TM-4539/000/01

USER ADAPTIVE LANGUAGE (UAL): A STEP TOWARD MAN-MACHINE SYNERGISM

Aiko Hormann, Antonio Leal, David Crandell

June 28, 1971

Sponsored by the Advanced Research Projects Agency
ARPA Order No. 1327

SERIES	BASE NO. / VOL.	REISSUE
TM	4539 /000 /01	

(TM-4539/000/00 is a draft.

TECHNICAL MEMORANDUM

(TM Series)

This document was produced by SDC in performance of contract No. DAHC 15 67 C 0149 with the Advanced Research Projects Agency, Department of Defense

USER ADAPTIVE LANGUAGE (UAL): A STEP TOWARD MAN-MACHINE SYNERGISM

Aiko Hormann
Antonio Leal
David Crandell

June 28, 1971

SYSTEM DEVELOPMENT CORPORATION
2500 COLORADO AVE.
SANTA MONICA
CALIFORNIA
90406

(Page ii Blank)

ABSTRACT

The User Adaptive Language (UAL) is designed to provide a convenient and flexible means for man-machine communication in cooperative problem-solving/decision-making efforts. The language is extensible and functionally oriented with user control of evaluation and data manipulation. The interactive nature of UAL provides a "conversational" environment conducive to dynamic decision making.

The syntax of UAL is simple and straightforward. The function definition features provide a means of creating new terms and primitives allowing a higher level of sophistication in the communication of ideas. The new functions may be compact and stylized or English-like in their use. Consequently, the problem solver is free to concentrate on *what* to say rather than *how* to say it. UAL can be a powerful tool in building systems in which man and machine can work together to complement each other's capabilities.

(Page iv Blank)

TABLE OF CONTENTS

		Page
1.	INTRODUCTION	1
1.1	Need for Dynamic Extensibility	2
1.2	UAL As A Step Toward Man-Machine Synergism	3
2.	DESCRIPTION OF UAL (USER ADAPTIVE LANGUAGE)	7
2.1	Basic Elements	8
2.2	Assignment	9
2.3	Lists	10
2.4	Evaluation	12
2.5	Operationals	13
2.6	Functions	14
2.7	Argument Map	14
2.8	State Conditions	19
2.9	Built-in Functions and Language Extensibility	19
2.10	The Control Character #	21
2.11	Predefined Functions	23
3.	USE OF UAL: AN EXAMPLE	26
3.1	PO&M System (Planning Organization and Management System)	26
3.2	Primitive Functions and Building Blocks for PO&M System	29
	REFERENCES	38-40
	APPENDIX A. A UAL Terminal Session	41-53
	APPENDIX B. Precedence	54
	APPENDIX C. Character Conversions	55 & 56
	APPENDIX D. UAL Syntax	57

USER ADAPTIVE LANGUAGE (UAL):

A STEP TOWARD MAN-MACHINE SYNERGISM*

1. INTRODUCTION

This paper describes a communication language between man and machine designed to aid our research in man-machine synergism. The overall objective of our research is to develop a system and techniques with which man-machine talents can be utilized effectively and through which man and machine can "co-evolve" in a variety of decision-making and problem-solving situations.

Research efforts have been concentrated on answering three interrelated questions: (1) "What machine capabilities and features, including adaptivity, can be built in and what must be left to man-machine interaction?" (2) "What type of problems or problem environments from which substantial payoffs can be expected are especially in need of this approach?" and (3) "What type of language is needed for man-machine communication to enable the man to start exploiting the machine capabilities from the early stage of problem conceptualization and definition through the exploratory and the intuition-guided stage of problem solving?"

Rudimentary models constructed in an attempt to give partial answers to the three questions are Gaku,**a system of computer programs which has been evolving for several years; Shimoku [Hormann, 1966], a highly complex game-like environment with complex payoff-cost functions in which man and machine interact in generating and evaluating alternative courses of action to gain a high score; and the User Adaptive Language (UAL).

Only Shimoku and a small part of Gaku were implemented on the Q-32 computer, the rest of the Gaku design was hand-simulated at the pencil-and-paper level. However, a great deal of insight and experience with live subjects was gained, which led to several modifications of all three models.

---

*This document is a revision of an earlier draft document (TM-4539/000/00, dated April 10, 1970).

**Gaku is a Japanese word meaning adaptive. This system was first designed and implemented on the AN/FSQ-32 computer in the context of artificial-intelligence research [Hormann, 1962, 1964, and 1965]. It was later redesigned for man-machine cooperative problem solving and was partially implemented on the same computer [Hormann, 1966 and 1969]. The latest design features, which incorporate team planning and problem solving and are aimed at real-world problems, are currently being implemented on the IBM 360/67 computer under SDC's ADEPT time-sharing system.

1.1 NEED FOR DYNAMIC EXTENSIBILITY

The following is a summarization of the findings that combine our own observation and the reflective-introspective comments expressed by experimental Shimoku subjects. These findings confirmed our earlier belief that the dynamic extensibility of Gaku capabilities, which depends on dynamic extensibility of the communication language, is one of the essential features in achieving man-machine synergism.

- Even simple bookkeeping functions of the machine can greatly enhance the information-handling capacity of the man, especially in keeping track of interrelated elements in a complex situation. However, the need for such bookkeeping assistance can arise in so many different ways and different situations that a fixed set of predetermined machine functions cannot handle all such needs adequately. Provisions must be made to enable the man to formulate these requests to the machine as the need arises. Some clever ways of utilizing simple machine functions apparently come about dynamically during the interaction when the complexity of the situation exceeds the information-handling capacity of the man. When such clever ideas are not forthcoming or machine aids are not available, the man tends to oversimplify the situation, deliberately (or unconsciously) ignoring many interrelated elements and often leading to a premature decision or conclusion. This is especially true when complexity and time constraints are present simultaneously.

- Before actually executing their decisions, most subjects ask "what if" questions either overtly or covertly in order to estimate the consequences of the tentative decision steps that have been formulated. However, the breadth and depth of such "what if" questions varies greatly with individuals, even with machine assistance. Since exhaustive examination of alternatives in more than three-step depth is infeasible (in the Shimoku environment) even by the machine, very selective "what if" explorations are generated during the interaction. Understanding how such selectivity is formulated dynamically will answer one of the major questions about what separates a good problem solver from a poor one.

- The performance records of the subjects (in terms of the numerical "score") show a dramatic separation between those who attacked the problem incrementally (immediate or short-term payoffs were considered) and those who had strategic plans for the problem situation as a whole. Those who scored in between the two groups made statements such as: "I had a rather vague plan, but I couldn't follow through with it because so many unexpected possibilities opened up as I played that I got distracted by attractive new prospects and deviated from the plan. Then things got too confusing, so I gave up the plan and played incrementally."

Those who scored highest seemed to have utilized intuitive pattern recognition (both spacial and abstract patterns) to structure the problem space and to make a rough estimate of cause-and-effect relation patterns.

It is our belief that good planning and selectivity in asking "what if" questions are related and that they are highly problem specific. Cleverness in both activities seems to depend on how astutely the subject discerns the idiosyncracies of the problem situations and takes advantage of them in formulating his plans and his "what if" questions. Such characteristics dictate that dynamic extensibility of machine functions is needed beyond the predetermined set of machine functions; many of the relevant questions and much of the selective exploration cannot be formulated at the time of the problem definition. Predetermined machine functions, however, should include any machine assistance that will make it easier for the man to express his tentative ideas and requests for exploration and to delegate certain decision functions to the machine once they are defined and identified as useful.

