CONCEPTS AND STRUCTURES IN PROGRAMMING LANGUAGES

Peter Wegner

Technical Report
No. 68-14
April 1968

Department of Computer Science
Cornell University
Ithaca, New York 14850

Notes for a talk given at the San Francisco Symposium on the Effective Use of Higher Level Languages, April 19, 1968.
CONCEPTS AND STRUCTURES IN PROGRAMMING LANGUAGES

Peter Wegner
Cornell University
April 1968

Table of Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1.</td>
<td>Interpreters and Compilers</td>
<td>2</td>
</tr>
<tr>
<td>2.</td>
<td>Information Structure Models</td>
<td>3</td>
</tr>
<tr>
<td>3.</td>
<td>An Information Structure Model for Computers</td>
<td>5</td>
</tr>
<tr>
<td>4.</td>
<td>An Information Structure Model for Programming Languages</td>
<td>7</td>
</tr>
<tr>
<td>5.</td>
<td>Information Structures in FORTRAN</td>
<td>8</td>
</tr>
<tr>
<td>6.</td>
<td>Information Structures in ALGOL</td>
<td>11</td>
</tr>
<tr>
<td>7.</td>
<td>Syntax and Semantics</td>
<td>17</td>
</tr>
<tr>
<td>8.</td>
<td>Symbol Tables and Macros</td>
<td>19</td>
</tr>
<tr>
<td>9.</td>
<td>Binding Time</td>
<td>21</td>
</tr>
<tr>
<td>10.</td>
<td>Information Structures in SNOBOLAI</td>
<td>26</td>
</tr>
<tr>
<td>11.</td>
<td>LISP and the LISP APPLY Function</td>
<td>29</td>
</tr>
<tr>
<td>References</td>
<td></td>
<td>32</td>
</tr>
</tbody>
</table>

Abstract

These notes develop a set of concepts for describing program structures which arise in programming languages. These concepts are then used to discuss FORTRAN, ALGOL, macro languages, languages with definitional facilities, SNOBOLA and LISP from a unifying point of view. The central concept is that of an information structure model. Emphasis is placed on the sequence of information structures generated by programs of a programming language during their execution.

Section 1 considers program representation, interpretation, and compilation. Section 2 introduces the concept of an information structure model for characterizing classes of computations. Sections 3 and 4 show that computers and programming languages are examples of information structure models. Sections 5 and 6 respectively consider the information structures associated with FORTRAN and ALGOL programs during their execution. Section 7 shows how syntax and semantics may be specified for information structure models and introduces the notion of an interpreter-interpreter. Section 8 develops information structure models for macro languages. Section 9 introduces the concept of binding time and discusses declarations and definitional facilities in programming languages. Sections 10 and 11 consider information structure models for SNOBOLA and LISP.

Computer science may be defined as the study of representation and transformation of information structures. The present approach is concerned with the representation of programs as information structures and with the transformations of these structures during execution, and directly reflects the above definition of computer science.
1. Interpreters and Compilers

A program in a programming language can be defined either in terms of what it does or in terms of how it is represented.

Mathematically a program in a programming language is a realization of a function \( f \) which, given data \( x \) to which it is applicable, specifies a value \( y = f(x) \). The value \( y \) is said to be computed by applying the function \( f \) to its data.

When considering the mechanical evaluation of functions both the program and its data must be represented in an information storage medium. Written programs and data are represented by character strings on a sheet of paper. Programs and data in a computer are represented by strings of binary digits.

If \( \text{rep}(f) \) is a representation of a function \( f \) as a program and \( \text{rep}(x) \) is a representation of the data in the form required by \( \text{rep}(f) \), then the representation \( \text{rep}(y) = \text{rep}(f)(\text{rep}(x)) \) is computed by applying \( \text{rep}(f) \) to \( \text{rep}(x) \). A mechanical prescription for applying programs to their data is called an interpreter. When executing programs on a digital computer the interpreter is the processing unit of the computer. However, an interpreter may be implemented by manually performing the program execution steps. The process of interpretation is illustrated in figure 1.

![Figure 1: Application of Programs to their Data.](image)

Programs written by the programmer must be translated into the machine language of a computer before they can rapidly be executed. Translation of

A function \( f \) is a rule of correspondence between a set \( X \) of argument (data elements) and a set \( Y \) of values (results), which for every argument \( x \in X \) specifies a value \( y \in Y \). The set \( X \) is said to be the domain of the function and the set \( Y \) is said to be the range of the function.
programs from one language (the source language) into another language (the target language) is performed by programs called compilers, as illustrated in figure 2.

![Figure 2. A Compiler.](image)

Compilers are important technologically since they allow programs to be automatically converted into a form in which they can rapidly be executed. However, it is felt that the study of compilation has been overemphasized in the literature. The present approach deemphasizes compilation, and emphasizes the sequence of transformation that a program and its data undergo during execution.

2. Information Structure Models

A character string representing a program and its data is an example of an information structure.* We shall be concerned not only with individual information structures, but also with sets of information structures.

The set of all programs in a programming language are a set of information structures. The set of all programs with their data in a given programming language are also an example of a set of information structures. We shall be concerned with sets of information structures determined by programming languages.

Let \( I_0 \) be an information structure representing a program \( P \) and its data \( D \). The computation which computes the value \( V = P(D) \) can be broken down into a sequence of steps corresponding to primitive instructions. Each primitive instruction \( p \) transforms the information structure on which it

---

* An information structure can be defined as a finite string of characters over a finite alphabet. The strings considered here are composed of meaningful substrings which may in turn contain meaningful substrings. The term information structure is used in place of the term string in order to emphasize that the strings being considered have a definite internal structure.
operated and produces a new information structure. A finite computation
in which \( n \) primitive instructions are executed can be represented as fol-
lows:

\[
I_0 P_1 I_1 P_2 I_2 \ldots P_n I_n
\]

\( I_0 \) is referred to as the initial representation of the program and its
data. \( I_n \) is referred to as the final representation. The sequence of
information structures \( I_0, I_1, \ldots, I_n \) are snapshots of the state of the com-
putation at successive points during its execution, and are referred to as
instantaneous descriptions.

