UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NPS52-82-010	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Simple Metrics for Programming Languages		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Bruce J. MacLennan		8. CONTRACT OR GRANT NUMBER(s) N00014-82-WR-20162
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61152N: RR000-01--10 N0001482WR20043
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93940		12. REPORT DATE October 1982
		13. NUMBER OF PAGES 32
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

programming language metrics, software metrics, complexity measures, grammar size, language evaluation, empirical computer science, quantitative langauge design.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

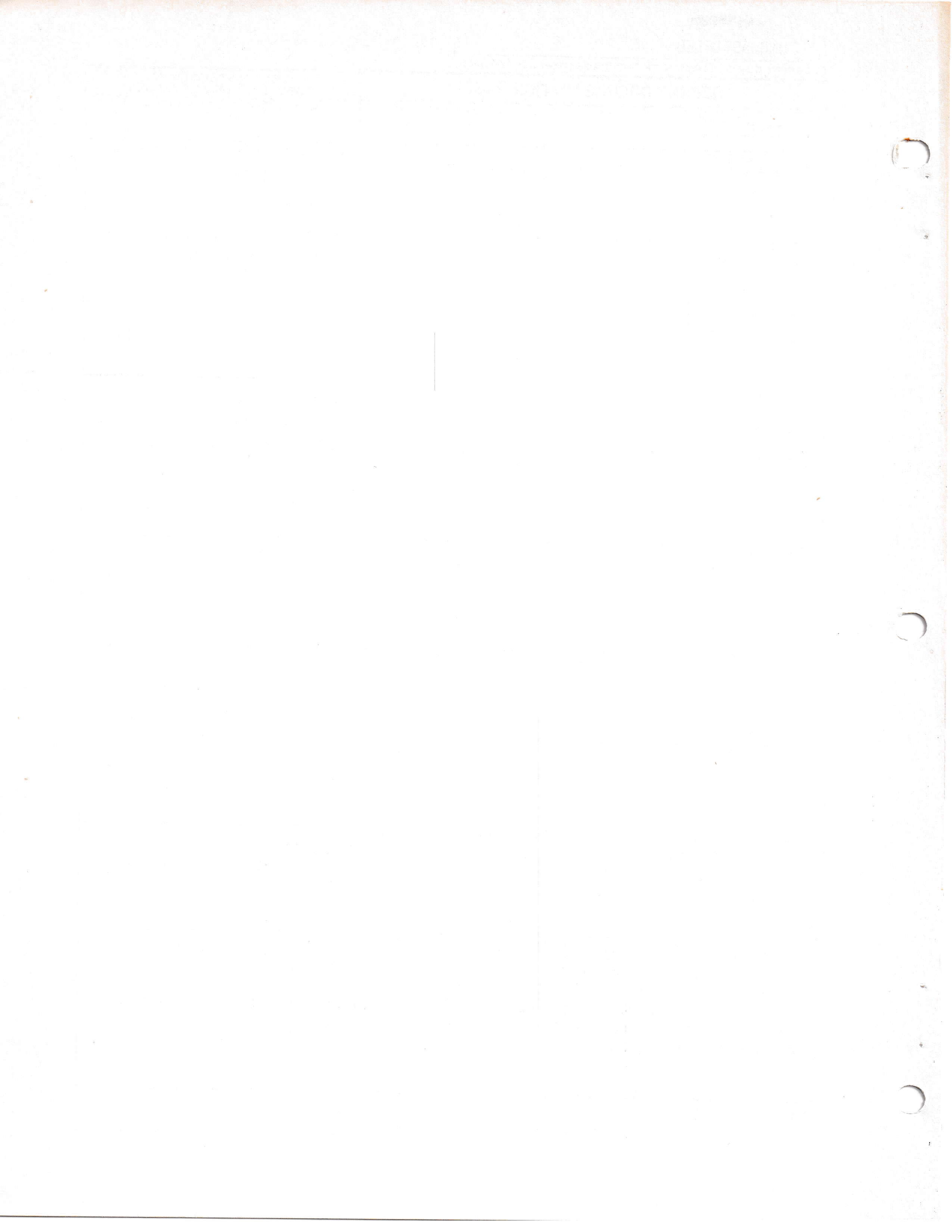
Several metrics for guiding the design and evaluation of programming languages are introduced. The objective is to formalize notions such as 'size', 'complexity', 'orthogonality', and 'simplicity'. Three different kinds of metrics are described: syntactic, semantic, and transformational. The use of these metrics is demonstrated using several complete languages and subsystems of several languages.

