CABAL: environmental design of a compiler-compiler

Richard K. Dove
Carnegie Mellon University
NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.
CABAL. ENVIRONMENTAL DESIGN OF A COMPILER-COMPILER

Richard K. Dove

Carnegie-Mellon University
Pittsburgh, Pennsylvania
May 12, 1968

Address: Carnegie-Mellon University
Computation Center
Pittsburgh, Pennsylvania 15213
Phone: 683-7000, Extension 268
CABAL is a compiler-compiler designed at Carnegie-Mellon University. In appearance its debt to ALGOL, PL/I, and FSL is apparent. Additionally, it features a formal co-routine structure; a unitized data structure which can function as an array, scalar, list, tree, stack, queue and plex; and a high level code generation facility. In addition to presenting these and other results of the CABAL design, the design process itself is examined. Primarily, the results of the design process are viewed as owing their existence to a hard look at defining the ultimate environment in which the system would function. Environmental analysis and goal definition are separated from the design process and held accountable for the result.

This project was funded by ARPA.
I. INTRODUCTION

CABAL\(^1\), a compiler-compiler design project at Carnegie-Mellon University, was fostered in the wake of Jerry Feldman's FSL thesis\(^2\). Initial plans were to use Feldman's basic Production Language - Formal Semantic Language structure, include extensive storage allocation procedures, and implement a number of extensions suggested by feedback from an FSL implementation of FORMULA ALGOL\(^3,4\).

As the project progressed a study of the current state of the art for both translator writing systems and language design methods gradually changed the design goals of the development group. Working papers published throughout the design reflected this continuous change. Eventually a well-defined set of goals evolved and the design phase progressed to a solution not closely resembling the original direction.

It is an accepted fact that large design projects very rarely end up as they are initially conceived. A learning process goes on when the problem is seriously attacked which precludes a static set of goals. However, in the course of checking the pertinent literature and at the same time noting the causes for progressive quantum jumps in the CABAL design effort, two things became apparent: first, as a strong set of goals developed the design solutions came rapidly; and second, the literature notably lacks in design methods as well as goal definitions and language design considerations.

Shifting now to the results of compiler-compiler or translator writing system design, the current literature reflects the youthfulness of the field. New concepts and solutions to new problems affect systems design in a predictable way: the primary concern is to demonstrate a working system.
Only after we relegate these new problems to an innocuous position and comfortably understand the new concepts involved do we turn our attention to usable systems as opposed to workable systems.

Although compiler-compiler techniques are by no means near to being a closed issue, there is enough foundation material currently available that a little bit of polish can produce a palatable system.

Thus, the concern of this paper is twofold. First, in an effort to increase documentation of design processes as well as stimulate some criticism toward better methods, I have included a section covering the goals set forth for CABAL as well as the information responsible for setting them. The point of view was stimulated by something T. E. Cheatham lightly dropped: "We are concerned hardly at all with the extremely important and often neglected problems of the environment in which a compiler or code resulting from a compiler is to operate."*

The second aim of this paper is to present a substantial portion of the CABAL design results. Briefly, the language owes much of its appearance to ALGOL, PL/I, and FSL. Additionally, it provides a formal co-routine structure syntactically patterned after ALGOL procedures, and a unitized data structure capable of functioning as a scalar, array, list, stack, queue, tree and plex. There is a clean separation between syntax, semantics, and code generation facilities without sacrificing a unified language. Lacking, however, is a discussion of general purpose input/output facilities as their inclusion at this point is not considered enlightening.

The remainder of this paper is sequenced in five sections followed by an evaluation of how well the design satisfied the goals. After discussing

---

*Cheatham, T. E., p. 65
design criteria in the next section, the language is presented as structure and control flow, semantics processing, syntax parsing, and code generation. Although this breakdown sounds like a delivery of semantics language, syntax language, and code generation language, the individual language distinction has been carefully bred out of them. The presentation sequence was chosen for how much light each shed on its followers.

II. DESIGN CRITERIA

CABAL Note 3 presented the initial thoughts of the CABAL Development Group on design criteria for compiler-compilers in general. The following discussion presents the criteria governing CABAL specifically and includes pertinent material from Note 3 as well as additional constraints determined since then.

