View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Calhoun, Institutional Archive of the Naval Postgraduate School

LIBRARY
RESEARCH REPORTS DIVISION
NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA 93940

NPS52-81-009

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THE STRUCTURAL ANALYSIS

OF PROGRAMMING LANGUAGES

B. J. MacLennan

September 1981

Approved for public release; distribution unlimited

Prepared for:

Chief of Naval Research
Arlington, VA 22217

FEDDOCS
D 208.14/2:NPS-52-81-009

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral J. J. Ekelund
Superintendent

D. A. Schrady
Acting Provost

The work reported herein was supported by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

Reproduction of all or part of this report is authorized.

This report was prepared by:

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NPS52-81-009	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) THE STRUCTURAL ANALYSIS OF 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)
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 N0001481WR10034
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93940		12. REPORT DATE ~~9 June 1981~~ SEPT. 81 13. NUMBER OF PAGES 25
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)

Structural Analysis, Programming Language Metrics, Software Metrics, Software Measurement, Metrics, Programming Language Design Methods.

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

A language's structures are some of its most important characteristics. These include the data structures: those mechanisms that the language provides for organizing elementary data values. They also include the control structures, which organize the control flow. Less obviously, they include the name structures, which partition and organize the name space.

Languages can be compared relative to their structures in the data, control, and name domains. This report describes a syntax-independent

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

method of representing the structures of a language which facilitates visual complexity comparisons and is amenable to measurement. The data, control, and name structures of a number of languages are analyzed, including Pascal, LISP, Algol-60, Algol-68, the lambda calculus, FORTRAN, and Basic.

The Structural Analysis of Programming Languages.

B. J. MacLennan
SEPT. 81
~~81/06/09~~

1. Introduction

It is common to find articles in the programming language literature riddled with unsupported claims. Words and phrases, such as 'better', 'simpler', 'more structured' and 'less error prone', are used with abandon. If we were selling aspirin and made such unsupported claims, we would probably be sued. We clearly need more precise ways of measuring our languages.

A language's structures are some of its most important characteristics. These include the data structures: those mechanisms that the language provides for organizing elementary data values. They also include the control structures, which organize the control flow. Less obviously, they include the name structures, which partition and organize the name space.

Languages can be compared relative to their structures in the data, control and name domains (and others, such as the syntactic domain). To make this comparison precise, we need a precise method of describing the structural properties of a language. Further, this method should be syntax independent; it should "look through" the syntax of a language to its underlying structure. In the next section we discuss a means by which pro-

gramming language structures can be described.

2. Describing Structure

The number of different structures that a programmer can use are essentially unlimited. For instance, there are an infinite number of ways he can organize his data or control flow. Since programming languages are finite, there must be some finite means of generating this infinite number of structures.

The means, of course, is to have some number of _primitive_ structures and some number of _constructor_ functions which take existing structures and compose them into new structures. For instance, Pascal data types are built by applying the data type constructors (array, record, set, etc.) to the primitive data types (real, integer, char, etc.). This results in hierarchical structures. Similarly, control flows may be organized by applying the control flow constructors ('sequence', 'if,' and 'while') to the control flow primitives (those constructs that do not alter the control flow).

The hierarchical application of constructors to primitives is the most common method of building structures. Thus, we can use this as a starting point for our analysis of structures. For instance, as a first approximation, we can compare the complexity of structures of two programming languages by comparing the number of primitives and constructors in each. For instance, we can see from Table 1 that Pascal has 5 primitive data types and 7

data type constructors.

TABLE 1. Data Structures

	Pascal	
5	primitives:	real, integer, Boolean, char, text
7	constructors:	subrange, enumeration, set, array, file, pointer, record.
	Algol - 60	
3	primitives:	real, integer, Boolean.
1	constructor:	array.
	Lisp 1.5	
1	primitive:	atom
1	constructor:	list
	Algol - 68	
11	primitives:	int, real, bool, char, format, compl, bits, bytes, string, sema, file
6	constructors:	long, ref, array, struct, union, proc

Since Algol-60 has 3 primitives and 1 constructor, it is probably simpler than Pascal. Conversely, since Algol-68 has 11 primitives and 6 constructors it is likely to be more complex. However, the number of primitives and constructors is not the entire story.

A significant aspect of the structuring mechanisms provided by a language is the complexity of the inter-relationships among

the primitives and constructors. For instance, if the output of every constructor is a legitimate input to every constructor, and every primitive is a legitimate input to every constructor, then the system will be more regular than if this is not the case. This is often called 'orthogonality'. It is also part of what is involved when we call a language 'structured'. In the next section we will develop means for analyzing these relationships.

3. Data Structures

3.1 Semantic Grammars

We will begin with data structures to illustrate our technique for analyzing structure. Our goal is to analyze the interrelationships among the primitives and constructors of a system of data structures. How are we to go about this? We can begin by looking at syntax because, in most languages, there is a close relation between the syntax and the structures it embodies (i.e., form follows function). In particular, there will usually be exactly one syntactic construct for each data primitive. Consider Pascal. We can see from Table 1 that the primitives are denoted by the predefined type identifiers, 'integer', 'Boolean', 'real', 'char' and 'text'. There are constructors for enumerations, subranges, sets, arrays, records, files and pointers. We know that these are constructors because each can generate a potentially unlimited number of structures (types). Since the Pascal grammar tells us what <u>syntactic</u> entities can go together this will be a big help in deciding what <u>semantic</u> entities can go

together.

Consider the array type. We can write its syntax as

```
array-type ::=  array [  index-type ,... ] of type
```

The index-type must be a type isomorphic to a subrange of the integers. _Syntactically_, this can take the form:

```
index-type:  scalar-type | subrange-type | type-identifier

scalar-type:  ( identifier ,...)

subrange-type:  constant .. constant
```

What we are interested in, however, is the _semantics_ of the array constructor. Since we know that the index type must be isomorphic to a subrange of the integers, we know that the type-identifier must either name a scalar-type or a subrange-type or one of the predefined _finite_ _discrete_-_types_, Boolean and char. Also, a subrange must be constructed from a _discrete_ _constant_ (i.e., an integer, or an element of a scalar or finite discrete type). We can write this as a "semantics-oriented grammar":

```
array-type:  array [ index-type ,... ] of type
index-type:  scalar-type | subrange-type | discrete-type
scalar-type:  ( identifier ,...)
subrange-type:  constant .. constant
discrete-type:  Boolean | char
```

One further simplification can be made here. Recall that in Pas-

cal

```
array [ i, j ] of t
```

is just an abbreviation for

```
array [i] of array [j] of t
```

Thus, without loss of generality, the definition of array-type can be written

```
array-type:  array [ index-type ] of type
```

We have not altered the syntax; we have just eliminated some syntactic sugar. The semantics of most of the rest of Pascal's constructors closely follows their syntax.