A programming language is defined by a set of information structures to-
gether with a set of interpretation rules corresponding to primitive instruc-
tions which specify transformations of information structures. A computer
may similarly be identified with the set of all information structures that
it can contain and a set of interpretation rules corresponding to primitive
instructions for transforming information structures.

Both a programming language and a computer defines a class of computation
A specification of a class of computations in terms of information structure
transformations will be called an information structure model.

An information structure model is a pair \((I, P)\) where \( I \) is a set of
information structures and \( P \) is a set of transformations which operate
on elements of \( I \).

A finite computation in an information structure model consists of a
sequence \( I_0 P_1 I_1 P_2 I_2 \ldots P_n I_n \) where \( I_0 \in I \) is referred to as an initial
representation, and \( I_{j+1} \) is obtained from \( I_j \) by execution of a primitive
instruction \( i \in P \).

The set \( I \) is the set of initial representations of computations in
the information structure model. Operations of \( P \) may in general produce
instantaneous descriptions \( I_j \) which are not in the set \( I \). The closure
\( \text{cl}(I) \) of \( I \) is defined as the set of all instantaneous descriptions \( I_j \)
which can be generated from \( I \) by finite sequences of operations of \( P \).
3. An Information Structure Model for Computers

Both digital computers and programming languages can be characterized by information structure models. An information structure model for a digital computer with a processing unit, a memory with n memory registers, and no input-output facilities is illustrated in figure 3.

![Diagram of a simple computer with processing unit and memory registers](image)

Figure 3. A Simple Computer.

Let \( s_0 \) be the initial configuration in the processing unit, \( S \) be the set of all possible processing unit configurations, \( T \) be the set of all memory register configurations, and \( \tau \) be the set of all \( n \)-element strings of \( T \).* The set \( I \) of all initial representations can be characterized as triples of the form \( (s_0, \tau, 1) \) where the first component specifies the state of the processing unit, the second component specifies the contents of the tape, and the third component specifies the register pointed to by the instruction pointer. The set \( cl(I) \) can be specified by triples of the form \( (s, \tau, r) \) where \( s \in S \) is a processing unit state and \( r \) is one of the integers \( 1 \) through \( n \). Instructions \( f \) of the computer can be specified in terms of how they transform instantaneous descriptions of the form \( (s, \tau, r) \) into new instantaneous descriptions.

In specifying primitive instructions more precisely the format of the information structures \( s \) and \( \tau \) must be specified. The information structure \( s \) associated with the processing unit usually contains an accumulator field \( AC \), and a multiplier quotient field \( MQ \). The information

---

*Readers familiar with automata theory will recognize that this definition parallels the definition of automata. Definition of classes of automata as information structure models are given in (2).
structure \( t \) consists of a sequence of information fields \( t_1 t_2 \ldots t_n \) each of which may contain a data item or an instruction. If the information field \( t_i \) represents an instruction, the the format consists of an operation field \( OP \) followed by address fields \( A \) and index register fields \( i \).

If the set \( F \) of transformations is taken to be the set of operation codes of the computer, then each \( f \in F \) is a parameterized operation in the sense that the transformation it determines depends on parameters such as the address field of the instruction. An instruction such as "ADD \( A_i \)" is an instance of ADD whose action depends both on explicit parameters such as \( A \) and \( i \), and on implicit parameters such as the accumulator.

The operations \( F \) of an information structure model \((I,F)\) are generally parameterized transformation rules, requiring the specification of explicit and implicit parameters in order to determine the transformation to be performed.

The interpretation rule associated with an information structure model \((I,F)\) may be specified as a conditional expression of the form "if \( p_1 \) then \( A_1 \) else if \( p_2 \) then \( A_2 \ldots \) else if \( p_n \) then \( A_n \)\", where each \( p_i \) is a parameterized primitive instruction and \( A_i \) is the action to be taken when that parameterized instruction is recognized.

The interpretation rule for the above computer model might by specified by a conditional expression of the form

\[
\text{if } r(t) = "\text{ADD } A_i" \quad \text{then} \quad [\text{AC}=\text{AC}+\text{C}(A+C(i)) \quad \text{and} \quad r=r-1] \\
\text{else if } r(t) = \text{MULT } A_i \quad \text{then} \quad [\text{perform multiplication action}]
\]

where \( r \) applied to \( t \) selects the content of the \( r \)th memory register.

The computer model described above has three information components \( F_j, \text M \) and \( r \). A formal information structure model for a computer would specify how each of the primitive instructions changes the information components \( F_j, \text M \) and \( r \) as a function of its parameters.

Computers usually have additional information components, such as an input stream \( \text{IN} \), an output stream \( \text{O} \), and an auxiliary memory structure \( \text{A} \) which may in turn be arbitrarily complex. Primitive instructions exist
for performing information structure transformations on these additional information components. The set I of the corresponding information structure model can be specified in terms of the six components FU, N, r, IN, O, A. Different elements f ∈ F transform different information components. All primitive instructions may be specified in terms of how they change these six information components.

4. An Information Structure Model for Programming Languages

The component I of an information structure model for a given programming language can be characterized by a program component P, a data component D, and a stateword component W which is the analogue of a processing unit in an actual computer.

![Diagram of Information Components]

Figure 4. Information Components for a Programming Language.

Each of the three components W, P and D has a characteristic structure and mode of access. Transformations of the W component are usually restricted so that it does not change its size during execution. The P component is often implemented reentrantly, so that it cannot be modified during its execution. The D component contains the data structures being created, deleted and modified during their execution, and is therefore the component of greatest interest.