- The subjects unanimously agreed that visual display of their performance in graphic form and of environmental changes caused by their actions was helpful in assessing previous decisions and formulating new ones. However, more sophisticated techniques of summarizing the current "state of the environment" are needed to display information in a variety of formats and at varying levels of aggregation. At one point, the subject may be interested in the overall relational aspects; at another point, he may be interested in detailed information about one small portion of the environment. Again, the need for dynamic definability of man's ideas and requests became clear.

1.2 UAL AS A STEP TOWARD MAN-MACHINE SYNERGISM

UAL has been designed to fulfill the essential need for dynamic extensibility demonstrated by the Shimoku experiments. This extensibility and other features of UAL are discussed in Section 2.

An additional advantage can be gained by the use of UAL for the implementation of Gaku. Traditionally, a system is implemented in one language, and another language is specially developed to allow for user-oriented and/or problem-specific expressions. We have, instead, geared the design of UAL and associated techniques to serve both purposes without the compromises that would usually be required.

The basic UAL will be used for initial Gaku implementation and an extended UAL will be used for user-Gaku interaction and also for designer-Gaku

interaction in system modification. This will be possible through the use of the extensible features of UAL and of the techniques of building problem-oriented primitives from which higher-level, problem-oriented functions and capabilities can be constructed for the users' convenience. Thus the basic UAL can be used in the same manner as other programming languages but extended UAL can be made into a higher-level, English-like, and/or highly problem-specific language, depending on the users' needs and convenience.

The extensibility of a language refers to its ability to modify itself--that is, its ability to create new primitive terms and functions and to define new infix operators. This becomes important when the problem situation dictates a new notation that the original language does not accept or when new primitive terms and functions are required to reduce complexity. These new terms and functions may not be known at the outset but, through interaction, new ideas may be generated and the need for new terms and procedures realized. Thus, extensibility permits dynamic definition.

With these features, the user can start interacting with Gaku from the initial problem-conceptualization and problem-definition stage and continue to the intuition-guided, hunch-generation-and-testing stage of problem-solving (and perhaps back to the problem-definition stage to repeat the process). The conventional separation between the problem-definition and the problem-solving stages caused by specialists or programmers, or by extra languages, is not necessary. Gaku implemented on UAL can handle user-defined terms and procedures directly, without internal translation or mode changes, since expressions in UAL are the basic items that Gaku can understand, generate, and manipulate. For partially-defined or ill-defined problems, the problem-definition stage cannot be separated from the problem-solving stage because of the iterative nature of the two. The former can be thought of as a model-building process and the latter as model exercising, expecially when the model includes a man to provide the behavioral-procedural information that was not known or could not be made explicit at the outset. Gaku can bring these together by allowing the undefined portions to be supplied by the user in the light of new information and insight as supported by his own background knowledge and past experience. Use of the basic UAL in constructing Gaku and also in defining a given problem environment is shown schematically in Figure 1.

Whenever a new problem environment is considered, the man first introduces a set of primitive terms and functions to UAL in order to establish a semantic link between basic UAL and the problem environment. Once the semantic linkage is established, the Gaku system, which is built on top of UAL, becomes a problem-oriented or special-purpose system. Then, given that the primitives are adequate, the man can communicate any statement or questions about the problem environment and can define any operations in it. In Figure 1, this set of primitives is represented by the solid box on the right.

In practice, however, it will be easier and more efficient conceptually if the modeling system is equipped with higher-level terms and operations that are

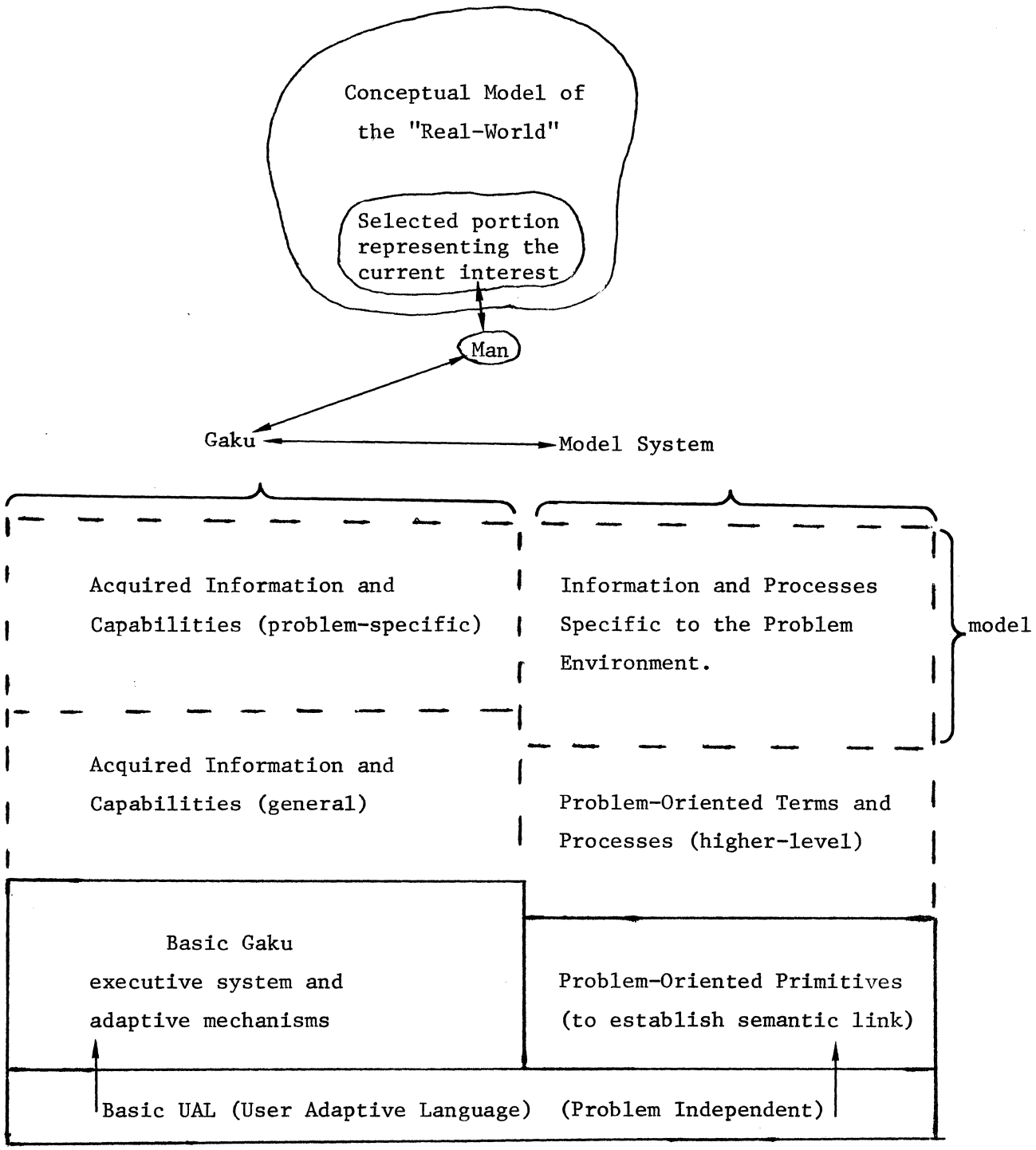


Figure 1. A schematic diagram depicting man-machine cooperative endeavor in both defining a problem (or building a model) and solving the problem (or experimenting with the model).

more amenable to man's levels of thinking. Representing these is a dotted box labeled "Problem-Oriented Terms and Processes (higher-level)," placed on top of the box of "Problem-Oriented Primitives." The model is built on top of this. Dotted boxes imply that contents are subject to change and expansion.