If we are to be able to compare structures in different languages, we must obviously ignore any syntactic differences that exist between them. This we can do by writing the grammar in a neutral, functional form. For instance, for arrays:

```
array-type:  array (index-type, type)
index-type:  scalar-type | subrange-type | discrete-type
scalar-type:  scalar ( identifier+ )
subrange-type:  subrange (constant, constant)
discrete-type:  Boolean | char
```

3.2 Interpretation

Now, let us make some observations about these rules. Consider a typical string generated by this grammar:

array (char, array (Boolean, real))

This string describes a particular Pascal data type. Now suppose BOOLEAN = { true, false } is the set of all Boolean values and REAL is the set of all real values. Then, the set of all arrays with Boolean indices and real elements in just the set of functions mapping BOOLEAN into REAL: [BOOLEAN -> REAL]. Therefore, we can see that the string shown above describes the set of data values:

[CHAR -> [BOOLEAN -> REAL]]

This suggests that we can define an _interpretation function_, I, that associates a set of data values with each string generated by the grammar. This can be defined recursively:

I [array (t, t')] = [I[t] -> I[t']]

I [scalar ($i_1, \ldots, i_n$)] = { $i_1, \ldots, i_n$ }

I [subrange (C, C')] = { x | $C \leq x$ & $x \leq C'$ }

I [Boolean] = BOOLEAN

I [char] = CHAR

I [real] = REAL

To make this interpretation more obvious, we will write _subrange_ (C, C') as C..C', and _scalar_ ($i_1, \ldots, i_n$) as { $i_1, \ldots, i_n$ }. Figure 1 shows the complete Pascal type system using these conventions.

Defining the interpretation for record-type and pointer-type is quite complicated without the notations of a relational

```
type:  simple-type | structured-type | pointer-type
simple-type:  index-type | integer | real
index-type:  scalar-type | subrange-type | discrete-type
scalar-type:  { identifier + }
subrange-type:  constant .. constant
discrete-type:  Boolean | char
structured-type:  [packed] unpacked-structured-type
unpacked-structured-type:  array-type | record-type | set-type |
      file-type
array-type:  array (index-type, type)
record-type:  record ([field*]) [variant-part]
field:  field (identifier, type)
variant-part:  field (identifier, index-type)
               X (constant X record-type)*
set-type:  set (index-type)
file-type:  file (type)
pointer-type:  pointer (type)
```

Figure 1. The Pascal Type System

calculus, so they will not be shown here. The interpretation of set and file types are easy to define:

I [set (t)] = P (I[t])

I [file (t)] = I[t]*

where P is the power-set function.

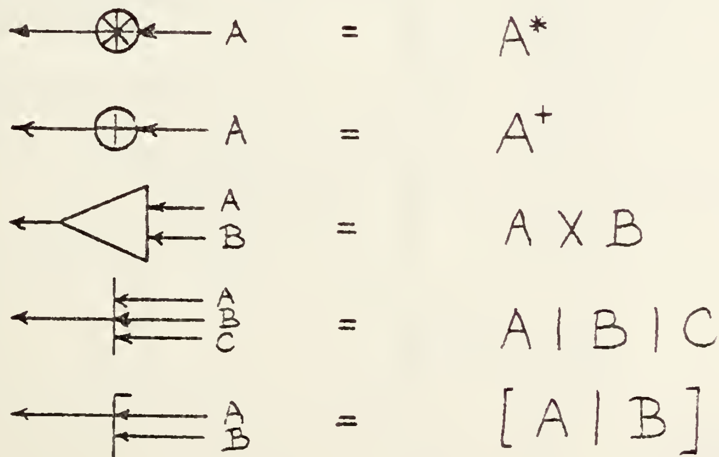
It should be noted that the above equations imply <u>structural equivalence</u> of Pascal types, as opposed to <u>name equivalence</u>. The Revised Report on Pascal [4] does not define the form of type equivalence used. It is simple to alter the above definitions to accommodate name equivalence; we just represent each type by a pair where the first element of the pair is the type's identifier and the second element of the pair is the type in the structural sense. Thus we have,

```
type:  identifier X unnamed-type
unnamed-type:  simple-type | structured-type | pointer-type
```

It should be pointed out that there are limitations to the descriptive power of this notation. For instance, it does not express the fact that the identifiers in scalar-types must be distinct, or that type identifiers must be distinct, etc. To include all this information would clutter the notation to the point of unusability.

4. Structure Diagrams

We have said that the complexity of a collection of structures is reflected by the complexity of the semantic grammar. It is still a little difficult to see this complexity in the traditional BNF form. For this purpose we have found a diagrammatic form enlightening. This is really a dependency graph (showing which nonterminals depend on which others) coupled with special symbols for various operations, viz.



where [A|B] means either A or B or nothing.

In our semantic grammars (as in syntactic grammars) common

structural patterns are factored out and given names. This reflects the fact that these structural patterns only have to be learned once. In the structure diagrams this factoring is represented by an edge that forks and goes to each of the uses of that structure. For example, since 'index-type' is used both as a part of 'discrete-type' and as a part of array and set types, the edge from index type goes to the subgraphs defining each of these structures. We have adopted the convention of only using binary forks; since edges represent dependencies, this simplifies complexity estimation by edge counting.

Structures from other systems are represented by T-shaped terminations. Given this explanation, the reader is encouraged to compare the diagram of Pascal's data structures in Figure 2 with the semantic grammar in Figure 1. The data structures of LISP, Algol-60, and Algol-68 are diagrammed in Figures 3 - 5.