DD FORM 1473, 1 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)



SIMPLE METRICS FOR PROGRAMMING LANGUAGES

Bruce J. MacLennan
Computer Science Department
Naval Postgraduate School
Monterey, CA 93940

Abstract:

Several metrics for guiding the design and evaluation of programming languages are introduced. The objective is to formalize notions such as 'size', 'complexity', 'orthogonality', and 'simplicity'. Three different kinds of metrics are described: syntactic, semantic, and transformational.

Syntactic metrics are based on the size of a context-free grammar for a language or a part of a language. They can be used to judge the size of a language and the relative sizes of its parts. These techniques are demonstrated by their application to Pascal, Algol-60, and Ada.

Syntactic metrics make no reference to the meaning of a language's constructs. For this purpose we have developed several semantic metrics that measure the interdependencies among the basic semantic ideas in a language. This technique has been applied to the control, data, and name structures of FORTRAN, BASIC, Lisp, Algol-60, and Pascal.

Finally, we suggest that a useful measure of a programming language is the complexity of the relationship between its

syntactic and semantic structures. For this purpose we introduce a transformational metric and demonstrate its use on subsystems of several languages.

The paper concludes by discussing the general principles underlying all of these metrics and by discussing the proper method of validating metrics such as these.

1. Introduction

Since programming languages are the primary tools used in the programming process, it is not surprising that the choice of programming language is an important element of the life-cycle cost of a software development project. Sometimes the design of a new programming language seems the appropriate approach, as has been the case with the Ada language for embedded computer applications. In either case, it is necessary to be able to compare languages and judge their suitability for various applications.

Programming languages are frequently compared informally. One language may be described as more "structured" than another, or simpler, or more powerful, or better "human engineered", or less procedural, or smaller, or more "orthogonal", and so forth. These claims are particularly common in the descriptions of new programming languages.

Unfortunately, there do not exist objective methods for validating these claims. A claim that one language is preferable to another may be supported by arguments, but these are

frequently unconvincing. Also, these arguments fail to provide any quantitative measure of how languages compare along these axes. This eliminates any meaningful evaluation of the tradeoffs among language design decisions. Thus, language comparison and evaluation remains a mostly subjective art, not unlike literary criticism (see, for example, [1]). This is unsatisfactory for a tool of the importance of a programming language.

2. Related Work

The importance of language metrics makes the lack of research in this area quite astonishing. Perhaps this can be attributed to the relative youth of the craft of language design. Also, it may in part result from some of the problems inherent in formulating language metrics; a subject discussed later. In any case, there are few reported attempts to place language comparison and evaluation on an objective basis.

One such attempt was reported by Sammet [2] in 1971. This approach might be described as "quantified subjectivity." There are several steps: first, a list of language properties, such as "English-like" and "high-level", is made. Each property is assigned a weight depending on its relevance to an application (or application class) as judged by the evaluator. In the second stage the evaluator judges how well each language satisfies each property and assigns a corresponding numeric score. A final score for each language is computed by summing the weighted individual scores. Sammet admits that this technique is subjective

but claims that it at least has the advantage of making the evaluator's biases explicit.

Other attempts to measure languages can be found in the psychological experiments of Gannon [3, 4] and others [5] which compare specific language features (such as terminating versus separating semicolons) with respect to properties such as readability and susceptibility to error. Although these studies are valuable, their application will be limited unless psychological properties can be related to more general language properties (e.g., degree of structure).

How might we go about measuring objective language properties? What properties are amenable to such measurements? One candidate is the size of a language. It is common to speak of one language (say, PL/I) being larger than another (say, Pascal) based on a subjective assessment of the number of features in each language. The size of the reference manuals may even be cited as evidence in such a judgement. A more promising approach to comparing language sizes is to compare the size of their grammars. Since a smaller, more regular language will tend to have a shorter grammar than a larger, less regular language, we can measure the size of a language by the size of its description in a grammar in an appropriate normal form. The grammar itself can be measured in a variety of ways (number of tokens, graph-theoretic measures, etc.).

