 + 2 \times (2 + 2 + 1 + 1 + 2 \times 5) = 1 + 2 \times 16 = 33
\end{aligned}
$$

The metric value is equal to the maximum observed value and is approximately 3 times the survey values.

7.2.9 Item 9

```
TYPE V = ARRAY[L..K] OF REAL;
```

Structural Data Structure Metric

$$
\begin{aligned}
| \, Item\ 9 \, | &= | \, V \, | + C_a \times (| \, [L..K] \, | + | \, REAL \, |) \\
&= 1 + 2 \times (1 + 1) = 5
\end{aligned}
$$

The metric agrees with the survey values.

7.2.10 Item 10

```
TYPE Range1 = [1..N];
```

Structural Data Structure Metric

$$
\begin{aligned}
| \, Item\ 10 \, | &= | \, Range1 \, | + | \, [1..N] \, | \\
&= 1 + 2 = 3
\end{aligned}
$$

The metric value is slightly greater than the survey values.

7.2.11 Item 11

```
TYPE B = ARRAY [1..20] OF BOOLEAN;
```

Structural Data Structure Metric

$$\begin{aligned} \mid Item\ 11 \mid &= \mid B \mid + C_a \times (\mid [1..20] \mid + \mid BOOLEAN \mid) \\ &= 1 + 2 \times (3 + 1) = 9 \end{aligned}$$

The metric value is slightly less than twice the survey values.

7.2.12 Item 12

```
TYPE P = RECORD
           A, B: INTEGER;
         END;
TYPE M = ARRAY[1..N] OF P;
```

Structural Data Structure Metric

$$\mid Item\ 12 \mid = \mid P \mid + C_r \times (\mid A : INTEGER \mid + \mid B : INTEGER \mid)$$

$$
\begin{aligned}
& + \mid M \mid + C_a \times (\mid [1..N] \mid + \mid P \mid) \\
= \quad & 1 + 2 \times (2 + 2) + 1 + 2 \times (2 + 1) \\
= \quad & 9 + 7 = 16
\end{aligned}
$$

The median, mode, and mean are at 10 or 11 and the standard deviation is small at 3.2 . In fact, 9 of the participants set 10 as this item complexity. The metric value is 16, less than twice the survey values.

The complexity count of important tokens in the declarations (TYPE, P, RECORD, A, B, INTEGER, TYPE, M, ARRAY, 1, N and P) is 12, higher than the survey values. It appears that the participants evaluation of complexity did not include appropriate consideration of the tokens. It is possible that declarations of the two simple types included in the item occur so often that they were ignored by programmers.

7.2.13 Item 13

```
TYPE A = RECORD
           X, Y: INTEGER;
           P1: POINTER TO B;
           P2: POINTER TO C;
         END;
TYPE B = RECORD
```

```
            I: INTEGER;
          END;
TYPE C = RECORD
            Z: INTEGER;
            J: INTEGER;
          END;
```

Structural Data Structure Metric

$$
\begin{aligned}
|\, Item\ 13\,| &= |\, A\,| + C_r \times (|\, X:\ INTEGER\,| + |\, Y:\ INTEGER\,| \\
&\quad + |\, P1\,| + |\, POINTER\ \ TO\ \ B\,| + |\, P2 : POINTER\ \ TO\ \ C\,|) \\
&\quad + |\, B\,| + C_r \times (|\, I : INTEGER\,|) \\
&\quad + |\, C\,| + C_r \times (|\, Z:\ INTEGER\,| + |\, J:\ INTEGER\,|) \\
&= 1 + 2 \times (2 + 2 + 1 + C_p \times 1 + 1 + C_p \times 1) \\
&\quad + 1 + 2 \times 2 \\
&\quad + 1 + 2 \times (2 + 2) \\
&= 1 + 2 \times (2 + 2 + 1 + 4 \times 1 + 1 + 4 \times 1) + 5 + 9 \\
&= 1 + 24 + 5 + 9 = 41
\end{aligned}
$$

The metric value is 2 to 2.5 times the survey values.

7.2.14 Item 14

```
TYPE L = RECORD
           A, B: INTEGER;
           Prev, Next: POINTER TO L;
         END;
```

Structural Data Structure Metric

$$
\begin{aligned}
|\ Item\ 14\ | \;&=\; |\ L\ | + C_r \times (|\ A:\ INTEGER\ | + |\ B:\ INTEGER\ | + |\ Prev\ | \\
&\quad + |\ Next\ | + 2 \times |\ POINTER\ \ TO\ \ L\ |) \\
&=\; 1 + 2 \times (2 + 2 + 1 + 1 + 2 \times 5) = 1 + 2 \times 16 = 33
\end{aligned}
$$

The metric value is approximately 3 times the survey values.

7.2.15 Item 15

```
TYPE A = RECORD
           X, Y: INTEGER;
           P1: POINTER TO B;
           P2: POINTER TO C;
         END;
```

```
TYPE B = RECORD
           I: INTEGER;
           PC: POINTER TO C;
         END;
TYPE C = RECORD
           Z: INTEGER;
           J: INTEGER;
         END;
```

Structural Data Structure Metric

$$
\begin{aligned}
|\, Item\ 15\,| &= |\, A\,| + C_r \times (|\, X:\ INTEGER\,| + |\, Y:\ INTEGER\,| \\
&\quad + |\, P1\,| + |\, POINTER\ \ TO\ \ B\,| + |\, P2\,| + |\, POINTER\ \ TO\ \ C\,|) \\
&\quad + |\, B\,| + C_r \times (|\, I:\ INTEGER\,| + |\, PC\,| + |\, POINTER\ \ TO\ \ C\,|) \\
&\quad + |\, C\,| + C_r \times (|\, Z:\ INTEGER\,| + |\, J:\ INTEGER\,|) \\
&= 1 + 2 \times (2 + 2 + 1 + 5 + 1 + 5) + 1 + 2 \times (2 + 1 + 5) + 1 + 2 \times (2 + 2) \\
&= 1 + 2 \times 16 + 1 + 2 \times 8 + 1 + 8 = 59
\end{aligned}
$$

The metric value is approximately 2.5 times the survey values.

7.2.16 Item 16

```
TYPE W = RECORD
           X, Y: REAL;
         END;
TYPE L = ARRAY[0..M] OF W;
```

Structural Data Structure Metric

$$
\begin{aligned}
| \, Item\ 16 \, | &= | \, W \, | + C_r \times (| \, X : REAL \, | + | \, Y : \ REAL \, |) \\
&\quad + | \, L \, | + C_a \times (| \, [0..M] \, | + | \, W \, |) \\
&= 1 + 2 \times (2 + 2) + 1 + 2 \times (2 + 1) \\
&= 1 + 8 + 1 + 6 = 16
\end{aligned}
$$

The metric value is higher than the survey values. This is the same declaration as Item 12 with INTEGER replaced by REAL and the array lower bound 1 replaced by 0. The complexity values reported by the survey for the two items are essentially the same with median and mode at 10 and mean near 11.

If we reduce the complexity for the array index range to 1, the metric value becomes 14. Furthermore, if we reduce C_r to 1, as was indicated by other items, the metric value would become 10.

This may indicate two aspects of complexity perceived by programmers. First, the index range complexity on one-dimensional arrays is small and may be a constant independent of the items used to specify the range. The metric assigns higher complexity if a literal is used. At least for 0 and 1 as the lower bound, perhaps the minimal complexity should be assigned.

Second, records do not add significant complexity over the same fields defined as variables. It is possible that placing items in one or more records gives the programmer a way to organize the items and so does not increase the complexity.

7.2.17 Item 17

```
TYPE A = RECORD
           X, Y: INTEGER;
           P1: POINTER TO B;
           P2: POINTER TO C;
         END;
TYPE B = RECORD
           I: INTEGER;
           P: POINTER TO A;
         END;
TYPE C = RECORD
           Z: INTEGER;
```

```
    J: INTEGER;
    P: POINTER TO A;
  END;
```

Structural Data Structure Metric

$$
\begin{aligned}
|\, Item\ 17\,| \;=\;& |\, A\,| + C_r \times (|\, X:\ INTEGER\,| + |\, Y:\ INTEGER\,| \\
& + |\, P1\,| + |\, POINTER\ \ TO\ \ B\,| + |\, P2\,| + |\, POINTER\ \ TO\ \ C\,|) \\
& + |\, B\,| + C_r \times (|\, I:\ INTEGER\,| + |\, P\,| + |\, POINTER\ \ TO\ \ A\,|) \\
& + |\, C\,| + C_r \times (|\, Z:\ INTEGER\,| + |\, J:\ INTEGER\,| \\
& + |\, P\,| + |\, POINTER\ \ TO\ \ A\,|) \\
=\;& 1 + 2 \times (2 + 2 + 1 + 5 \times 2 + 5 \times 2) \\
& + 1 + 2 \times (2 + 1 + 5 \times 2) \\
& + 1 + 2 \times (2 + 2 + 1 + 5 \times 2) \\
=\;& 1 + 2 \times 25 + 1 + 2 \times 13 + 1 + 2 \times 15 = 109
\end{aligned}
$$

The metric value is 4 times the survey values and well out of the mean interval. The record complexity factor, C_r, may account for some of the increase, but setting it to 1 will only reduce the computed complexity to 53. If the recursive pointer complexity factor, C_{rr}, is reduced to 2 the complexity becomes 32, barely within the mean interval.

However, we believe the survey results are low on this example. On Item 12 the participants

judged a simple array of records to have complexity 10. Item 17 has three records connected by pointers and our intuition is that it should be more than 2.5 times the complexity of Item 12.

7.2.18 Item 18

```
TYPE L = RECORD
           A, B: INTEGER;
           Left, Right: POINTER TO L;
         END;
```

Structural Data Structure Metric

$$
\begin{aligned}
|\ Item\ 18\ | &= |\ L\ | + C_r \times (|\ A:\ INTEGER\ | + |\ B:\ INTEGER\ | + |\ Left\ | \\
&\quad + |\ Right\ | + 2 \times |\ POINTER\ \ TO\ \ L\ |) \\
&= 1 + 2 \times (2 + 2 + 1 + 1 + 2 \times 5) = 1 + 2 \times 16 = 33
\end{aligned}
$$

The metric value is approximately 3 times the survey values. Again, the survey results are low. This item involves a record with two pointers; yet the survey results assign it the same complexity as an array of records. It is possible that the participants' past experience with similar the declarations used for binary tree or queue implementations gave them a high

comfort level with the type definitions given and resulted in their assigning low complexity values.

7.2.19 Item 19

```
TYPE A = RECORD
           X, Y: INTEGER;
           P1: POINTER TO B;
           P2: POINTER TO C;
         END;
TYPE B = RECORD
           I: INTEGER;
           P: POINTER TO A;
           PC: POINTER TO C;
         END;
TYPE C = RECORD
           Z: INTEGER;
           J: INTEGER;
           P: POINTER TO A;
           PB: POINTER TO B;
         END;
```

$$
\begin{aligned}
|\, Item\ 19\,| &= |\, A\,| + C_r \times (|\, X : INTEGER\,| + |\, Y : INTEGER\,|) \\
&\quad + |\, P1\,| + |\, POINTER\ TO\ B\,| + |\, P2\,| + |\, POINTER\ TO\ C\,| \\
&\quad + |\, B\,| + C_r \times (|\, I : INTEGER\,| + |\, P\,| + |\, POINTER\ TO\ A\,| \\
&\quad + |\, PC\,| + |\, POINTER\ TO\ C\,|) \\
&\quad + |\, C\,| + C_r \times (|\, Z : INTEGER\,| + |\, J : INTEGER\,| \\
&\quad + |\, P\,| + |\, POINTER\ TO\ A\,| \\
&\quad + |\, PB\,| + |\, POINTER\ TO\ B\,|) \\
&= 1 + 2 \times (4 + 1 + 20 \times 5 + 1 + 6 \times 5) \\
&\quad + 1 + 2 \times (2 + 1 + 4 \times 5 + 1 + 6 \times 5) \\
&\quad + 1 + 2 \times (2 + 2 + 1 + 6 \times 5 + 1 + 6 \times 5) \\
&= 1 + 2 \times 136 + 1 + 2 \times 54 + 1 + 2 \times 66 \\
&= 515
\end{aligned}
$$

The metric value exceeds even the highest survey value for this item. If C_r is set to 1 instead of 2, the metric value is 259, still higher than all survey values. Reducing the complexity further would require reducing C_{rr}.

Note that Item 17 is a subset of the declarations in Item 19. Both the survey results and the metric give Item 19 a higher complexity than Item 17.

7.2.20 Item 20

```
TYPE A = RECORD
           X,  Y: INTEGER;
           P1: POINTER TO B;
           P2: POINTER TO C;
         END;
TYPE B = RECORD
           I: INTEGER;
           P: POINTER TO D;
         END;
TYPE C = RECORD
           Z: INTEGER;
           J: INTEGER;
           P: POINTER TO D;
         END;
TYPE D = RECORD
           L: INTEGER;
           R: INTEGER;
           P: POINTER TO A;
         END;
```

$$
\begin{aligned}
|\, Item\ 20\,| &= |\, A\,| + C_r \times (|\, X : INTEGER\,| + |\, Y : INTEGER\,| + |\, P1\,| \\
&\quad + |\, POINTER\ TO\ B\,| + |\, P2\,| + |\, POINTER\ TO\ C\,|) \\
&\quad + |\, B\,| + C_r \times (|\, I : INTEGER\,| + |\, P\,| + |\, POINTER\ TO\ D\,|) \\
&\quad + |\, C\,| + C_r \times (|\, Z : INTEGER\,| + |\, J : INTEGER\,| \\
&\quad + |\, P\,| + |\, POINTER\ TO\ D\,|) \\
&\quad + |\, D\,| + C_r \times (|\, L : INTEGER\,| + |\, R : INTEGER\,| \\
&\quad + |\, P\,| + |\, POINTER\ TO\ A\,|) \\
&= 1 + 2 \times (4 + 1 + 5 \times 3 + 1 + 5 \times 3) \\
&\quad + 1 + 2 \times (2 + 1 + 5 \times 3) \\
&\quad + 1 + 2 \times (4 + 1 + 5 \times 3) \\
&\quad + 1 + 2 \times (4 + 1 + 5 \times 6) \\
&= 73 + 37 + 41 + 71 = 222
\end{aligned}
$$

The metric value is over twice the highest survey value for item 20. Replacing C_r by 1 still leaves the metric value well above the mean.

7.2.21 Item 21

```
TYPE A = RECORD
           X, Y: INTEGER;
           P1: POINTER TO B;
           P2: POINTER TO C;
         END;
TYPE B = RECORD
           I: INTEGER;
           PC: POINTER TO C;
           PD: POINTER TO D;
         END;
TYPE C = RECORD
           Z: INTEGER;
           J: INTEGER;
           PB: POINTER TO B;
           PD: POINTER TO D;
         END;
TYPE D = RECORD
           L: INTEGER;
           R: INTEGER;
           PB: POINTER TO B;
           PC: POINTER TO C;
```

```
END;
```

Structural Data Structure Metric

$$
\begin{aligned}
|\ Item\ 21\ | \;=\; & |\ A\ | + C_r \times (|\ X : INTEGER\ | + |\ Y : INTEGER\ | + |\ P1\ | \\
& + |\ POINTER\ TO\ B\ | + |\ P2\ | + |\ POINTER\ TO\ C\ |) \\
& |\ B\ | + C_r \times (|\ I : INTEGER\ | + |\ PC\ | + |\ POINTER\ TO\ C\ | \\
& + |\ PD\ | + |\ POINTER\ TO\ D\ |) \\
& + |\ C\ | + C_r \times (|\ Z : INTEGER\ | + |\ J : INTEGER\ | + |\ PB\ | \\
& + |\ POINTER\ TO\ B\ | + |\ PD\ | + |\ POINTER\ TO\ D\ |) \\
& |\ D\ | + C_r \times (|\ L : INTEGER\ | + |\ R : INTEGER\ | + |\ PB\ | \\
& + |\ POINTER\ TO\ B\ | + |\ PC\ | + |\ POINTER\ TO\ C\ |) \\
= \;& 1 + 2 \times (4 + 1 + 4 + 1 + 4) \\
& + 1 + 2 \times (2 + 1 + 5 \times 6 + 1 + 5 \times 6) \\
& + 1 + 2 \times (4 + 1 + 5 \times 6 + 1 + 5 \times 6) \\
& + 1 + 2 \times (4 + 1 + 5 \times 6 + 1 + 5 \times 6) \\
= \;& 29 + 129 + 133 + 133 = 424
\end{aligned}
$$

Both the survey and metric assign higher complexity to Item 21 than Item 20. Since this item has two extra pointers and many of the others are identical, this seems reasonable.

7.2.22 Item 22

```
TYPE A = RECORD
           X, Y: INTEGER;
           P1: POINTER TO B;
           P2: POINTER TO C;
         END;
TYPE B = RECORD
           I: INTEGER;
           P: POINTER TO C;
         END;
TYPE C = RECORD
           Z: INTEGER;
           J: INTEGER;
           P: POINTER TO D;
         END;
TYPE D = RECORD
           L: INTEGER;
           R: INTEGER;
           P: POINTER TO B;
         END;
```

Structural Data Structure Metric

$$
\begin{aligned}
|\ Item\ 22\ | \;=\; & |\ A\ | + C_r \times (|\ X,Y : INTEGER\ | + |\ P1\ | + |\ POINTER\ TO\ B\ | \\
& + |\ P2\ | + \ |\ POINTER\ TO\ C\ |) \\
& + |\ B\ | + C_r \times (|\ I : INTEGER\ | + |\ P\ | + |\ POINTER\ TO\ C\ |) \\
& + |\ C\ | + C_r \times (|\ Z : INTEGER\ | + |\ J : INTEGER\ | + |\ P\ | \\
& + |\ POINTER\ TO\ D\ |) \\
& + |\ D\ | + C_r \times (|\ L : INTEGER\ | + |\ R : INTEGER\ | + |\ P\ | \\
& + |\ POINTER\ TO\ B\ |) \\
= \;& 1 + 2 \times (4 + 1 + 4 + 1 + 4) \\
& + 1 + 2 \times (2 + 1 + 5 \times 3) \\
& + 1 + 2 \times (4 + 1 + 5 \times 3) \\
& + 1 + 2 \times (4 + 1 + 5 \times 3) \\
= \;& 29 + 97 + 41 + 41 = 208
\end{aligned}
$$

Item 22 is a subset of the Item 21 structure with 2 fewer pointers. Both the survey results and the metric assign it a lower complexity value. The reduction in complexity is approximately 13% in the survey data. At 208 the metric value is still high but the metric reduction from Item 21 is 50%. This is a good example of how the metric values can grow quickly with small changes in the code.

7.2.23 Item 23

```
TYPE A = RECORD
           X, Y: INTEGER;
           P1: POINTER TO B;
         END;
TYPE B = RECORD
           I: INTEGER;
         END;
TYPE C = RECORD
           Z: INTEGER;
           J: INTEGER;
           P: POINTER TO B;
         END;
TYPE D = RECORD
           L: INTEGER;
           R: INTEGER;
           P: POINTER TO B;
         END;
```

Structural Data Structure Metric

$$| \, Item \ 23 \, | \quad = \quad | \, A \, | + C_r \times (| \, X, \, Y : INTEGER \, | + | \, P1 \, | + | \, POINTER \ TO \ B \, |)$$

$$
\begin{aligned}
& + \mid B \mid + C_r \times (\mid I : INTEGER \mid) \\
& + \mid C \mid + C_r \times (\mid Z : INTEGER \mid + \mid J : INTEGER \mid + \mid P \mid \\
& + \mid POINTER\ TO\ B \mid) \\
& + \mid D \mid + C_r \times (\mid L : INTEGER \mid + \mid R : INTEGER \mid + \mid P \mid \\
& + \mid POINTER\ TO\ B \mid) \\
= \quad & 1 + 2 \times (4 + 1 + 4) \\
& + 1 + 2 \times 2 \\
& + 1 + 2 \times (4 + 1 + 4) \\
& + 1 + 2 \times (4 + 1 + 4) \\
= \quad & 19 + 5 + 19 + 19 = 62
\end{aligned}
$$

This is a simple structure compared with the other pointer items since three records point to a single record type with no pointers. This is reflected in the lower complexity values assigned by the survey and the metric.