Intensive research is still required to make the semantic linkage as easy to build as possible so that a user need not feel committed to his first choice of primitives but can start with a "quick-and-dirty" version to experiment with and change it as he discovers its inadequacies. After we have gained experience by building primitives for a number of different problem types, we may be able to extract a set of commonly used steps for primitive building. From this, a set of "primitive-building primitives" may be made available for a quicker and easier definition of new problems.

When model building becomes truly quick and cheap, man may be encouraged to try out different formulations or representations of the same problem situation, thus gaining different viewpoints, one of which may reveal a lead to an answer or to solution methods that could not have been discovered in any other way.

2. DESCRIPTION OF UAL (USER ADAPTIVE LANGUAGE)

The User Adaptive Language (UAL) is a functionally oriented, extensible, problem-solving language. In addition to many new concepts and ideas, the language incorporates desirable features from existing languages in a unified and consistent manner that makes them easy to use and learn. As well as being user adaptive, the language is user oriented. It is hoped that UAL can be effectively used by those not directly in the computing field.

UAL is designed to be used interactively through a remote terminal so that an immediate response from the computer is received for each and every input. Every expression is evaluated (executed) as soon as it is entered, rather than being stored for evaluation at a later time. However, it is possible to inhibit evaluation if desired. In addition to a number of available predefined functions that aid the user in problem-solving, the language has five different data types: (1) numbers, (2) character strings, (3) quoted expressions, (4) argument maps, and (5) lists. "Character strings" are useful if nonnumerical applications are involved. "Quoted expressions" and "argument maps" provide a convenient, flexible, and powerful means of defining new functions. These functions may be made English-like in use or compact and stylized; they may be general-purpose functions or problem-specific primitives. A "list" is composed of groups of data elements which can be treated as a unit. Sublists and superlists can be formed into structures or networks. These data types, coupled with predefined functions plus the rules for their manipulation, make up the User Adaptive Language.

Some important features of the language are summarized below:

- The basic data structure in UAL is the list. Elements in a list may be numbers, character strings, other lists, or even functional expressions. Thus, procedures may be stored and manipulated in the same manner as simple data items. In adddition, two or more lists may share members so that each is "aware" of a change in the other.

- There are two types of assignment in UAL: pointer changing and value changing. Each variable "points to" and is separate from its value. Thus, either the value itself may be changed, or the variable may be caused to point to a new value.

- Arbitrary functions and infix operators may be defined or redefined, depending upon the user's preference. New terms and primitives may be created to fit specific needs in a given problem situation and to express complex ideas in a clear, readable fashion. The user may also make his new functions general and more English-like.

- The language is extensible, which means it is capable of redefining its own parts. The user may change the action of any built-in function or even the way expressions are processed after entry.

- The language is capable of supporting a semantic linkage for a given problem environment by defining a set of problem-oriented primitives. This allows more specialized problem-oriented languages to be built upon it.

- The normal mode of UAL is evaluation. Expressions are evaluated (executed) immediately upon entry. This makes the language naturally suited for interaction. However, a means is provided to suppress evaluation when desired so that expressions may be stored and subsequently manipulated without evaluation.

- A great deal of power and flexibility is provided for functional definition. The user has control over argument evaluation, the exact place where the function name is to appear, the scope of variables, and other features that allow a large variety of functions to be defined.

- The user can specify directly to UAL environmental conditions that describe the context within which he will work. UAL will not allow these conditions to be violated and will warn the user if he attempts to do so.

The language is described using a somewhat idealized character set for optimum understanding of the concepts involved. UAL has been implemented at System Development Corporation in Santa Monica, California on the IBM 360/67 ADEPT time-sharing system (Linde, et al. [1969]) using LISP 1.5 as the source language (Weissman [1967]).

2.1 BASIC ELEMENTS

The formation of numbers, variables, and character strings follows standard rules. Numbers may be integer or real, that is, with or without a decimal point. Variables consist of one letter followed by zero or more letters or digits. A character string is a sequence of characters enclosed in double quote marks. In addition, a number of primitive functions are available for forming arithmetic, relational, and logical expressions:

+ addition
- subtraction
* multiplication
/ division
÷ integer division
↑ exponentiation

%	percent
<	less than
≤	less than or equal to
=	equal to
≠	not equal to
≥	greater than or equal to
>	greater than
==	identical to
~	negation (not)
∨	disjunction (or)
&	conjunction (and)
→	implication
≡	equivalence
≢	nonequivalence

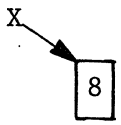
These functions are used with parentheses in forming expressions. Each has an imposed precedence. (See Appendix B)

2.2 ASSIGNMENT

Values may be assigned to variables by using the assignment function, back-arrow (←). For example:

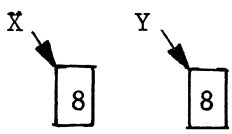
```
X←8
```

assigns the value 8 to the variable X. The above expression is read, "X is defined as 8." The result of this assignment may be represented graphically as follows:



The picture is intended to show that the variable X is separated from and points to its value 8. If the number 8 were assigned to another variable, that variable would point to a second representation of the number. For example:

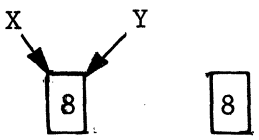
```
Y←8
```



It is possible for two or more variables to point to the same value. For example:

```
Y←X
```

would reassign Y to the value that X has:

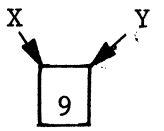


Since no variable points to Y's old value, it is lost and its storage space may be reused. This "garbage collection" process occurs periodically as the need arises.

The backarrow, then, causes a variable to point to a new value. It is also possible to change the actual value itself. This, of course, means that every variable which points to that value would also be changed. The function that performs this operation is the double backarrow (←←). For example:

```
Y←←9
```

may be read "Y is changed to 9."



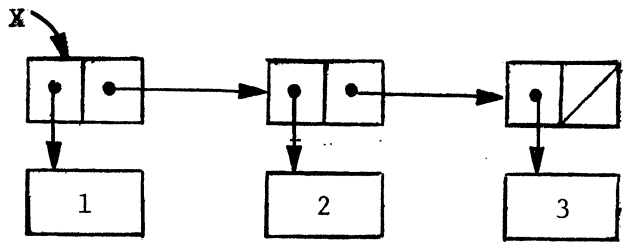
The primitive function "is identical to" (==) may be used to test whether or not two variables point to the same value. This function is stronger than equals (=), which only tests for equality.