3. Method of Approach

In the section Introduction we described the use of context-free grammars to measure syntactic complexity. This is based on the idea that the difficulty in learning a language is a function of the length of its grammar. The reason for this is that the programmer must, in essence, internalize these rules. Section 4 shows how this approach can be used to measure the total size of a language's syntax, and how it can be used to compare the relative sizes of a language's parts.

There must be more to complexity than just grammar size, however, since the shortest programming language grammar (for any infinite language) is that whose statements are sequences of identical tokens, e.g.

```
<program> ::=  1 | <program> 1
```

The reason that such a language is not simple is that the translation mapping programs to their meanings is very complicated. We could say that the translation is not continuous (this is more than a metaphor if these issues are placed in a lattice-theoretic framework). To measure this complexity we use translation grammars rather than simple generative grammars: the complexity of a language is a function of the relation between its syntax and its semantics. Measurement of this is accomplished by writing a translation grammar that maps the language in question into an abstract language that embodies its semantics. The size of this translation grammar can then be measured in a variety of ways.

This transformational metric is demonstrated in Section 5.

The technique described above measures the transformational complexity of a language, i.e., the complexity of the relation between a language's syntax and semantics, but it does not address the complexity of the underlying semantics. That is, a language might have a simple grammar that is simply related to its semantic constituents, but these semantic constituents might themselves be complicated. (Of course, with a continuous translation, a complicated semantics will to some extent induce a complicated syntax.) For instance, we can observe that the data structuring methods of Pascal are more elaborate than those of Algol-60. How can we measure this fact?

One technique comes from denotational semantics (see, for example, [6], [7]). By using these techniques one can formulate a set of domain equations that describe, for instance, the data types provided by a language. It is then often possible to rank the complexity of the data structuring methods provided by several languages by comparing the complexity of the associated domain equations. To convert this into a quantitative measure it is necessary to measure the complexity of these equations quantitatively. This technique has already been used by the author to compare FORTRAN, Algol-60, Pascal, and Ada on the basis of the complexity of their data, control, and name structuring facilities [8].

Some subsystems of a language, such as the control struc-

tures, are not readily amenable to formulation as domain equations. Thus, a more generally applicable technique has been developed. We can observe that all structures in programming languages are produced by applying a set of constructors to a set of primitives. The various ways in which these constructors can be combined can be represented as grammar-like rules or as simple graphs. More formally, sets of structures can be taken as objects, and constructors as morphisms, in a category corresponding to the structural system.

How does this permit comparison or evaluation of languages? Intuitively, we might expect the complexity of a structural system to be related to the number of dependencies between parts of the system. These are represented by the number of morphisms, or by the number of edges in the diagram representing the system. Therefore, by ranking the complexity of the diagrams, we have an ordinal measure for system complexity, and by counting the edges in the diagram, we have a cardinal (quantitative) measure of complexity. Of course there are many other measures that can be applied to graphs, and several of these are investigated in Section 6.

The important issue of the validation of programming language metrics is discussed briefly in Section 7.

4. Syntactic Metrics

We define a context-free grammar G to be a quadruple,

$$G = \langle T, N, P, g \rangle$$

where T is a finite set of terminal symbols, N is a finite set of non-terminal symbols, $V = T \cup N$ is the vocabulary, $P \subseteq N \times V^*$ is a finite set of productions, and $g \in N$ is the goal symbol. We use lower-case letters for elements of the vocabulary and upper-case letters for sequences and sets.

For a string S in V^*, let $|S|$ be the length of S. Then, we define the size $|\pi|$ of a production $\pi = \langle n,S \rangle$ in P as $|n|+|S| = |S|+1$. The size $|G|$ of a context-free grammar G is defined

$$|G| = \sum_{\pi \in P} |\pi| = p + \sum_{\langle n,S \rangle \in P} |S|$$

where $p = |P|$ is the cardinality of P. This definition of size is essentially the same as S(G) defined in [9] and [10]. We also define $R(G) = |G|-p$ to be the total size of the right-hand sides of the productions.