Programming languages can be characterized by the kinds of transformations permitted in the D component. FORTRAN is carefully designed so that the structure of the D component in cl(I) is fixed prior to execution. ALGOL is designed so that the D component is essentially a stack with last-in-first-out order of creation and deletion of information structures. List processing and string manipulation languages require the D component to
take an even more flexible form, with unpredictable creation and deletion requirements. One method of organizing D so that information structures can be flexibly created and deleted is to organize D as a (free storage) list structure from which blocks can be returned.

The information structures which arise during the execution of programs in FORTRAN and ALGOL will now be examined.

5. Information Structures in FORTRAN

A FORTRAN program consists of a main program and a number of subroutines. The main program and subroutines will be referred to as function modules since they are modules of the function which constitutes the complete program. The data associated with a FORTRAN program may either be local data, local to a function module or COMMON data, in one of a number of named COMMON data storage areas. The term "block" will be used to denote "function module or COMMON data storage area."

FORTRAN is organized so that the storage requirements of each block can be determined prior to program execution, and remain fixed throughout program execution. This implies that an upper bound on the size of all data structures must be specified prior to execution, and that creation and deletion of data structures with unpredictable storage requirements is not permitted. This in turn leads directly to the source language restrictions that upper bounds on array size must be specified prior to execution and that recursive subroutines are not permitted.

The information structures associated with a given function module during execution consist of a program part containing the sequence of instructions to be executed, a local data part containing the values of variables local to

*Creation and deletion of data structures by overlay of a data structure defined prior to execution is permitted. This is the way list processing is implemented in the FORTRAN implementation of SLIP.*
the program unit, and a working space part which is used during execution for temporary storage.

<table>
<thead>
<tr>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Data</td>
</tr>
<tr>
<td>Working Space</td>
</tr>
</tbody>
</table>

Figure 5: Principal Information Structure Components of a Function Module During Execution.

The storage requirement of the program part, local data part and working part of a function module must be known prior to execution. This allows internal references within a function module to be specified by relative addresses relative to the beginning of the function module.

Although relative addresses within a function module are assumed known prior to execution, it is assumed that the origin of function modules and COMMON data blocks remain unknown until execution begins, and may vary for different instances of execution. Thus internal cross reference within a function module must be represented by relative addresses (relocatable addresses) relative to an origin which is supplied at load time immediately prior to execution.

FORTRAN allows function modules to be independently compiled into an information library. The origin of function modules and COMMON data blocks is assumed to be unknown during compilation, and becomes known only at load time when programs are loaded into the memory and absolute addresses are determined.

It is usually assumed in FORTRAN that the absolute addresses of function modules are fixed during execution of a given program, so that referencing during execution could in principle be performed by absolute addresses substituted for relative addresses at load time. However, it is usual to
perform cross referencing during execution by indirect addressing a transfer vector as indicated in reference 1 section 1.8.

During execution of a given function module, cross references may be made to other function modules and to COMMON data blocks as illustrated in figure 6.

![Figure 6. Program with two Function Modules and one COMMON Data Block.](image)

The FORTRAN source language is carefully constructed so that each function module can determine prior to execution the relative addresses within COMMON data blocks of all COMMON variables that it uses. Thus COMMON variables can be represented prior to execution as relative addresses relative to an origin that is supplied when execution begins. If the language allows only a single COMMON data block as in FORTRAN II, then all COMMON data references are relative to the same origin, and it is convenient to represent relative addressing information prior to execution by a three element code indicating no relocation, function module relocation or COMMON relocation. However, when there may be an arbitrary number of named COMMON blocks, COMMON relocation must be performed by a transfer vector technique.

Cross reference between function modules is restricted in FORTRAN to entry points of function modules. The addresses of entry points to function modules are supplied at load time in a manner similar to that for origins of named data blocks. Cross references between function modules could be implemented by absolute addresses substituted for symbolic references at load time, but are usually implemented by indirect addressing through base registers using transfer vector techniques.
Cross referencing between function modules in FORTRAN is restricted to subroutine calls. The method of access of FORTRAN subroutines to their parameters will be described in some detail so that it can be compared with that used in ALCOL.

FORTRAN subroutines access parameters by relative addressing relative to their point of call. It is usual to store addresses of successive parameter values immediately following the point of call.

```
TSR S, A1, A2, A3
```

This segment of code might represent a transfer (TSR) to a three parameter subroutine S. The addresses A1, A2, and A3 of the three parameters are stored in registers immediately following the point of call.

If an actual parameter of a FORTRAN subroutine is an expression, then an instruction sequence for computing the expression is compiled immediately preceding the point of subroutine call. The compiled instructions store the value of the expression in a location pointed to by the parameter address in the locations immediately following the TSR instruction.

This method of accessing parameters is referred to as a call by reference, and is the only method available in FORTRAN. Other methods of accessing parameters are illustrated in the next section.

6. Information Structures in ALCOL

An ALCOL program consists of a block which may have other blocks nested inside it. A block consists of the word `begin` followed by a sequence of `declaration` followed by a sequence of `statement` followed by the symbol `end`. The declarations may include data declarations for identifiers of the type `real`, `integer`, and `boolean`, array declarations for arrays of data quantities, procedure declarations for procedures, implicit `label` declarations, and

Labels are recognized by context rather than by explicit declaration. It is sometimes convenient to think of declarations for labels as being explicitly created during a "compilation" phase, so that all identifiers can be treated in a uniform manner.
switch declarations for arrays of labels. The statements may include
assignment statements for assigning the values of variables, and conditional
and iteration statements. A block is itself an example of a statement,
so that a block may have blocks nested inside it to an arbitrary level.