2.3 LISTS

A useful data type of UAL is the list. A list is a series of double-pointer nodes. The left-hand side points to the value and the right-hand side points to the next node. A variable whose value is a list points to the first node in the list. The right-hand side of the last node points to an indicator that signals the end of the list. To form a list, the elements are written sequentially with no punctuation between them and are enclosed in braces. For example:

```
X←{1 2 3}
```

may be pictured as follows:



In addition to the brace formation of a list, there is a predefined function that causes a list to be formed from its arguments. The function is the comma. The above list could have been equivalently specified:

```
X←1,2,3
```

It is sometimes necessary to have lists of numbers that are in arithmetic progression. However, such lists may be too long to specify explicitly. A shorthand notation for such lists is available. For example:

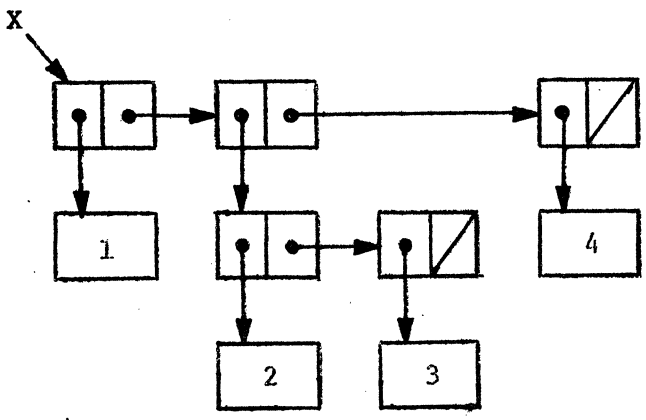
```
{2 5 ... 35}
```

The above list contains 2 as the first element, 5 as the second element, and continues in this manner (that is, incrementing by 3's) until 35 is reached or exceeded.

Sublists and superlists may be formed simply by nesting braces. In this event, the left-hand side of the particular node will point to the first node of the sublist. For example:

```
X←{1 {2 3} 4}
```

would produce the following list:



Individual elements of a list may be referenced by the locative functions ST, ND, RD, and TH. Using the above example:

```
3 RD X    produces 4
1 ST 2 ND X    produces 2
```

Variables may be assigned to particular parts of a list simply by using the assignment arrow and the desired locative functions.

```
Y ← 1 ST X
```

will point Y to the number 1, which is the first element of the list X. Since the locatives are actually functions, any expression at all may precede them as long as an integer is returned as a value. Similarly, in the case of lists, any expression at all may be placed inside the braces. The expression will be evaluated and the result used as the list member.

The individual nodes of a list may be accessed with the node-locative functions STN, NDN, RDN, and THN. These functions must be used when changing the value of elements of a list.

Most primitive arithmetic and relational functions are defined to iterate over the elements of a list. For example:

```
3+{2 4}    produces {5 7}
```

If two lists are used with these functions, their elements are taken one by one in parallel from each list.

```
{3 5}↑{0 2}    produces {1 25}
```

If one of the lists is short in elements, it is extended with zeros or ones, depending upon the particular function involved.

The list of no elements at all is called the "empty list" and may be written in either of the following two ways:

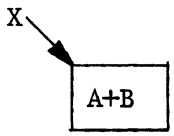
```
{}    ϕ
```

2.4 EVALUATION

Evaluation occurs when an expression is entered into the computer through the remote terminal. However, it may be desirable to inhibit the evaluation of all or part of the expression. In this way, it can be stored and saved for evaluation at a later time. Any expression that is enclosed in single quote marks will be inhibited from evaluation. For example:

```
X←`A+B´
```

produces the following:



The expression A+B is inhibited; X← is not inhibited, so that the assignment takes place. Later, if X is called upon to be evaluated, either alone or as part of another expression, its value (that is, A+B) will be evaluated also. The act of inhibiting is called "quoting" and may be done to any depth. In addition to the quote-marks, there is a function that quotes the next complete expression after it. It is the colon. The above example could also have been written:

```
X←:A+B
```

The predefined function EVAL causes an extra evaluation each time it is used. Extra evaluations may be necessary if an expression is quoted more than once. The predefined function VALUE may be used to access the value of a variable without evaluating it.

2.5 OPERATIONALS

An operational is a sequence of two or more expressions that is treated as a group--that is, as one expression. The expressions in an operational are enclosed in parentheses and may be placed wherever any other single expression may be placed. For example, the following is an operational of three expressions:

```
(X←3   Y←{1 2}   Z←"ABC")
```

Since an operational is a single expression, it must have a single value. By convention, the value of an operational is the value of the first expression in it. The value of the above operational is 3. In the following example, the intention is to evaluate (B/3)*H and assign this value to A. However, the assignment B←A is meant to be performed after the division by 3 but before the multiplication by H:

```
A←(B/3   B←A)*H
```

The return value of an operational may be reset by using the function SETOP. An interrupt of the normally sequential evaluation of the expressions in an operational may be caused with the function RETURN. An operational may also be thought of as a single expression followed by a series of side effects.

2.6 FUNCTIONS

A quoted expression is a simple function of no arguments and is evaluated when its name is evaluated. To define a function with arguments, the bound variables of that function must be listed, enclosed in brackets, [], and placed in front of the quoted expression (function definition). The list of bound variables is called the argument map.

For example:

```
F←[X Y]:X+Y+2
```

defines a function F of 2 arguments X and Y such that F equals X+Y+2. In order to call F, its name should be given, followed by 2 arguments (expressions), either alone or in another expression. For example:

```
F 2 3
```

would produce 7 as a result. No parentheses, commas, or other punctuation are used in a function call. However, since a function name plus its arguments form a complete expression, parentheses may be placed around the entire group.

The arguments may be any expression at all including other function calls. A function always has as many arguments as bound variables in the argument map. If a function is called with fewer arguments than it needs, a sufficient number will be supplied. For example:

```
(F 4)
```

would produce 6 with zero being supplied for Y.

2.7 ARGUMENT MAP

The argument map has many features that make function definition powerful and flexible.

An argument may be received in quoted form, even though it is not explicitly quoted at call time. This is indicated by placing single quote marks around the bound variable in the argument map. For example:

```
F←[`X´ Y]:(P←X  Q←Y)
```

The above function assigns its two arguments to P and Q, respectively, but asks for its first argument in quoted form.

```
F A+2   3+2
```

would assign the expression A+2 to P and the value 5 to Q.

Function calls may be made more readable by using "noise words." These are variables that are enclosed in parentheses and may be used as optional words in the function call. For example:

```
MOVE←[(THE) X (FROM) Y (TO) Z]:definition
```

The function MOVE could be called in either of the following two ways:

```
MOVE PIECE SQUARE1 SQUARE2
MOVE THE PIECE FROM SQUARE1 TO SQUARE2
```

Functions may possess variables that do not pick up a value from an argument. These "local variables" are considered undefined every time the function is called. They are listed after the regular bound variables and separated from them by a semicolon. In the following example, K is a local variable:

```
F←[X Y;K]:definition
```

All functions defined as described above are used in prefix form--that is, the function name first, followed by the list of arguments. It is possible to cause the function name to appear anywhere among the arguments by placing a caret (^) at the point in the argument map where the name is to appear. For example, suppose that two numbers are to be combined in a predefined manner. The function that performs this combination could be defined so that its name appeared before, in between, or after its two arguments:

```
COMBINE←[X (WITH) Y]:definition
COMBINE 3 WITH 5
```

or

```
COMBINED←[X ^ (WITH) Y]:definition
3 COMBINED WITH 5
```

or

```
COMBINATION←[X Y ^]:definition
3 5 COMBINATION
```

No noise words may precede the caret.