The size of a context-free grammar is easy to determine from its written form. For example, to determine the size of the grammar with these productions:

g = h

g = gh

h = i

h = sjg

j = i

j = ji

we simply count all the tokens except for the equal-signs. In this case the size is 16.

Context-free grammars may be written in various kinds of extended notations. For example, the BNF notation allows productions of the form

$$n = S_1 + S_2 + \ldots + S_k$$

as an abbreviation for the context-free productions

$$n = S_1$$
$$n = S_2$$
$$\ldots$$
$$n = S_k$$

We define the size of the BNF production in terms of the size of the corresponding context free productions, namely

$$k + \sum_{i=1}^{k} |S_i|$$

Since there are k-1 plus-signs in the BNF production, the size of BNF productions can also be determined by simply counting the tokens they contain.

Another common notation for context-free grammars allows the use of parenthesized lists of alternatives. A production of the form

$$n = R\ (S_1 + \ldots + S_k)\ T$$

means the same as

$$n = RsT$$

$$s = S_1 + \ldots + S_k$$

The size of the latter can be computed from the extended production if each of the parentheses is counted as one token. Similar conversions can be found for other notations for context-free grammars.

Note that the number of productions in a BNF or extended BNF grammar is just n, the number of non-terminals. We define the right-hand size of a BNF or extended BNF grammar G to be $R(G) = |G|-n$. Obviously, this is obtained by counting everything to the right of the equal-signs.

In Table 1 we show the size of the context-free grammars for BASIC, Pascal, Algol-60, and Ada. Since several of these languages are expressed in extended-BNF notations, conversion factors like those described above have been used.

The size measure we have defined can also be applied to parts of a language's grammar. This is useful for comparing the relative size of a language's subsystems and for comparing the amount of syntax used by different languages for corresponding subsystems. Table 2 shows the size of the major subsystems of Algol-60. Table 3 compares Algol-60, Pascal, and Ada on the basis of the proportion of their syntax devoted to various purposes. The greater proportion of Pascal devoted to declarations

is a result of its more elaborate type system; this trend has continued in Ada.

5. Transformational Metrics

As discussed in Method of Approach, the goal of transformational metrics is to measure the complexity of the relationship between the syntax and semantics of a language. We do this by measuring the size of a context-free translation grammar that maps the source constructs into an abstract language representing the meaning of the constructs.

Translation grammars are commonly written as sets of transformation rules. For example, the following production is a transformation rule that maps certain expressions from infix to prefix form:

```
E  =  E+T  ⇒  +ET
   +  E-T  ⇒  -ET
   +  T    ⇒  T
```

(Of course, the left-most plus-sign in each line indicates alternation in the BNF rule; the other plus-signs are terminal symbols.)

The notation above is not general since there may be several occurrences of the same non-terminal on the left. This results in an ambiguity in the correspondence with the non-terminals on the right. For this reason, a more general notation for transla-

tion grammars uses natural numbers on the right to refer to corresponding non-terminals on the left. For example:

$$\begin{array}{llll} E & = & E+T & \Rightarrow +12 \\ & + & E-T & \Rightarrow -12 \\ & + & T & \Rightarrow 1 \end{array}$$

Thus, in '+12', '1' refers to the first non-terminal on the left, namely 'E'.

These considerations lead to the following definition: A _context-free translation grammar_ is a quintuple,

$$G = \langle T, S, N, P, g \rangle$$

where T is a finite set of _analysis_ terminal symbols, S is a finite set of _synthesis_ terminal symbols, N is a finite set of non-terminal symbols, and P is a finite set of transformation rules. A _transformation rule_ is an element of

$$N \times V^* \times W^*$$

where $V = T \cup N$ is the _analysis vocabulary_, and $W = S \cup Nat$ (where Nat is the natural numbers) is the _synthesis vocabulary_.