Much of the resultant criteria is strongly based on my evaluation of where Carnegie's computation center and computing in general is going in the next five to ten years. Additionally, the fact that we are "going" is most important. The truth of change must be recognized if anything lasting is to be built.

A Data Base

Rather than arbitrarily choosing guidelines which sound intelligent and meaningful, we should first examine the ultimate environment in which CABAL will function. In fact, there are three environments of concern: physical system, user, and maintenance. In addition, both present and future definitions of these environments must be investigated, else another obsolete system be designed.
The physical system environment contains both hardware and software, the things CABAL must directly interact with. The overall environment I see as a computer utility, already started and surely on its way to great expansion. This means a hardware configuration consisting of many different kinds of remote terminals as well as a complex processing facility. This processing facility contains a multiple number of central processors of unlike design, some will be interconnected and some not. Additionally, parallel processing is expected.

The software environment is felt on two levels. Operating systems capable of time sharing, batch processing, and parallel processing, will buffer all interactions between the processing facility and running programs. On another level we find that many different kinds of languages are present ranging from assemblers and compilers to interpreters as well as from general to special purpose.

Academic surroundings make the maintenance environment different from its commercial and industrial counterparts. At least, defining the maintenance environment with the least amount of restrictions on it proves such. The academic computation center has a high turnover of employees due to student labor and lure of better wages in industry. Additionally, many employees, barring permanent staff, are not overly endowed with systems experience (maintenance procedures or systems comprehension). Then, too, we seem to have more systems than maintenance personnel. Thus, inexperienced personnel are used for much of our maintenance. Teaching them complex systems and techniques takes so much time that few of them have the chance to become indispensable. Only in this last respect is our maintenance environment enviable.
The user environment is also unique to an academic atmosphere. Researchers working on compiler-compiler techniques, languages, and large programming systems are only outnumbered as a group by the mongol hordes from the undergraduate ranks. Then we have the graduate students whose theses involve special purpose languages to implement their main concern. Finally, there is enough on-site talent looking for means to express itself that lowering the amount of pain incurred to implement a system will considerably increase the amount of talent used.

So far we have only been concerned with environmental factors. I have tried not to introduce any artificial constraints which would result by drawing conclusions from these factors. Separating the one from the other clearly defines a base for design criteria. When considering change it will be easier to evaluate what is being given up against what is being gained without confusing the issue with secondary effects.

**General Design Goals**

Directing our attention now to the design goals of CABAL, we can separate our concern into three areas: syntax parsing, semantic processing, and code generation. Although CABAL as a whole should exhibit unity, the problems associated with each of these three areas are distinct and therefore warrant individual consideration. Additionally, one more area of concern germane to programming languages in general, structure/control flow considerations, is readily isolated. However, before considering each of these four areas individually, I will outline a list of unifying overall design goals and justify each by its relevance to our data base.
Modularity is important. In designing a system one seeks to divide it into distinct sub-sections. The criteria governing this separation is to produce sub-sections which have a minimum of interaction and inter-communication and a maximum of structural similarity. The resultant modularity offers a system which can be easily changed, debugged, and extended; has fewer initial bugs; is easily and quickly tested; is quickly understood by design and maintenance personnel; is less confusing to users; and possesses an inherent division of labor for both the initial coding and subsequent maintenance. Then, too, if this modularity shows through at the user level, it becomes a system whose use is quickly learned, readily understood, and easily retained.

The language must be readable and conceptually simple, facilitating a quick grasp by the occasional or one shot user as well as providing for comprehension by maintenance personnel and programmers faced with the problem of understanding someone else's program. For the same reasons interactions with system parts concealed below the surface must be kept to a minimum. At the same time the language should be sufficiently powerful and terse to warm the heart of the most esoteric systems programmer, providing him with handles on any part of the system.

Flexibility must be inherent as the environment will most definitely change. Machine dependence should be kept to a minimum as should well-defined interactions with the system environment. Note that modularity does much for flexibility but does not insure it.

Most important is applicability. CABAL is useless unless its performs some meaningful unique function geared to the problem at hand: compiling compilers. Thus, the language must apply to the problem area.
Compatibility with the physical system environment must be insured. Reflecting the physical system in CABAL buys power from another angle. Not only should CABAL fit into its environment, but it must provide access to all parts that are usable.