7.2.24 Item 24

```
TYPE A = RECORD
           X, Y: INTEGER;
           P1: POINTER TO B;
           P2: POINTER TO C;
```

```
          END;
TYPE B = RECORD
            I: INTEGER;
            PC: POINTER TO C;
            PD: POINTER TO D;
          END;
TYPE C = RECORD
            Z: INTEGER;
            J: INTEGER;
            PD: POINTER TO D;
            PA: POINTER TO A;
          END;
TYPE D = RECORD
            L: INTEGER;
            R: INTEGER;
            PA: POINTER TO A;
            PB: POINTER TO B;
          END;
```

Structural Data Structure Metric

$$|\ Item\ 24\ | \quad = \quad |\ A\ | + C_r \times (|\ X,Y : INTEGER\ | + |\ P1\ | + |\ POINTER\ TO\ B\ |$$

$$+ |\ P2\ | + |\ POINTER\ TO\ C\ |)$$

$$
\begin{aligned}
& + \mid B \mid + C_r \times (\mid I : INTEGER \mid + \mid PC \mid + \mid POINTER\ TO\ C \mid \\
& + \mid PD \mid + \mid POINTER\ TO\ D \mid) \\
& + \mid C \mid + C_r \times (\mid Z : INTEGER \mid + \mid J : INTEGER \mid + \mid PD \mid \\
& + \mid POINTER\ TO\ D \mid \\
& + \mid PA \mid + \mid POINTER\ TO\ A \mid) \\
& + \mid D \mid + C_r \times (\mid L : INTEGER \mid + \mid R : INTEGER \mid + \mid PA \mid \\
& + \mid POINTER\ TO\ A \mid \\
& + \mid PB \mid + \mid POINTER\ TO\ B \mid) \\
= \quad & 1 + 2 \times (4 + 1 + 5 \times 16 + 1 + 5 \times 6) \\
& + 1 + 2 \times (2 + 1 + 5 \times 6 + 1 + 15) \\
& + 1 + 2 \times (4 + 1 + 5 \times 8 + 1 + 5 \times 6) \\
& + 1 + 2 \times (4 + 1 + 5 \times 12 + 1 + 5 \times 6) \\
= \quad & 233 + 99 + 153 + 193 = 678
\end{aligned}
$$

This item has the greatest number of pointers and there is no particular pattern within the linking of records. This structure would be the equivalent of spaghetti code. Both the survey and the metric give this item the highest complexity.

7.2.25 Item 25

```
TYPE W = RECORD
```

```
            A, B: INTEGER;
            P1: POINTER TO X;
            P2: POINTER TO Y;
          END;
TYPE X = RECORD
            I: INTEGER;
            P: POINTER TO Z;
          END;
TYPE Y = RECORD
            C: INTEGER;
            D: INTEGER;
            P: POINTER TO Z;
          END;
TYPE Z = RECORD
            L: INTEGER;
            R: INTEGER;
            P: POINTER TO W;
          END;
```

Structural Data Structure Metric

$$|\ Item\ 25\ | \quad = \quad |\ W\ | + C_r \times (|\ A, B : INTEGER\ | + |\ P1\ | + |\ POINTER\ TO\ X\ |$$

$$+ |\ P2\ | + |\ POINTER\ TO\ Y\ |)$$

$$
\begin{aligned}
& | X | + C_r \times (| I : INTEGER | + | P | + | POINTER\ TO\ Z |) \\
& | Y | + C_r \times (| C : INTEGER | + | D : INTEGER | + | P | \\
& + | POINTER\ TO\ Z |) \\
& | Z | + C_r \times (| L : INTEGER | + | R : INTEGER | + | P | \\
& + | POINTER\ TO\ W |) \\
= \quad & 1 + 2 \times (4 + 1 + 5 \times 3 + 1 + 5 \times 3) \\
& + 1 + 2 \times (2 + 1 + 15) \\
& + 1 + 2 \times (4 + 1 + 15) \\
& + 1 + 2 \times (4 + 1 + 30) \\
= \quad & 73 + 27 + 41 + 71 = 222
\end{aligned}
$$

Item 25 is the same as Item 20 with names changed. The survey results are consistent between the two items indicating some consistency in assigning complexity values. The metric values are identical as the metric ignores names.

Despite the large differences between the survey and computed complexity values on items 17 through 25, there is considerable correlation between the two sets of values. In particular, the ranking among the 9 items between the two is very similar as shown in Table 7.8. The rankings are in agreement except on items 22 and 25 which have only small differences in median and metric values. Since the structure metric values for 22 and 25 are over 200

and differ by only 4, the two values are essentially the same. It is not surprising that, with those surveyed generally unfamiliar with estimating data structure complexity, the absolute values are not very close to the metric values for complex data structures.

	Survey	Survey	Metric	Metric
Item	Median	Ranking	Value	Ranking
17	25	3	109	3
18	10	1	33	1
19	30	7	515	7
20	26	4	222	4
21	38	8	424	8
22	30	6	208	5
23	22	2	62	2
24	45	9	678	9
25	28	5	212	6

Table 7.8: Rankings

Given the strong ranking order agreement we have just observed, it is reasonable to study the correlations between the survey results and the structural metric values.

7.2.26 Correlation Analysis

Correlation analysis using SAS/STAT[1] was performed on the observed mean values and structural metric values on each item. Similarly, a second analysis was performed on the observed median values and structural metric values for each item. All data values, including extreme values, were used for all items in the analysis results shown. Similar tables were generated with extreme values removed, increasing the correlation.

[1]SAS/STAT from SAS Institute., Inc. is a system for data analysis that provides, among many other statistical options, correlation routines.

Two correlations, Pearson's r (also called the sample correlation) and Spearman's r_s were requested. Pearson's r is computed by dividing the sample covariance by the product of the standard deviations of the two data sets. Let X and Y represent the two sets of N values each and S_X, S_Y be their respective standard deviations. The sample covariance is $S_{XY} = \sum(X - \overline{X})(Y - \overline{Y})/(N-1)$ and $r = S_{XY}/(S_X \times S_Y)$. The Spearman's r_s is computed by converting numeric values to ranks before computing the correlation. Pearson's is the most commonly applied correlation method and is preferred if the data is normal whereas the use of Spearman's is preferable for non-normal data. In looking at the data, it was observed that the mean and median values are low. Since the lowest complexity value that could be assigned was 0, there was little room below the observed values for value selection. On the other hand, the right side was not bounded. This resulted in skewed distributions, often with long tails extending to the right. Thus we decided to calculate both correlations because for most items the data is not normal.

The Pearson's analysis of the observed mean and metric yields a correlation factor of 0.88534 which indicates a strong correlation. It also produced a two sided p-tail factor of 0.0001. The one sided p-tail[2] is .00005, well below the standard 5% criterion for rejecting the null hypothesis. Values lower than 5% provide greater confidence in the correlation. The Spearman analysis indicates an even higher correlation at 0.95983 with the same p-value.

The median also has strong correlation with the metric values. In fact, the correlation coefficients, 0.88839 and 0.96975, are slightly closer to 1 than for the mean values. The two

[2]If we were interested in testing for both positive and negative correlation, the two sided p-tail value would be used. Since we are interested in positive correlation ($r_s > 0$), we use the one-sided p-tail which is half the two sided p-tail.

sided p-tail factor was 0.0001 for both. As mentioned above, extreme values were included. A single extreme value can greatly effect the mean. However, if several items have values at or near the median, there will be little if any change in the median caused by one extreme value.

7.3 Discussion

Even with a small sample size, the experiment provided useful results for estimating the metric parameter assignments.

The median and mode proved to be more useful than the mean and mean interval. The median was always equal to or larger than the mode with the difference usually small. On items with small differences between the median and the mode, the most common value appears in the middle of the range of observations, indicating that the data may be close to normal. The mean was always larger than the median and the difference was often substantial because a few very high values moves it substantially away from the mode and median values.

Also, the standard deviation was high on most items. This indicates that participants did not share the same perception of the complexity for the items. The high standard deviations also resulted in large 95% confidence intervals. If we exclude Item 6 with a standard deviation of 136.8, the average mean interval size was 17. To put this in perspective, the largest median value was 12 and the largest mode value was 6. In most cases the median

and mode fell in the extreme lower end of the mean interval. Originally, it was believed that the end points of the mean interval might provide a range of values to consider for the constants. As it turned out, the mode and mean were better choices for the range of constants to consider.

The survey showed that participants judged the complexity of data structures to be lower than the values returned by the structural metric. This was consistent on all items except the first which had the median and mode less than the metric but the mean greater than the metric.

The standard deviations were lower than those observed in the experiment and the average mean interval size was 7.58 which was significantly smaller than the 17 of the experiment. The median and mode were within the interval in all but 3 cases and even then they were within 0.1 of the lower end point. In the survey introduction two items and their complexities were given with the hopes of normalizing the results but this only worked moderately well.

Analysis of the survey results indicate several changes to the structural data metric that would bring its values more in line with the observed values. They are:

1. The complexity of all range types should be assigned the value 1. In developing the metric, we believed the introduction of a new literal constant added an extra item to remember so [1..2] should have a higher complexity than [N..M]. It appears that this is not the case, at least for numeric ranges when the lower bound is a "standard" value such as 0 or 1. Therefore, we recommend replacing the Range Types portion of

the structural metric definition by $| a..b | = 1$.

2. The value 2 for the record structure parameter, C_r, appears to be too high for small records. The survey indicates that an assignment of 1 would be more appropriate. We believe that the record structure should add some additional complexity so we do not propose to reduce the constant to 1. Instead, we replace C_r by 1.3 and truncate the result. For records of small internal complexity the resulting metric value will be the same as if the value was 1. Complex records will see a slight increase in the complexity due to the record structure.

3. The value of the recursive parameter C_{rr} may be too high at 5, so we reduce it to 3. The survey results suggest a value around 2. We set it at 3 instead of 2 to avoid inconsistences with the C_p value.

   Consider the two declarations A and B below. The complexity of the TYPE A declaration is $| A | + C_p \times | INTEGER | = 4 \times 1 = 5$. If C_{rr} was set to 2, the complexity of POINTER TO C in type B would be $C_{rr} \times number\ of\ paths \times longest\ path\ length = 2 \times 1 \times 2 = 4$. With the reduced C_r, the final value for TYPE B is $| B | + C_r \times 4 = 1 + TRUNC(1.3 \times 4) = 6$. It seems obvious that the TYPE B declaration with its mutual recursion with TYPE C should be more complex than the TYPE A declaration. Therefore, a value of 2 for C_{rr} seems too small.

```
TYPE A = POINTER TO INTEGER;


TYPE B = RECORD
            X: POINTER TO C;
```

```
            END;


    TYPE C = RECORD
               Y: POINTER TO B;
            END;
```

The correlation analysis indicates that there is strong correlation between the metric values and complexity values assigned by participants in the survey. In particular, the rank correlation is strong. Although the metric may give inflated values with the parameters given in Chapter 5, its values follow the same trends as human assigned values.

One surprise with the survey was the small number of replies received - only 17 from the Internet and 9 returned locally. There were two initial concerns with posting the survey on the net.

The first was that the background and level of the user could not be determined. The local distribution went to faculty and graduate students so some level of expertise could be assumed. Since on the net anyone could respond, we sent the survey to groups dealing with software engineering issues which did not appear to have large undergraduate and inexperienced programmer followings.

A second concern with placing the survey on the net was that such a large number of replies might be returned that handling the mail involved would consume a significant amount of time. Given the number of postings in comp.software-eng and comp.object, there are a potentially large number of people involved.

Why did only a small number respond? One possibility is that people did not feel comfortable judging complexity as an expert. Another possibility is that people did not wish to take the time, either because they are busy or do not see any benefit in participating. If a survey is placed on the network again, it would be advisable to make it also available as a Word Wide Web document with automatic collection of results. That way, responders would not have to save and edit a file, thereby reducing the effort required. Also the novelty of collecting information this way should attract some people.

Chapter 8

Evaluation of Metrics

In this chapter, we will evaluate our metrics in two ways. First, we will look at several examples of declarations from textbooks and will evaluate the metrics on these examples. Second, we will look at a set of properties proposed by others and will see which ones hold for our metrics.

8.1 Examples

Only the data and procedure declarations will be given for each example. Each example is prefixed by comments that indicate the example, its purpose, and the reference.

8.1.1 Random Number Examples

The first three examples are different implementations of random number generators.

```
------------------------------------------------------------------------
Example 1 - From page 71 of McCracken, Salmon.
MODULE Random;
FROM InOut IMPORT WriteCard, WriteLn;
CONST
  HowMany = 100;
VAR
  I, Number, Seed: CARDINAL;
  MODULE Calc;
  IMPORT Seed;
  EXPORT Rand;
  CONST
    A = 21;
    B = 400;
    PROCEDURE Rand() : CARDINAL;
------------------------------------------------------------------------
```

```
-------------------------------------------------------------------------

Example 2 - From page 240, Gabrini and Kurtz.

MODULE PseudoRandom;

  EXPORT Random;

  VAR Seed: REAL;

  PROCEDURE Random(): REAL;

-------------------------------------------------------------------------



-------------------------------------------------------------------------

Example 3 - From page 620 of Cooper.

IMPLEMENTATION MODULE Random;

CONST DEFAULTseed=1;

VAR Seed: LONGINT;

PROCEDURE View(): LONGINT;

PROCEDURE Generate(VAR Seed: LONGINT);

  CONST MULTIPLIER = 16807;

        QUOTIENT = 127773;

        REMAINDER = 2836;

  VAR Low, Hi, Test: LONGINT;

PROCEDURE Get(): LONGINT;

  VAR Temp: LONGINT;

-------------------------------------------------------------------------
```

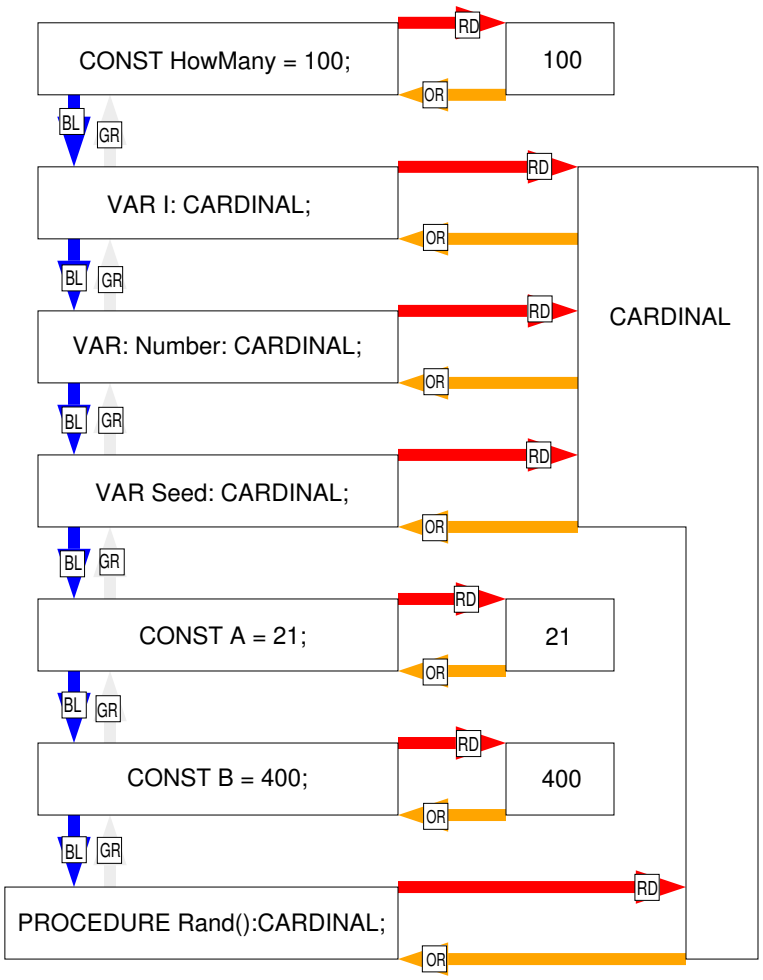


Figure 8.1: Graph for Example 1.

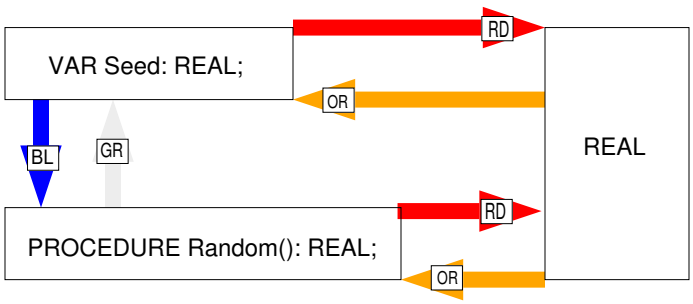


Figure 8.2: Graph for Example 2.

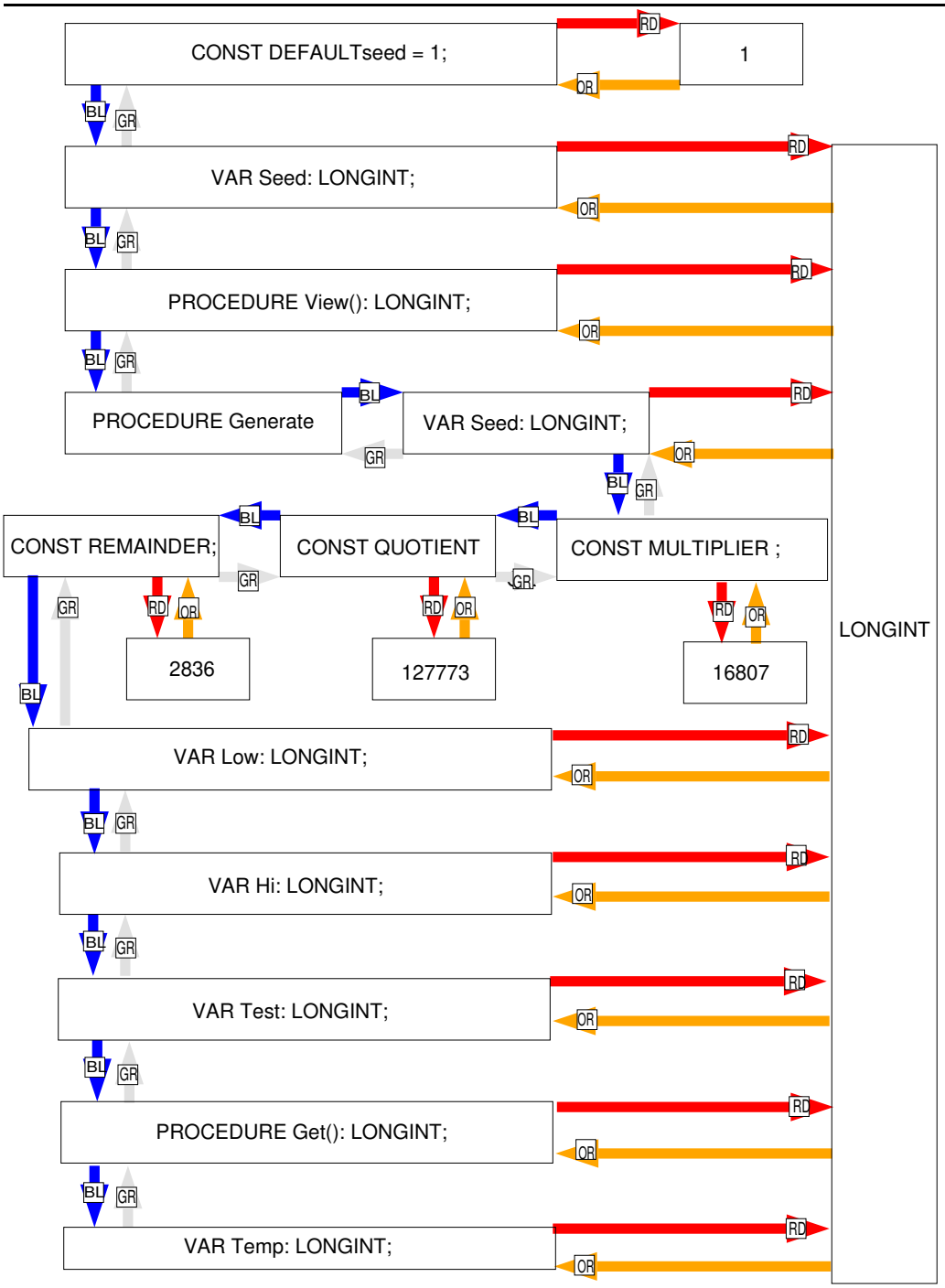


Figure 8.3: Graph for Example 3.