A BNF translation rule such as

$$\begin{array}{llll} E & = & E+T & \Rightarrow +12 \\ & + & E-T & \Rightarrow -12 \\ & + & T & \Rightarrow 1 \end{array}$$

can be translated into the equivalent context-free translation

rules

$$E = E+T \Rightarrow +12$$

$$E = E-T \Rightarrow -12$$

$$E = t \Rightarrow 1$$

which are represented by the triples

$\langle E, \langle E, +, T\rangle, \langle +, 1, 2\rangle\rangle$

$\langle E, \langle E, -, T\rangle, \langle -, 1, 2\rangle\rangle$

$\langle E, \langle T\rangle, \langle 1\rangle\rangle$

We define the <u>analysis size</u> of a translation grammar G to be the total size of the analysis parts of the rules:

$$A(G) = \sum_{\langle n,S,T\rangle \in P} |S|$$

Similarly, the <u>synthesis size</u> is the total size of the synthesis parts of the rules:

$$S(G) = \sum_{\langle n,S,T\rangle \in P} |T|$$

Finally, the total size of the grammar is defined:

$$|G| = |P| + A(G) + S(G)$$

Note that $|P|+A(G)$ is the size of the context-free grammar corresponding to the translation grammar G.

As with the syntactic metrics defined earlier, this transformational metric can be computed by counting the tokens in a translation grammar, ignoring the '=' and '⇒' signs.

Consider the simple translation grammar in Figure 1, which maps infix arithmetic expressions into prefix. Measuring it yields:

$$A(G) = 19$$

$$S(G) = 17$$

$$|P| = 9$$

$$|G| = 45$$

The author used one variant of this approach to design the external appearance of the 8086 microprocessor for Intel Corporation. In this case a translation grammar was formulated that mapped an assembly-language level view of the machine into the various primitive operations it provided. The complexities of alternate views were then estimated by measuring the size of the associated translation grammars. The premise underlying this approach was that the syntactic complexity of a language was a function of the complexity of the mapping from the language into its semantic constituents. This mapping was, in essence, what the programmer had to learn in order to use the machine. This technique resulted in a number of improvements in the apparent simplicity of the 8086.

6. Semantic Metrics

In this section we consider methods for measuring the semantic complexity of structural subsystems of a programming language. That is, we are interested in measuring the complexity of the

semantic interrelationships without regard for the complexity of the syntax with which they are expressed.

Consider a subset of the Pascal type system with primitive types real, integer, Boolean, and char and with the array and set type constructors. The allowable interrelationships among these types can be expressed by domain equations such as these:

```
T  =  D + R + array(X,T) + set(X)
D  =  X + I
X  =  B + C + subrange( el(D), el(D))
```

where the plus-sign denotes disjoint union, upper-case letters represent domains (T=type, D=discrete type, R=real, X=index type, I=integer, B=Boolean, C=char), and words beginning with lower-case letters denote functions on the domains. For example, 'set(S)' is the power-set of S and 'array(D,R)' is the set of all (continuous) functions from D to R.

The number of restrictions and special cases inherent in a subsystem of a programming language will be reflected in the complexity of the domain equations required to describe that subsystem. We can measure the complexity of these equations by replacing them by an equivalent context-free grammar:

```
T  =  D + r + aXT + sX
D  =  X + i
X  =  b + c + deDeD
```

This has the terminal symbols 'r', 'i', 'b', and 'c'

corresponding to the primitive types, and the terminal symbols 'a', 'd', 's', and 'e' corresponding to the type constructors. We have eliminated parentheses by representing function applications in prefix form (hence, we essentially have a tree grammar). The resulting grammar generates the language of all type structures defined by the equations, i.e.,

{ r, b, c, i, sb, sc, abr, abb, abc, abbi, absb, ... }

We can measure the size of this grammar: 25.

A <u>semantic grammar</u> is a BNF grammar in which the right-hand sides of the productions are representations of domain expressions. That is, the strings between the plus-signs are either (1) non-terminals, (representing non-primitive domains), (2) niladic terminals (representing primitive domains), or (3) n-adic terminals (representing constructor functions) followed by n argument strings, each representing either a domain (primitive or non-primitive), or a constructor function with its arguments. Figure 2 shows a <u>syntactic</u> grammar for the Pascal type system; Figure 3 shows the corresponding <u>semantic</u> grammar.

Another way to view a semantic subsystem of a language is through a <u>dependency graph</u> like that in Figure 4, which corresponds to the semantic grammar:

T = D + r + aXT + sX

D = X + i

X = b + c + deDeD

In the graph the dependencies among the parts of the type system become apparent: a type depends on the definition of another type if there is an edge leading from the latter to the former. Hence, recursive definitions are represented by cycles and primitive domains are represented by initial nodes. The output from a node can lead to exactly one other node, although this latter node may be a <u>fan-out</u> operation (represented by a small dot), which can have any number of outputs. The output of the entire graph is always required to be a fan-out operation.