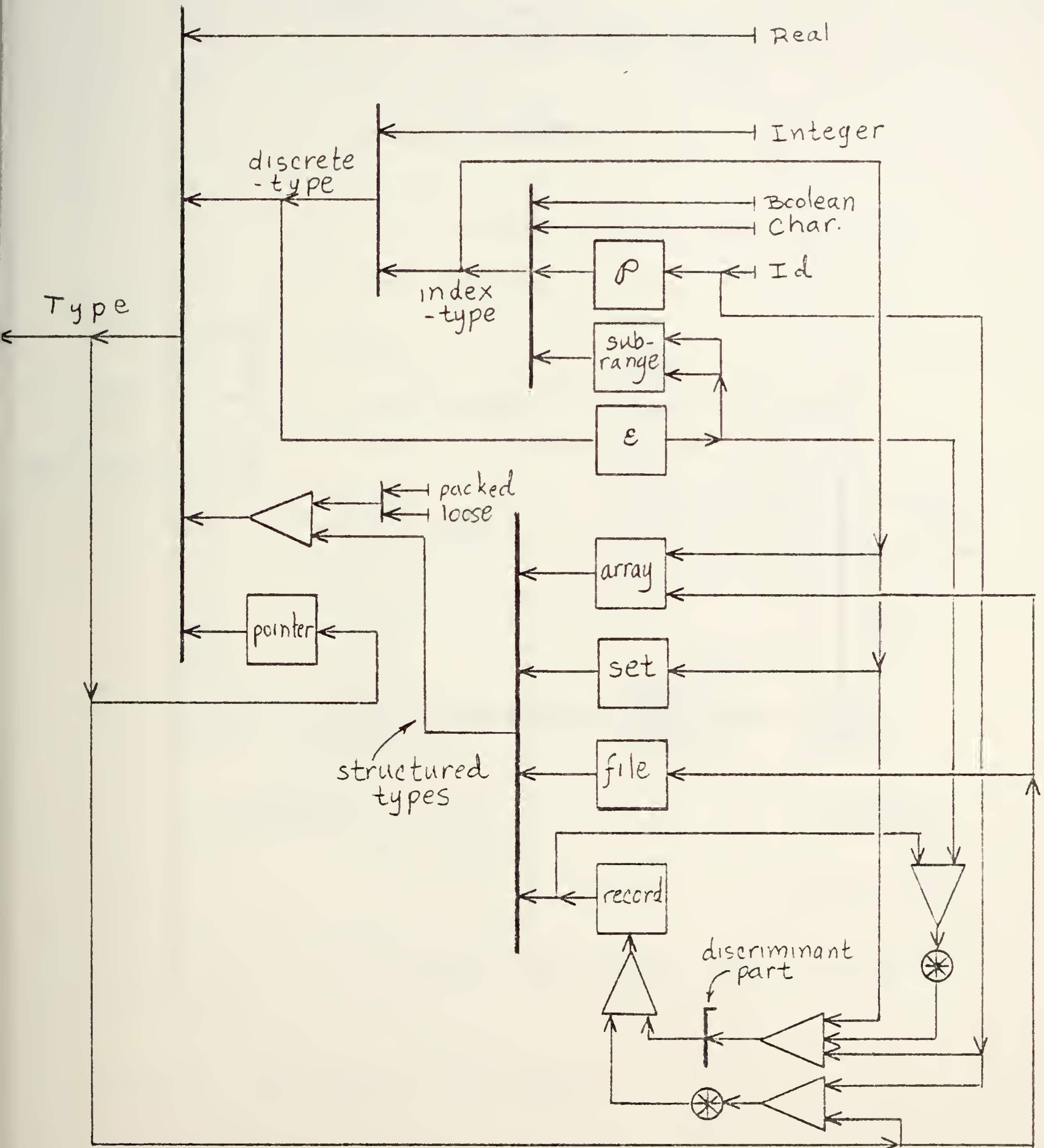


Figure 2. The Pascal Type System

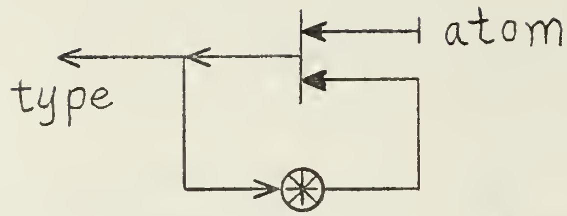


Figure 3. The LISP Type System

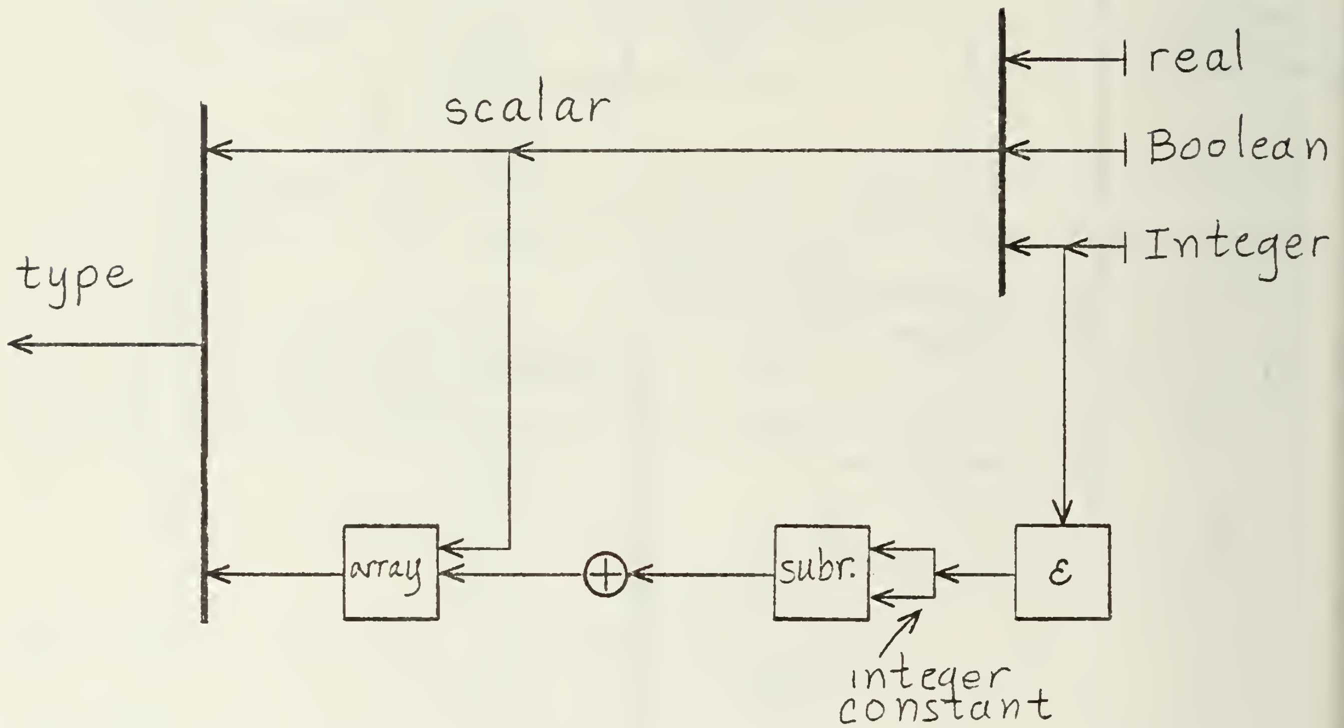


Figure 4. The Algol-60 Type System

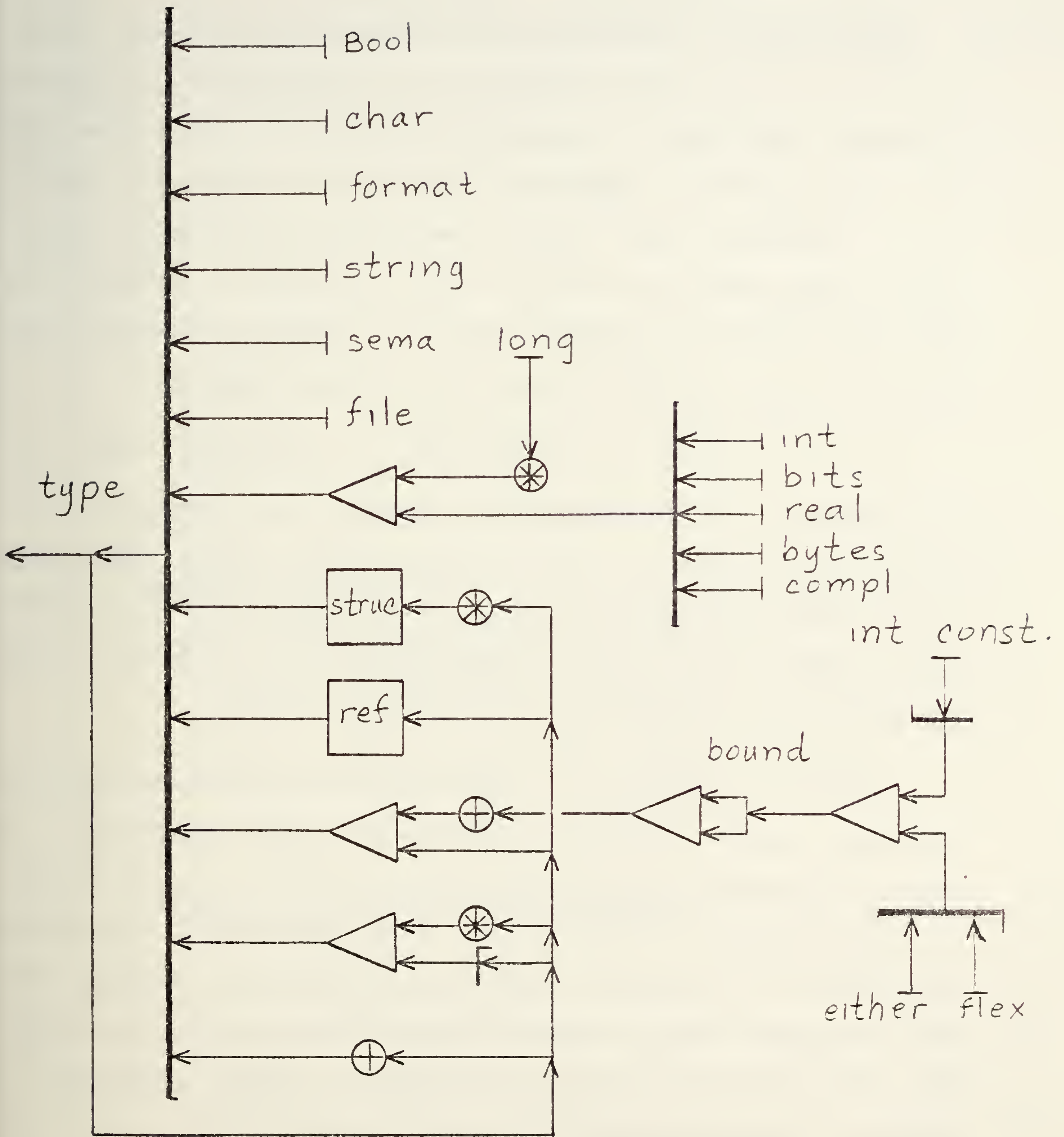


Figure 5. The Algol-68 Type System

5. Name Structures

Next, we will demonstrate the application of these techniques to the name structures, another subsystem of programming languages. The name structures of programming languages are often described by terms such as "block-structured", "monolithic", "disjoint", etc. To get a better grasp on these structuring techniques we must ask, "What is being structured?" To put it more precisely, "What relation or relations are being controlled by the structuring mechanisms in question?"

For name structures this relation is visibility, that is, the relation that holds between a binding and a use of an identifier when that use can refer to that binding. Thus, the primitives from which names structures are assembled are bindings and uses of identifiers, and the constructors used to assemble these structures are mechanisms such as block structure.

How can we abstract the name structures from a programming language? Again, we can use syntax as a guide. In Figure 6 we show the fragments of Algol-60 syntax relevant to visibility. Irrelevant parts of the syntax have been