Its range must be comprehensive enough to warrant its development. The ability to produce conversational and incremental compilers should be considered. It should be able to compile assemblers and interpreters as well as compilers. One benchmark would be the ability to compile itself.

Last but by no means least is reliability. This aspect has been discussed at length by Peter Naur. I have saved mentioning reliability until this point as many of the preceding goals do much to insure it. However, other goals aside, reliability must be recognized as a primary goal in and of itself.

Collecting these goals into one concise list we find the following:

1. applicability to compiling language translators;
2. wide range of producible translators;
3. compatibility with total dynamic environment;
4. modularity;
5. flexibility in dynamic environment;
6. power for the professional systems programmer;
7. transparency for the non-professional user;
8. readability;
9. terseness for the professional user;
10. machine independence;
11. reliability.

This list is not meant to be composed of first principles and therefore we need not be concerned about overlap. In the following section we will
see how these goals have affected the design of CABAL. The success of the design effort is measured by how well it satisfies these goals. Consequently, the formation of design goals is seen to be of prime importance, for the design effort is determined by them.

III. STRUCTURE AND CONTROL FLOW

In order to avoid confusion among the various levels of processors and languages associated with a compiler-compiling system, I will adopt a convention defined in CABAL Note 3. $C^2$, $C^1$, and $C^0$ will refer respectively to the meta-compiler CABAL, an object compiler written in CABAL, and an object program written in the language of the object compiler. When necessary, subscripts $P$ and $L$ will further distinguish processors from languages. "Thus, $C^1_P$ is a processor written in $C^2_L$, $C^0_P$ is a program written in $C^1_L$, and $C^0_L$ might be considered the 'language' in which data for $C^0_P$ is written."*

In this section we are concerned with structure and control flow associated with $C^2_L$. Structure and control at lower levels will be discussed in the sections concerned with code generation.

Structure

CABAL is an ALGOL-like language and in fact has ALGOL's block structure complete with BEGIN-END pairs and identifier scope. In addition, the internal structure of a block is divided into a declaration part followed by a statement sequence part. Among other things, the declaration part includes declarations for two programming structures: co-routines and ALGOL procedures. A co-routine declaration is syntactically similar to an ALGOL procedure declaration and will be discussed shortly.

Another kind of structure inherent in CABAL results from the distinction between syntax parsing, semantics processing and code generation.

The next three sections will discuss each of these at length so at this point I wish only to indicate the nature of their separation. Syntax parsing is provided by reduction statements modeled after those used by FSL. However, they are not format dependent in keeping with the ALGOL-like nature of CABAL. Code generation uses FSL's code bracket concept but the contents are entirely different. Finally, semantics processing is provided by constructs similar in number and nature to ALGOL statements.

At a lower level modular concepts are applied to the structure of the various language constructs. Concepts with similar meanings exhibit similar structure. For example, the syntax for accessing an element within a multiple element data structure uses square brackets around the indexing information. An extension of this concept demanded square brackets around the indexing information associated with accessing bit fields within elements. Thus, A [3].[4] requests bit number 4 of element number 3 in data structure A.

Applying modular structural concepts again resulted in declarations resembling those of PL/I rather than ALGOL's. Whereas ALGOL has individual declarations for each kind of typed storage structure, PL/I has a single DECLARE construct which is parameterized. Extensions can easily be accommodated without adding a new declaration concept.

Modular structure concepts and simplicity considerations were responsible for unitizing five data structures into a single concept. An examination of data structure declarations will finish the structural concepts. CABAL has only one data structure. However, it has the capability to function
as a scalar, array, threaded list, stack, or queue. The syntax is:

\[
\text{<declaration> ::= DECLARE <declaration list>}
\]

\[
\text{<declaration list> ::= <name list> ( <structure> )}
\]

| \text{<declaration list>, <name list> ( <structure> )}
\]

\[
\text{<structure> ::= <nature> <element linkage>}
\]

\[
\text{<nature> ::= NUMBER , <bound> <number option>}
\]

| \text{Logic , <bound>}

| \text{STRING , <bound>}

| \text{NAME <bound>}

\[
\text{<bound> ::= <number expression> | ? | ? <number expression>}
\]

\[
\text{<number option> ::= , FIXED | <empty>}
\]