How can we measure the complexity of such a graph? The nodes represent the concepts (types, in this case) that are defined by the system and the edges represent the dependencies among the definitions. Therefore, since one notion of the complexity of a system is just the number of dependencies among its parts, one way to measure the complexity is to count the edges in the dependency graph. In this example it is 22.

We now relate the complexity measures for semantic grammars and dependency graphs.

<u>Theorem</u>: Let G be a semantic grammar and let Γ be the corresponding dependency graph. Let $E(\Gamma)$ represent the number of edges in Γ, and $F(\Gamma)$ represent the number of fan-out nodes in Γ. Then:

$$R(G) = E(\Gamma)$$

$$N(G) = F(\Gamma)$$

$$|G| = E(\Gamma)+F(\Gamma)$$

where N(G) is the size of the non-terminal vocabulary of G (which is also the number of productions in a BNF grammar).

<u>proof</u>: We sketch the proof informally. The method of constructing the dependency graph from a grammar will make the truth of the theorem obvious. Repeat the following procedure for each production in the grammar:

For each production 'n = S', add a fan-out node labeled 'n' to the graph. Hence, the number of fan-out nodes will equal the number of non-terminals, since in a BNF grammar the number of productions is the same as the number of non-terminals. Thus, $N(G) = F(\Gamma)$.

Suppose that S (in the production 'n=S') has the form U+V; add to the graph a plus-node whose inputs are U and V and whose output is the fan-out node for n. The plus-sign in the production corresponds to the edge from the plus-node to the fan-out node. Continue this process if either U or V contains plus-signs by adding new plus-nodes whose outputs lead to previously added plus-nodes. Hence, the number of edges leading from plus-nodes is the number of plus-signs in the grammar.

Next consider a terminal string S that does not contain a plus-sign. If S is a single niladic terminal symbol t, then add an initial node to the graph with an edge leading out from it. Hence, the number of edges leading from initial nodes is the number of occurrences of niladic terminal symbols.

If S is a single non-terminal symbol n, then construct an edge leading from the fan-out node labeled n. Hence, the number of edges leading from fan-out nodes is the number of occurrences of non-terminal symbols on the right-hand side of rules.

Finally, suppose S is a string

$$fS_1S_2...S_k$$

where f is a non-niladic terminal symbol representing an operator and the S_i are strings representing the arguments of that operation. Add a node representing an operation f and recursively process its arguments. Hence, the number of edges leaving operator nodes is the number of non-niladic terminal symbols in the grammar.

Since every edge must leave either a fan-out node, an initial node, or an operator node, the total number of edges is the total of the number of occurrences of non-terminals, niladic terminals, and non-niladic terminals. Hence, the number of edges is just the total number of symbols that occur on the right of the BNF rules, so $R(G) = E(\Gamma)$. QED.

Both the grammar-oriented and graph-oriented approaches have been applied to measuring the semantic complexity of the data, control, and name structures of several programming languages. These studies are reported in [8] and [11].

7. Validation of Metrics

There remains the important question, How are these measures validated? To put it another way, we have an informal understanding of complexity; how can we make it formal? Firstly, our formal measure must agree with our informal judgements in most cases. For instance, the measure should show that the data structures of Algol-60 are simpler than those of Pascal. This aspect of the validation could be backed up with formal psychological tests, but this does not seem necessary. Psychological validation has not been required for concepts such as "computable": the formal definition seems to correspond to the informal, although no formal proof of the correspondence is possible.

Secondly, we can determine if the formal measure satisfies the same properties as the informal. For instance, the measure should be additive in those aspects that the informal idea is additive. An example of this comes from information theory: we expect the information capacity of two pages to be approximately the sum of the information capacities of the separate pages. It is easy to see that the formal definition of information capacity satisfies this property.

Finally, the formal measure should be productive; that is, it should lead to a rich theory with good predictive abilities and explanatory power. Information theory is a perfect example. Of course, it is difficult to evaluate a measure on this basis until a substantial amount of experience in its use has accumu-

lated.