Function names may be picked up as arguments in character-string form by enclosing the bound variable in the argument map between double quote marks. For example:

```
F←["V"]:definition
```

This differs from picking up the argument in quoted form because the function is not structured with its arguments into a complete expression--only the name is passed.

It may be well to digress for a moment and explain that it is to character strings that assignment is made. The correct (and acceptable) form of the assignment expression is:

```
"A"←5
```

The double quotes may be omitted because the function ← has an argument map that picks up its first argument in string form.

```
["X" ∧ Y]:  assign Y to X
```

If any other expression appears in the place where a string argument is to be picked up, it is evaluated normally. Consequently, indirect assignment is possible. For example:

```
M← "N"
```

assigns "N" to M, and

```
(M)← 7
```

assigns 7 to N

Bound variables in a function have no connection or reference to variables with the same spelling that may be defined outside the function. For example:

```
X←4
F←[X]:X↑4
```

The bound variable X in the function F is not the same X as the global or free X defined above and given a value of 4. Only the variables listed inside the argument map are bound. If X appeared in the definition of F but did not appear in F's argument map, then X would indeed refer to the free X defined outside. For example:

```
X←4
F←[Y]:X+Y
```

In this case, the X in the definition of F does refer to the X whose value is 4.

Once a variable has been bound, it may be necessary to refer to the free variable of the same spelling. This is done by placing an exclamation point after the variable name. For example:

```
X←4
F←[X]:X+X!
F 2      produces 6
```

A third type of variable, in addition to the bound and free variables, is the globally bound variable. It is defined simply by placing an exclamation point after any bound or local variable in the argument map. A globally bound variable has the property that it will be bound not only in the scope of the defining function, but also in any function that is called in which it occurs free. For example:

```
X←4
F←[W X!]:X + G W
G←[Y]:Y+X
```

In the above example, G is defined to be a function of one argument, which adds it to the value of the free variable X. A call to G, such as:

```
10 + G 2
```

would produce 16 as a result. However, F is defined with X as a globally bound variable in a definition that calls G. This effectively binds the X in G so that it is no longer referring to the free X.

```
F 2 10   produces:
10 + G 2   which produces:
22
```

In all of the examples given above, the expression that was the function definition was specified explicitly as a quoted expression. The function definition must be quoted, but it need not be given explicitly. For example:

```
F←[X Y]:SQRT X↑2+Y↑2
```

could be written as follows once EXPR is defined:

```
EXPR←::SQRT X↑2+Y↑2
F←[X Y] EXPR
```

The expression EXPR must be quoted twice when written so that it will return an expression when evaluated. In addition, the argument map itself may be saved by including a colon somewhere inside the brackets.

```
ARGMAP←[: X Y]
```

Now F may be defined:

```
F ← ARGMAP EXPR
```

The variables in the argument map ARGMAP are bound in the expression EXPR.

The argument map may be overridden at call time by explicitly specifying the number of arguments a variable is to possess. If the argument-forcing option is used, the function call must be in prefix form without any of the features normally available in the argument map. The number of arguments follows the variable name and is separated from it by a semi-colon. For example:

G;3 *argument argument argument*

causes the function G to pick up 3 arguments regardless of its current definition.

Argument forcing is useful for functions that have not yet been defined, for functions that are intended to be redefined, or for forcing a function to pick up more arguments than its definition allows. In the latter case, the values of the extra arguments are assigned respectively to any local variables the function may possess.

The function definition may be any expression at all, including other function definitions. For example:

```
F←[X]: G←[Y]: Y↑X
```

In the above example, F is a function of one argument X, which, when called, defines another function G of one argument Y. The function G is not defined until F is called for the first time. Notice that the variable X is free in G but bound in F. The expression

```
F 4
```

would define G as Y↑4. Every time F is called, G is redefined.

A function could contain instructions to redefine itself:

```
F←[N]:(1/N  F←[X]:X/N)
```

The above function returns 1/N on the first call and X/N on every call afterward. The following example shows a function F that redefines itself after every call:

```
G←[Y]:F←[X]:G Y+X
```

2.8 STATE CONDITIONS

A state condition is a declaration of the environment in which further investigation is to take place. In order to specify a state condition, the user simply encloses any UAL expression between vertical bars. The expression is evaluated at the time it is specified and returns a value of true or false according to the current state of affairs. From that time on, however, no actions will be allowed that would cause the state condition to become false. For example:

```
|-1≤FACTOR≤1|
```

If, after the above specification, an attempt were made to redefine FACTOR outside the given range:

```
FACTOR←2.8
```

the redefinition would not occur and FACTOR would retain its old value.

The user may specify many state conditions that are active simultaneously either in a global context or local to a particular function.

2.9 BUILT-IN FUNCTIONS AND LANGUAGE EXTENSIBILITY

UAL has a vast number of helpful built-in functions. The functions fall into the following groups: arithmetic expressions, logical expressions, relational expressions, list manipulation, evaluation, iteration, conditional expressions, input/output, function editing, and function debugging.

The iteration function is FORALL. This function evaluates an expression once for each member of a specified list and returns a list containing the result of each evaluation. For example:

```
FORALL   Xε{1 2 ... 50}   2↑X
```

returns a list of powers of 2 from 1 to 50. The variable X in the above example is a control variable that assumes successive values from the specified list during iteration. The control variable is optional and has the status of a local variable throughout the scope of the FORALL function.

There are also options for skipping elements of the specified list or prematurely terminating the iteration. They are WHENEVER, UNLESS, WHILE, and UNTIL. Each of these functions must be followed by a logical expression and controls the iteration in a different manner. WHENEVER prohibits evaluation when its logical expression is false; UNLESS prohibits evaluation when its logical expression is true. WHILE and UNTIL terminate the iteration when their logical expressions are false and true, respectively. Any number of the qualification functions may be used in a FORALL and in any order. They must appear immediately after the specified list when used. For example:

```
FORALL I∈DATA WHENEVER I<0   I←→0
```

places a lower limit of zero on the elements of the list DATA.

Conditional expressions may be formed using the IF function of two arguments, which evaluates an expression only if the outcome of a given logical expression is true. The IFE function of three arguments evaluates one of two expressions, depending on whether the given logical expression is true or false. For example:

```
IF FLAG="ON" & 0≤X≤1 THEN (X↑2 FLAG←"OFF")
GRADE←IFE SCORE≥70 THEN "PASS" ELSE "FAIL"
```

The noise words "THEN" and "ELSE" are optional.