\[
\text{<element linkage> ::= , <bound pair list> <pop linkage>}
\]

| \text{<empty>}

\[
\text{<pop linkage> ::= , LIFO | , FIFO | , RANDOM | <empty>}
\]

\[
\text{<bound pair list> ::= <bound> : <bound>}
\]

| \text{<bound pair list>, <bound> : <bound>}

Examples follow for scalars A and B, array C, stack D, and list E:

```
DECLARE A,B(NUMBER,6),
    C(LOGIC,4,[0:10,0:10]),
    D(STRING,10,[1:?15],LIFO),
    E(STRING,?,[1:?],RANDOM);
```

Thus, if there is no <element linkage> a scalar is signified, <element linkage> without <pop linkage> results in an array, and the presence of <pop linkage> signifies how elements will be pushed and popped for stacks, queues, and lists. Plexes and trees can be built from lists.
The expressions following the type indicate how many significant figures associated with the type in question are to be retained. Limits on these will be set by the implementation. A ? signifies an unknown number and results in dynamic allocation. An expression following a ? indicates the user's guess and allows the compiler to take advantage of this information for efficient structuring.

CABAL has only four types: NUMBER, LOGIC, STRING, and NAME. These will be discussed in the semantics processing section.

**Control Flow**

Control flow is altered with GOTO and call statements. The GOTO statement, in addition to its standard GOTO <label identifier> appearance, also appears as GOTO <name variable>; where <name variable> can be any type NAME data structure element with a LABEL value. Thus, the ALGOL switch label construct looks like GOTO A[N] where A is any multiple element data structure of type LABEL and A[N] has a LABEL value.

There are three kinds of call statements, however, each is syntactically identical. The difference is in the item being called which can be function designators, procedures, and co-routines. The method of calling is standard ALGOL in nature: <identifier> ( <parameter list> ).

Syntactically co-routines are identical to procedures with the exception of a COMMON declaration which may appear in a co-routine but has no meaning in a procedure. Briefly, co-routines are programming structures which, when they receive control, continue processing from the internal point which last gave up control. Thus, although control is passed to them by name, they can be considered as having multiple entry points. The COMMON declaration is simply a group of statements which are executed everytime
control enters the co-routine in concern. Immediately after execution of the COMMON statement control passes to the internal point which last gave up control.

Calling a co-routine demands that control be returned to the point following the call, just as with procedure and function designators. However, co-routines may also be activated by GOTO statements. When this is the case, a co-routine must relinquish control with a GOTO statement rather than a RETURN as could be the case had it been called.

Minor control changes associated with IF and FOR statements are similar to ALGOL and need no explanation. The following section on general semantics processing will present their exact nature. Reduction statements exhibit control changes similar to IF statements and will be further explained in the section concerned with syntax processing.

IV. GENERAL SEMANTICS PROCESSING

T. E. Cheatham has noted that - "While there exist reasonable elegant schemes for 'automatically' doing syntactic analysis (and even much of code synthesis), the handling of declarations is generally messy with any but the simplest of languages. For this reason (among others) it will prove highly useful in any general purpose compiling system to have the ability to do arithmetic and relationals - i.e., the 'action language' should contain at least the rudiments of a good algebraic language."*

Rather than just containing "rudiments", CABAL has all the power of ALGOL as well as some significant extensions, notably co-routines, stacks, field and bit level addressing within storage elements, and the reduction statement which is not limited to syntax parsing alone.

*Cheatham, T. E. 10, p. 65.
Expressions and Types

Expressions have four types as do data structures. However, typing in CABAL does not follow the usual rules. An expression type is determined purely by the operators appearing in it unless it has only a single operand, in which case it takes on the operands type. Thus, a string variable could be numerically added to a logic variable and would produce a value of type NUMBER. The main reason for typing variables is so that information within them may be partially accessed. For example: `<variable>.[<field designator>]` is a means to address a portion of the variable. If the construction was A.[2:4] we would be addressing the second, third, and fourth digits within element A where digit is defined according to the type of A. Digit definitions are as follows: NUMBER-decimal digit, STRING-alphanumeric character, LOGIC-one bit, NAME-one entire name.