8. Conclusions

In this paper we have defined three simple metrics that can be applied to programming language design. The first is a syntactic metric that is determined by counting the tokens in a context-free grammar for a language or a part of a language. This allows a language designer to estimate the total syntactic complexity of a language and to measure the relative proportion of a language's syntax devoted to different purposes.

The second metric is a transformational metric that is determined by the number of tokens in a translation grammar that maps the source language into an abstract language reflecting the basic semantic notions of the language. This metric allows the language designer to evaluate the complexity of the relationship between a language's syntax and semantics. Like the syntactic metric, it can be applied to the entire language or to particular parts.

Next we defined a semantic metric that is determined by the number of tokens in a context-free grammar that describes the dependencies among the semantic primitives. This metric was shown to be equivalent to a metric based on the number of nodes and edges in the corresponding semantic dependency graph. The semantic metric is most usefully applied to well-defined semantic subsystems of a programming language, such as its control struc-

ture, name structure, and data type systems. This permits the comparison of the complexity of the dependencies in corresponding systems in different languages.

Finally we discussed the validation of metrics like those defined in this paper. We argued that these metrics must be validated by their integration with existing theories and by their usefulness, rather than by psychological demonstrations of their relationship with perceived qualities. As it has in the natural sciences, the objective approach is more likely to produce testable, widely applicable theories than is the subjective approach.

9. Acknowledgements

The work reported herein was supported by the Office of Naval Research under contract number N00014-82-WR-20162.

10. References

[1] A.R. Feuer and N.H. Gehani, A Comparison of the Programming Languages C and PASCAL, Comp. Surveys 14, 1, March 1982, pp 73-92.

[2] J.E. Sammet, Problems in, and a Pragmatic Approach to Programming Language Measurement, AFIPS Fall Joint Computer Conf., 1971, pp 243-251.

[3] J.D. Gannon and J.J. Horning, Language Design for Program-

ming Reliability, IEEE Trans. Software Eng. SE-1, 2, June 1975, pp 179-191.

[4] J.D. Gannon, An Experimental Evaluation of Data Type Conventions, CACM 20, 8, Aug. 1977, pp 584-595.

[5] B. Schneiderman, SOFTWARE PSYCHOLOGY: Human Factors in Computer and Information Systems, Winthrop Publishers, Inc., Cambridge, Mass. (1980).

[6] M.J.C. Gordon, The Denotational Description of Programming Languages, Springer-Verlag, New York (1979).

[7] R. Milne and C. Strachey, A Theory of Programming Language Semantics, Chapman and Hall, London (1976).

[8] B.J. MacLennan, The Structural Analysis of Programming Languages, Naval Postgraduate School Computer Science Department Technical Report NPS52-81-009, September 1981.

[9] S. Ginsburg, and N. Lynch, Size Complexity in Context-Free Grammar Forms, JACM 23, 4, October 1976, pp 582-598.

[10] J. Gruska, On the Size of Context-Free Grammars, Kybernetica 8, 1972, pp 213-218.

[11] B.J. MacLennan, Measuring Control Structure Complexity Through Execution Sequence Grammars, Naval Postgraduate School Computer Science Department Technical Report NPS52-81-015, November 1981.

TABLE 1. Comparison of Sizes of Entire Languages

Language	Total Grammar Size
BASIC	396
Pascal	541
Algol-60	603
Ada	1614

TABLE 2. Sizes of Subsystems of Algol-60

Subsystem	Size (tokens)
Lexics	69
Expressions	210
Statements	177
Declarations	147
Total	603

TABLE 3. Subsystem Proportions of Algol-60, Pascal, and Ada

Subsystem	Algol-60 (%)	Pascal (%)	Ada (%)
Lexics	11	14	8
Expressions	35	23	16
Statements	29	22	22
Declarations	24	41	54
Total	99	100	100

```
E    = E + T    ⇒  sum 1 2
     + E - T    ⇒  dif 1 2
     + T        ⇒  1
T    = T * F    ⇒  prd 1 2
     + T / F    ⇒  quo 1 2
     + F        ⇒  1
F    = ( E )    ⇒  1
     + I        ⇒  1
     + N        ⇒  1
```