Table 8.1 gives the metric values for these three examples.

Metric	Size	Connection	Layout	Volume	Structural
Example					
1	7	7	7	1	10
2	2	2	2	1	3
3	13	12	12	1	20

Table 8.1: Random Number Generator Complexities

Note that all three examples have declarations that are essentially independent. This would be true of many library modules. This is reflected in the metric values in two ways. First, the Volume metric value is small for all three examples. Second, the Connection and Layout metrics are identical for all three examples. All of the complexity for these two metrics on these examples results from red arcs. Thus, only the Size metric among the graph metrics distinguishes these three examples. In the last column we give the value of the Structural metric defined in Chapter 5. Note the correspondence between the Size and Structural metrics for this set of examples.

8.1.2 String Examples

The next two examples are implementations of string.

```
------------------------------------------------------------------------

Example 4 - From page 94 of Gabrini, Kurtz

CONST Max = 80;

TYPE String = POINTER TO CharSequence;

     CharSequence = ARRAY [0..Max] OF CHAR;

------------------------------------------------------------------------
```

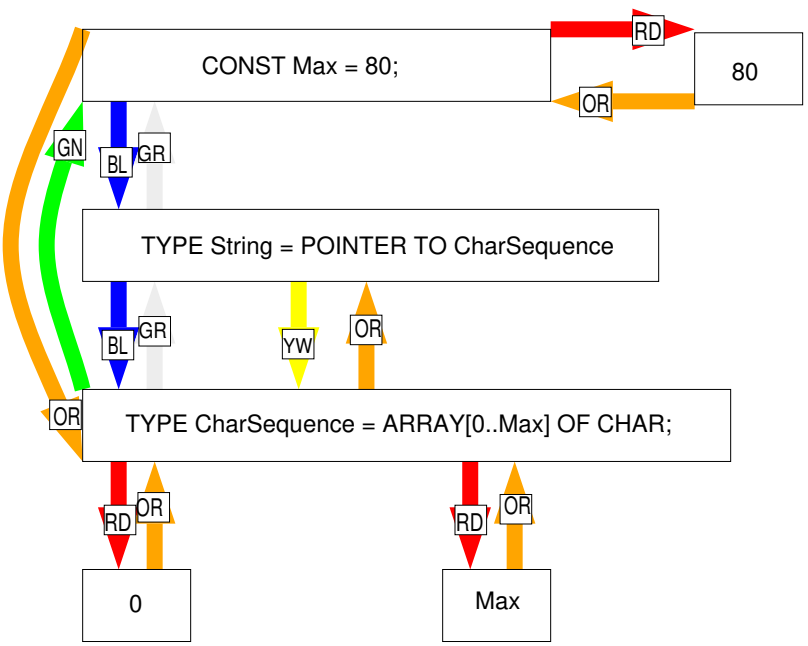


Figure 8.4: Graph for Example 4.

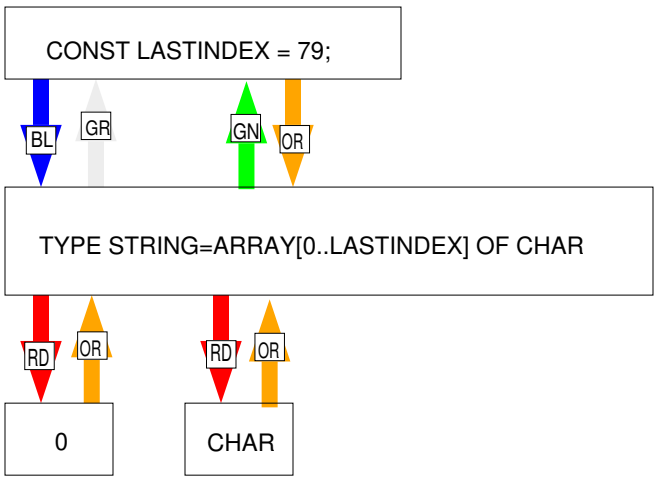


Figure 8.5: Graph for Example 5.

```
------------------------------------------------------------------------

Example 5 - From page 719 of Walter.

CONST LASTINDEX = 79;

TYPE STRING = ARRAY [0..LASTINDEX] OF CHAR;

------------------------------------------------------------------------
```

Table 8.2 gives the graph and Structural metric values for these two implementations.

Metric	Size	Connection	Layout	Volume	Structural
Example					
4	3	8	9	2	17
5	2	3	3	1	6

Table 8.2: String Complexities

Although the number of declarations involved is small (indicated by the small Size values), the Connection, Layout, Volume and Structural metrics rank Example 4 of higher complexity, capturing the extra complexity due to dependencies between items. Furthermore, they assign a significantly higher complexity value.

Since the Layout value is greater than the Connection value in Example 4, a programmer should consider if the order of the declarations can be changed. In this case, switching the order of the second and third declarations in Example 4 will reduce both the Connection and Layout values to 5.

It should be noted that our complexities should never be used by themselves to say that the declarations in Example 5 are better than those in Example 4. All that we can say is that are metrics indicate that Example 5 is less complex to read. If a program requires moving strings between variables, it may be that Example 4 is better for the application.

8.1.3 Stack Examples

Four implementations of stacks follow. Table 8.3 gives the corresponding metric values.

Metric	Size	Connection	Layout	Volume	Structural
Example					
6	9	20	48	27	70
7	10	18	49	24	84
8	10	20	41	40	64
9	9	24	47	48	80

Table 8.3: Stack Complexities

```
-------------------------------------------------------------------------

Example 6 - From page 199 of Gabrini, Kurtz.  A static implementation.

IMPLEMENTATION MODULE Stacks;

FROM StackElements IMPORT Max, StackElementType;

FROM Storage IMPORT ALLOCATE, DEALLOCATE;

TYPE Stack = POINTER TO StackType;

     StackType = RECORD

                   Top: [0..Max];

                   Data: ARRAY [1..Max] OF StackElementType;

                 END;

PROCEDURE CreateStack( VAR S: Stack);

PROCEDURE Full(S: Stack): BOOLEAN;

PROCEDURE Push( VAR S: Stack; Elt: StackElementType);

PROCEDURE Empty( S: Stack): BOOLEAN;

PROCEDURE Pop( VAR S: Stack; VAR Elt: StackElementType);

PROCEDURE StackTop( S: Stack; VAR Elt: StackElementType);

PROCEDURE DisposeStack( VAR S: Stack);

-------------------------------------------------------------------------
```

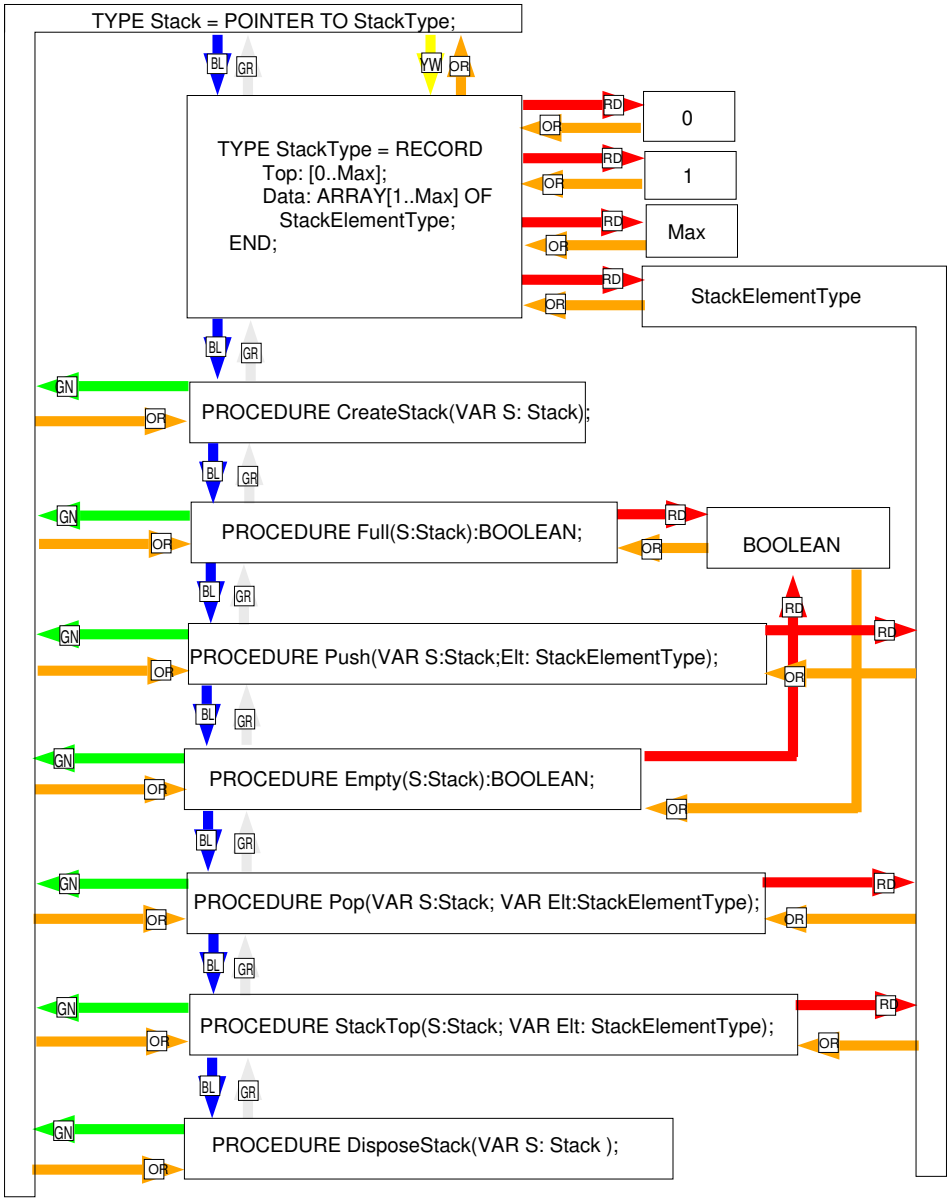


Figure 8.6: Graph for Example 6.

```
-------------------------------------------------------------------------
Example 7 - From Gabrini, Kurtz. A dynamic implementation.
VAR StackError: BOOLEAN;
IMPLEMENTATION MODULE Stacks;
FROM StackElements IMPORT StackElementType;
FROM Storage IMPORT ALLOCATE, DEALLOCATE;
TYPE Stack = POINTER TO StackNode;
     StackPtr = Stack;
     StackNode = RECORD
                   element: StackElementType;
                   next: StackPtr;
                 END;
PROCEDURE CreateStack(VAR S: Stack);
PROCEDURE Empty(S: Stack): BOOLEAN;
PROCEDURE Push(VAR S: Stack; Elt: StackElementType);
PROCEDURE Pop(VAR S: Stack; VAR Elt: StackElementType );
PROCEDURE StackTop(S: Stack; VAR Elt: StackElementType);
PROCEDURE DisposeStack(VAR S: Stack);
-------------------------------------------------------------------------


-------------------------------------------------------------------------
Example 8 - From page 609 of Cooper.  A static implementation.
CONST TOP = 100;
```

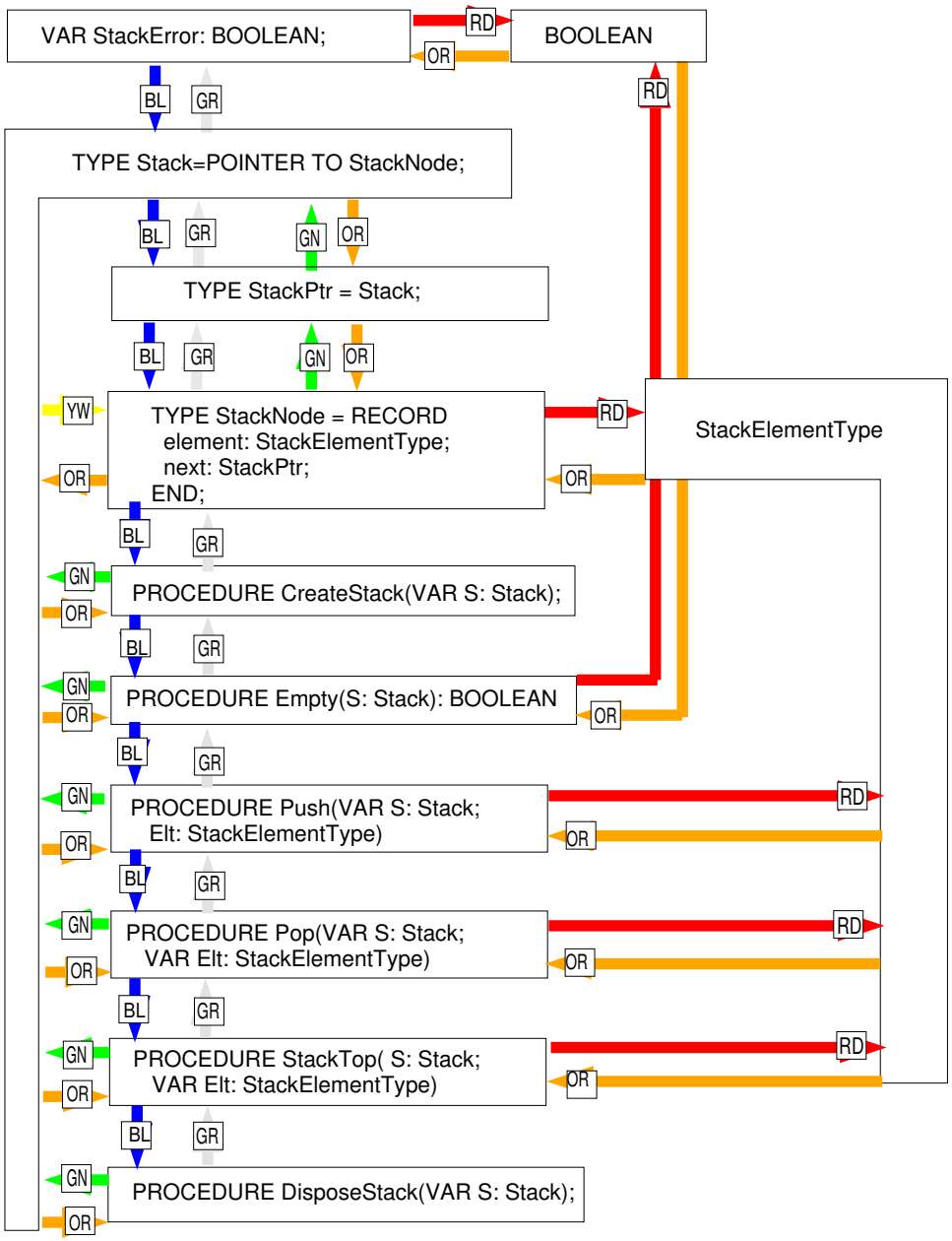


Figure 8.7: Graph for Example 7.

```
TYPE DataTYPE = ARRAY[1..TOP] OF CHAR;

TYPE T = RECORD

           Info: DataTYPE;

           Position: INTEGER[0..TOP];

         END;

IMPLEMENTATION MODULE Stack;

PROCEDURE Inspect(S: T; VAR Ch: CHAR);

PROCEDURE Push(VAR S: T; Ch: CHAR);

PROCEDURE Pop(VAR S: T; VAR Ch: CHAR);

PROCEDURE Empty(S: T): BOOLEAN;

PROCEDURE Full(S: T): BOOLEAN;

PROCEDURE Initialize(VAR S: T);

PROCEDURE Remove(VAR S: T);
```

```
Example 9 - From page 577 of Koffman. A static implementation.

CONST

  MaxStack = 100;

TYPE

  Stack = POINTER TO StackObject;

  StackObject = RECORD

                  Top : [0..MaxStack];
```

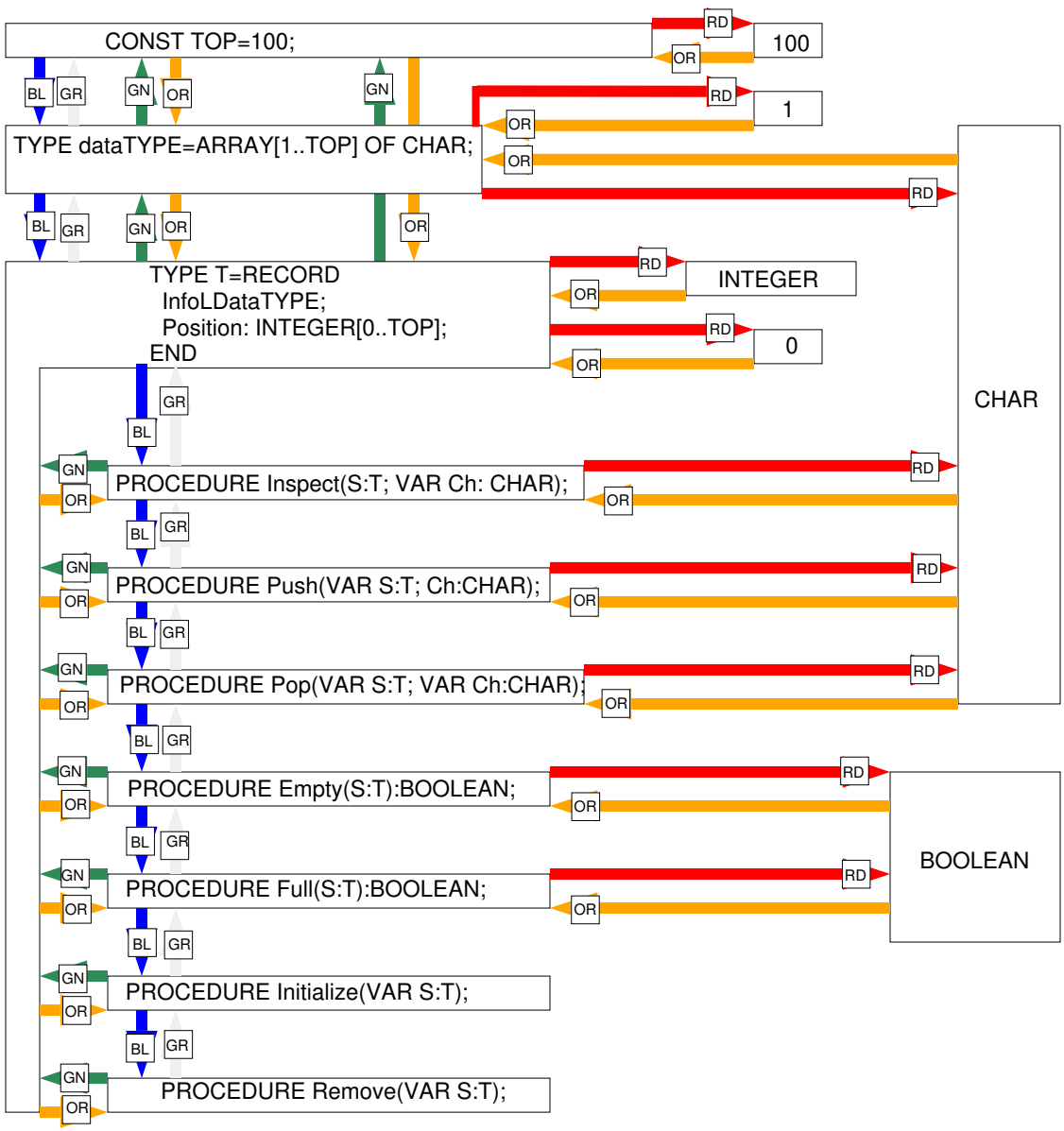


Figure 8.8: Graph for Example 8.

```
                    Items : ARRAY [1..MaxStack] OF CHAR;
                END;
PROCEDURE CreateStack( VAR S : Stack);
PROCEDURE Push (VAR S : Stack; X : CHAR; VAR Success : BOOLEAN);
PROCEDURE Pop (VAR S : Stack; VAR X : CHAR; VAR Success : BOOLEAN);
PROCEDURE Retrieve (S : Stack; VAR X : CHAR; VAR Success : BOOLEAN);
PROCEDURE IsEmpty (S : Stack) : BOOLEAN;
PROCEDURE IsFull (S: Stack) : BOOLEAN;
-------------------------------------------------------------------------
```

Examples 6 and 7 use a generalized stack element type while examples 8 and 9 define stacks of characters. Since the generalized stack element is imported in both cases, it is treated for complexity purposes the same as primitive types and system types so the difference does not affect the graph-based and Structural metric values.