The unary and binary operators are classified according to the expression type they produce.

```plaintext
<unary numerical operator> ::= - | + | ↓
<binary numerical operator> ::= - | + | / | * | ↑, o, c, C
<unary logical operator> ::= ¬
<binary logical operator> ::= < | > | ≥ | ≤ | = | ≠ | ∧ | ∨
<binary string operator> ::= &
```

Some of the unfamiliar operators are &:concatenation, o:mod, C: is a substring of, and ↓:truncation. The substring operator provides a value of 0 if the left string is not contained in the right string and a value corresponding to the digit position of the left string's first character match in the right string if it is contained.
Name expressions never contain operators as their function is to convey pointers. For example, a name variable N might be assigned the name of a label, \texttt{N:=NAME(LABELX)}, and then appear in a subsequent GOTO statement. Name variables may be used in place of any CABAL name provided they have been assigned either the name, copy, or form of the item they are representing. Assigning \texttt{N[X]:=A} allows \texttt{N[X]} to be used in place of \texttt{A} as \texttt{N[X]} has a pointer to \texttt{A}. However, if the value of \texttt{A} is changed then the value of \texttt{N[X]} has also been changed. To circumvent this, \texttt{N[X]:=COPY(A)} will duplicate \texttt{A} and assign a pointer of the copy to \texttt{N[X]}. Additionally, \texttt{N[X]:=FORM(A)} will allow \texttt{N[X]} to be referenced as if it had been declared the same way \texttt{A} had. The name variable concept is particularly useful in building up threaded lists or trees.

\textbf{Statements}

There are four statements specifically supplied for semantics processing: assignment, conditional, iterative, and push-pop. The assignment statement is straightforward:

\[
<\text{assignment statement}> ::= <\text{variable}>\leftarrow <\text{expression}>
\]

\[
<\text{expression}> ::= <\text{name expression}> \mid <\text{number expression}>
\mid <\text{logic expression}> \mid <\text{string expression}>
\]

The replacement operator does not affect type as the value of the \texttt{<expression>} is stored in the variable with no regard to type differences. If the length of the expression value cannot fit in the variable, then the replacement is undefined.

The conditional is standard ALGOL and needs no further explanation.

The iterative statement is as follows:
<iterative statement> ::= <iterative condition> DO
<iterative condition> ::= WHILE <logic expression>
| FOR <variable> ← <number expression> <terminal condition>
<terminal condition> ::= TO <number expression> BY <number expression>
| TO <number expression> WHILE <logic expression>
| WHILE <logic expression>

Although the syntax has been shortened, the semantics here are similar to ALGOL for statements.

The push-pop statement is used to push and pop elements associated with LIFO, FIFO, and RANDOM linked storage.

<push-pop statement> ::= <stack operator> <name>
| <stack operator> <group reference>
<stack operator> ::= Δ | <stack operator> Δ
| Ψ | <stack operator> Ψ
<group reference> ::= <name> [ <group definition> ]
<group definition> ::= <index range list> <number expression list>
<index range list> ::= <empty> | * | <index range list> *

To exemplify the range of manipulation this buys consider the following structure:

DECLARE MST(Logic,32,[1:X,1:Y,1:Z],RANDOM);

This gives us a master symbol table similar to the one which must be declared by the user at a block level sufficiently high to encompass all code bracket statements. Z is the maximum element number in the list and corresponds to the maximum number of identifiers the user's compiler will handle; Y specifies the number of 32 bit storage elements needed per
identifier; and X specifies the number of remembered nested declarations any one identifier may have. By asking for RANDOM linkage we can push and pop any group of elements in the list at any time. The result of pushing element \([*,*,2]\) is to have its address now by \([*,*,3]\); \([*,2,2]\) goes to \([*,3,2]\); and \([2,2,2]\) goes to \([3,2,2]\). Popping any one of these, though, removes it entirely from the list and changes group \([*,*,3]\) to \([*,*2]\) and so on.

Consequently, using this as our master symbol table allows us to push and pop information concerning every declaration of a given identifier name as well as only that information associated with its declaration at a specific block level.

The master symbol table (MST) serves as a communication link between the semantics processing and code generation. As will be discussed in the code generation section, storage allocation and other necessary MST information is entered into the MST by the code generators. During semantics processing this information as well as additional data the semantics routines may store in the MST is available for use.