elided. Each string generated by this grammar (ignoring reordering of declarations, etc.) defines a unique name structure, i.e., structural arrangement of visibility relations. In Figure 7 we have formulated a semantics oriented grammar for these relations.

```
<identifier> ::= ....

<block> ::= <block head>; <compound tail>

<block head> ::= begin <declaration> | <block head>; <declaration

<compound tail> ::= <statement> end

                  | <statement>; <compound tail>

<program> ::= <block> | <compound statement>

<procedure declaration> ::= [<type>] procedure

                            <proc.heading> <proc.body>

<proc.heading> ::= <proc. identifier> <formal par.part>;

<formal par.part> ::= ( <identifier> ,... )

<declaration> ::= <proc.decl.> | <other decl.>
```

Figure 6. A Fragment of Algol-60

```
program:  executable

block:  scope (declaration+, executable)

declaration:  simple-decl | proc-decl

proc.decl:  identifier X scope (simple-decl*, executable)

simple-decl:  identifier

executable:  {identifier | block}*
```

Figure 7. The Algol-60 Name System

Notice that, from the visibility standpoint, a procedure declaration is the same as a block; they both bind local identifiers and delimit a scope. Figure 8 shows the Algol-60 name system in diagrammatic form. The following figures (9-11) show the name systems of the lambda calculus, FORTRAN and Pascal.