In FORTRAN the information structures in which values of variables are
stored during execution are assumed to be created prior to execution and to
endure throughout the lifetime of the program. In ALGOL the information
structures in which values of identifiers are stored are assumed to be cre-
during execution on entry to the block in which they are declared, and
deleted on exit from the block in which they are declared. It will be ass-
that, on entry to a block, a composite information structure is created
which contains space for values of all identifiers declared in the block
head, together with pointer and bookkeeping information further discussed
below. The information structure created on entry to a block head will be
referred to as its activation record.

Entry to and exit from blocks is always in a last-in-first-out order,
so that block activation records are always created and deleted in a last-
in-first-out order and can therefore be stored in a stack.

\[
\begin{array}{c}
B \\
B1 \\
B2 \\
B3 \\
\end{array}
\quad \leftarrow 
\begin{array}{c}
P \\
Q \\
\end{array}
\begin{array}{c}
B2 \\
B1 \\
B \\
B3 \\
\end{array}
\]

\( a) \) Fixed Program \hspace{2cm} \( b) \) Execution at \( P \) \hspace{2cm} \( c) \) Execution at \( Q \)

**Figure 7. Activation Record Stack During Execution.**
Figure 7a shows an ALGOL program with an outer block B in which blocks B1 and B3 are nested. B2 is in turn nested in the block B1. On linear execution of this program, entry to the block B causes creation of an activation record for B. Successive entry to B1 and B2 causes creation of the activation records for B1 and B2. When execution is at the point P, the activation record stack is as shown in Figure 7b. Exit from B2 and B1 deletes activation records for B2 and B1. Entry to B3 then creates an activation record for the block B3. When execution is at Q, the activation record stack is as shown in Figure 7c. Exit from B3 and B deletes the activation records for B3 and B, so that the stack is empty on completion of the program.

Whereas a FORTRAN program is composed of a set of independent function modules, an ALGOL program is composed of a set of nested function modules. The principal function modules of ALGOL are the block and the procedure. Procedure declarations consist of a procedure heading in which the name and type of the procedure and formal parameters are specified, and a procedure body which consists of a single statement (usually a block). Procedures are assumed to be created on entry to the block in which they are declared, and deleted on exit from the block in which they are declared, just as other identifiers declared in block heads. A procedure may be called only in the block in which it is declared. A procedure call creates an activation record for the parameter and linkage information just as a block creates an activation record for its declarations and linkage information.

Blocks and procedure function modules differ in their mode of entry and exit. Blocks are entered in line, and exit from a block is normally through its end statement. Procedures are entered by an explicit function call and return control to the statement following the function call on exit. However, both procedures and blocks cause an activation record to be created on entry and deleted on exit. Moreover, entry to and exit from procedures is always in a last-in-first-out order, and entry to and exit from blocks and procedures
is initially in a last-in-first-out order. This allows activation records for blocks and procedures to be stored in a single run-time stack.

The set of identifiers which may meaningfully be used at a given point in an ALGOL program include identifiers declared in enclosing blocks and parameters of enclosing procedure headings. The static level of nesting at a given point of the ALGOL program is defined as the sum of block heads and procedure headings in which that point of the program is enclosed. When execution is at level of nesting k there are precisely k levels of nomenclature at which names of identifiers may be defined. The set of names accessible at a given point of execution is sometimes referred to as the environment of that point of execution.

The dynamic level of nesting at a given point of execution is defined to be the sum of the blocks and procedures which have been entered but whose execution has not yet been completed. This corresponds precisely to the current number of activation records on the activation record stack.

The static level of nesting at a given point of the source program is an invariant of the source program. The dynamic level of nesting at a given point of the source program may differ on different occasions. For example a call of a procedure P from different internal levels of nesting will give rise to different dynamic levels of nesting. A recursive procedure enters a deeper dynamic level of nesting on each recursive call.

During execution, the dynamic sequence in which partially executed functions have been entered may be remembered by a chain of links referred to as a dynamic chain, which links backwards through successive activation records in the activation record stack. The sequence of activation records which correspond to statically enclosing function modules of the source program can be remembered by a static chain which links each activation record to the most recent activation of the next physically enclosing one.
Figure 7a, illustrates an ALCOL program consisting of a block B with a procedure declaration P having body B2, and an inner block B1 which contains a call to P. When execution is in the block B2 at the point X, the static nesting level is 3 and the dynamic nesting level is 4. The activation record stack with its static and dynamic chains is indicated in figure 7b.

Parameters in ALCOL60 can be called either by value or by name. A parameter call by value corresponds to evaluation of the parameter at the time of entry to the program and substitution of the evaluated parameter for all instances of use. A parameter call by name, corresponds to evaluation of the parameter every time it is used during execution. Parameter calls by value are implemented by storage of the value in the procedure activation record and reference to the local value on use. Parameters called by name are implemented by procedure calls to a function for evaluating the parameter in the context of the point of call, and result in creation of a parameter activation record during parameter evaluation.

Functions may be called in expression evaluation during execution of assignment statements. Function modules which occur in expressions are dynamically nested at their point of call. It is therefore convenient to store values of variables which occur during expression evaluation at the top of the run time stack. Execution of blocks, procedures and expressions are
all dynamically nested with respect to each other, so that information structures in all three categories can share a single stack.

The only class of run-time data structures in ALGOL whose values cannot be stored in a stack during execution are _own_ variables. _Own_ variables have a lifetime which endures between activations of the block in which they are declared. The _own_ storage concept becomes confusing when _own_ variables may change in size during execution as in arrays with dynamic bounds and recursive procedures. It is assumed below that the storage requirements for all _own_ variables can be determined at translation time.