V. SYNTAX PARSING

Syntax parsing is accomplished with a reduction statement similar to FSL's production except the stack manipulation is different. An example will clarify:

\[
\text{L: } \text{"WHILE"}, \text{EXP} \rightarrow \text{STA} \mid \text{WHCODE,RETURN};
\]

The sequence of events is as if an IF-THEN statement were executed. If the top stack entries are matched with the left section then the right
section receives control, just before the right section gives up control
totally the top two stack elements are popped and the contents of the
middle is pushed on. Thus, although \texttt{WHCODE} is the semantics routine
procedure called for processing, the stack remains the same until just
prior to the \texttt{RETURN} execution.

Another difference from FSL is that there is not a semantics stack
automatically pushed and popped as the syntax stack is. The user must
provide his own semantics sequencing. This eliminates a lot of unnecessary
stack manipulation as well as the confusion which arises in the following:

\[
|A,A,B| \rightarrow |A,B|; \\
\]

The question as to which \texttt{A} semantics should be retained does not occur without
a parallel semantics stack.

Elements in the left and middle sections of the reduction statement
are of two kinds: literals and meta-characters. The literals cause straight
forward comparisons between the character string enclosed in quotes and the
character string associated with the corresponding stack position. Meta­
characters represent a group of literals and are matched if the corresponding
stack position matches any one of the group.

\texttt{CABAL} associates a meta-character with its member literals through a
define declaration. An example covering unary and binary numerical operators
follows:

\[
\text{DEFINE UNOP}=\"-\"\mid \text{"+"}, \\
\text{BNOB=}\text{UNOP} \mid \text{"/"\mid \text{"*"\mid \text{"^"\mid \text{"_"\mid \text{"C"; \\
\]}

Additionally, meta-characters can be defined for non-printing characters
like end-of-record, end-of-file, carriage return, tab, space, backspace,
etc. by using the numerical equivalent of the character in question as a
\texttt{definens}. 
DEFINE ECC = 64;

The matter of the sigma function, a meta-character which always matches, is taken care of by the absence of either a meta-character or literal, when one is clearly called for by the comma placement. Alternatively, for those who demand a printable sigma function, a meta-character may be used whose definens is empty.

Actually, reductions are not restricted to operating on stacks, any data structure will do. However, trying to match a two element reduction to a scalar is undefined as is any match which is larger than the declared size of the coupled data structure. A multiple element data structure with ? number of elements is permissible though.

Couple statements are used to dynamically associate reductions to a specified data structure. Reductions are coupled to the structure specified in the last executed couple statement at the same or higher block level.

There is a system supplied routine which will produce a unique integer value for every unique character string supplied to it. The inverse function is also available. Normally, reductions will expect these unique integers to be stack entries rather than character strings as a considerable time and space saving can be realized. However, should a particular parsing job consist of short strings only, it might be profitable to bypass the string translation. Communicating the nature of stack contents is accomplished by the stack type, either NUMBER or STRING, associated with the stack specified by the appropriate couple statement. An example of a couple statement follows:

COUPLE(SYNXSTK);
Reductions match only a certain field within NUMBER stack elements extending from digit one to X where X will be defined by the implementation. Thus, the user may declare a wider or multiple dimensioned stack and gain parallel storage if he so desires. Stacks of type STRING will have the entire first element of each stack row matched by the reductions and consequently the only way to gain parallel storage here is with multiple dimensioning.

The left two sections of a reduction statement act as a pattern recognition and generation device for syntax parsing and bear little resemblance to the rest of CABAL even though reduction control is similar to IF-THEN statements. The third section, however, exhibits the full range of CABAL as its content is syntactically defined as a statement sequence, with the exception that another reduction statement may not appear unless imbedded within a BEGIN-END pair.

This structure, along with the placement freedom of reductions allows diverse ways to organize a language translator. Notably, the two ways most usually desired: a complete set of productions following one right after another with their associated semantic routines also grouped in one sequential mass disjoint from the reductions; and alternations of syntax and semantics statements with each syntactic mechanism containing or followed by its associated semantic routines.

VI. CODE GENERATION

Code generation is accomplished through the use of code statements and item statements. A code statement simply encloses in code brackets constructs which for the most part compose the CABAL language as a whole.
Item statements are used to provide parameters in conjunction with the indicated code from a code statement. Thus, the translator writer merely translates his input stream into valid CABAL and encloses it in code brackets.