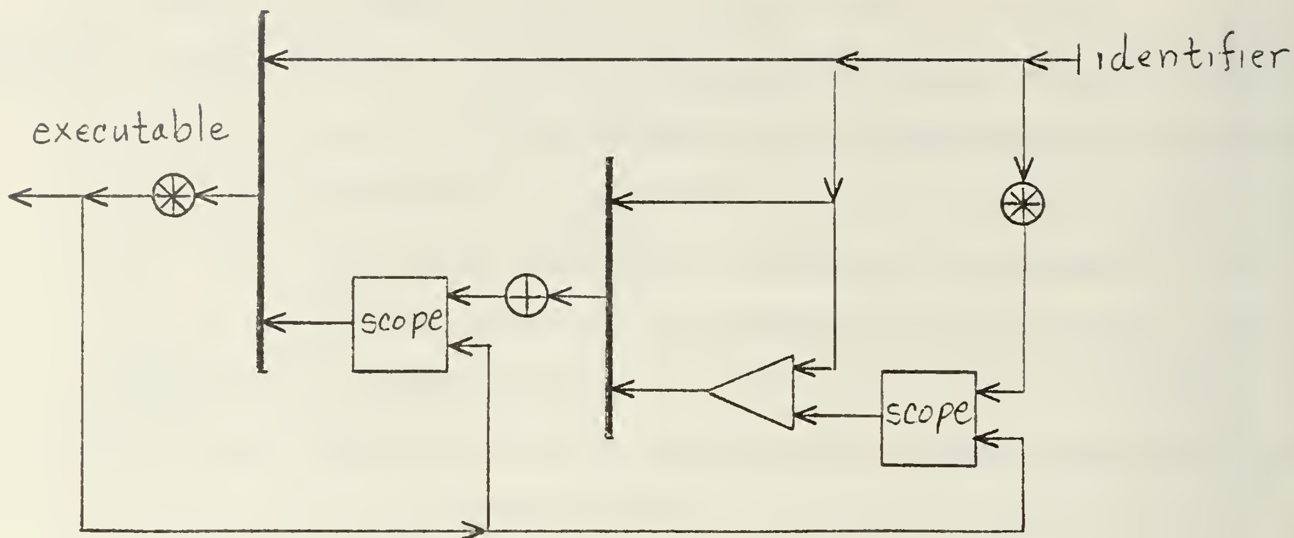


Figure 8. The Algol-60 Name System

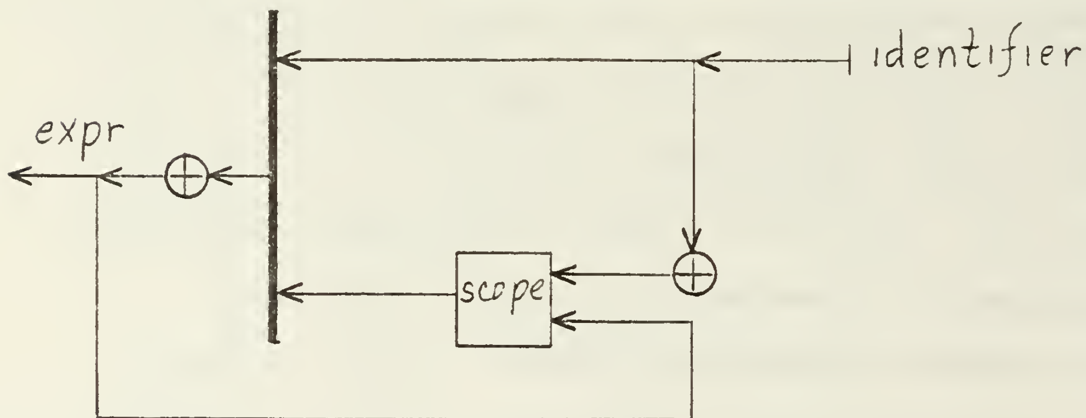


Figure 9. The Lambda-Calculus Name System

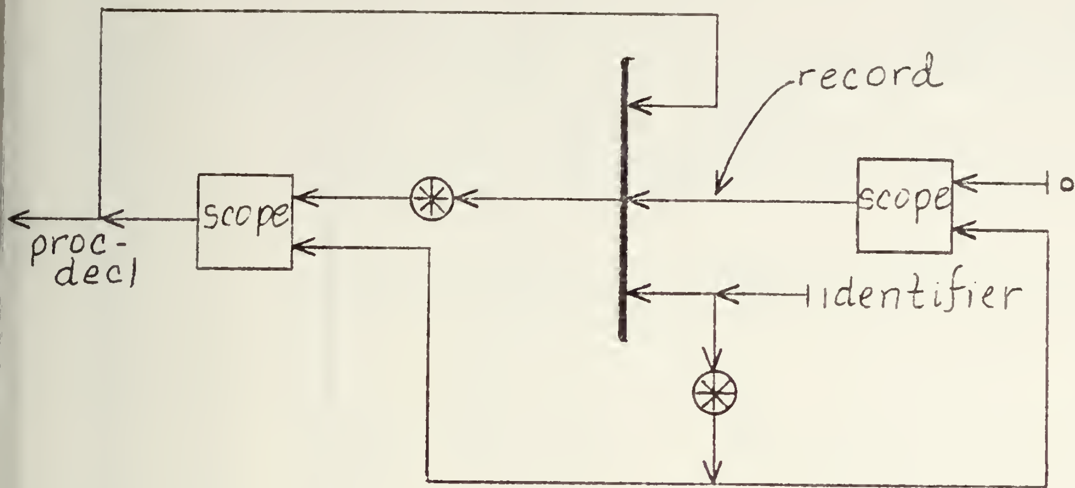


Figure 10. The Pascal Name System

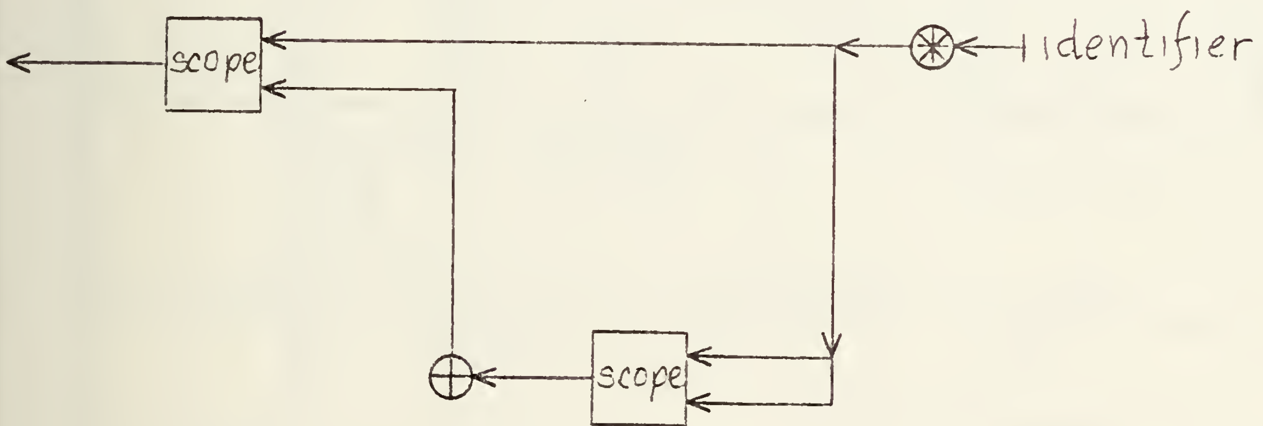


Figure 11. The FORTRAN Name System

In the latter case (Pascal), note that we have analyzed the record declaration as a scope defining (or name grouping) constructor. Figure 12 compares the complexities (as measured by edge-count) of these name systems along with the complexities of

their type systems.