The Size metric values and Connection values for all four Stack examples are approximately the same. The Layout values are significantly higher than the Connection values. This is in part due to the parameter types on the access routines. However, some reordering of declarations will also decrease the Layout value. For example, reversing the order of the first two declarations in Example 6 reduces the Connection value to 17 and the Layout value to 38.

Large differences appear for the Volume and Structural metric values. In particular, Example 7 has a low Volume metric value but a high Structural metric value due to the mutual

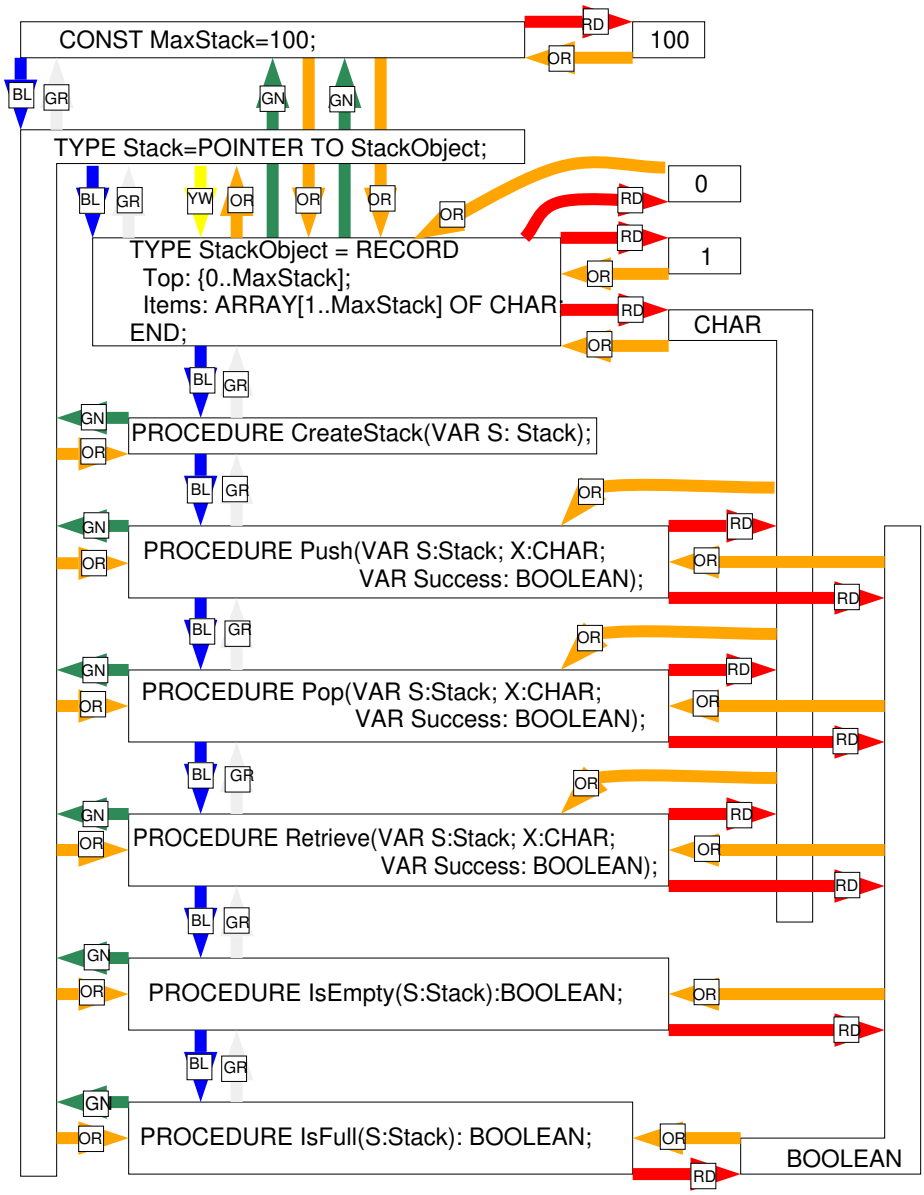


Figure 8.9: Graph for Example 9.

recursion of three declarations. A look at the declarations leads us to conclude that the declaration of StackPtr can be eliminated and its only use in the StackNode declaration can be replaced by Stack. This reduces the Size to 9, the Connection metric to 16, the Layout metric to 37, the Volume to 18, and the Structural metric to 72. We can reduce the Layout metric value by 6 to 31 by reversing the Stack and StackNode declarations. This shows that the metric values can be used to suggest problems and then to effect improvements. In this case, the Layout metric value is high compared with the Connection metric value, indicating that the position of declarations is increasing the complexity. By considering the arcs contributing to the increased Layout value, we can consider reorderings of nodes attached to the arcs to search in an effort to reduce the Layout metric.

Examples 6, 8, and 9 are of static sized arrays, that is, array implementations of stacks. Example 7 is a dynamic implementation using recursive pointer based structures. In comparing the graph-based metric values, the dynamic example has approximately the same complexity as the majority of the static-based examples. It receives lower Connection metric and Volume metric values than any of the others. With the improvements mentioned above, Example 7 has a Structural value that is only slightly larger than one static example (Example 6) and is lower than another (Example 9).

Example 8 has a low Structural metric and Layout metric values while its other metric values are not particularly lower than the other examples. There are two reasons for the lower metric values for Example 8. First, the example does not use a pointer to the stack information. Instead it directly passes the record containing the stack information. Such an approach cannot be used in some languages. For example, C does not support passing

its records (struct) as a VAR parameter. Also, passing the information this way requires making the record structure known to the user of the stack module. This would allow the user to manipulate the stack without using the module routines, possibly leaving the stack in an inconsistent state.

Second, it separates the declaration of the stack information data type into a declaration outside of the record. This introduces a new name, adding a complexity of 2 (1 for definition and 1 for use). Since the array is now outside of the record, its complexity of 6 is not multiplied by the record complexity constant of 2, reducing the complexity by 6.

The structure of Example 8 is the least complex of the stack examples. The Size metric values are nearly identical so are of little use. The only metric giving Example 8 a high value with respect to the other examples is Volume and this is only because it uses a greater number of constants and primitive types.

8.1.4 Graph Examples

Finally, we will look at three implementations of graphs. We will assume ElementType is a name imported from another module. Therefore, in the Structural metric, ElementType has complexity 1. Table 8.4 gives the corresponding metric values.

The Connection and Layout metrics indicate that the declarations may not be ordered in the best way to reduce complexity. As a case in point, Example 10 could be reordered to place the declaration of Struct immediately before the declaration of Graph. This would

Metric	Size	Connection	Layout	Volume	Structural
Example					
10	3	19	31	6	46
11	4	17	30	6	39
12	5	22	42	8	75

Table 8.4: Graph Complexities

reduce the length of the backward references from Struct and convert the forward reference from the Graph node to a backward reference.

```
------------------------------------------------------------------------
Example 10 - From page 582 of Gabrini, Kurtz.
CONST MaxVertices = 100;
TYPE Graph = POINTER TO Struct;
  Struct = RECORD
             N: KeyType;
             Matrix: ARRAY [1..MaxVertices], [1..MaxVertices] OF INTEGER;
             Value: ARRAY [1..MaxVertices] OF ElementType;
           END;
------------------------------------------------------------------------


------------------------------------------------------------------------
Example 11
CONST MaxVertices = 100;
TYPE Graph = POINTER TO Struct;
     Connections = RECORD
```

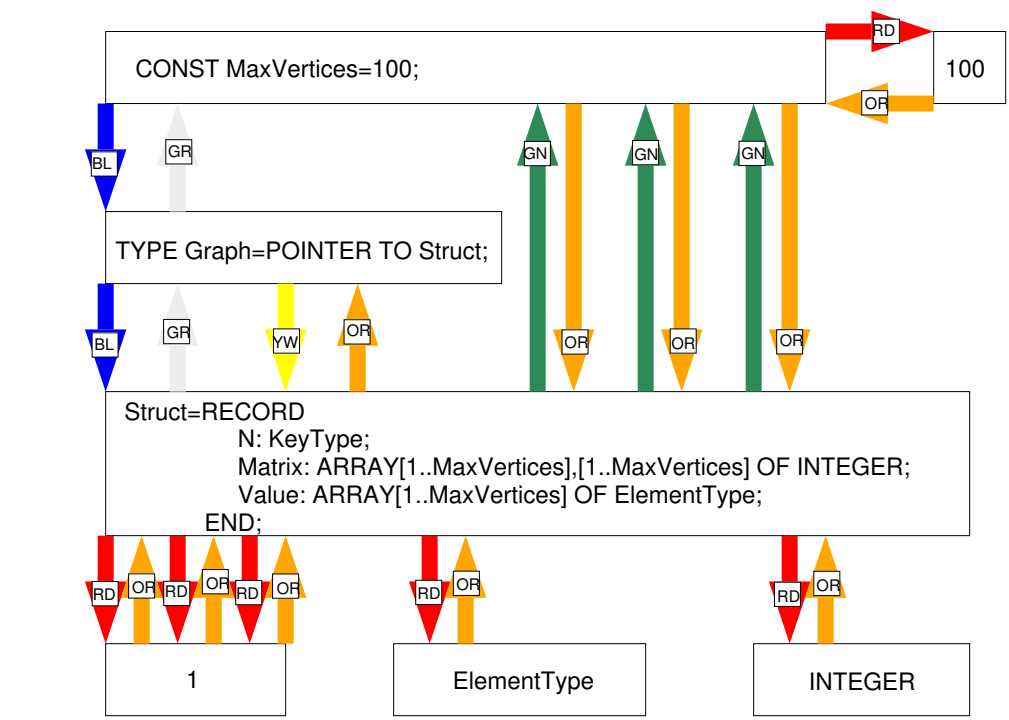


Figure 8.10: Graph for Example 10.

```
                    FromNode, ToNode: [1..MaxVertices];
                    Value: ElementType;
                  END;
    Struct = RECORD
               X: ARRAY [1..MaxVertices] OF Connections;
             END;
```

Example 12

```
CONST MaxVertices = 100;
TYPE Graph = POINTER TO Struct;
    ConnectionPtr = POINTER TO Connection;
    Connection = RECORD
                   FromIndex, ToIndex: [1..MaxVertices];
                   Value: ElementType;
                   NextFrom, NextTo: ConnectionPtr;
                 END;
    Struct = RECORD
               FromNode, ToNode: ARRAY [1..MaxVertices] OF ConnectionPtr;
             END;
```

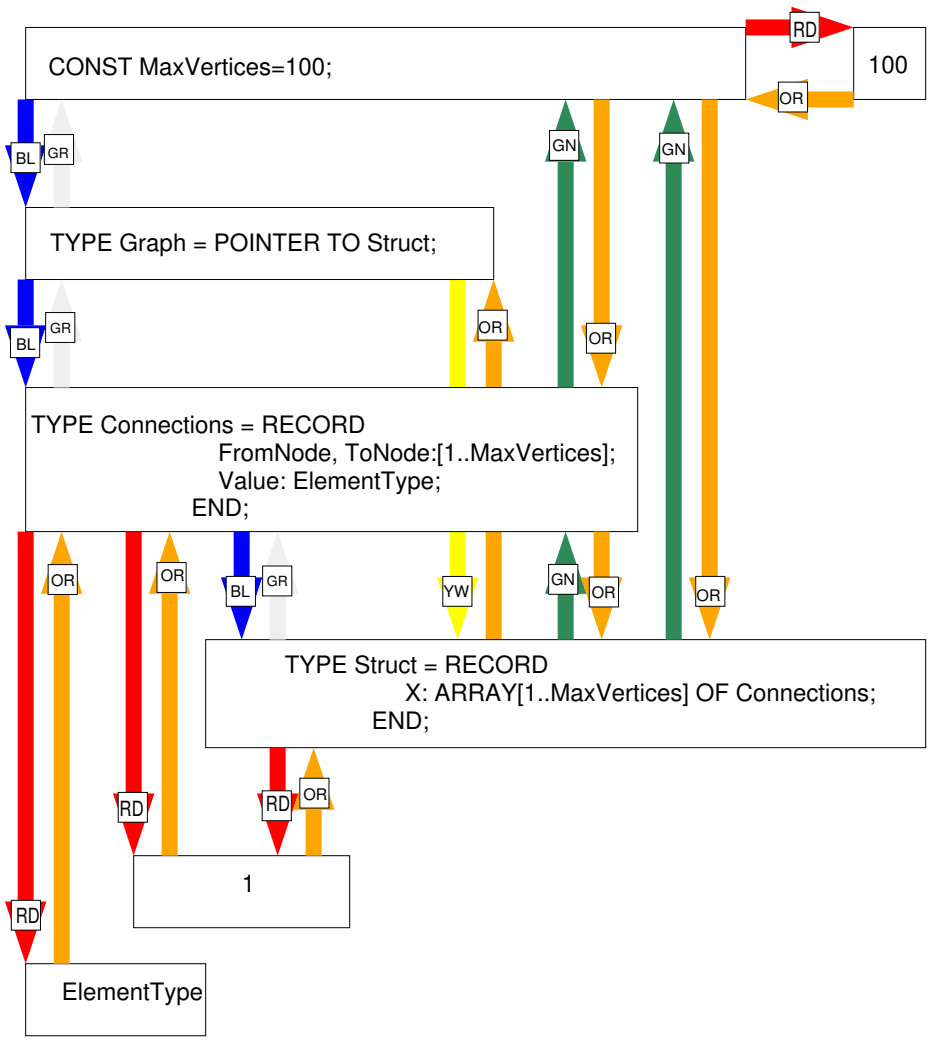


Figure 8.11: Graph for Example 11.

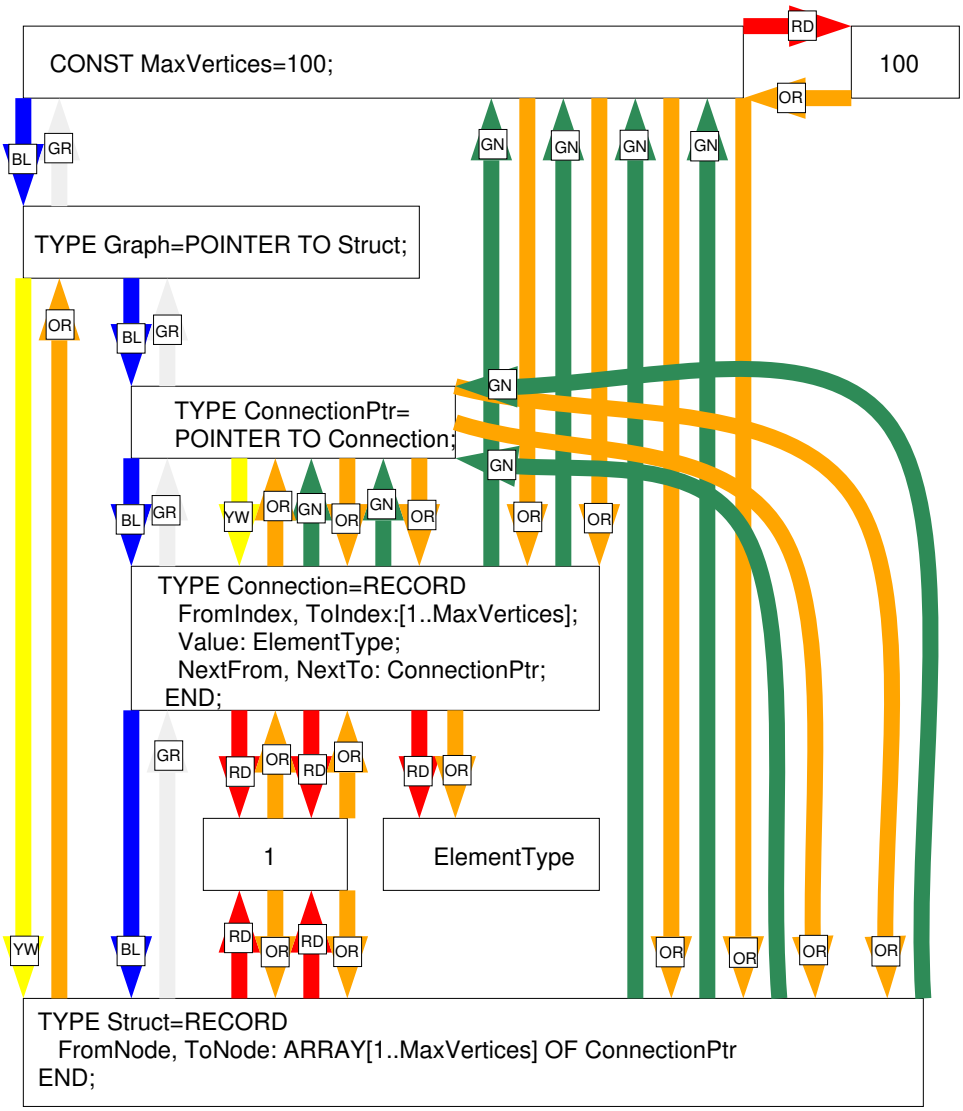


Figure 8.12: Graph for Example 12.

The high Structural metric values are due to the nesting of structures with high Structural metric complexity within records. All three examples include array declarations within records which raises the complexity since the index and base type complexities are multiplied by not only the array constant $C_a = 2$ but also the record constant $C_r = 2$.

Example 10 includes a two dimensional array plus a single dimension array within a record with Structural complexities of 10 and 6 respectively. Since they are declared within the record, the sum of their complexities is multiplied by the record constant, 2, resulting in 32. The complexity of this example could be greatly reduced by declaring types for the arrays outside of the record. The two declarations would add complexity 18 but would reduce the complexity of the two fields within the record to 4. The total complexity would be 36.

Similarly, Example 11 uses an array withing a record structure that could be declared outside of the record. This example also has two fields that are ranges which add 8 to the total complexity.

Example 12 has a nested array but also has extra complexity due to ConnectionPtr. This pointer complexity is also reflected in the Connection and layout metrics.

8.2 Properties

In this section we consider our metrics in light of nine properties that have been proposed for all software metrics[Weyuker,1988]. The ordering of properties is as presented in [Cher-

niavsky,1991].

In what follows, we use Weyuker's notation. The operation ; is the sequencing operator. That is, if P and Q are programs, $P;Q$ represents the code of P followed by the code of Q. Also, $P = Q$ if programs P and Q are syntactically identical. Both can be used unchanged for data structures.

Weyuker uses $P \equiv Q$ to denote that P and Q are equivalent. Two programs are equivalent if they halt on the same inputs with the same values for outputs. Since this definition of equivalency depends on evaluation of statements and halting, it does not apply to code units consisting exclusively of data structures. However, if P and Q are complete program units (a program, module, or routine), the definition can be applied.

In the discussion below, P and Q may include both declarations and executable statements but the metric is applied only to the declaration portions. Alternately, one can view the domain of the metrics extended from declarations by defining that all other statements have complexity 0.

8.2.1 Property 1

Property 1 states that there are at least two programs with different measures. That is, the measure cannot be completely trivial and give the same value for all programs. It is expressed as $(\exists P)(\exists Q)(\mid P \mid \neq \mid Q \mid)$.

Property 1 holds for all of the Data Graph based metrics. Let P and Q be defined as shown below.

```
P:
  VAR I: INTEGER
Q:
  CONST MaxStr = 100;
  TYPE Str = ARRAY[1..MaxStr] OF CHAR;
  VAR S: Str;
```

For the Size metric, $\mid P \mid = 1$ and $\mid Q \mid = 3$ so $\mid P \mid \neq \mid Q \mid$. Hence the Size metric satisfies this property.

For the Connection metric, $\mid P \mid = 1$ while $\mid Q \mid = 4$ so the Connection metric satisfies Property 1.

For the Layout metric, $\mid P \mid = 1$ while $\mid Q \mid = 4$, so P and Q satisfy the property for the Layout metric.

Computing the Volume metric for P and Q give 1 and 4 respectively, again satisfying Property 1.

For the Structural data metric, $\mid P \mid = 2$ while $\mid Q \mid = 8$ so property 1 is satisfied.

8.2.2 Property 2

Property 2 requires that there are no arbitrarily long programs of fixed measure assuming there are a finite number of identifiers available. This property is stated as "For C any nonnegative number, there are only finitely many programs of complexity C."

If complete programs are used, no metric computed using only data declarations can satisfy this property. To see this, consider the program below that contains an assignment statement with no side effects.

```
VAR X, sum: INTEGER;
BEGIN
  sum := 25;
  WRITELN("Sum = ",sum);
END
```

Without changing any declarations, we can duplicate the assignment statement as many times as we wish and still have a functioning program with the same data structure complexity. For $0 \leq i$, let P_i be the above program with the assignment statement replaced with i copies of itself. Any metric that depends only upon data structures will assign the same complexity value to each P_i. So there are an infinite number of programs with the same data structure complexity as the original, violating property 2.