The details of an implementation of ALGOL60 along the above lines are discussed in (6) and in chapter 4 of (1). An information structure model of ALGOL60 which captures the characteristics of the above implementation can be formulated as follows:

Instantaneous descriptions in ALGOL can be denoted by sextuples \((P,W,S,IN,O,X)\) where \(P\) is a program component, \(W\) is the state word component, \(S\) is the stack component, \(IN\) is the input component, \(O\) is the output component and \(X\) is an _own_ storage component. Each of these components has a characteristic substructure and mode of access. The \(P\) component is pure procedure, and is accessed through \(W\) in a read-only mode. The \(W\) component is of fixed size and contains pointers to the current instruction in \(P\) and the current point of execution in the stack \(S\). The \(S\) component is a stack with respect to creation and deletion but allows arbitrarily deep reading and writing in the interior of the stack. The \(I\) component is a one way read-only tape and the \(O\) component is a one way write-only tape. The usual interpretation of _own_ variables results in an \(X\) component whose memory requirements do not change during execution.

Execution of an ALGOL program consists of scanning successive primitive operator-operand combinations of \(P\) pointed to by \(W\) and performing transformations determined by the scanned string. The transformations of greatest interest are transformations of the stack component. The stack component is augmented by an activation record on entry to a block or procedure or on evaluation of a parameter called by name. An activation record is deleted
on exit from a function module. Creation and deletion of stack elements also occurs during expression evaluation. Access to a variable in the interior of the stack occurs when a variable is first accessed during expression evaluation, and writing of a value in the interior of the stack occurring during execution of an assignment statement.

Primitive operations $f$ of an information structure model for ALCOL can be specified in terms of how they transform the above information components. An interpretation function for ALCOL execution can be specified as a conditional expression of the form "if $p_1$, then $A_1$ else if $p_2$ then $A_2$ ... else if $p_n$ then $A_n$" where $p_1, p_2, \ldots, p_n$ are parameterized information structures in the program and stateword components, and $A_1, A_2, \ldots, A_n$ specify transformations of the components $W, S, O, X$ when corresponding elements $p_1$ are recognized.

7. Syntax and Semantics

The first component $I$ of an information structure model is referred to as the syntactic component, and the second component $F$ is referred to as the semantic component. A notation for specifying the set of all elements of a class $I$ or $cl(I)$ of information structures is referred to as a syntactic notation. A specification of the set $I$ for a given information structure model in a syntactic notation is referred to as a syntax or grammar for the information structure model. The term syntax will be used also to denote a syntactic specification of $cl(I)$.

The notation of context-free languages is one of the most common notations for specifying syntax. In this notation the principal components of $I$ would be specified by concatenation, by means of a production of the form $I = p_1 \cdot p_2 \cdot \ldots \cdot p_n$. The internal structure of each of the components $p_j$ would in turn be specified by further productions until a specification in terms of primitive information components is reached.

The primitive information components may be either at the bit level or at a very high level, depending on the degree of detail at which the information structure is to be specified. In the model for ALCOL above, the
production. "T = P W S E O X" specifies the principal information components. For certain purposes the principal information components may also be regarded as the primitive information components. However, a model of greater precision is obtained by considering finer primitive information components. Thus the stack may be specified to be a sequence of activation records.

Detailed specifications of information structure models are facilitated by a language specifically designed for this purpose. A language for specifying information structure models will be called a modelling language. Work on the development of a modelling language is currently in progress at Cornell.

A modelling language must contain a syntactic specification language for specifying the sets \( E \) and \( \text{cl}(I) \), a mechanism for selecting substructures corresponding to primitive instructions, and a facility for specifying transformations of \( \text{cl}(I) \) to be performed on execution of primitive instructions.

Execution of a primitive instruction in an information structure model consists of a recognition phase during which the primitive instruction is recognized using syntactic facilities of the modelling language, followed by a transformation phase using transformational facilities of the modelling language. The execution process is syntax directed in precisely the same sense that the execution of a syntax directed compiler is syntax directed. However, information structure models are concerned with interpretation rather than with compilation, so that a specification of an information structure model in the modelling language will be called a syntax directed interpreter.

An implementation of a modelling language allows information structure models to be specified by specifying their syntactic and semantic components. In this respect it is similar to a compiler-compiler which allows specification of a compiler by means of a syntactic and a semantic specification.
However, a modelling language is concerned with specification of interpreters. An implementation of a modelling language will be called an interpreter-interpreter since it executes (interprets) interpreters specified in the modelling language.

8. Symbol Tables and Macros

A symbol table may be thought of as a function \( f \) with a finite domain and a finite range, which permits flexible modification of the rule of correspondence by modifying entries in the symbol table, and permits functions to "grow" by adding entries in the symbol table. Evaluation of the function \( f \) is performed by symbol table look up. If entries in a symbol table \( f \) have the form \( (s_i, t_i) \), then symbol table look up for the element \( s_i \) corresponds to application \( f \) to \( s_i \) and may be written as \( f(s_i) = t_i \).

Because of the flexibility of symbol tables, function modules are represented by symbol tables in many diverse applications. Information structure models for symbol tables are therefore of great importance.

The simplest functions represented by symbol tables are encoding functions, which involve transliteration of a source symbol in a source code to a target symbol in a target code. Assemblers are essentially encoders with some additional facilities. A symbol table \( f_1 \) is required for encoding operation codes and a symbol table \( f_2 \) is required for encoding addresses. The symbolic operation code is directly encoded by symbol table look-up. The rule of correspondence between symbolic and machine addresses is, however, constructed by the computer. This requires construction of the symbol table followed by its use. The assembly process is discussed in greater detail in section 2.1 of (1).

Macro assemblers are more elaborate than assemblers without macros principally in allowing a more elaborate rule of correspondence between parameterized macro names and their values than is permitted between instructions and their values. Macro definition are implemented by a growing symbol table called a macro definition table in which first components may be parameterized and second components have a considerably richer structure than second
components of operation code or address field symbol tables. Moreover, second components of the macro table are processed during table look-up, and may initiate further table look-up operations during their execution.