There are a few restrictions and extensions which may appear in code brackets. With the exception of having to put out complete statements, structure is non-existent. Declarations and statements may be interspersed at will. A declaration in code brackets causes the generator to push the MST at the appropriate place and store the necessary information. Subsequent usage of declared items in statements causes retrieval of MST data in order to produce code.

The output from code brackets is a stream of items which are interpreted by the generator into code. Code is produced whenever a sufficient amount of information comes through the code brackets. Thus, one statement may be executed such as CODE[^1], producing no code until the construct is sufficiently completed by, perhaps, CODE[^1 + ^2].

All names within code brackets which must be identified have a syntax of ^<integer>. This relates them to parameters in the most recently executed item statement. For example:

ITEM(SMT[4], SMT[1], #A, #B);
CODE[FOR#1←#4 TO #3 BY #2 DO];

Here we have the parameters in the item statement implicitly numbered starting with 1 at the left. The code statement designates which parameter it wants by signifying the number. The # in the item statement is used to indicate that a parameter represents a constant rather than a declared name.
When the generator receives a parameter representing a declared name, it looks in the MST to get the appropriate storage address and related information. When a parameter name has more than one declaration, the top one is always used.

Should it be necessary to pass a name parameter to the generator which has not yet been declared, like a label name for instance, the CHAIN function is useful. This function need only be used once per name as it continued to chain all instances of the name until the ASSIGN function is invoked. ASSIGN is used after the appropriate declaration has been made and retraces the chained list inserting appropriate addresses and completing any unfinished coding.

There are times when $C^1_p$ wishes to transfer some of its data to $C^0_p$. This is facilitated quite easily as:

```
ITEM(RUNNAME,#TABLE);
CODE(#1<#2;);
```

This works if both data structures have the same declarations; TABLE is declared for the $C^1_p$ level and during a compilation collects data which must be passed to the $C^0_p$ where RUNNAME has been declared.

Control can be passed to compiled code by generating a GOTO <label name> NOW; where <label name> is some pre-declared label in the generated code. Return control to the point following the code statement which gave up control will happen if execution runs into a pre-generated RETURN NOW. Execution of HALT by either $C^0_p$ or $C^1_p$ will give control to the operating system.
VII. SUMMARY

Looking now with an eye for comparing the language description with the design criteria, there are a number of points worth mentioning. For applicability to compiling-compilers there are included co-routines for natural phase and pass separation, reductions for syntax parsing, full algebraic power for semantics processing, structures and operators for string manipulation, system supplied routines for handling strings comfortably, a master symbol table easily accessible by both code generation system routines and user written semantic routines, and a code generation facility that couples the easy and nonchalant use of a high level language with optimal system routines capable of streamlining the resultant code to a degree determined by how much time is deemed worthwhile.

Producible translators include interpreters, which make use of the output facility for compile time data and code as well as reduction statements; conversational compilers, aided by ease of control flow between environment, compiler, and generated code; and multiple language systems using co-routine and reduction stack coupling. Compatibility is maintained with a dynamic environment through machine independence and an open ended design ready to accept extensions. In this respect a macro facility is anticipated and seems reasonably easy to implement.

Modularity is provided not only by the language structure and its program and data structures, but also at a lower level its statement and declaration syntax is modular to a degree that makes extension trivial. Flexibility, besides benefiting from points already mentioned, is further assured by an absolute minimum of communication linkages and interdependencies within and among the language system subdivisions adn the system environment.
There is full algebraic power as well as the ability to reference all system variables and even drop down to an assembly language sequence. Readability and transparent learning is assured from the ALGOL nature as well as keeping the concepts to a minimum and using a high level language for code generation. The syntax was specially geared for terseness to such an extent that FOR-STEP-UNTIL-DO was superceded by FOR-TO-BY-DO.

Finally, reliability is facilitated through an easily understood and well-partitioned language system with no major hidden subdivisions.

In summation, I feel satisfied with the CABAL design for two reasons. First, I think it demonstrates that programming languages aimed at complex problems not fully formalized or understood need not wait for the absolute insight before user problems are considered. In fact, making the usage of such a system as painless as possible will do much for increasing the number of users and thereby quicken the time when language translation is yesterday's problem.