One might object that the example above is contrived because all of the programs generate the same output. It is not difficult to construct an infinite family of programs which must

be assigned the same value by data structure metrics but each program generates a unique output. The code below will be the initial program. To create the next program in the sequence, duplicate the assignment statement adding X to sum. This creates a sequence of programs, all of which compute distinct values and all using the same data structures.

```
VAR X, sum: INTEGER;
BEGIN
  X = 5;
  sum := 0;
  sum := sum + X;
  WRITELN("Sum = ",sum);
END
```

We must conclude this metric is not relevant to data structure metrics.

8.2.3 Property 3

Property 3 requires that there be distinct P and Q such that $\mid P \mid = \mid Q \mid$. That is, although two programs may be different, is is possible for their complexity values to be the same.

All four Data Graph based metrics satisfy property 3. If we can show that two programs generate the same Data Graph, the four metrics will give the same values to the two programs. P and Q defined below both generate the same Data Graph; hence property 3 is satisfied for the graph based metrics.

```
P:
  TYPE Element = CHAR;
  TYPE Node = RECORD
                E: Element;
                I: INTEGER;
              END;
  TYPE S = Node;
Q:
  CONST MaxStr = 100;
  TYPE Str = ARRAY[1..MaxStr] OF CHAR;
  VAR S: Str;
```

For the Structural data metric use Q from above and P given below. Both have complexity 8. So Property 3 is also satisfied by the Structural metric.

```
VAR I: REAL;
VAR J: INTEGER;
VAR K: CHAR;
VAR L: BOOLEAN;
```

8.2.4 Property 4

Property 4 requires that there exist P and Q which are functionally equivalent yet have different measures. That is, $(\exists P)(\exists Q)(P \equiv Q$ and $\mid P \mid \neq \mid Q \mid)$. From the stack examples in Section 8.1 it is clear that equivalent stack modules can be defined with a variety of metric values. In particular, consider two versions using the same DEFINITION module but different IMPLEMENTATION modules, one using a static array and the other using dynamic memory allocation but fixing the depth to the same limit as the static array.

8.2.5 Property 5

$(\forall P)(\forall Q)(\mid P \mid \leq \mid P;Q \mid$ and $\mid Q \mid \leq \mid P;Q \mid$

This is trivially true for the Size metric as the number of blue arcs can never decrease by adding more declarations. Except when P or Q is empty, $\mid P;Q \mid$ will be strictly greater than $\mid P \mid$ and $\mid Q \mid$.

The other three Data Graph based metrics also satisfy this property, although the color of arcs need not be preserved. For example, P and Q below would each have a red arc to a simple type node.

```
P:
  TYPE INTEGER = CHAR;
Q:
```

```
  VAR I: INTEGER;
```

If P and Q are combined into a single declaration, the red arc for the second declaration would be replaced by a green arc to the first declaration. Such reusing of the name of a simple type is allowed in some languages, although most people would agree that it should be avoided. Even if the language does not allow reuse of simple type names, there usually are system names that can be reused.

Such redefinitions always involve converting a red arc to either a green or yellow arc. With the assignment of weights for the metrics, this can never decrease the amount added to the metric value by the arc and for yellow arcs will increase the complexity.

The Volume metric does not strictly satisfy this property. Converting the red arcs to INTEGER to green arcs in Q below decreases the width of Q from 3 to 1 with no change in the height of Q. Therefore $\mid P;Q \mid = 2 \times 4 = 8$ while $\mid Q \mid = 3 \times 4 = 12$. Since $\mid Q \mid > \mid P;Q \mid$, the property is not satisfied.

```
P:
   TYPE INTEGER = CHAR;
Q:
  VAR I: INTEGER;
  VAR J: INTEGER;
  TYPE Step1 = CHAR;
  TYPE Step2 = Step1;
```

```
TYPE Step3 = Step2;
```

It should be noted that this situation only arises in the rare cases when a declaration redefines primitive types such as INTEGER or redefines system names. If restrictions are placed on P and Q to disallow redefining types and system types, the property will hold. Given the semantics associated with primitive types and system names, it is highly unlikely that two programs combined in this manner would compile, much less execute properly. However, it does expose a mechanism by which a programmer could reorganize a program with many references to primitive types so as to lower the program's Volume metric value. If X is the name of a primitive type, the programmer could define a type USERX as TYPE USERX = X; and replace all other occurrences of X by USERX. This will collapse all red pointers to type X to one red pointer while at most increasing the height by 1. If initially there were n red arcs to X and the height was h, the Volume was $n \times h$. After the transformation the Volume is $1 \times (h + 1)$. If n is large, the difference could be significant.

The Structural data metric satisfies this property since $\mid P; Q \mid = \mid P \mid + \mid Q \mid$.

8.2.6 Property 6

Property 6 requires that in some cases adding code of the same complexity either before (or after) a piece of code will give different complexities. Another way of stating this is that there are interactions between pieces of code. Formally, this property is stated in two parts.

$(\exists P)(\exists Q)(\exists R)(\mid P \mid = \mid Q \mid \; and \; \mid P; R \mid \neq \mid Q; R \mid)$

$(\exists P)(\exists Q)(\exists R)(\mid P \mid = \mid Q \mid and \mid R;P \mid \neq \mid R;Q \mid)$

The Size metric does not satisfy this property. Let m be the blue arcs in R. If P and Q have the same complexity, they have the same number of blue arcs n. Adding the graph of P or Q to R will result in graphs with the same number of blue arcs, $m + n + 1$. The extra blue arc is added to connect the last node of the leading code unit to the first node of the following unit. The failure of the Size metric to satisfy this property is no surprise. Since no information about the contents of the data declaration or the other declarations that it may reference are used by Size, any interactions between declarations cannot be captured by the metric.

The other three graph-based metrics satisfy this property. To show the first half for the Connection and Layout metrics, let P, Q, and R be defined as below. Since the Data Graphs are the same, $\mid P \mid = \mid Q \mid$ for both metrics. Also, since there are only two declarations in P;R and Q;R, the distances between the head and tail of an arc used in the Layout metric is always 1. Hence, the Layout and Connection metrics will have the same values. The Data Graph for P;R includes a yellow arc from the the node from the P declaration to the R declaration node and a red node for the R node so $\mid P;R \mid = 3$. The graph for Q;R has no yellow node but has two red nodes, one for Q and one of R, so $\mid Q;R \mid = 2$.

```
P:
  TYPE X = INTEGER;
Q:
  TYPE X = CHAR;
```

```
R:
  TYPE INTEGER = CHAR;
```

To show the second half of the property for the Connection and Layout metrics, let P, Q, and R be defined as below. Clearly, for each of the metrics $| P | = | Q |$. Yet, $| R;P | \neq | R;Q |$ because the first introduces a yellow arc with greater weight than red arc it replaces while the second retains the red arc. Of course, there may be languages in which the reuse of a name may not be allowed after it has been used within the scope but the metric does not impose any such restriction.

```
P:
  TYPE INTEGER = CHAR;
Q:
  TYPE X = CHAR;
R:
  TYPE S = INTEGER;
```

For the Volume metric, let P, Q, and R be defined as below. Clearly, $| P | = | Q | = 1$. Yet, $| P;R | = 4$ while $| Q;R | = 8$.

```
P:
  TYPE INTEGER = CHAR;
Q:
  TYPE X = CHAR;
```

```
R:
  VAR I: INTEGER;
  TYPE Step1 = CHAR;
  TYPE Step2 = Step1;
  TYPE Step3 = Step2;
  TYPE Step4 = Step3;
```

The same example works if R precedes P and Q.

The Structural data metric does satisfy this property for the P, Q, and R defined below.

```
P:
  TYPE INTEGER = RECORD
                   X: POINTER TO CHAR;
                 END;
Q:
  TYPE AChar = RECORD
                   X: POINTER TO CHAR;
                 END;
R:
  TYPE CHAR = RECORD
                   X: POINTER TO INTEGER;
                 END;
```

As individual data declarations, P, Q, and R have the same structure and would each be assigned a complexity value of 11 by the Structural data metric. Since Q and R do not share any type names, $\mid Q;R \mid = \mid Q \mid + \mid R \mid = 11 + 11 = 22$. On the other hand, P and R are mutually recursive so the circular reference weight is applied in evaluating the POINTER fields in each. The result is that both P and R have complexity 21 and $\mid P;R \mid = \mid P \mid + \mid R \mid = 21 + 21 = 42$ so $\mid P;R \mid \neq \mid Q;R \mid$. It should be noted that $\mid P;R \mid = \mid R;P \mid$ and $\mid Q;R \mid = \mid R;Q \mid$ for this example also shows $\mid R;P \mid \neq \mid R;Q \mid$.

8.2.7 Property 7

Property 7 requires that the order of some statements is important. Formally, property 7 is $(\exists P)(\exists Q)$, Q is a permutation of the statements in P, and $\mid P \mid \neq \mid Q \mid$

Only green and yellow arcs capture directional information so only the metrics that use such arcs satisfy this property. Because the Size metric uses only blue arcs, it does not depend on the ordering of the statements.

The Volume metric satisfies this property. Consider P and Q below which are identical except for the location of the X: POINTER TO B statement. In both cases, the only leaf node is for INTEGER and there will be 3 red arcs to it so the width for P and Q is 3. In P the maximum height is 3 while Q has a maximum height of 2, so $\mid P \mid = 3 \times 3 = 9$ while $\mid Q \mid = 2 \times 3 = 6$.

```
P:
```

```
  TYPE A = RECORD
             I: INTEGER;
             X: POINTER TO B;
           END;
  TYPE B = RECORD
             I: INTEGER;
             Y: POINTER TO C;
           END;
  TYPE C = RECORD
             I: INTEGER;
           END;
Q:
  TYPE A = RECORD
             I: INTEGER;
           END;
  TYPE B = RECORD
             I: INTEGER;
             X: POINTER TO B;
             Y: POINTER TO C;
           END;
  TYPE C = RECORD
             I: INTEGER;
           END;
```

For the Connection and Layout metrics it is easy to see that moving an item at the head of a green arc to after the node at the tail of the arc will convert a green arc to a yellow arc, resulting in an increase in the complexity value. P and Q defined below have the same statements but a different ordering. Since there are only two declarations, the lengths computed for the Layout metric will always be 1. Hence, the Layout and Connection metrics will give the same value on P and the same value on Q. Both have one red arc but P has a green arc while Q has a yellow arc. Thus, $\mid P \mid = 2 \neq 3 = \mid Q \mid$.

```
P:
  TYPE S = INTEGER;
  TYPE T = S;
Q:
  TYPE T = S;
  TYPE S = INTEGER;
```

P and Q below show that the Structural data metric satisfies this property. They are identical except for the location of the statement Y: POINTER TO A. Yet, the Structural metric is different for the two programs. P involves two mutually recursive types, A and C, and gives a metric value of 59. In Q, TYPE B references TYPE A and TYPE A references TYPE C, but no mutual recursion is involved resulting in a complexity of 35.

```
P:
  TYPE A = RECORD
             I: INTEGER;
```

```
            X: POINTER TO C;
          END;
  TYPE B = RECORD
            J: INTEGER;
          END;
  TYPE C = RECORD
            K: INTEGER;
            Y: POINTER TO A;
          END;
Q:
  TYPE A = RECORD
            I: INTEGER;
            X: POINTER TO C;
          END;
  TYPE B = RECORD
            J: INTEGER;
            Y: POINTER TO A;
          END;
  TYPE C = RECORD
            K: INTEGER;
          END;
```

8.2.8 Property 8

Property 8 requires that if variable names are systematically replaced in a program, then the resulting program should have the same complexity as the original. That is, $(\forall P)(\forall Q)$, if P is a renaming of Q, then $\mid P \mid = \mid Q \mid$.

All of the metrics satisfy this property. Variable names are not used in the metrics so the metric values will not change if variable names are changed.

8.2.9 Property 9

Property 9 allows a program to be more complex than its parts.

$(\exists P)(\exists Q)(\mid P \mid + \mid Q \mid < \mid P;Q \mid)$

The Size metric does not satisfy this property. Consider and P and Q. Let n be the number of blue arcs in the graph for P and m be the number of blue arcs in Q. The Size metric gives $\mid P \mid = n + 1$ and $\mid Q \mid = m + 1$ resulting in $\mid P \mid + \mid Q \mid = n + m + 2$. Now consider the graph of P;Q. It contains all of the blue arcs from P and Q plus an arc connecting the last node in the graph of P to the first node in the graph of Q so there are $n + m + 1$ blue arcs. Hence, $\mid P;Q \mid = n + m + 2 = \mid P \mid + \mid Q \mid$.

The Layout and Connection metrics satisfy this property. Consider P and Q defined below. For both metrics, $\mid P \mid = 1$ and $\mid Q \mid = 1$ so $\mid P \mid + \mid Q \mid = 2$. When the graph for P;Q is generated, there is one red arc and one yellow arc so $\mid P;Q \mid = 3$.

```
P:
  TYPE INTEGER = CHAR;
Q:
  VAR I: INTEGER;
```

For the Volume metric, consider P and Q such that one has a large width but low height and the other has a small width but large height. Under these conditions, P;Q will exceed the sum of the two. P and Q below are examples that work. P has height 1 and width 8 while Q has width 1 and height 8. Both have a Volume of 8 so $\mid P \mid + \mid Q \mid = 16$. The width of P;Q is 7 and the height is 8 so the Volume is 56.

```
P:
  VAR I: INTEGER;
  VAR J: INTEGER;
  VAR K: INTEGER;
  VAR L: INTEGER;
  VAR M: INTEGER;
  VAR N: INTEGER;
  VAR O: INTEGER;
  VAR P: INTEGER;
Q:
  TYPE I = INTEGER;
  TYPE J = I;
```

```
  TYPE K = J;

  TYPE L = K;

  TYPE M = L;

  TYPE N = M;

  TYPE O = N;

  TYPE P = O;
```

The Structural data metric satisfies this property. For P and Q below, $\mid P \mid = \mid Q \mid = 11$. But P;Q involves mutual recursion $\mid P;Q \mid = 46$. Since $\mid P \mid + \mid Q \mid = 11 < 46 = \mid P;Q \mid$, property 9 is satisfied.

```
P:
  TYPE INTEGER = RECORD
                   X: POINTER TO CHAR;
                 END;
Q:
  TYPE CHAR = RECORD
                   X: POINTER TO INTEGER;
                 END;
```

8.3 Summary

In this chapter we have shown how our metrics can be evaluated for sample programs. Also, the metrics have been evaluated on a set of properties proposed for software metrics. The results are summarized in Table 8.5.

Property	1	2	3	4	5	6	7	8	9
Metric									
Size	yes	no	yes	yes	yes	no	no	yes	no
Connection	yes	no	yes	yes	yes	yes	yes	yes	yes
Layout	yes	no	yes	yes	yes	yes	yes	yes	yes
Volume	yes	no	yes	yes	yes	yes	yes	yes	yes
Structural	yes	no	yes	yes	yes	yes	yes	yes	yes

Table 8.5: Property Summary

Property 2 is satisfied by none of the metrics since no metric based solely on declarations and accepting complete programs as input can satisfy property 2. This is not a concern as we would not expect the data structures of programs to capture all of the complexity in a program. If we restrict the input to declarations, property 2 would hold for the Structural metric. Of the graph-based metrics, Layout, Connection, and Volume metrics satisfy property 2 when inputs are restricted to data structures.

To see that Size does not satisfy property 2 even when the input is restricted to data structures, consider the data declaration DS below. We can create a declaration, D_i, of any length by replacing X: POINTER TO DS by i copies of itself. Since no new blue arcs are created when a new copy of the field is added, the Size metric will be the same for all D_i.

```
TYPE DS = RECORD
            X: POINTER TO DS;
          END;
```

Properties 6, 7, and 9 are contextual properties. Size simply counts the number of data declarations so it cannot take contextual relationships between nodes into account. Hence, it cannot satisfy the three properties.

Chapter 9

A Mechanism for Practical Utilization of Multiple Metrics

In this thesis multiple metrics have been presented for data structure complexity. One might ask whether the different metrics can be replaced by one metric that captures all aspects of data structure complexity? Some results [Fenton, 1994] on program complexity metrics in general based on measure theory indicate that no single metric can be expected to capture all of complexity. Since data structure complexity has many aspects, the metric or metrics one should use will depend on the goals of the analysis and on the questions one is trying to answer [Basili,1988].

When several metrics are used to compare programs, it becomes difficult to evaluate which of

several programs is the most complex. Providing a quality assurance individual or manager a list of numeric values is not very useful for a quick determination of complexity. For example, it is not easy to find the program that is different from all others in Table 9.1.

But look now at Figure 9.1 in which the data in Table 9.1 are plotted as polygons constructed with the distance from a fixed point to a vertex equal to one of the metric values. In this example, the top vertex corresponds to metric 1, the right vertex corresponds to metric 2, the bottom vertex corresponds to metric 3, and the left vertex corresponds to the matrix 4. One can easily observe that most of the items form diamond shapes with the long axis in the vertical direction. Only item 7 is displayed as a diamond with the long axis in the horizontal direction.

Metric	1	2	3	4
Pgm				
1	100	60	100	55
2	50	10	80	20
3	15	5	20	6
4	30	10	40	10
5	95	50	86	45
6	82	41	76	20
7	30	60	32	55
8	76	17	69	23
9	26	3	28	5
10	92	45	83	35

Table 9.1: Different Program.

Figure 9.1 illustrates a general approach to help people evaluate metrics by providing graphical views of the metric values. An Athena widget X-based tool called XFace [Curtis,1992] [Scarfone,1992] was developed at SUNY Buffalo to provide such views for metrics for any multivariant data. The tool is based on displaying faces with values mapped to features.

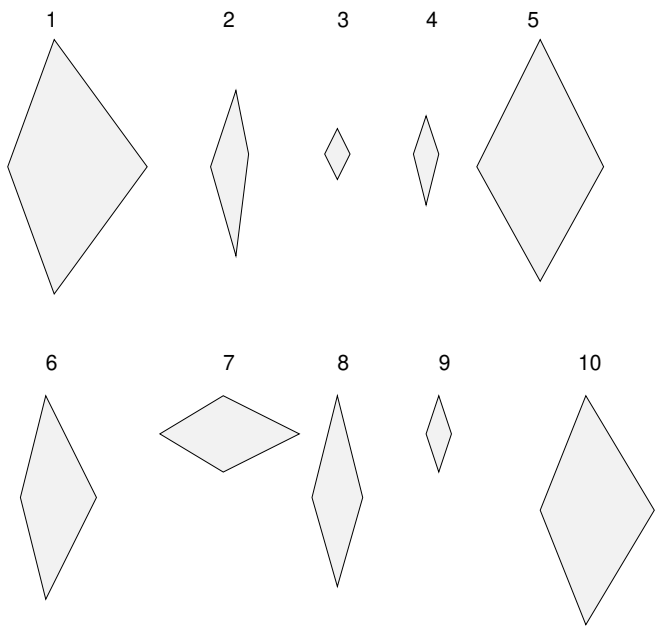


Figure 9.1: Graphical Display of Different Program.

The basic concept is that people have developed the ability to recognize quickly faces that are similar. The use of faces was first proposed by Herman Chernoff [Chernoff, 1973]. He believed it is easier to get a gestalt based upon faces. In presenting multivariant data, such as metrics, the data can be mapped to specific facial features and the human visual facilities can be utilized to quickly determine "good" and "bad" or similar faces. Newer faces [Flury, 1981] were developed so that each feature controls a limited portion of the face. This limits the dramatic changes that could be seen in Chernoff faces. Figure 9.2 shows a set of faces derived from six metrics describing fourteen programming projects. The labels, starting with "prj" in this example, are assigned by the user and are useful for associating the printed face with the original metric data.