The format of macro calls in macro assemblers and some more general macro systems is restricted to being a name followed by parameters in parentheses. This format may be generalized by allowing a macro call to be an arbitrary pattern. Such a generalization increases the complexity of first components of the symbol table. Matching of successive first components of the symbol table is now no longer trivial but may involve a complex pattern matching process. However, the greater flexibility of format allows language structures which are not in functional notation to be defined as macros and has resulted in such macro systems being called language independent macro processors. (7)

Macro definitions may contain nested macro calls, including recursive macro calls, and may include macro definitions. Execution of a macro call may therefore result in nested macro calls to an arbitrary level of nesting, and may result in the definition of new macros during execution of macro calls at any level of nesting.

Actual parameters of macros may also contain macro calls and macro definitions. Evaluation of macro parameters may be performed either prior to macro expansion (corresponding to a call by value) or at the time of the use of the macro parameter (corresponding to a call by name). Macro definitions which occur during parameter evaluation can be defined to be either temporary macro definitions during evaluation of the current macro body or permanent definitions having the same status as definitions which occur in macro bodies.

An information structure model for macro expansion requires at least the following two run-time structures.

1. A macro definition table which is augmented or modified on macro definition.

2. A macro expansion stack which contains an activation record for each partially expanded macro, with information about actual
parameters of the partially expanded macro. If the convention is adopted that macro parameters are called by value and that macro definitions in parameters are valid only within the macro expansion, then the activation record stack for a given macro call will contain both parameter values and macro definitions executed during parameter evaluation.

The macro expansion stack is analogous to the run time stack of ALCOL. ALCOL has no analogue of the macro definition table since all created information structures are deleted on exit from the function module in whose head they are created.

When macro calls may be complex patterns, the information structure model must allow for information structures generated during the recognition process as well as for information structures generated during the expansion process. For example if the class of patterns permitted for macro calls is that specified by a context free language\(^8\), then the recognition process will require a stack for storing partially recognized subpatterns.

9. Binding Time

In procedure oriented languages like ALCOL, creation and naming of information structures is performed dynamically during execution just like assignment of a value to an information field. Creation of an information structure is accomplished by a declaration and is referred to as a declarative action. Value assignment is referred to as an imperative action.

Declarations permit attributes of an information structure that remain invariant during execution to be specified independently of attributes that are modified during execution. The name number and size of information fields of an ALCOL information structure remain invariant during its lifetime and are conveniently specified by declarations. The values of information fields are modified by imperative actions.

The moment during execution at which a given set of attributes is fixed is said to be the binding time of that set of attributes. When there is a choice of binding time of attributes of a given information structure,
the following two extreme binding strategies can be used.

1. Perform binding at the earliest possible point of time.
2. Perform binding at the latest possible point of time.

Binding of attributes as early as possible results in more efficient program execution since it saves repeated binding of attributes. Binding of attributes as late as possible allows greater flexibility in specifying the attribute. For example ALCOL parameter calls by values, which result in early binding, are more efficient than parameters called by name, which result in late binding. However, it is sometimes necessary to have the extra flexibility of parameter calls by name.

If a given attribute \( A \) is assigned the value \( V \) at time \( T_1 \) by binding strategy \( A \) and at time \( T_2 \) by binding strategy \( B \), then the two binding strategies have different effect if and only if the computation changes the value \( V \) between the time \( T_1 \) and \( T_2 \). Let \( V \) be the value of an ALCOL procedure parameter, let binding strategy \( A \) be call by value and let binding strategy \( B \) be call by name. Then \( T_1 \) is the time of entry to the procedure and \( T_2 \) is the time of use of the parameter. The two binding strategies yield different results if and only if the parameter value can change between the time of entry to the procedure and the time of use of the parameter. A procedure which may change externally-defined parameter values is said to have side effects (see chapter 4 of (1)).

If it is convenient to distinguish between three different binding times for attributes of an information structure.

1. The template binding time which binds the form of a class of information structures.
2. The creation time of an instance of an information structure whose form is defined by a template.
3. The assignment time at which values of fields associated with a given information structure are assigned.

In early programming languages templates were specified by the system and all structures were created prior to the beginning of the computation.
In ACOOL structure creation for declared identifiers occurs at the time of block entry. In FL/I\(^{(9)}\) the CONTROLLED storage allocation facility allows template definition to occur at the time of block entry, and creation of multiple instances of a structure to occur during execution by execution of ALLOCATE commands.

A function declaration defines a template for a function in precisely the same way that a data template specification defines a template for a data structure. The activation record created on function call is analogous to creation of an instance of a data structure. In both ACOOL and FL/I creation and deletion of function activations is constrained to be in a last-in-first-out order, so that activation records may be stored in a stack. Controlled storage allocation for functions would allow activation records to be created and deleted in a more flexible fashion. Some form of controlled storage allocation for functions is required for coroutines, event handlers and tasks (see section 4.10 of reference 1).

The binding of attributes of an information structure may be distributed over the template binding time, instance creation time and assignment time in a variety of ways. Some declarative languages like Dider\(^{(10)}\), adopt the extreme strategy of having only one system template (new) which allows new names of information structures to be created. Attributes are associated with a structure not at the time of its creation, but at the time of its use. This strategy is also adopted in non-declarative languages like SIMULA\(^{(11)}\), and in list processing languages. The later binding time of attributes in such languages introduces some execution time inefficiency, but allows greater flexibility in assigning and reassigning attributes of structures. One of the principal features which distinguishes list processing languages from languages for numerical computation, is the later binding time of structure attributes in list processing languages.

The question of binding time is important not only in programming languages, but also in machine organization. One of the most important developments in machine organization during the last fifteen years has
been to delay binding of data addresses until execution time, and to allow later and more flexible binding of information structures into storage structures. In early machines, absolute addressing required data addresses to be bound prior to their use. Indexing, relocation registers and base registers permitted successively more flexible binding of data addresses at the time of use. Paging techniques allowed later and more flexible binding of information structures into storage structures. Each of the above facilities introduces inefficiencies by delaying the binding time, but permits greater flexibility of computation.