For the input/output group there are, among others, a PRINT function that causes values and lists to be printed and a READ function that requests an expression to be entered at the terminal. The expression is evaluated and the result returned as the value of READ. The function READQ treats the input as though it were a quoted expression. Through the use of these functions, new supervisors may be defined that handle input in nonstandard ways. For example:

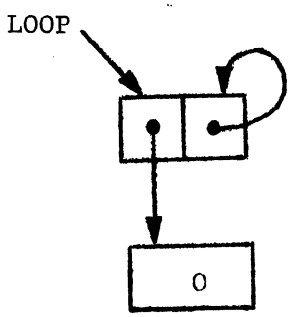
```
FORALL LOOP EVAL PRINT READQ
```

echos back the input before evaluating it normally.

```
PROGRAM←FORALL LOOP UNTIL (EXPR←READQ)=`END´ EXPR
```

EXPR stores a list of quoted expressions into PROGRAM for later evaluation.

As shown above, the user may create an indefinite number of iterations with the predefined variable LOOP.



UAL has no special forms or special functions that are handled differently than user-defined functions. If the user is not satisfied with the operation of a particular function, he is free to redefine it to suit himself. For example:

```
+ ← [X ∧ Y]:(X-Y)↑2
```

The above example redefines + to return (X-Y)↑2 rather than X+Y.

```
7+3  produces 16
```

or, + could be redefined as follows:

```
+ ← [X ∧ Y]:(SUM X,Y   INCREMENT C)
```

where SUM is a built-in function that adds elements of a list and INCREMENT is a function that adds 1 to a variable. The redefined + still adds X and Y, but it also keeps track of the number of additions that have been made.*

UAL, then, is extensible, which means that it is capable of redefining its own parts. (For other extensible languages, see Christensen and Christopher [1969], and Smith [1970].)

2.10 THE CONTROL CHARACTER

The character # is a metacharacter used to control the input line that is being typed on a remote terminal keyboard. A carriage return is enough to enter a line for evaluation. The current input line may be deleted by typing # followed immediately by the letter D.

---

*The function + could not be used in its own definition in this case, since that would have produced a recursive function.

```
1+2+3+#D4+5
```

enters only 4+5. A specific number of immediately previous characters may be deleted by typing one or two digits after #:

```
V+W+X+Y+#5+Z
```

enters V+W+Z. Comments may be placed anywhere (even inside numbers and variable names) as follows:

#[*comment*]#

The current input line may be edited character by character if a mistake is detected before the line is entered. The # is followed by two character strings separated by a comma and enclosed in parentheses. The current input line is scanned for the first occurrence of the first character string and, when a match is made, the duplicate is replaced by the second character string. For example:

```
F←[X Y]:X+Y+Z#("Y","Z")
```

would enter the line as if it had been written

```
F←[X Z]:X+Y+Z
```

A second example:

```
QUES←"WHERE ARE Y#("ERE","O")OU?"
```

would enter as:

```
QUES←"WHO ARE YOU?"
```

Omitting the comma and second character strings deletes the match when found. A number may precede the first character string to indicate the occurrence.

```
#(4"E","N'T")
```

matches the fourth occurrence of E.

The example above would enter as:

```
QUES←"WHERE ARN'T YOU?"
```

The following list gives the possible characters that may appear after the control character and their meanings.

#D	Delete all of the current input line so far.
#C	Request for one continuation line.
#[	Begin Comment.
#]	End Comment (also may be]#).
#(	Begin Edit Field.
#W	Wait - Request for an indefinite number of continuation lines.
#E	Evaluate - Process the input regardless of how many continuation lines have been requested.
#P	Print out the current input so far and request one continuation line.
#!	The # character itself.
##hh	Enter the character with hexadecimal representation hh.
#dd	Delete dd characters back. (One or two digits may be specified.)

2.11 PREDEFINED FUNCTIONS

Many predefined UAL functions are available to the user. The argument map and a description of the definition is given with each function below. The type of argument required is indicated by its letter.

N - Number
I - Integer
L - List
S - Character string
V - Variable name
R - Relational or logical expression
X - Any expression

Any expression may serve as an argument for the above types as long as the value returned is of the correct type.

Arithmetic Expressions:

SQRT←[N]:	square root
SUM←[L]:	adds elements in the list L
SIN←[N]:	sine
COS←[N]:	cosine
ARCTAN←[N]:	arctangent
LOG←[N]:	log to the base 10
ELOG←[N]:	log to the base e
EXP←[N]:	e^n
MAXIMUM←[L]:	maximum element in L
MINIMUM←[L]:	minimum element in L
INCREMENT←["V"]:	adds 1 to V
DECREMENT←["V"]:	subtracts 1 from V
CONVNS←[N]:	converts N to a character string

Character Strings:

```
CONCAT←[S1 (AND) S2]:   string concatenation
CHAR←[I (IN) S]:  selects Ith character from S
EXPLODE←[S]:   returns list of single characters of S
COMPRESS←[L]:  concatenates elements of L
CONVSN←[S]:    converts S to a number if possible
NAME←[S]:      converts S to a variable
```

Logical Expressions:

```
TRUE←1
FALSE←Ø
CONVØ1←[X]:    interprets an expression as true or false (0 or 1)
```

Lists:

```
COPY←[L]:      copies list L
ST←[I ∧ L]:    selects Ith element of L
ND←ST
RD←ST
TH←ST
ON←[I ∧ L]:    returns remainder of list L from Ith element on
STN←ON
NDN←ON
RDN←ON
THN←ON
APPEND←[L1 (AND) L2]:  appends L2 onto the end of a copy of L1
LAST←[L]:      last element of L
LENGTH←[ (OF) L]: number of elements of L
JOIN←[L1 (AND) L2]: attaches L2 at node L1.
```

Evaluation:

```
EVAL←[X]:      evaluates X
VALUE←["V"]:   returns what V points to without evaluating V
EVALST←[S]:    forms an expression out of the string S and evaluates it
QUOTE←[X]:     evaluates X and quotes the result
```

Operationals:

```
RETURN←:       premature return from an operational (abbreviated RET)
RETURNR←[X]:   premature return from an operational with the result
                 (abbreviated RETR)
SETOP←[X]:     resets the value of an operational
OPVAL←:        current value of the operational
```

Conditionals:

```
IF←[R (THEN) `X´]:  evaluates X if R is true
IFE←[R (THEN) `X1´ (ELSE) `X2´]:  evaluates X1 if R is true and X2 if R
                                  is false
```

Iteration:

```
FORALL←[L (DO) `X´]:  iterates X for each element of L
WHENEVER←[R `X´]:     returns X if R is true, and "SKIP" if R is false
UNLESS←[R `X´]:       returns "SKIP" if R is true, and X if R is false
WHILE←[R `X´]:        returns X if R is true, and "STOP" if R is false
UNTIL←[R `X´]:        returns "STOP" if R is true, and X if R is false
IN←["V" ∧ L]:         returns {"IN" V L}
LOOP←:                list of indefinite number of elements
```

State Conditions:

```
STATES←:       prints out a list of state conditions with reference numbers
DELSTATE←[I]:  deletes state condition number I
```

Input/Output:

```
READ←:          returns an evaluated expression from the terminal
READQ←:         returns a quoted expression from the terminal
READST←:        returns terminal input as a character string
SPECIFY←["V"]: asks for an input from the terminal and assigns it to V
PRINT←[X]:      formatted print
OUTPUT←[X]:     unformatted print
STASH←[X]:      holds X until the next PRINT or OUTPUT is called, then
                outputs X
PRINTOFF←:      deactivates automatic printing of value of every expression
                that is entered
PRINTON←:       activates automatic print
```

Debugging:

```
LINEPROMPT←[S]:changes line prompt to string S
STACKSTATUS←:  prints out current status of waiting argument stack
FREESPACE←:    calls garbage collector and prints out number of words of
               freespace left
FREE←["V"]:    returns variable V to free storage
MORENAMESPACE←:gets more space for variable names
TYPE←[X]:      gives type number of X
```

Protection:

```
UNPROTECT←["V"]:  unprotects variable V for redefinition
PROTECT←["V"]:    protects V
```

3. USE OF UAL: AN EXAMPLE

3.1 PO&M SYSTEM (PLANNING ORGANIZATION AND MANAGEMENT SYSTEM)

The following example is taken from "An On-Line Interactive Hierarchical Organization and Management System for Planning," [Kleine and Citrenbaum, 1970].