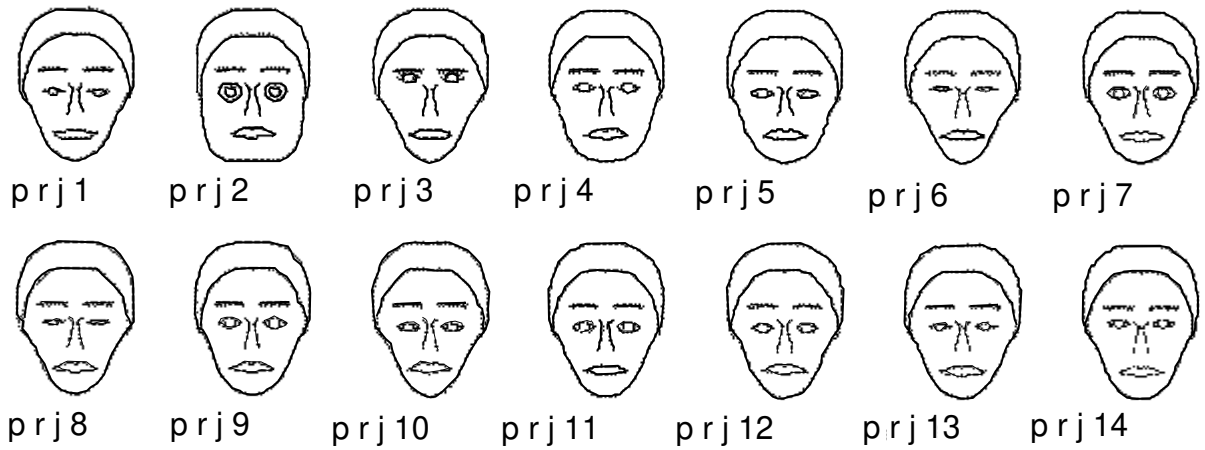


Figure 9.2: Set of Faces.

The user may select which values are mapped to specific facial features. The eighteen features available are eye size, pupil size, position of pupil, eye slant, horizontal position of eye, vertical position of eye, curvature of eyebrow, density of eyebrow, horizontal position of eyebrow, vertical position of eyebrow, upper hair line, lower hair line, face line, darkness of hair, hair shading slant, nose, size of mouth, and curvature of mouth. Normally 18 features are sufficient and symmetric faces are used. However, it is possible to assign 36 different data values by mapping different data values to the left and right sides resulting in asymmetric faces.

Each feature has a level of salience associated with with it of *high*, *mid*, *low*, or *unknown*. The *unknown* items were added after the salience studies[De Soete, 1986]. High salience items should be assigned to metrics that are considered of great importance and lower salience features to secondary metrics. The high salience features are eye size, vertical position of eye, density of eyebrow, face line, and curvature of mouth.

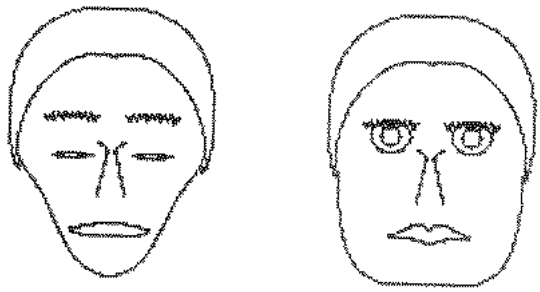


Figure 9.3: Salient Features - Low and High.

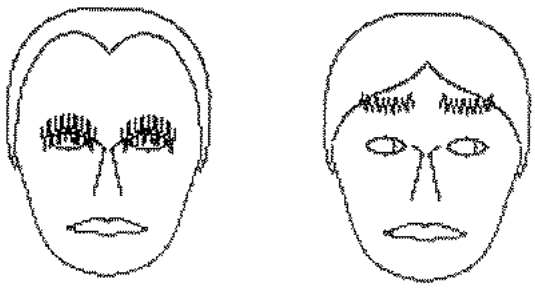


Figure 9.4: Low Salience Features - Low and High.

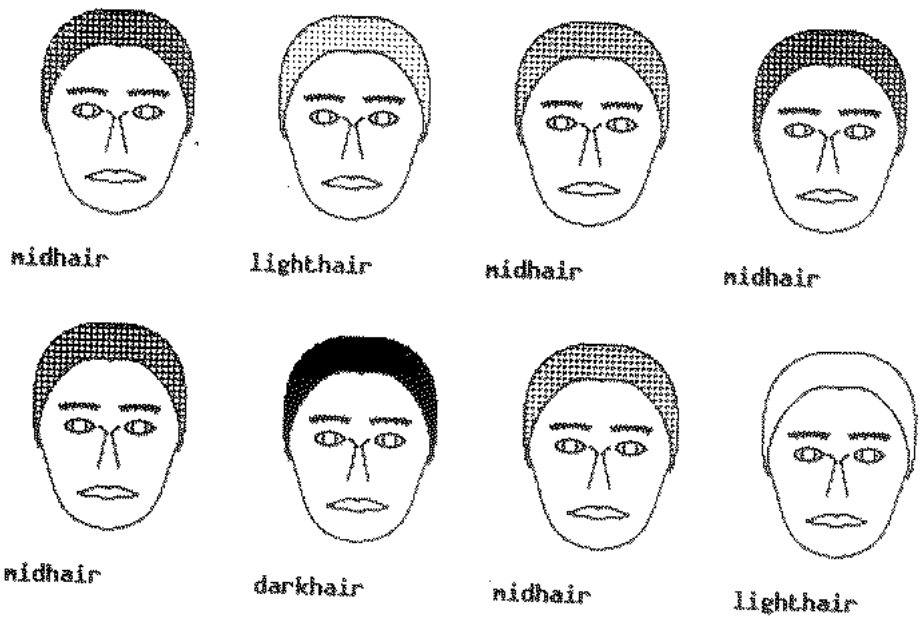


Figure 9.5: Hair Color Example.

Hair darkness was not included in the original salience study but informal observation with the tool indicates that it has very high salience. In fact, it has such a strong visual effect that it overpowers other features. While the other features are continuous, the hair color feature is discrete valued (10 choices). A small change in a value can dramatically change the hair color. For example, in Figure 9.5 the dark hair face stands out in the crowd although the difference between the value mapped to hair for the midhair face is close to the darkhair face value.

If the user wishes to view two items next to each other and there are 18 or fewer values, the values for the items can be mapped to different sides of a single face. For example, in Figure 9.6 values of one project are displayed as features on the left side of the face and

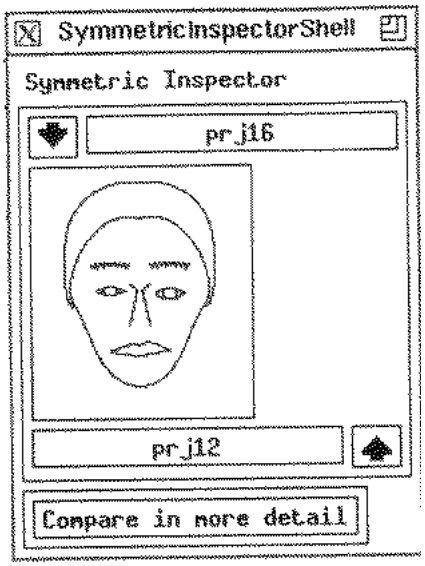


Figure 9.6: Symmetric Inspector.

values of a second project are displayed on the right side. This allows the user to inspect the two visually for differences. Furthermore, the user can select the "Compare in more detail" button and the numeric values are displayed in a Detail Shell as shown in Figure 9.7. An asymmetric face displayer is also available that allows multiple faces to be placed next to each other for viewing.

Currently, another student is incorporating the XFace display into a package with other methods of displaying multivariant data. The new tool should support glyphs, daises, stars, scatter diagrams, and several forms of histogram shown in Figure 9.8. The difference histogram may be particular useful. When two items are identical, only the horizontal separator line is displayed which is easily detected. Items with very similar metric values will have vary small area in the bars while if two items are very different the area in the histogram will be large.

Symmetric Detail Shell

	Values		Normalized Values	
LOC	50	62	0.152542	0.355932
NPath	1950	610	0.196816	0.058301
Prather	130	177	0.156863	0.259259
PratherP	394	297	0.239357	0.161446
McCabe	17	22	0.428571	0.785714

Figure 9.7: Inspector Detail Display.

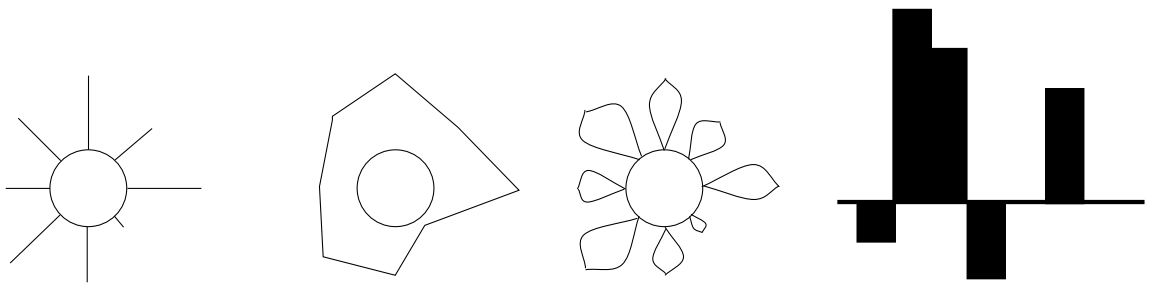


Figure 9.8: Future Display Forms.

There are many other possible applications for such a tool in research, teaching, and in industry. In research the tool could be detect interesting cases for detailed study. It could also be used to compare two sets of metrics. One set could be mapped to the left side of the face while the second set is mapped to the right side.

A teacher could use the tool to quickly detect student programs that are "different" according to some set of metrics. Of course, a more detailed look would be required to determine if the difference is good or bad. For example, a student program could be judged more complex by the metrics due to extensive error handling or extensions to the project. Unless the project specification excludes such things, a teacher would not wish to penalize the student for doing more. The teacher might also wish to quickly compare student programs against a standard reference program. Finally, a teacher could use the tool as a first level detection of programs to review for possible cheating. Often, class size does not allow comparison of all programs in pairs to detect cheating. Also, different sections of the same course with different graders may have the same project. Although the tool cannot detect cheating, it can greatly reduce the search space if students submit source code in electronic form. After a set of metrics is computed for each program, the tool can display the results. One person can then display the faces. The faces can then be partitioned into groups of similar faces. Pairs of faces may then be placed next to each other in the symmetric inspector. A Details Shell can be opened to look at the metric values. If there is a good match, the graders must then look at the programs involved to determine if, in their judgement, there was cheating. Since more than one program can usually generate the same metric value, identical metric values need not imply cheating. However, if the metrics are appropriately selected, two

programs with very different faces and metric values need not be checked for cheating.

A manager in industry might use the tool to compare the modules. Modules in a project might be compared against a standard module. If a module is very different than the standard, it should be investigated. A module that is very complex compared with the standard might need to be split into smaller functional units or watched carefully for possible problems. A module with very low metric values may be looked at to see if is should be consumed in another module. The tool could also be used in module change analysis. Each face might correspond to metric values of the module at different times and ordered according to time. By scanning the faces the user may notice that a module is becoming very complex. One of the metric values might be the rate of bug detection for the module. If the module is increasing in complexity and there is a trend for the bug rate to increase, maybe more effort will be placed on reducing its complexity instead of continuing with bug corrections.

Returning to Figure 9.1, the viewer could identify item 7 as significantly different from the others. The importance of the difference depends upon the metrics assigned to the vertices. If metric 1 is the number of declarations, metric 2 is the number of loop statements, metric 3 is the number of assignment statements, and metric 4 is the number of selection statements, the shape clearly indicates that item 7 has a different basic structure than the other items. It is primarily constructed of loops and selections with few declarations and assignments while the other items are dominated by declarations and assignment statements. Difference is not always bad. In the metric assignments stated above, it might be the best program. Suppose each of the other programs declares a large number of integer variables such as A1,

A2, and A3 and then performs individual assignments. Item 7 may declare an array and use a loop to perform a single assignment statement with different array elements.

The viewer could also use Figure 9.1 to identify the "largest" items and "smallest" items. In this example items 1, 5, and 10 might be reviewed if they are larger than an established standard or estimated final size. It is possible that the original system design was poor resulting in excessive code in item 1. Or maybe item 5 was assigned to an inexperienced person who has placed excessive data structures in the code.

The display tool does not directly identify the problem. With the appropriate data and mapping of metrics to the display, it can be used to detect possible items of interest.

Similarly, items 3, 4, and 9 might be investigated if they are below established limits or expectations for the particular module. Of course, it is also possible that the programmers working on it have hit upon a very efficient implementation. The display is used to identify modules that *may* have problems.

Suppose the items are 10 modules in a system under development that are expected to be of approximately the same complexity when finished. Projects are sometimes divided such that each programmer or group is expected to have approximately the same amount of work. If a company is well into a project it could have a sequence of metric displays which it could use to keep managers informed on the progress to date with the system. In this case item 3 is small which may indicate that item 3 is "lagging behind" so the manager would investigate to see if there is a problem. It may be that a problem has developed and attention needs to be focused on that module. Possibly the programmers did not understand

the specification. Of course it is always possible that the estimate developed for item 3 was too high. We cannot tell which is the case from the tool, only that it is something that is different from expectations and so needs to be investigated.

In summary, visual interfaces provide a mechanism for quick review of multiple metric values for a collection of programs or modules. By using the ability of the human visual system to detect quickly differences and similarities between visual objects, an initial analysis can be performed on a large set of metric values to determine items that warrant detailed investigation.

Chapter 10

Conclusions and Further Work

In this chapter, we summarize our conclusions about the data structure metrics presented here, and discuss future work for carrying forward the research presented in this thesis.

10.1 Conclusions

Different aspects of data structure complexity are captured by the metrics developed in this dissertation. The Structural metric, based on the axioms developed, captures complexity due to the type of declaration, the nesting of declarations, and the interaction of pointer fields. Each construct in the language is covered by a formula specifying how its complexity is computed. The graph-based metrics use the arcs of the Data Graph to capture differ-

ent aspects of the complexity. Since the Data Graph does not contain the details of the individual data declarations, the graph metrics capture relationships between declarations.

Complexity values of composite structures in the Structural metric are based on multiplying a constant for the structure times the complexity of the declarations within the structure. For example, the complexity of an ARRAY declaration with index complexity I and base type complexity B is defined as $C_a \times (I + B)$ with C_a a parameter of the metric. A critical aspect of the metric is assigning values to the multiplicative constants. An initial assignment based on the experiment and other considerations appears under the Original Value column in Table 10.1.

Even with the initial assignment of constants, a strong correlation between the Structural metric's values and the complexity values provided by humans was observed. However, actual values from the Structural metric were consistently high.

Experimental results indicated that some of the initial values assigned to the parameters of the Structural data structure may be high. To reduce the structural metric values, values of 1.3 for the record constant, C_r and 3 for the recursive record constant are suggested.

The Data Graph based metrics capture different aspects of complexity by evaluating different sets of arcs, as shown in Table 10.2. The Size metric uses the blue arcs to capture the length in number of declarations. The Connection metric captures the interconnections between declarations. It takes into account arcs to primitive and system types (red arcs) as well as reference arcs between declarations. It does take into account the direction of a reference (yellow arcs for forward references, green arcs for backward references, and brown

Parameter	Data Structure	Original Value	Adjusted Value
C_a	ARRAY	2	2
C_c	CASE	2	2
C_e	EXPORT	8	8
C_p	POINTER	4	4
C_P	PROCEDURE	2	2
C_r	RECORD	2	1.3
C_{rr}	RECURSIVE FIELD	5	3
C_s	SET	2	2

Table 10.1: Structural Metric.

arcs for self-references) but does not utilize "distance" between references and declarations. The Layout metric uses the arcs of the Connection metric and it uses the blue and gray arcs to incorporate the distance between a reference and the corresponding declaration. Finally, the Volume metric uses the red arcs to compute the width and the orange and brown arcs to compute a height.

Metric	Green	Red	Brown	Yellow	Blue	Gray	Orange
Size	no	no	no	no	yes	no	no
Connection	yes	yes	yes	yes	no	no	no
Layout	yes	yes	yes	yes	yes	yes	no
Volume	no	yes	yes	no	no	no	yes

Table 10.2: Arcs Used by Graph Metrics.

The Connection and Layout metrics use weights for the different color arcs. The weight assignments appear in Table 10.3. A higher weight penalizes data structures with graphs using that color arc. The weights in the table assign low weights for using primitive and system supplied types and backward references (that is, a type is referenced only after it is declared). The intuition is that readers of the program already know the type at the point of reference either because they have read an earlier declaration or because it is a global type

common to all programs. Self references, represented by brown arcs, are given a slightly higher weight. Finally, forward references are given the highest weight because readers must scan forward into unread portions of the program to understand the declaration fully.

Metric	Green	Red	Brown	Yellow
Connection	1	1	2	4
Layout	1	1	2	4

Table 10.3: Arc Weights.

Analysis of the metrics against a set of properties (see Chapter 8, section 8.2) proposed for comparing metrics, summarized in Table 10.4, indicates that most of the properties are satisfied. The Size metric fails to satisfy properties that require a metric to produce different values for a construct in different contexts. All of the data structure metrics fail the property that requires that there be only a finite number of programs assigned the same complexity value. Since data structure metrics ignore the control structures in a program, it is possible to generate an infinite number of programs which use the same declarations. This property cannot hold for any data structure metric.

Property	1	2	3	4	5	6	7	8	9
Metric									
Size	yes	no	yes	yes	yes	no	no	yes	no
Connection	yes	no	yes	yes	yes	yes	yes	yes	yes
Layout	yes	no	yes	yes	yes	yes	yes	yes	yes
Volume	yes	no	yes	yes	yes	yes	yes	yes	yes
Structural	yes	no	yes	yes	yes	yes	yes	yes	yes

Table 10.4: Property Summary

This work has produced data structure axioms and a set of software metrics for use in understanding the complexity of programs. Since the metrics capture different aspects of data structure complexity, it is possible to select a subset that captures aspects of interest

to a particular application. Each metric was checked against a standard set of properties for complexity metrics, providing information on the behavior of the metrics. A set of textbook examples were used to give examples of metric evaluation and to show how they may be used to improve the declarations.

Although our survey results show that the Structural metric orders declarations in the same general order as human participants, it appears to increase complexity rapidly while humans assign a modest increase. The nesting of structures does not result in a great increase in complexity by readers. Multiplying by constants 2 and greater forces the Structural metric to return much higher values as the nesting increases. To avoid the increase, either constants 1 or greater but less than 2 must be used or a change to the metric, such as adding instead of multiplying the constant, is required. Both changes would be consistent with the original axioms. The remaining metrics remain untested against human assignments of complexity.

The graph-based metrics introduce the ability to focus on certain aspects of complexity. For example, by using the Connection and Layout metrics, it is possible to investigate how different orderings of the declaration change the complexity. Although we have looked at examples and some properties for these metrics, experiments are needed to evaluate their utility in evaluating data structures.

10.2 Future Work

To continue the research started in this thesis, experiments need to be performed and other types of programming languages investigated.

10.2.1 Experiments

More experiments are needed to evaluate the metrics. A survey similar to the one developed for the Structural metric would allow further investigation of the characteristics of each metric. In particular, experiments are needed to investigate how the graph-based metrics related to reader evaluations of complexity.

Experiments also need to be performed to establish which of the metrics or which subsets are most useful. Such experiments would require constructing situations in which data structure metrics could identify important conditions.

Experiments need to be performed on larger programs. The experiment and survey in this thesis involved small data structure samples but future work should include both larger program units and multiple units implementing a program. Small items are needed to probe perception of specific data structures. Larger programs allow more of the data structure interactions to be probed and better approximate "real life" programming activity.

Another interesting extension would involve performing experiments on data structure portions of design languages.

Experiments also need to be performed to answer some questions that developed during the survey analysis. These include:

- How does the context of presenting an item change the programmer's perception of an item's complexity? We observed that after participants saw complicated pointer structures, they assigned lower complexities to items.
- Should the complexity of an array index be stated as a single constant? That is, are arrays with bounds given as literal constants of the same complexity as ones that use declared constants? Also, are 0 and 1 special since, in the survey, it appears lower bounds of 0 and 1 do not add complexity?
- Do all programmers perceive sets as having high complexity? In our experiment, this was the case. Will this be true with other groups?

10.2.2 Data Structure Metrics in Other Styles of Programming Languages

Future extensions to the work presented in this thesis should include investigating data structures in other programming styles such as object-oriented, functional, and logic.

The metrics presented here should extend to object-oriented languages. The module related portions capture a part of the complexity involved with an object.

Changes to the metrics or additional metrics would be required to capture additional aspects

of object complexity. For example, the existing metrics do not allow for inheritance, an important relationship in object-based systems.