As a second point of satisfaction, I note that the results of the CABAL design phase are directly related to the goal system established. Hopefully, refined techniques for defining the initial goals of a design project will make this process quicken and the ultimate designing a matter of cause and effect.
References


COMPUTER AUGMENTED DESIGN-A BIBLIOGRAPHY

The following bibliography covers those papers and publications that deal with the graphic, ill-defined or heuristic aspects of computer augmented design. It does not deal with traditional optimization or linear programming techniques.

The following outline is utilized. Where a particular study is relevant to several categories -- it has been listed more than once.

CLASSIFICATION OUTLINE

I. Design-general considerations
   A. Definitions and descriptions of the design process
   B. Design task analysis
   C. Analysis of information retrieval systems for design

II. Operational computer programs augmenting design
   A. Graphic output
   B. Automated evaluation and simulation
   C. Scheduling
   D. Cost controls
   E. Synthesis
   F. Computer-aided design systems

III. Representations and models for design
   A. Morphological and syntactic analysis of natural design languages
   B. Representations of surfaces, spaces, and forms
   C. Possible representations
      1. Graphs
      2. Lists
      3. Associative nets
      4. Numerical systems

IV. Design operations
   A. Planning
   B. Heuristic search
      1. Theory
      2. Examples
   C. Processes that learn
   D. Pattern recognition
   E. Other operational forms

V. Applications
   A. Computer design
   B. Space allocation
   C. Structures
   D. Urban models
   E. Transportation
   F. Circuits and Logic
   G. Other
Bibliography


5. Alexander, Christopher, "The Theory and Invention of Form", in Architectural Record, April 1965, pp. 177-186. (IIIC1, VB, VD)


7. Alexander, Christopher, Notes on the Synthesis of Form, Harvard University Press. (IA, IIE, IIIC1, IVA, VB, VD)


20. Black, W. L., "Discrete Sequential Search", in *Info. and Control*, 8, 2, April 1965, pp. 159-162. (IIIC4, IVE)


46. Ernst, George, Newell, Allen, "Some Issues of Representation in a
General Problem Solver", Proceedings SJCC, 1967, Spartan Books,
Washington, D. C. (IIE, IIIA, IIIC4, IV1)

47. Evans, D. H., "Modular Design - A Special Case of Non-Linear Programming",

48. Fair, G. R. et al, "Note on the Computer as an Aid to the Architect",
in Computer J., June 1966. (IIB, VB)

49. Falk, H., "Computer Programs for Circuit Design", Electro-Techn, 77, 6,
pp. 54-7, June 1966. (IIB, IIE, VF7

50. Feder, J., Linguistic Specification and Analysis of Classes of Patterns,


New York, 1965. (IIA)

53. Flagle, C. D., Higgens, W. H., Roy, R. H., Operations Research and

54. Fletcher, J. C., "A Program to Solve the Pentomino Problem by the

55. Fogel, Lawrence J., Biotechnology: Concepts and Applications, Prentice-
Hall, Englewood Cliffs, New Jersey, 1963. (Chapters 1, 2 - IIIC4)


Drafting, ABAJ 50, 1 Jan. 1964, 41-43.

59. Freund, Louis and Sodosky, "Linear Programming Applied to the Optimiza-
tion of Instrument Panel and Workplace Layout", Human Factors, 9, 4,

60. Frisch, R., Inseeking by the Nonplex Method in Non-Linear Programming,
Inst. Natl. Planning, Documentation Center, Cairo, Egypt, Memo. 396,

Holt, Rinehart and Winston, New York, 1962. (Chapters 5, 6 IB)


69. Gosling, W., The Design of Engineering Systems, Haywood, London, 1962. (Chapters 1, 3 IA; Chapter 2 IIIC1; Chapter 5 IVB2)

70. Grason, John, "A Dual Graph Representation for Space Filling Location Problems of the Floor Plan Type", Carnegie-Mellon University, paper presented at the DMG Conference, M.I.T., June, 1968. (IIE, IIIC1, VB)


131. McCarn, D. B., "Large Scale System Design Techniques", in Proc. 2nd Congress Inf. and System Sciences, pp. 95-98.