The problem is that of convention planning. Let us suppose that a chairman has already been selected for setting up a large convention or conference. The chairman, possibly with other co-chairmen and assistants, now proceeds to define a number of areas of responsibility or activities that can be delegated to other individuals. He might define the areas such as "publicity" and "accommodations" and assign individuals to head the areas and their assistants. These heads, who have been appointed by the chairman, may themselves delegate portions of their tasks to their subordinates. Thus a hierarchical structure results, with "nodes" representing subdivided areas of activities.

A representative hierarchical organization created by a planning group to meet the needs of convention planning is shown schematically in Figure 2. Here the CHAIRMAN has set up a CHAIRMEN's COMMITTEE and TREASURER, PUBLICATION, MEETING ROOM, PUBLICITY, and LUNCHEON ACTIVITIES. He will define the duties and responsibilities of each Activity, name a head and staff members as appropriate, and communicate with and request reports or information from them as the planning proceeds. He will also create additional Activities as the need arises and delete those whose work is completed. The TREASURER establishes formal communications and reporting relationships with the major Activities, and PUBLICATIONS sets up a subactivity, PRINTING, which PUBLICITY also uses for its printing needs.

Formalizing the Concept of Activity. Let us consider the elements needed in the PO&M system in order to assist the planners in setting up their organization and communicate within it. First, all of the information which might be associated with an activity is listed (Figure 3). This includes the name or title of the Activity; a description of the job to be done by that Activity; a status indicator to tell others whether the activity is unstarted, finished, working, or waiting. It also includes a set of links connecting the Activity with superiors, subordinates, and associates on the same level. These links are used for normal instructions, reports, and communications. Communication can take place outside of these channels, but automatic distribution of information down through the organization and the forwarding of reports upward takes place along these lines.

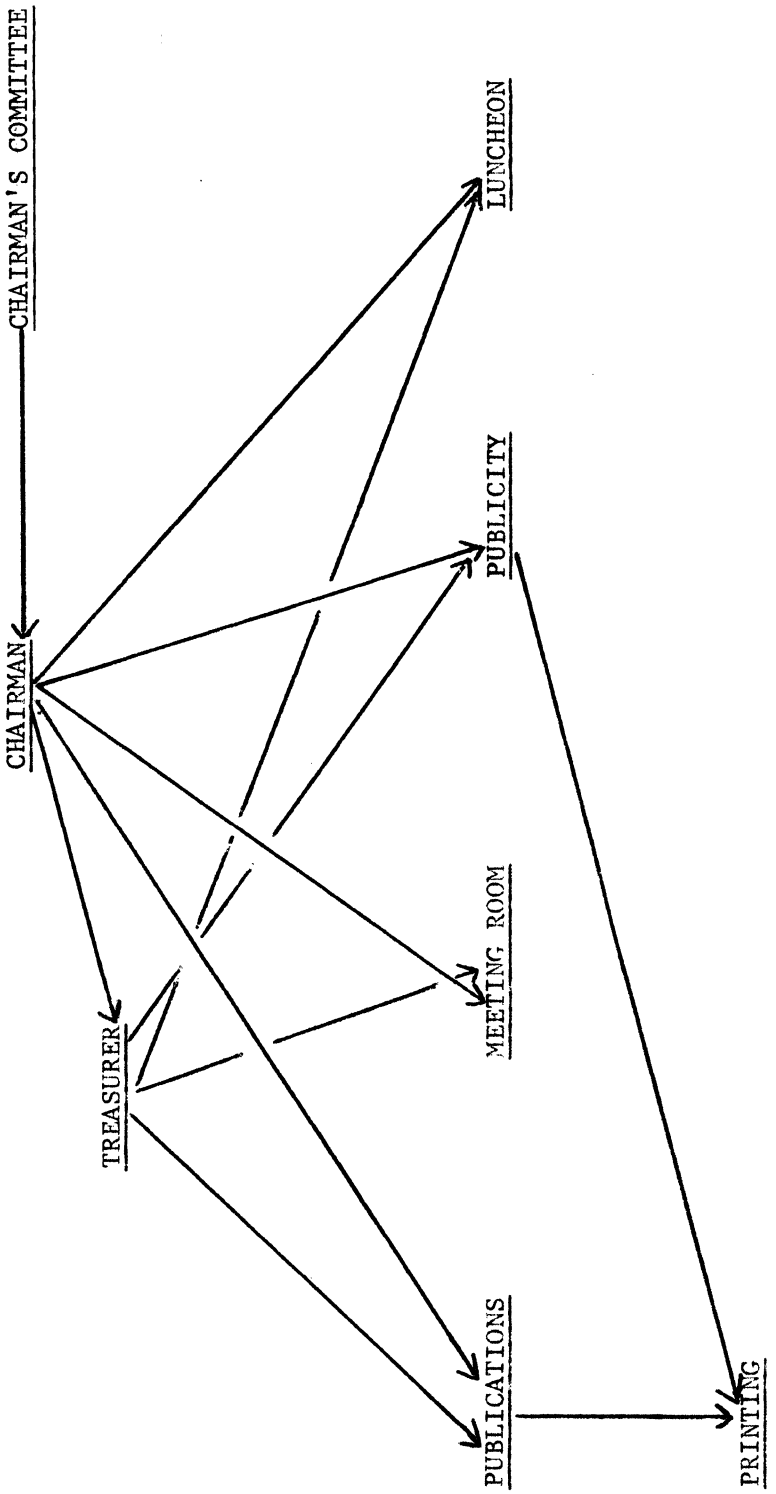

CHAIRMAN'S COMMITTEE
CHAIRMAN
LUNCHEON
PUBLICITY
MEETING ROOM
TREASURER
PUBLICATIONS
PRINTING

Figure 2

ACTIVITY

NAME
DESCRIPTION
STATUS
LINKS:
SUPERIOR:
PARENT: LINK, NAME, FORWARD, DATA
OTHERS
SUBORDINATE
LATERAL
MEMBERS: HEAD, OTHER MEMBERS
DURATION
RESPONSE: REQUESTED, EXPECTED
TASK_INPUT
TASK_OUTPUT
DATA

GROUP

NAME
LINKS: SUPERIOR, SUBORDINATES, LATERAL
MEMBERS:
.
.
.