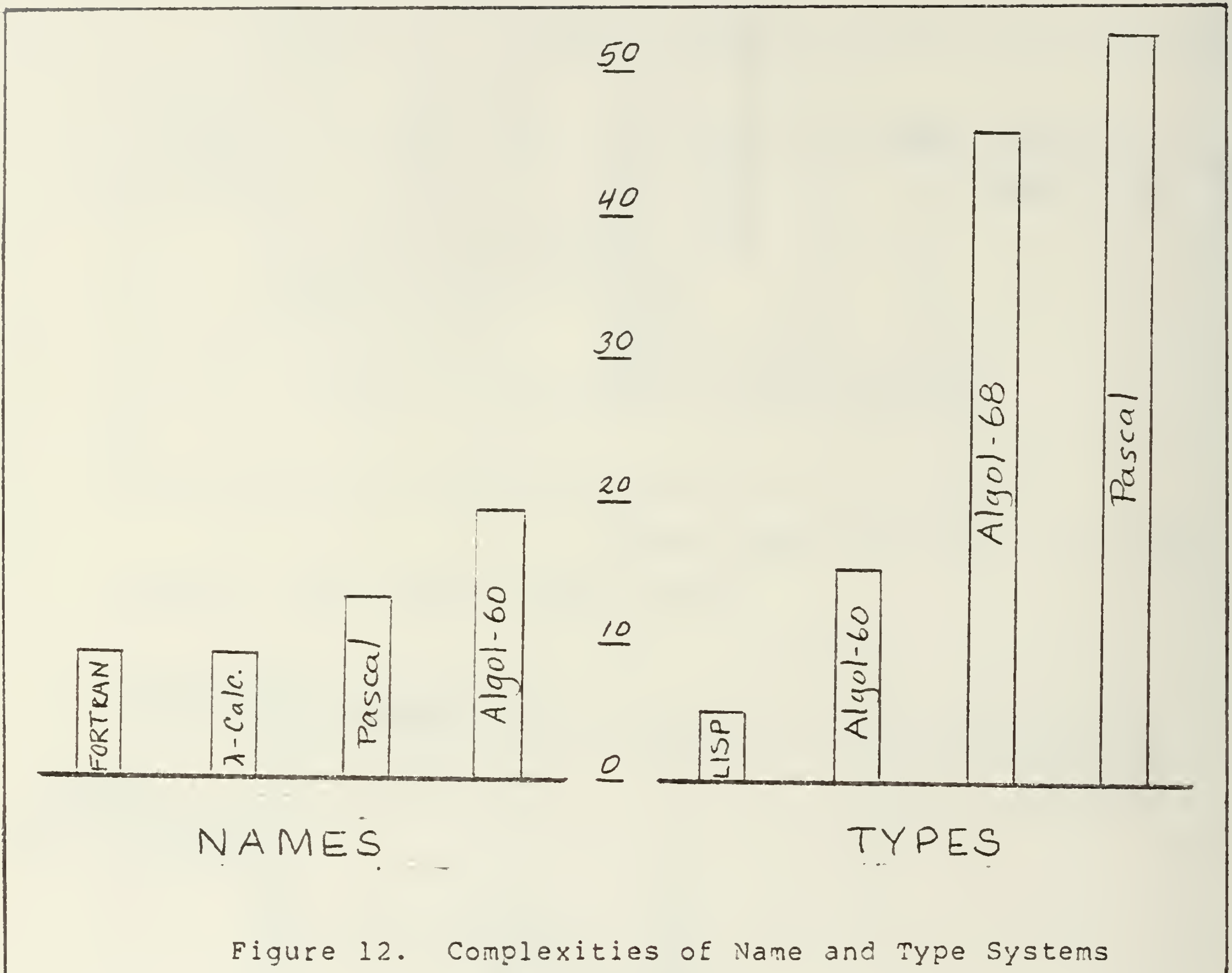


Figure 12. Complexities of Name and Type Systems

6. Control Structures

Control structures are analyzed in the same way as the other structures. These are reflected in the equations and structure diagrams shown in Figures 13-16.