The distinction between compilation and interpretation may be phrased in terms of the concept of binding time. Compilation results in binding of a source language program to its target language representation at translation time, while interpretation delays binding of the source language program till execution time. Here again, early binding results in greater efficiency, since it saves repeated binding, and late binding allows greater flexibility in introducing tracing and other modification at execution time.

The distinction between macros and procedures may similarly be stated in terms of binding time. When a procedure-oriented language contains macro facilities, then macro calls result in physical substitution of the macro definition in the program text prior to execution. Procedure calls result in simulated substitution of the procedure definition during execution. The earlier binding time of macros results in greater efficiency during execution. However, the delay of binding till execution time allows considerable greater flexibility during execution by delaying the binding of procedure parameters. The cost is normally only a small proportion of the total execution time of the procedure.

The above examples illustrate that the concept of binding time is very useful when distinctions between events occurring at different points of time in a computation are discussed. The present notes emphasize the sequence of dynamic representations of a program at successive stages of
execution, and the concept of binding time is an important one in this context.

The question of binding is relevant not only in the context of creation and deletion of information structures, but also in the context of specifying structural forms (templates) for classes of information structure. At this level, the question arises whether the structural form of program segments and data of a programming language should be specified (bound) at the time of language design or whether the programming language should contain facilities for specifying new structural forms suited to particular problem areas.

Facilities within a programming language for the creation of new structural forms are referred to as definitional facilities. Designers of a general-purpose programming language cannot provide suitable program and data structures for all problem areas in which it might be used, so that general-purpose programming languages of the future will probably contain definitional facilities. However, the choice of primitive program and data structures from which non-primitive structures are to be built up still presents a problem. Languages which contain a relatively small set of primitive program and data structures, and rely heavily on definitional facilities are sometimes referred to as core languages, while languages which attempt to provide an adequate set of primitives for general-purpose use are sometimes referred to as shell languages. Programming languages such as ALGOL and PL/I were developed as shell languages. However, a better understanding of macros allows macros to be used in developing definitional facilities. Definitional facilities have been discussed by Chastain (12), and by Galler and Perlis (13). Recently developed programming languages which contain definitional facilities include CPL (14) and APL2 (15).

The class of structures defined by a template specification serve a useful function only if operations for transforming or interpreting instances of the class of structures are defined. A template specification together with a specification of a class of operators applicable to structures
specified by the template is said to define a new type for the programming language. Primitive types of a programming language may be thought of as being defined by system templates. Languages with definitional facilities permit specification of new types during execution.

The operators associated with a new type may be derived implicitly from the template specification. Consider a template specification of the form "template T(F1,F2,F3)" where T is the name of the new type and F1,F2,F3 are names of information components of the new type. The system in which this template specification facility is embedded may automatically construct a predicate operator T for testing whether a structure is an instance of the template, and may automatically construct selectors F1,F2,F3 for selecting components of created instances of the template. In a language with definitional facilities, both implicitly and explicitly defined operators on defined data types should be permitted.

10. Information Structures in SNOBOL4. SNOBOL4(11) is a language whose structure is different from the ALGOL-FPL/I family. It does not have a block structure, and does not have scope rules for nomenclature. However it has a large number of primitive data types, facilities for data definition, and many interesting data manipulation facilities. An information structure model for SNOBOL4 indicates how differences in the source language are reflected in the information structures generated during execution.

The primitive SNOBOL4 data types include the types INTEGER, REAL, STRING, ARRAY, PATTERN, NAME and CODE. Each type of object has a characteristic representation for "constants" and allows constants of the given type to be assigned as the values of variables. It is convenient to introduce a uniform terminology for describing information structures which represent constants, and to specify assignment of values to variables in a uniform manner.

An information structure which represents a data constant will be referred to as a data reference block. Each SNOBOL data type have a data reference block of characteristic format. For example the data reference block for the data type STRING (string reference block) has the following format.
The block heading field specifies the size of the string reference block and a pointer to its successor in linked list storage. The descriptor specifies the "value" of the string and is further discussed below. The attribute list field points to an attribute list which specifies attributes of the string not explicitly stored in the string reference block. The hash address field does not correspond to any logical string attribute, but is used internally for string accessing. The BCD name field contains the literal BCD string name.

A data reference block for an object essentially specifies that object in terms of its list of attributes. Some of the attributes are specified explicitly in the data reference block, while others such as the "value" and attributes of the attribute list are specified by pointers to other information structures. Attributes on the attribute list specify whether the given string is a function name and/or a label in the current program. This allows a given string name to be used simultaneously as a function name, label name, and data item.

Values of strings are specified in their descriptor field. The format of descriptors is as follows:

<table>
<thead>
<tr>
<th>Pointer to Value or literal value</th>
<th>Usage Information</th>
<th>Type of Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>value field</td>
<td>mode field</td>
<td>type field</td>
</tr>
</tbody>
</table>

Figure 9. Format of a String Reference Block.

Figure 10. Format of a descriptor.
The value field of a descriptor specifies the literal value for objects of the type INTEGER, and a pointer to the data reference block for other types of objects. The mode field specifies usage information for garbage collection and other purposes. The type field specifies the data type.

Assignment of a value to a variable is accomplished by replacing the descriptor field of that variable. Since assignment results in replacement of the type field as well as the value field, a given string name may have different types of values associated with it at different points of time. The delay in binding time of the type till the time of assignment introduces greater flexibility at the expense of some inefficiency.

A pattern in SNOBOL4 is an ordered set of strings. Patterns may have operations performed on them and may be assigned as values of variables, just as other types of objects. The operations which may be performed on them include concatenation and alternation. If P1 and P2 are patterns, then "P1 P2" (P1 followed by P2) denotes any instance of P1 followed by any instance of P2. "P1|P2" (P1 or P2) denotes any instance of P1 or P2.