Figure 1. Translation Grammar for Arithmetic Expressions

```
type              = type_id
                  + id_list
                  + constant .. constant
                  + ↑ type_id
                  + PACKED structured_type
                  + structured_type

id_list           = id  +  id , id_list

structured_type   = ARRAY [ type_list ] OF type
                  + RECORD field_list END
                  + RECORD field_list variant_part END
                  + FILE OF type
                  + SET OF type

field_list        = ε  +  id_list : type ; field_list

variant_part      = CASE opt_id type_id OF variant_list

opt_id            = id :  +  ε

variant_list      = variant  +  variant ; variant_list

variant           = case_labels : ( field_list )

case_labels       = constant  +  constant , case_labels
```

Figure 2. Syntactic Grammar of Pascal Type System

```
type                = REAL
                    + discrete_type
                    + PTR type
                    + PACKED structured_type
                    + structured_type

discrete_type       = INTEGER  +  index_type

index_type          = BOOLEAN  +  CHAR  +  POWERSET id
                    + SUBRNG const const

const               = SELECT discrete_type

structured_type     = ARRAY index_type type
                    + SET index_type
                    + FILE type
                    + RECORD field_list
                    + RECORD field_list variant_part

field_list          = ε  +  CONS_PAIR id type field_list

variant_part        = CASE opt_id index_type variant_list

opt_id              = id  +  ε

variant_list        = variant  +  CONS variant variant_list

variant             = PAIR constant field_list
```

Figure 3. Semantic Grammar for Pascal Type System

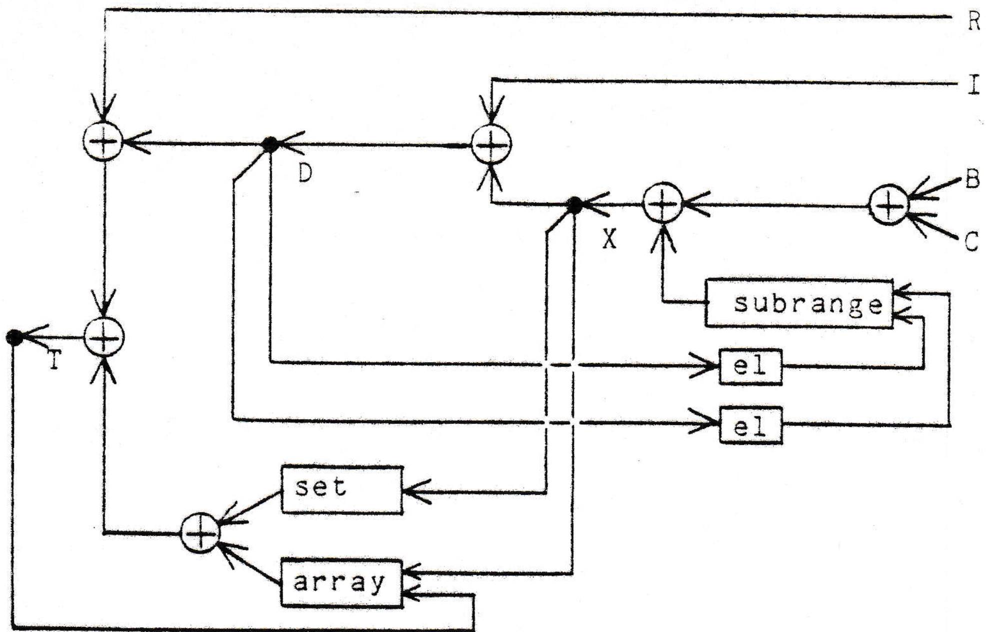


Figure 4. Diagram of Subset of Pascal Type System

INITIAL DISTRIBUTION LIST

Defense Techical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93940	2
Office of Research Administration Code 012A Naval Postgraduate School Monterey, CA 93940	1
Chairman, Code 52Hq Department of Computer Science Naval Postgraduate School Monterey, CA 93940	40
Professor Bruce J. MacLennan, Code 52Ml Department of Computer Science Naval Postgraduate School Monterey, CA 93940	12
Dr. Bob Grafton Code 433 Office of Naval Research 800 N. Quincy Arlington, VA 22217	1