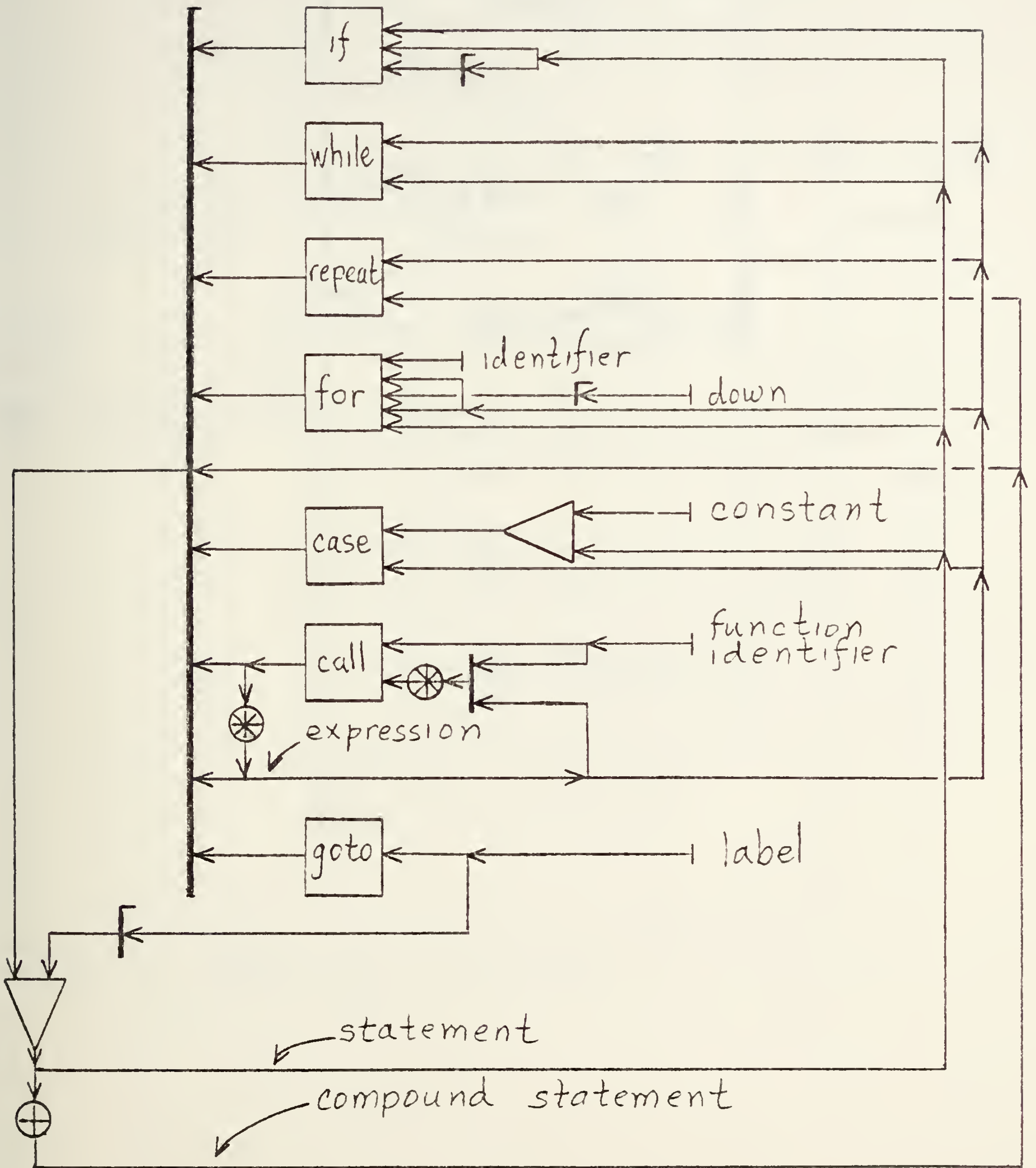


Figure 13. Pascal Control Structures

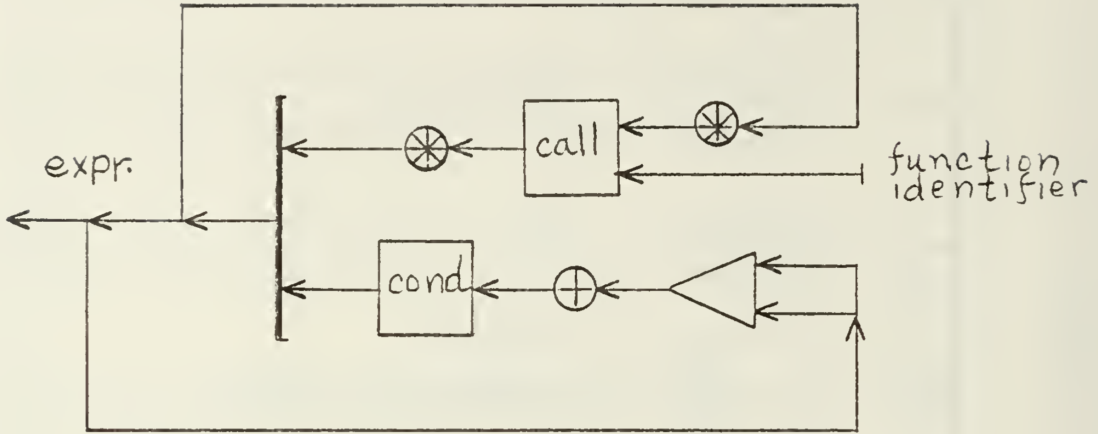


Figure 14. LISP Control Structures

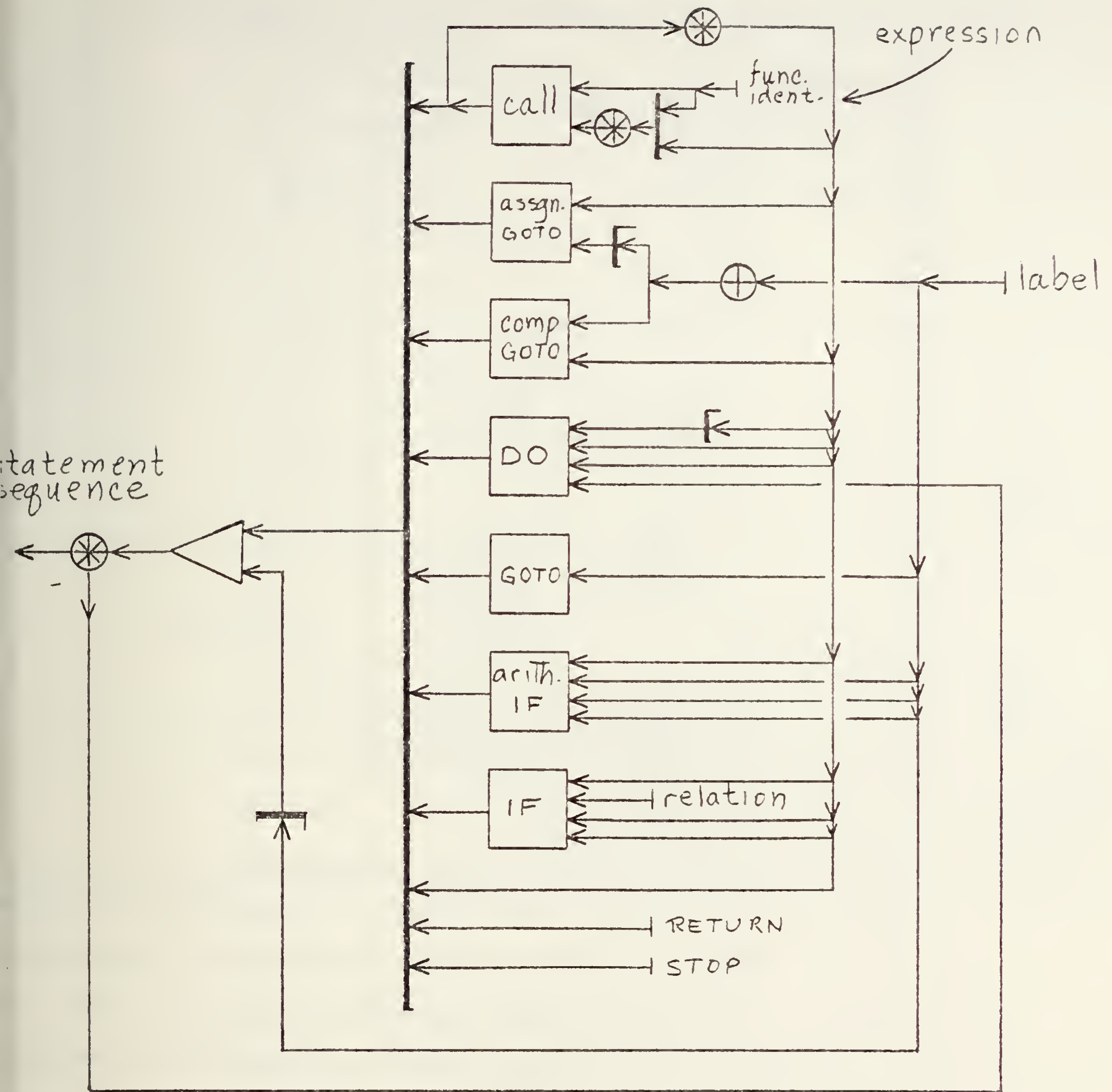


Figure 15. FORTRAN Control Structures

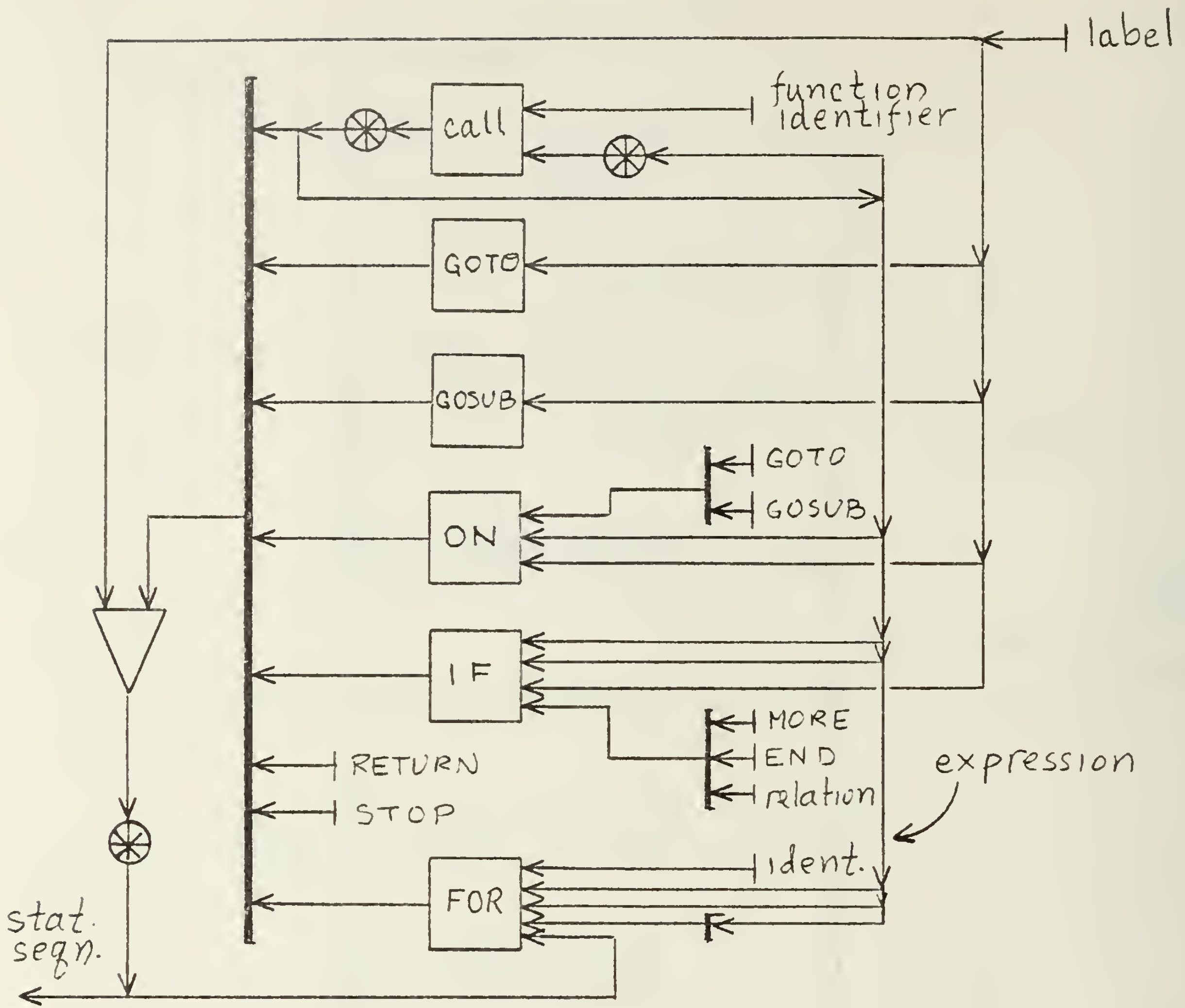


Figure 16. BASIC Control Structures

Consider Pascal; the relevant parts of the grammar are shown in Figure 17. These diagrams are somewhat deceptive because they do not reflect the extraordinary complexity introduced into the control structures by the goto statement. An analogous complexity is caused in data structures by the pointer construct. These are both examples of non-local references, whose proper treatment

```
simple-statement:  assign-stat | proc-stat | goto-stat | empty

assign-stat:  expr

function-desig:  call (fid, exprlist)

exprlist:  expr*

expr: function-desig*

proc-stat:  call (fid, {expr | fid}*)

goto-stat:  goto (label)

statement:  [label] x unlab-stat

unlab-stat:  simple-statement | struc-stat

struc-stat:  comp-stat | cond-stat | rep-stat | with-stat

comp-stat:  statement+

cond-stat: if-stat | case-stat

if-stat:  if (expr, stat, [stat])

case-stat:  case (expr, case-list-element+)

case-list-element:  const+ x statement

rep-stat:  while-stat | repeat-stat | for-stat

while-stat:  while (expr, stat)

rep-stat:  rep (stat+, expr)

for-stat:  for (id, forlist, stat)

forlist:  expr x [down] x expr

with-stat:  with (expr+, stat)
```

Figure 17. Pascal Control Structure Grammmar.

---

remains an open question.

7. Conclusions

The techniques we have described provide a simple, visual method of comparing the structuring methods provided by programming languages. Languages can often be ranked as to their structural complexity by comparing the complexity of their structural grammars or structure diagrams. In addition, the diagrams allow the language designer to appraise the regularity or irregularity of a structural subsystem and to identify areas where they can be simplified.