Patterns may be used in pattern matching operations to determine whether a given string contains an instance of a given pattern. Since a given string may in general contain more than one instance of a given pattern, the order of pattern matching is carefully defined, and the match found during pattern matching is always used.

Patterns are represented by a data reference block called a pattern reference block. A pattern which consists of the concatenation and alternation of more primitive patterns, is represented by a linked structure of pattern reference blocks of its constituents.

SNOBOL4 does not have a nested block structure, and does not therefore require a mechanism for creating and deleting data objects on entry to and exit from blocks. However, SNOBOL4 does have facilities for subroutine definition and subroutine call, and therefore requires facilities for pushing down of information on subroutine entry and popping up of information on subroutine call. Pattern matching may also require a stack. SNOBOL4
permits function execution during pattern matching, and permits pattern matching during function execution, so that the same stack is used for subroutines and pattern matching.

Subroutine call results in pushdown of an activation record on the stack. Pattern matching gives rise to stack entries when matching at a given level is suspended to match a subpattern. The information pushed down during matching of a subpattern may be called a pattern activation record.

Stacks arise during interpretation over dynamically nested structures. A new stack entry is required on entry to a deeper level of nesting and can be deleted on exit from the deeper level of nesting. The stack entry associated with a given nested substructure may uniformly be called an activation record, since it is created on activation of the substructure and deleted when activation is completed.

A more detailed description of SEDIC as an information structure model is given in (16).

11. LISP and the LISP APPLY Function.

LISP is a programming language whose primitive data objects are atoms, and lists of atoms. There are a number of types of atoms each of which is defined by a set of attributes to which certain transformations are applicable. However, the principal operations of LISP are operations which operate on lists independently of the atoms to which they point.

List elements in LISP consist of two pointers. The first may point either to an atom or to a list element, and the second may point to a list element or to the list termination symbol NIL. The three-element list \((A, (B, C, D), D)\) is represented as follows.

![Diagram](image)

**Figure 11.** The list \((A, (B, C, D), D)\).
In Figure 12, A, B, C, D, E are assumed to be pointers to information structures which represent atoms.

The operators on lists include a predicate for testing whether a given object is an atom or a list, and an operation for testing whether two data objects represent the same atom.

\(\text{ACC}(x)\)

The operator \(\text{ACC}\) applied to a data object \(x\) yields the value \(T\) (true) if \(x\) is an atom, and yields the value \(F\) (false) is \(x\) is a list.

\(\text{EQ}(x,y)\)

The operator \(\text{EQ}\) applied to two objects \(x,y\) yields the value \(T\) if \(x\) and \(y\) denote the same atom and yields the value \(F\) otherwise.

There are two selectors \(\text{HEAD}\) and \(\text{TAIL}\) which, when applied to a list element respectively yield the data object pointed to by the first and second components of the list element. By repeated use of \(\text{HEAD}\) and \(\text{TAIL}\) any sublist or atom of a given list can be selected.

There is a list construction operator \(\text{CONS}\), for forming new lists out of components. The operator \(\text{CONS}\) has the property that \(\text{CONS}(\text{HEAD}(L),\text{TAIL}(L)) = L\), where \(L\) is a non-empty list.

There is a conditional operator \(\text{COND}\) which has the following form:

\(\text{COND}(P_1,A_1;P_2,A_2;\ldots;P_n,A_n)\)

This LISP expression is equivalent to the ALGOL expression "if \(P_1\) then \(A_1\) else if \(P_2\) then \(A_2\) else if \(P_n\) then \(A_n\)".

There is also an operator \(\text{LABEL}\) for specifying bound variables and an operator \(\text{LABEL}\) for naming functions:

\(\text{LABEL}((\text{NAME},(\text{LABEL}(X,EXPR))))L\)

In this example \(\text{EXPR}\) is a LISP program called \(\text{NAME}\), in which occurrence of \(L\) are treated as bound variables. During execution the list \(L\) is substituted for \(X\) in \(\text{EXPR}\) and instances of \(\text{NAME}\) in \(\text{EXPR}\) are treated as recursive function calls.

LISP programs may be represented as lists for purposes of execution. When programs and data are represented as lists, then the sets \(I\) and \(el(I)\)
Associated with an information structure model for LISP will be lists. Application of a program to its data can be represented by the three element list \((\text{APPLY}, P, D)\).

![Diagram showing the representation of \((\text{APPLY}, P, D)\).](image)

Figure 12. Representation of \((\text{APPLY}, P, D)\).

The \text{APPLY} function for applying an arbitrary LISP program to its data, is a function for performing transformations on elements of \(c_l(I)\). Since elements of \(c_l(I)\) are lists, and LISP is a language for performing transformations on lists, the function applying LISP programs to their data may be specified in LISP. LISP may thus be used as the modelling language for specifying the transformations \(F\) of an information structure model for LISP. McCarthy\(^\text{[17]}\) has used LISP as a modelling language for LISP, and has specified the LISP \text{APPLY} function as a LISP conditional expression.

Any "universal language" powerful enough to compute any computable function, may in principle be used as a modelling language. However, some universal languages are more suitable as modelling languages than others. In particular, a modelling language should have syntactic recognition facilities for the class of languages it is to model, and should have transformation facilities for specifying required classes of transformations. LISP has adequate syntactic recognition facilities for recognition of LISP program constituents (the ATOM and EQ predicates). It has adequate facilities for selection of sublists (HEAD, TAIL) and for forming new list structures out of components (CONS). LISP is therefore an adequate modelling language when programs and data are represented in LISP format. It would, however, be an inadequate modelling language for information structure models whose information set \(I\) and \(c_l(I)\) were not in LISP format.