Functional and logic programming languages provide special problems. No explicit type information is provided in these languages, requiring more work by the metric analyzer to determine the structures used. In such languages, the main data structure is the list. Connections between data structures in such languages may be unknown until program execution. For complete analysis, simulation of execution may be required.

In logic programming languages, care is required not to apply blindly the existing metrics. For example, treating the facts in Prolog as procedures would be incorrect. A large number of simple facts should not greatly increase the data structure complexity of a program. For example, in the segment below the facts related to HeightOf should not be treated as individual items but as instances in the HeightOf database.

```
HeightOf("Joe",72).
HeightOf("Pam",60).
HeightOf("Dave",64).
HeightOf("Mary",66).
```

In terms of the metrics presented in this thesis, HeightOf should be treated in the same manner as an array. An array of 10 items should be given the same complexity as an array of 1000 items of the same type. Complexity is introduced by the structure of the indexing and base type information.

Similarly, the number of elements is unimportant for simple facts. If the four facts above are replaced by a hundred, the complexity isn't changed. With facts the indexing and base type are combined and interchangeable. For example, HeightOf(Joe,X) can be used to extract 72 while HeightOf(Y,72) can be used to extract "Joe".

A further complication is presented by the flexibility allowed in supplying fact information. The same fact name can be used in multiple ways as shown below.

```
HeightOf("Joe",72).
HeightOf("Pam","Joan",60).
HeightOf("Dave","Tom","Betty",64).
HeightOf("Mary",66).
```

It is also possible for the name to be used as two independent concepts with two different parameter counts or with parameters of different shape. For example, HeightOf might be used for the height of people when the first parameter is a string and for the height of airplanes above the ground when the first parameter is a list including aircraft number and location. In some cases, the uses are disjoint as in this example and would be best treated as two different data structures. In other cases, it may be appropriate to treat them as one type.

Other programming paradigms introduce new issues and will require rethinking of existing metrics and development of new approaches.

Part I

Appendix

Appendix Experiment

Experiment 1

Constants Determination

for a Data Structure Metric

Researcher: Ronald Curtis

SUNY at Buffalo

Introduction:

The SUNY at Buffalo software metrics group is investigating the complexity of source programs in terms of readability and understandability. It is interested in the complexity as perceived by programmers. Earlier research has concentrated on control flow and data flow. One portion of the current research is interested in complexity of data structures.

This experiment's goal is to determine values of six constants that appear in the definition of a locally designed data structures complexity metric. In the experiment you will be given data structures and will be asked your opinion of a constant that determines the complexity of each data structure. The purpose of this experiment is solely to determine the constants and not to evaluate the appropriateness of the data structures. Take the expression as a given. Although I am more than willing to discuss the expressions we use to calculate the complexity of data structures with you after the experiment, for now please concentrate on the constant determination. Feel free to place comments below each item as you are performing the experiment.

In the table below are the expressions we use to calculate the complexity of various data structures. You will be given the appropriate expression for each item. Note that each expression involves a constant multiplier. These are the constants we are trying to determine.

To calculate complexity of	Compute this expression	Constant
ARRAY	AC × (complexity of the index + complexity of the base type)	AC
SET	SC × complexity of the base type	SC
POINTER	PC × complexity of the base type	PC
CASE	CC × sum of complexity of fields	CC
RECORD	RC × sum of complexity of fields	RC
recursive RECORD field	RRC × sum of areas of recursive fields	RRC

The *area* of a recursive field is defined to be the maximal circular reference length times the number of recursive paths. The circular reference length we will call depth and the number of paths will be called width.

For example, in the declaration of Node below, the maximal circular reference is from Next to List type to Node type, for a depth of 2 and there is 1 recursive path for a width of 1. Thus the area is 2×1 or 2.

```
TYPE List = POINTER TO Node;

  Node = RECORD

           Data: INTEGER;

           Next: List;
```

```
        END;
```

In the example below, there are several recursive fields. The field NextDept in DeptNode has depth 2 and width 1 for an area of 2. Manager and Members in DeptNode each have depth 4 (because each points to Employee to EmpNode to Dept to DeptNode) and width 1. So each will have width 4. The total recursive area for DeptNode is $2 + 4 + 4$ or 10.

Similarly, EmpDept in EmpNode has depth 4. It has width 2 because there are 2 paths back to EmpNode - one through Manager and one through Members. Its area is 8. Finally, NextInDept has depth 2 and width 1 for an area of 2.

```
TYPE
  Dept = POINTER TO DeptNode;
  DeptNode = RECORD
               Manager: Employee;
               Members: Employee;
               NextDept: Dept;
             END;

  Employee = POINTER TO EmpNode;
  EmpNode = RECORD
              Name: String; (* plus other info *)
              EmpDept: Dept;
              NextInDept: Employee;
```

```
END;
```

Modula-2 will be used throughout the experiment to display data structure definitions. A knowledge of Modula-2 or Pascal or any other language including pointers is required for this experiment. None of the samples are complex and no reading of complete programs is required.

For each item in the experiment, you will be given the complexity of the item's components such as the complexity of the array index and base, or the area of the recursive pointers. For each of the items in the experiment, please give your best estimate of the entire item's complexity. To normalize the complexity values, we define the complexity of `CONST Pi = 3.14` is 1 and if BaseType is the name of a type, the complexity of `TYPE X:BaseType` is 2. You may specify any number 1 or greater as the complexity of an item.

To make certain you understand the process a few sample items appear on the next page. The experiment supervisor will be present while you work on the samples. Please ask any questions during the samples. During the experiment, the supervisor will not be able to help or give directions.

When you are ready to start the samples, please turn to the next page.

Sample Items:

Sample 1:

```
TYPE
  MySet = SET OF SmallCard;
```

Set complexity is SC × complexity of the base type. The complexity of SmallCard is 1. What complexity do you assign to MySet? ____

Sample 2:

```
TYPE
  MyRecord = RECORD
               X: INTEGER;
               Y: REAL;
             END;
```

Record complexity is RC × the sum of the complexity of the fields in the record. The complexity of `X: INTEGER` is 2 and the complexity of `Y: REAL` is 2 so the fields sum to 4. What complexity do you assign to MyRecord? ____

Please wait until the supervisor has departed before turning the page.

Item 1:

```
TYPE
  AType = ARRAY [1..MaxA] OF INTEGER;
```

ARRAY complexity is AC $\times$ (complexity of the index + complexity of the base type) . The complexity of the index [1..MaxA] is 2 and the complexity of base type INTEGER is 1. What complexity do you assign AType? ___

Item 2:

```
TYPE
  BType = RECORD
            C:CHAR;
            D:CHAR;
          END;
```

RECORD complexity is RC $\times$ sum of the complexity of the fields. In this example, the complexity of each field is 2 so the sum is 4. What complexity do you assign BType? ___

Item 3:

```
TYPE
  FType = POINTER TO INTEGER;
```

The complexity of a pointer type is PC $\times$ complexity of the base type. In this case, the base type INTEGER has complexity 1. What complexity do you assign FType? ___

Item 4:

```
TYPE
  HType = RECORD
            CCode: CARDINAL;
            ICode: INTEGER;
          END;
```

The complexity of a RECORD is RC $\times$ sum of the complexity of the fields in the record. In this case, each field has complexity 2 giving a sum of 4. What complexity do you assign HType? ___

Item 5:

```
TYPE
  SetType = SET OF [1..MaxSetType];
```

The complexity of a Set type is SC $\times$ complexity of the set base type. In this case, the complexity of the base type [1..MaxSetType] is 2. What complexity do you assign SetType? ___

Item 6:

```
TYPE
  JType = POINTER TO JRec;
  JRec = RECORD
           CCode: CARDINAL;
           ICode: CARDINAL;
         END;
```

The complexity of a pointer is PC $\times$ complexity of the base type. The complexity of a record type is RC $\times$ the complexity of the sum of the fields. In this case, the sum is 4. What complexity do you assign JType? ___

Item 7:

```
TYPE
  LClass = (Hourly, Salaried, Commission, Consultant);
  WType = RECORD
            FName, LName, MI: String;
            Address: String;
            IdCode: CARDINAL;
            Category: LClass;
            TelephoneNumber: String;
            Wage: CARDINAL;
            Salary: CARDINAL;
            BaseSalary: CARDINAL;
            CommissionRate: CARDINAL;
            Fee: CARDINAL;
          END;
```

The complexity of a RECORD is RC $\times$ sum of the complexities of the fields in the record. The complexity of each of the 12 fields is 2 for a sum of 24. What complexity do you assign WType? ____

Item 8:

```
TYPE

  GType = ARRAY [1..MaxG] OF BOOLEAN;
```

ARRAY complexity is AC $\times$ the sum of the complexity of the index and the complexity of the base type. The complexity of [1..MaxG] is 2 and the complexity of BOOLEAN is 1. What complexity do you assign GType? ____

Item 9:

```
TYPE
  LClass = (Hourly, Salaried, Commission, Consultant);
  WType = RECORD
            FName, LName, MI: String;
            Address: String;
            IdCode: CARDINAL;
            TelephoneNumber: String;
            CASE Category: JobClass OF
              Hourly: Wage: CARDINAL;
            | Salaried: Salary: CARDINAL;
            | Commission: BaseSalary: CARDINAL;
                         CommissionRate: CARDINAL;
            | Consultant: Fee: CARDINAL;
            END; (* CASE *)
          END;
```

The complexity of a CASE is CC $\times$ sum of the complexities of the fields within the CASE. The Tag field (Category in the above is not included within the case. In the example above, the sum is 10. The complexity of a RECORD is RC $\times$ sum of the complexities of the fields in the record. The complexity of the fields outside the CASE is 14. What complexity do you assign WType? ___

Item 10:

```
TYPE
  FType = POINTER TO FElement;
  FElement = RECORD
               Data: REAL;
               Another: FType;
             END;
```

The complexity of a RECORD is RC $\times$ sum of the complexity of the record's fields. The complexity of the Data field is 2. The complexity of recursive fields (pointers) is RRC $\times$ the area of the pointer field. In this case the complexity of the Another field is 2. What complexity do you assign FElement? ___

Item 11:

```
TYPE
  EType = POINTER TO ERec;
  ERec =RECORD
          ECode: CARDINAL;
          Ptr1, Ptr2: EType;
        END;
```

The complexity of a RECORD is RC $\times$ sum of the complexity of the record's fields. The complexity of the ECode field is 2. The complexity of recursive fields (pointers) is RRC $\times$ area of the pointer fields. In this case the complexity of Ptr1 and Ptr2, the recursive fields, are each 2. What complexity do you assign ERec? ___

Item 12:

```
TYPE
  AType = POINTER TO AElement;
  AElement = RECORD
               AData: CARDINAL;
               PtrA1: BType;
               PtrA2: CType;
             END;
  BType = POINTER TO BElement;
  BElement = RECORD
               BData: REAL;
               PtrB: CType;
             END;
  CType = POINTER TO CElement;
  CElement = RECORD
               CData: String;
               PtrC: DType;
             END;
  DType = POINTER TO DElement;
  DElement = RECORD
               DData: INTEGER;
               Ptr1: AType;
```

```
    Ptr2: BType;
    Ptr3: CType;
END;
```

The complexity of a RECORD is RC $\times$ (sum of the complexities of the nonrecursive fields in the record + RRC $\times$ sum of the areas of the recursive fields). The area is the depth (maximum path length on the type recursion) times the number of distinct paths. In this case, the complexity of each of the Data fields is 2. The area of Ptr1 is 16 (depth is 8 and width is 2), the area of Ptr2 is 6 (depth is 6 and width is 1), and the area of Ptr3 is 4 (depth is 4 and width is 1). What complexity do you assign DElement? ___

BACKGROUND

Please give us some background information on yourself by answering the questions below.

1. What is your position? (circle one)

undergraduate student	graduate student	computer science faculty	research faculty
computer center staff	software engineer	other faculty	other _____

2. With which of the following languages are you very familiar?

Assembler	Basic	Cobol	C
C++	FORTRAN	FP	Lisp
Modula-2	Pascal	Prolog	other _____

3. What are the primary languages you use in your work?

Assembler Basic Cobol C

C++ FORTRAN FP Lisp

Modula-2 Pascal Prolog other

_____

4. How many years ago did you develop your first program? ___

5. How would you rate your programming activity over the last three years?

very light light moderate extensive very extensive

6. How large in lines of code (approximately) was the longest program you have written?

___

7. How would you rate yourself as a software developer?

a beginner some experience a reliable programmer
but not outstanding

better than most one of the best

RELEASE STATEMENT

Now that you have completed the experiment, please sign the authorization section below

so that we can use your estimates in our research. Please detach this page and the final page from the packet. Keep the final page. Please make certain your name does not appear on the experiment sheets. Hand the experiment sheets and this sheet to the supervisor.

I, _____, give permission for Ronald Curtis and the SUNY at Buffalo metrics research group to use my estimates of constants for data structures in their research and to distribute the data to other researchers. It is understood that names will not be stored with the data.

Date: _____

Signature: _____

Print Name: _____

FINAL WORDS

Thank you for participating in this experiment. You may ask the supervisor any questions that you might have. If questions arise after you leave, please contact Ronald Curtis by computer (email to curtis), telephone [(716) 636-3185], in person at 235 Bell Hall, or by letter to Computer Science Department, Bell Hall, SUNY at Buffalo, Buffalo, NY 14260.

The results of this experiment will appear in a Ph.D. thesis that should be out next year. Please remove this sheet and take it with you. Again, our many thanks for help with this research.

Appendix Survey

SURVEY

DATA STRUCTURE COMPLEXITY

A small research group from SUNY at Buffalo studies reader perception of program complexity. My area of interest is in data structures. As part of my Ph. D. work, I am collecting information on how people perceive data structures.

The language used in the survey will be Modula-2. No special features of the language are used and everyone who has a knowledge of Pascal, Fortran, C, or other similar language should have few problems understanding the code.

Only data structures are given so no part requires extensive code reading. The survey should take no more than 20 minutes to complete.
Do not spend large amounts of time on any part. We want your best intuitive feel for the complexity of the data structures.

It would be appreciated if you would fill in the survey below by June 1, 1992 and email it to curtis@cs.Buffalo.EDU. If for some reason you prefer to print it out and return a hard copy, then send it to:

Ronald Curtis

Computer Science Dept.

Bell Hall

SUNY at Buffalo

Buffalo, NY 14260

The results will be made available to other researchers in complexity in a form that includes only the complexity values. Names and email addresses will not be stored in the result files.

In this survey, each part will consist of a data declaration segment. Please give us your best estimate of the segment's complexity.

First, here are a few items to let us establish a base.

A constant declaration such as CONST MaxLine = 80 is taken as complexity 1. Similar, a name of a variable or type, including simple types such as

```
INTEGER, is given complexity 1.


Also, the declaration below has complexity 5.
 TYPE Line = ARRAY[1..MaxLine] OF CHAR
It is computed as follows:
  complexity of introducing the name "Line"    1
  complexity of [1..MaxLine]                   1
  complexity of base type CHAR                 1
  complexity of ARRAY[1..MaxLine] OF CHAR      2*(index type+base type)
                                               = 2*(1+1)
                                               = 4


  final complexity = 1 + 4 = 5


Ignore style issues and naming conventions as much as possible.


Given the above, determine the complexity that you perceive for the
segments below and place the value after the corresponding "?".
Note that each numbered segment should receive one complexity value for
ALL the declarations within the segment.


1. ?
TYPE DA = ARRAY[1..M][1..N] OF INTEGER;
```

2. ?

```
TYPE R = RECORD
   A: INTEGER;
   B: INTEGER;
                END;
```

3. ?

```
TYPE S = SET OF [1..20];
```

4. ?

```
TYPE T = ARRAY[1..2] OF INTEGER;
```

5. ?

```
TYPE A = ARRAY[N..M] OF INTEGER;
```

6. ?

```
TYPE RR = RECORD
    A, B: INTEGER;
    Next: POINTER TO RR;
  END;
```

7. ?

```
TYPE R1 = RECORD
    A: INTEGER;
    NextR2: POINTER TO R2;
  END;

TYPE R2 = RECORD
    A: INTEGER;
    NextR1: POINTER TO R1;
  END;
```

8. ?

```
TYPE L = RECORD
   A, B: INTEGER;
   P1, P2: POINTER TO L;
 END;
```

9. ?

```
        TYPE V = ARRAY[L..K] OF REAL;
```

10. ?

```
TYPE Range1 = [1..N];
```

11. ?

```
TYPE B = ARRAY [1..20] OF BOOLEAN;
```

12. ?

```
TYPE P = RECORD
   A, B: INTEGER;
                END;
TYPE M = ARRAY[1..N] OF P;
```

13. ?

```
TYPE A = RECORD
   X, Y: INTEGER;
   P1: POINTER TO B;
   P2: POINTER TO C;
 END;
TYPE B = RECORD
   I: INTEGER;
 END;
TYPE C = RECORD
   Z: INTEGER;
   J: INTEGER;
 END;
```

14. ?

```
TYPE L = RECORD
   A, B: INTEGER;
   Prev, Next: POINTER TO L;
 END;
```

15. ?

```
TYPE A = RECORD
   X, Y: INTEGER;
   P1: POINTER TO B;
   P2: POINTER TO C;
 END;
TYPE B = RECORD
   I: INTEGER;
   PC: POINTER TO C;
 END;
TYPE C = RECORD
   Z: INTEGER;
   J: INTEGER;
 END;
```

16. ?

```
TYPE W = RECORD
   X, Y: REAL;
```

```
                END;
TYPE L = ARRAY[O..M] OF W;
```

17. ?

```
TYPE A = RECORD
   X, Y: INTEGER;
   P1: POINTER TO B;
   P2: POINTER TO C;
 END;
TYPE B = RECORD
   I: INTEGER;
   P: POINTER TO A;
 END;
TYPE C = RECORD
   Z: INTEGER;
   J: INTEGER;
   P: POINTER TO A;
 END;
```

18. ?

```
TYPE L = RECORD
   A, B: INTEGER;
   Left, Right: POINTER TO L;
```

```
 END;
```

19. ?

```
TYPE A = RECORD
   X, Y: INTEGER;
   P1: POINTER TO B;
   P2: POINTER TO C;
 END;
TYPE B = RECORD
   I: INTEGER;
   P: POINTER TO A;
   PC: POINTER TO C;
 END;
TYPE C = RECORD
   Z: INTEGER;
   J: INTEGER;
   P: POINTER TO A;
   PB: POINTER TO B;
 END;
```

20. ?

```
TYPE A = RECORD
   X, Y: INTEGER;
```

```
   P1: POINTER TO B;
   P2: POINTER TO C;
 END;
TYPE B = RECORD
   I: INTEGER;
   P: POINTER TO D;
 END;
TYPE C = RECORD
   Z: INTEGER;
   J: INTEGER;
   P: POINTER TO D;
 END;
TYPE D = RECORD
                  L: INTEGER;
                  R: INTEGER;
                  P: POINTER TO A;
 END;
```

21. ?

```
TYPE A = RECORD
   X, Y: INTEGER;
   P1: POINTER TO B;
   P2: POINTER TO C;
```

```
 END;
TYPE B = RECORD
   I: INTEGER;
   PC: POINTER TO C;
   PD: POINTER TO D;
 END;
TYPE C = RECORD
   Z: INTEGER;
   J: INTEGER;
   PB: POINTER TO B;
   PD: POINTER TO D;
 END;
TYPE D = RECORD
                   L: INTEGER;
                   R: INTEGER;
                   PB: POINTER TO B;
                   PC: POINTER TO C;
 END;
```

22. ?

```
TYPE A = RECORD
   X, Y: INTEGER;
   P1: POINTER TO B;
```

```
   P2: POINTER TO C;
 END;
TYPE B = RECORD
   I: INTEGER;
   P: POINTER TO C;
 END;
TYPE C = RECORD
   Z: INTEGER;
   J: INTEGER;
   P: POINTER TO D;
 END;
TYPE D = RECORD
                   L: INTEGER;
                   R: INTEGER;
                   P: POINTER TO B;
 END;
```

23. ?

```
TYPE A = RECORD
   X, Y: INTEGER;
   P1: POINTER TO B;
 END;
TYPE B = RECORD
```

```
   I: INTEGER;
 END;
TYPE C = RECORD
   Z: INTEGER;
   J: INTEGER;
   P: POINTER TO B;
 END;
TYPE D = RECORD
                   L: INTEGER;
                   R: INTEGER;
                   P: POINTER TO B;
 END;
```

24. ?

```
TYPE A = RECORD
   X, Y: INTEGER;
   P1: POINTER TO B;
   P2: POINTER TO C;
 END;
TYPE B = RECORD
   I: INTEGER;
   PC: POINTER TO C;
   PD: POINTER TO D;
```

```
 END;
TYPE C = RECORD
   Z: INTEGER;
   J: INTEGER;
   PD: POINTER TO D;
   PA: POINTER TO A:
 END;
TYPE D = RECORD
                   L: INTEGER;
                   R: INTEGER;
                   PA: POINTER TO A;
                   PB: POINTER TO B;
 END;
```

25. ?

```
TYPE W = RECORD
   A, B: INTEGER;
   P1: POINTER TO X;
   P2: POINTER TO Y;
 END;
TYPE X = RECORD
   I: INTEGER;
   P: POINTER TO Z;
```

```
 END;
TYPE Y = RECORD
   C: INTEGER;
   D: INTEGER;
   P: POINTER TO Z;
 END;
TYPE Z = RECORD
                   L: INTEGER;
                   R: INTEGER;
                   P: POINTER TO W;
 END;
```

In advance, thank you for participating in this survey.