Of course, it is very desirable to be able to quantify these ideas, and there are many approaches to this quantification. One of the simplest, which was used in this paper, was to count the number of edges in the graph, since this reflects the dependencies within the system. In the cases we have investigated, this metric agrees with our informal evaluation.

These are, of course, other graph theoretic measures that can be applied, for instance, variants of McCabe's Cyclomatic Number [3], although which is the best remains an open question. It is also possible to apply the measures of Halstead's "Software Science" [1] to either the structural grammar or the structure diagrams. This has also been tried, but this work is still in progress [2].

Although the proper measure to be applied remains an open problem, the representation of structures in a measurable form such as the structure diagrams, is a first step towards

development of these metrics. Future research will attempt to refine the analysis of structures and their representation as graphs, and will attempt to develop appropriate measures of their complexity.

8. References

[1] Halstead, M.H., Elements of Software Science, New York, New York: Elsevier, 1977.

[2] MacLennan, B.J., Investigation of System Complexity, Naval Postgraduate School Computer Science Department Technical Report NPS52-81-010, 1981.

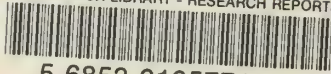
[3] McCabe, T.J., A complexity measure, IEEE Transactions on Software Engineering, 2, 4 (December 1976), pp 308-320.

[4] Wirth, N. The Programming Language Pascal (Revised Report),

INITIAL DISTRIBUTION LIST

Defense Technical 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 52Bz 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
Chief of Naval Research 800 N. Quincy Street Arlington, VA 22217	2

U199972



U199972