Appendix Intermediate Language

An intermediate language for data structures was developed as part of this dissertation in order to reduce the programming effort for implementing tools to compute the structural metric. We only discuss it briefly here because its purpose is technical rather than being a significant contribution in itself. This language simplifies extending metrics to new languages and in implementing new metrics as shown in Figure A3.1. Adding a new language requires adding one new front end accepting the new language as input and generating the intermediate language as output. Often an existing compiler can be modified to act as the front end. Similarly, only one program needs to be written to add a new metric. It must accept the intermediate language as input and output the complexity values. The documentation for the intermediate language appears starting on the next page.

Two front end programs and two back end metric programs have been produced. The front end ends are for Modula-2 [Wang1, 1991] and C [Democko1,1993]. The back end programs are for the Tsai metric [Wang2, 1991] and the structural metric [Democko2,1993].

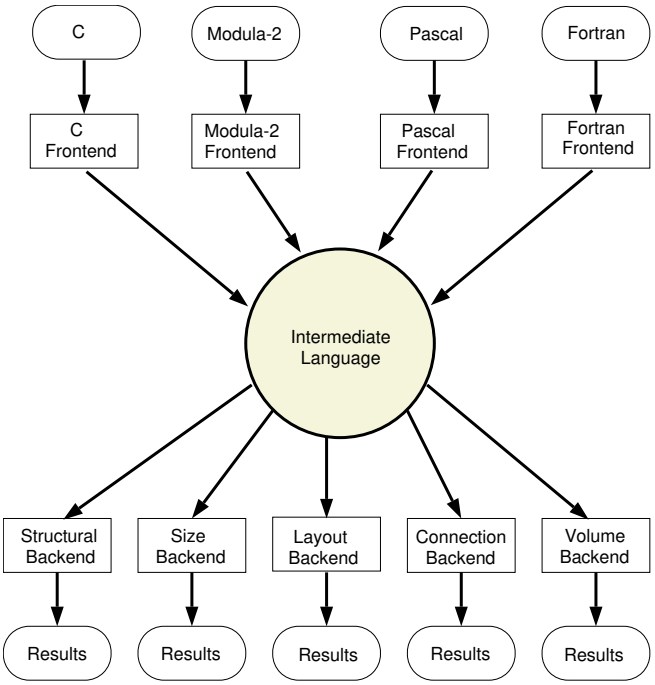


Figure A3.1: Intermediate Language.

Intermediate Language for Data Structure Definitions

Purpose: Provide a source programming language independent form for data structure declarations.

Why:

There are two reasons for creating an intermediate data structure language.

1. It will allow the implementation of software metrics for data structure declarations independent of the source language. This will allow each metric to be programmed once, independent of the number of languages. Adding a metric only requires writing the metric program for the intermediate language. Once that is done, it can generate

results for all the supported languages. A parser that generates the intermediate language will have to be developed for each new language but then all the metrics become available.

2. It allows distribution of the data structure portions of programs without giving away actual source code. This can be important with student programs and commercial software. For example, a bank might be reluctant to release actual sources with variable names and comments. They might be willing to release the structure of the data with all the names removed. A simple program could strip all names (NAME(X,Y) forms) from the intermediate language.

Assumptions:

1. Negative integers are used in NAME (see Intermediate Language Statements below) for items built into a language or system. The assignment of these items to integers is language dependent and determined by the writer of the parser.

2. If X is the string in the LANGUAGE form, then X.lan includes the default names for the language. For example, if LANGUAGE('PASCAL') is in the intermediate form, then PASCAL.lan is expected to contain the predefined Pascal types such as CHAR and INTEGER. Any combination of case can be used. Systems will check in the following order:

   (a) same case as appears in the string with lower case extension (.lan),

   (b) same case as appears in the string with upper case extension (.LAN),

(c) convert string to all upper case and try a and b again,

(d) convert string to all lower case and try a and b again.

The creator of the parser should also create this file.

3. If Y is the parameter string in SYSTEM (see below), then Y.sys includes the default names for the system. This item allows a generic language file with special files including changes for a particular system. For example, a generic Pascal language file might include standard Pascal items. The form SYSTEM('TURBOPASCAL') indicates that the file TURBOPASCAL.sys should include extensions for the Turbo version of Pascal. The SYSTEM specific file is always loaded after the generic language file. The same algorithm described above for LANGUAGE is used to handle case here. The creator of the parser should also create this file.

4. A NULL or 0 (zero) is used for an undefined field. NULL is used for undefined string fields while 0 is used for undefined integer fields.

5. Intermediate language files have an extension of .dst .

6. A Prolog-based language is used for the intermediate language. This simplifies syntax analysis by the metric backends. Also, it will allow quick implementation of metrics in Prolog for testing out new metrics.

7. Starting from 1, ids are assigned sequentially as new items are encountered. The intermediate language writes out what would be compiler symbol table entries in a linear form. The ids replace the pointers used in memory resident tables.

Intermediate Language Token Definitions:

A regular expression language is used to define the tokens in Table A3.1 with braces indicating an item may appear 0 or more times.

Token	Definition	Comment
string	'{character}' "{character}"	use '' (2 single quotes) to get a single quote in the string use "" (2 double quotes) to get a double quote in the string
NULL	'' ""	
integer	{digit}	digit indicates decimal digits
id	integer	

Table A3.1: Token Definitions.

A Pascal-like syntax is used for giving parameter types; parameter names appear followed by a colon (:) and a type.

Intermediate Language Statements:

1. LANGUAGE(languagename:string).

2. SYSTEM(systemname:string).

3. MAX(item:string , value:integer).

   The following strings may be used for the item: 'LINENO','LEVEL', or 'ID'.

4. NAME(name:string , nameid:id).

   This form creates a node for each item referenced in the data structure. For anonymous types, the name will be NULL but a nameid will be assigned.

5. CONST(nameid:id).

6. TYPE(nameid:id , constrid:id).

   The constrid is the id for the constructor used to create the type. If the type is a simple type, the constrid is the id for the type. Otherwise, it will be the id for a SHARESTORAGE, HOMOGENEOUS, or a HETEROGENEOUS form.

7. SHARESTORAGE(shareid:id , nameid:id , nextshareid:id).

   This form is used for structures that share storage such as the C union and the Pascal variant record. The first id is unique in this position to this sharestorage form. An earlier NAME form with a NULL name will have created the id. The nameid is the id of the item this sharestorage represents. The nextshareid is the id of the next name that shares the storage. The SHARESTORAGE entries form a simple linked list. The last item will have 0 as the nextshareid.

8. HOMOGENEOUS(constrid:id , indexid:id , baseid:id).

   The indexid is the id for the first element in a list of dimensions while baseid is the id of the base type.

9. INDEX(indexid:id , rangeid:id , nextindexid:id).

   A range for the index is given. The id may be for an enumeration or range type. The nextindexid item links to the next index or dimension.

10. RANGE(rangeid:id , lowvalue:string , highvalue:string).

    This form specifies a range of values to be used in an index or a range type.

11. HETEROGENEOUS(thisid:id , fieldid:id , nextid:id).

    This form corresponds to records in most languages. The second item is the id of a FIELD form describing this field. The last field is 0 or the id of the next HETEROGENEOUS form.

12. POINTER(pointerid:id , typeid:id).

    The id given as first item is a pointer to items of the type given as the second item.

13. VAR(nameid:id , typeid:id).

    This form relates a variable to its type.

14. ROUTINE(routinenameid:id , parameterid:id , resulttype:id).

    The routine has the parameter indicated by the second parameter. If the routine is a function, the third item gives the type it returns, otherwise it is 0.

15. PARAMETER(parameterid:id , fieldid:id , nextparameterid:id).

    This is a node in a linked list of parameters. The second item gives the id of a FIELD form describing the parameter.

16. FIELD(nameid:id , typeid:id).

    The nameid indicates the name of a field in a heterogeneous structure while the typeid gives the type of the field.

17. ENUM(nameid:id , enumelemid:id).

    The nameid is an enumeration type. The enumelemid is the id of the first ENUMELEM in the enumeration.

18. ENUMELEM(enumelemid:id , elemid:id , nextenumelemid:id).

    The elemid gives the current element in the enumeration. The enumelemid fields are used as pointers creating a simple list of the items in the enumeration.

19. LINENO(lineno:integer).

    Indicates the start of a new source line.

20. LEVELNO(levelno:integer).

    The level is used to indicate static nesting in the source program.

21. START(nameid:id , lineno:integer).

    Indicates that start of a item that may continue over multiple lines. For example, records and modules will have associated starts.

22. END(nameid:id , lineno:integer).

    Each START must be followed by an END with the same nameid.

23. UNIT(nameid:id , typeid:id , unitcompid:id).

    A unit is any portion of code that a program would normally treat as a unit including modules, packages, objects, and tasks. The typeid will be a built in language id. The unitcompid is the id of a UNITCOMPONENT which will contain information on the first component of the unit.

24. UNITCOMPONENT(unitcompid:id , item:id , nextcomponentid:id).

    The item indicated is a component and the nextcomponentid is the id of the next component in this unit. In Modula-2 a unit might be a module and the components

would be the items defined within it. This will include constants, types, variables, and routines.

25. INHERITFROM(itemid:id , parentid:id).

    This unit inherits the first item from the indicated parent. If all items are inherited from the parent, a 0 is used as the first parameter.

26. USES(itemid:id , sourceid:id).

    This form states that the item indicated by its id is used in this program unit. Its source is given as the second parameter.

27. EXPORTS(nameid:id).

    Gives id of an item that the program unit makes available to others.

28. EXTENSIONS(item:string).

    Item is an extension to try if this file uses items from other files. In the case of Modula-2 EXTENSIONS('def') and EXTENSIONS('mod') would be reasonable. The same algorithm is used for handling case as with LANGUAGE and SYSTEM. Normally, EXTENSIONS will be used in LANGUAGE and SYSTEM files.

Part II

References

Aho, A. and Ullman, J. (1972). *The Theory of Parsing, Translation, and Compiling*, Prentice-Hall.

Basili, V. and Turner, A. (1975). "Interactive Enhancement: A Practical Technique for-Software Development", *IEEE Trans. Software Eng.*, Vol. 1, No. 4, pp. 390-396, December 1975.

Basili, V. and Hutchens, D. (1983). "An Empirical Study of Syntactic Complexity Family", *IEEE Trans. Software Eng.*, Vol. SE-9, No. 6, pp. 664-672, November 1983.

Cantone, G. Cimitile, A and Sansone, L (1983). "Complexity in Program Schemes: The Characteristic Polynomial", *SIGPLAN Notices*, Vol. 18, No. 3, pp. 22-31, March 1983.

Cherniavsky, J. and Smith C. (1991). "On Weyuker's Axioms For Software Complexity Measures", *IEEE Trans. Software Eng.*, vol. 17, no. 6, pp. 636-638, June 1991.

Conte, S., Dunsmore, H., Shen, Y. (1986) *Software Engineering Metrics and Models*, Benjamin Cummings.

Cooper, D. (1990). *Oh My! Modula-2! An Introduction to Programming*, W.W. Norton & Company Inc.

Democko, N.(1993). *A Translator From C To a Data Structure Intermediate Language*, Masters Project, Dept. of Computer Science, Masters Project, Dept. of Computer Science, SUNY Buffalo.

Democko, N. (1993). *A Data Structure Complexity Estimator*, Masters Project, Dept. of

Computer Science, SUNY Buffalo.

Elshoff, J. (1976). "An Nalysis of Some Commercial PL/1 Programs", *IEEE Trans. on Software Engineering*, SE-2, No. 2, pp. 113-120, June 1976.

Fenton, N., Whitty, R. (1986). "Axiomatic Approach to Software Metrication", *The Computer Journal*, 29, 1986.

Fenton, N. (1994). "Software Measurement: A Necessary Scientific Basis", *IEEE Trans. Software Eng.*, SE-20, No. 3, pp. 199-206, March 1994.

Gabrini, P., Kurtz, B. (1992) *Data Structures and Algorithms with Modula-2*, D.C. Heath and Company.

Halstead, M. (1977). *Elements of Software Science*, Elservier North-Holland.

McCabe, T. (1976). "A Complexity Measure", *IEEE Trans. Software Eng.*, SE-2, No. 4, December 1976, pp. 308-320.

McCracken, D., Salmon, W. (1987) *A Second Course in Computer Science with Modula-2*, John Wiley & Sons.

Jackson, M. (1975). *Principles of Program Design*, Academic Press.

Oviedo, E. (1983). *Control Flow, Data Flow and Program Complexity*, Ph. D. Dissertation, Dept. of Computer Science, SUNY Buffalo.

Oviedo, E. (1980). "Control Flow, Data Flow and Program Complexity", *COMPSAC*,

December 1980, pp. 146-152.

Prather, R. (1984). "An Axiomatic Theory of Software Complexity Measure", *The Computer Journal*, Vol. 27, No. 4, pp. 340-347.

Tsai, W., Lopez, M., Rodriguez, V., and Volovik, D. (1986). "An Approach to Measuring Data Structure Complexity", *COMPSAC*, pp. 240-246.

Van Verth, P. (1985). *A System for Automatically Grading Program Quality*, Ph. D. Dissertation, Dept. of Computer Science, SUNY Buffalo.

Walter, R. (1992). *Introducing Computer Science with Modula-2*, West Publishing Company.

Wang, T. (1991). *A Translator From Modula-2 To a Data Structure Intermediate Language*, Masters Project, Dept. of Computer Science, SUNY Buffalo.

Wang, T. (1991). *A Data Structure Complexity Estimator*, Masters Project, Dept. of Computer Science, SUNY Buffalo.

Weyuker, E. (1988). "Evaluating Software Complexity Measures", *IEEE Trans. Software Eng*, vol. 14, pp. 1357-1365, Sept. 1988.

Contents

1 Introduction **20**

2 Overview of Complexity Metrics **23**

3 Data Structure Metrics **35**

3.1 Data Usage Metrics 36

3.2 Data Declaration Metrics 37

4 Data Structure Metric Axioms **43**

4.1 Axiom Description Information 43

4.2 General Principles 45

4.3 Simple Items 48

4.3.1 Constants 48

4.3.2 Simple Types 49

4.3.3 Range Types 49

4.4 Sequences of Declarations 50

4.4.1 Constant Declarations 50

4.4.2 Type Declarations 50

4.4.3 Variable Declarations 52

4.5 Record Axioms 53

4.6 Sequences of Ranges 56

4.7 Arrays 56

4.8 Sets 57

4.9 Pointers 57

4.10 Units 58

5 Structural Data Structure Metric **59**

5.1 Collapsing Specification and Implementation Files 60

5.2 Metric Definition 62

5.2.1 Simple Items 63

5.3 Range Types 64

5.4 Sequences of Declarations 65

5.5 TYPE and VAR Declarations 67

5.6 Sequences of Ranges 69

5.7 Arrays 70

5.8 Sets 71

5.9 Pointers 72

5.10 Records 72

5.11 PROCEDURE Declarations 79

5.12 Units 79

5.13 Values of Metric Parameters 80

6 Graph Data Structure Metrics **83**

6.1 Simple Size Metric 88

6.2 Connection Metric 88

6.3 Layout Metric 91

6.4 Volume Metric 93

6.5 Examples 95

7 Experimental Results **107**

7.1 Experiment Description and Results 108

7.1.1 Item 1 112

7.1.2 Item 2 113

7.1.3 Item 3 113

7.1.4 Item 4 114

7.1.5 Item 5 115

7.1.6 Item 6 115

7.1.7 Item 7 116

7.1.8 Item 8 117

7.1.9 Item 9 117

7.1.10 Item 10 119

7.1.11 Item 11 119

7.1.12 Item 12 120

7.2 Survey Description and Results 122

7.2.1 Item 1 128

7.2.2 Item 2 128

7.2.3 Item 3 129

7.2.4 Item 4 130

7.2.5 Item 5 130

7.2.6 Item 6 131

7.2.7 Item 7 132

7.2.8 Item 8 133

7.2.9 Item 9 134

7.2.10 Item 10 134

7.2.11 Item 11 135

7.2.12 Item 12 135

7.2.13 Item 13 136

7.2.14 Item 14 138

7.2.15 Item 15 138

7.2.16 Item 16 140

7.2.17 Item 17 141

7.2.18 Item 18 143

7.2.19 Item 19 144

7.2.20 Item 20 146

7.2.21 Item 21 148

7.2.22 Item 22 150

7.2.23 Item 23 152

7.2.24 Item 24 153

7.2.25 Item 25 155

7.2.26 Correlation Analysis 158

7.3 Discussion 160

8 Evaluation of Metrics **165**

8.1 Examples 165

8.1.1 Random Number Examples 166

8.1.2 String Examples 171

8.1.3 Stack Examples 174

8.1.4 Graph Examples 184

8.2 Properties 190

8.2.1 Property 1 191

8.2.2 Property 2 193

8.2.3 Property 3 194

8.2.4 Property 4 196

8.2.5 Property 5 196

8.2.6 Property 6 198

8.2.7 Property 7 202

8.2.8 Property 8 206

8.2.9 Property 9 206

8.3 Summary 209

9 A Mechanism for Practical Utilization of Multiple Metrics 211

10 Conclusions and Further Work 223

10.1 Conclusions 223

10.2 Future Work 228

10.2.1 Experiments 228

10.2.2 Data Structure Metrics in Other Styles of Programming Languages . 229

I Appendix 232

Experiment 233

Survey 252

Intermediate Language 267

II References **276**

List of Figures

2.1 Graph for Characteristic Polynomial Example. 33

6.1 Color Map. 85

6.2 Example 1 Code. 86

6.3 Example 1 Graph. 87

6.4 Example 2. 97

6.5 Example 3. 98

6.6 Example 4. 100

6.7 Example 5. 102

6.8 Example 6. 103

6.9 Example 7. 106

8.1 Graph for Example 1. 168

8.2 Graph for Example 2. 169

8.3 Graph for Example 3. 170

8.4 Graph for Example 4. 172

8.5 Graph for Example 5. 173

8.6 Graph for Example 6. 176

8.7 Graph for Example 7. 178

8.8 Graph for Example 8. 180

8.9 Graph for Example 9. 182

8.10 Graph for Example 10. 186

8.11 Graph for Example 11. 188

8.12 Graph for Example 12. 189

9.1 Graphical Display of Different Program. 213

9.2 Set of Faces. 214

9.3 Salient Features - Low and High. 215

9.4 Low Salience Features - Low and High. 215

9.5 Hair Color Example. 216

9.6 Symmetric Inspector. 217

9.7 Inspector Detail Display. 218

9.8 Future Display Forms. 218

A3.1 Intermediate Language 267

List of Tables

5.1 Structural Metric Parameters 81

7.1 Experiment Results 110

7.2 Experiment Statistics 111

7.3 Experiment Statistics with Trim 112

7.4 Network Survey Results 123

7.5 Local Survey Results 124

7.6 Survey Statistics 126

7.7 Survey Statistics with Trim 127

7.8 Rankings 158

8.1 Random Number Generator Complexities 171

8.2 String Complexities 173

8.3 Stack Complexities 174

8.4 Graph Complexities 185

8.5 Property Summary 209

9.1 Different Program. 212

10.1 Structural Metric. 225

10.2 Arcs Used by Graph Metrics. 225

10.3 Arc Weights. 226

10.4 Property Summary 226