Figure 3

Each of the three communication-link types consists of groups of links. The first of the superior links is the parent or direct superior link. Each link is itself made up of a link-pointer, which points to the associated activity; the name of the linkage; an indicator whether information is to be automatically forwarded along this path; and any data and/or textual information used with this communication link. MEMBERS is also a group in which each element is an individual member of the group associated with the Activity. The head of the group is listed as the first member and, if there is no head, that position is left empty.

DURATION is the best estimate of the total duration of this Activity. RESPONSE is composed of two parts: (1) REQUESTED, the response interval requested by the inquiring party, and (2) EXPECTED, the anticipated interval before this Activity responds to the inquiry. The TASK_INPUT is a combination of data and/or text information specifying the task requirements of this activity. Similarly, TASK_OUTPUT contains the results of the Activity's work, and DATA contains information accumulated for use in accomplishing the Activity's tasks. (For the sake of readability, the underline character will be used in the formation of variables and should be considered a letter.)

Organizational Elements. A given activity may not need or make use of all of the elements described above, but all categories remain available for use as required. Organizational types other than activities--such as groups--can be accommodated using some sublist of the elements in an Activity:

Designer/User Notation. A system can be built in UAL from the above general PO&M system requirements. Two types of instructions will be given in the language: the "system designer" will write instructions to build up the elements of the system, and the "project organizer" (the user) will write instructions in the new language of the augmented UAL system--that is, in the language of the UAL that has been augmented with PO&M system-building functions.

3.2 PRIMITIVE FUNCTIONS AND BUILDING BLOCKS FOR PO&M SYSTEM

In the following figures, the system designer's instructions will be in capital letters and those of the organizer-user will be in capitals with each instruction line set off with an asterisk. The designer will initially need to formalize the concept of an Activity and the elements of which it is composed. This can be done by treating an Activity as a list of data elements, some of which are lists themselves.

Figure 4 shows an example of the CHAIRMAN organized as an Activity. The lines represent the links of the Activity with other Activities and with individuals and their records. The entire Activity list is a rigidly formatted, non-user-oriented structure. It is necessary, for example, to ask for the

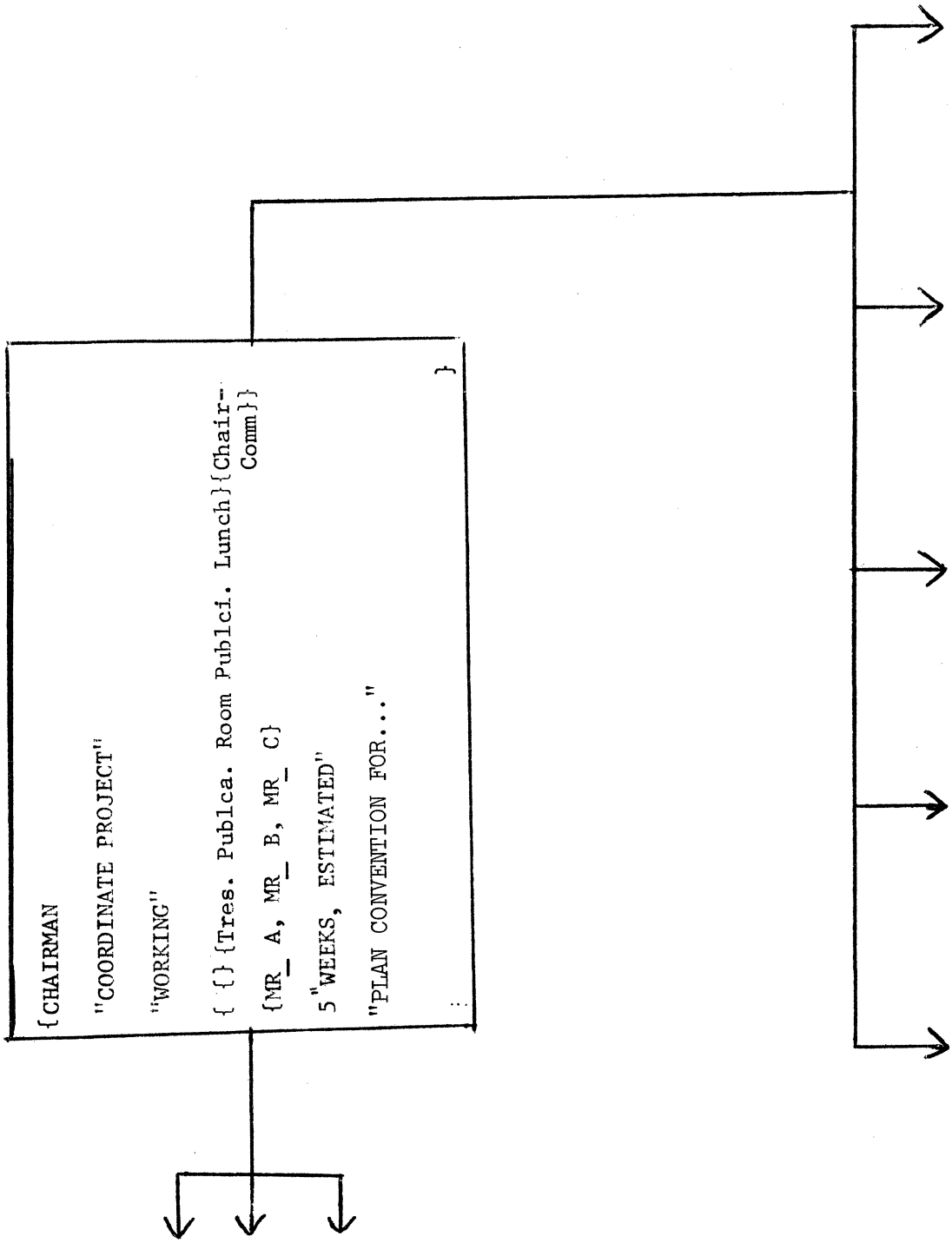

{CHAIRMAN
"COORDINATE PROJECT"
"WORKING"
{ {} {Tres. Publca. Room Publcl. Lunch}{Chair-
Comm}}
{MR_ A, MR_ B, MR_ C}
5 "WEEKS, ESTIMATED"
"PLAN CONVENTION FOR..."
}

Figure 4

value of the third part of an Activity in order to determine its status. The first step, then, in system design is to enable names to be used to access each part of an Activity while leaving the structural integrity of the Activity as a whole undisturbed.

Use of Names for Structure Parts. In Figure 5, NAME is defined as an expression that, when applied to an Activity ACT, will yield as its value the first or name part of that Activity. If NAME were applied to the chairman's Activity, it would yield "CHAIRMAN" as the value for the expression. Similarly, DESC and STATUS yield the description and status parts of an Activity. The list of superiors is the first item of a three-part list, which, in turn, is the fourth part of the Activity. The "parent" is the first item of the list of superiors, and so forth.

In a similar fashion, each of the elements or collections that make up an Activity is defined in a way that permits those elements to be referenced by name rather than by location within the structure.

Note that the definition of SUPERIOR has a "TO" inside the argument map. This permits the word to appear optionally in each use of the functional operator "SUPERIOR." For example, one could write SUPERIOR PUBLICATIONS or SUPERIOR TO PUBLICATIONS. Making use of these basic expression definitions, the designer can now build higher-level expressions.

Defining Activity-Creating Functions. One of the first things a user will want to do is to create one or more Activities. The first definition in Figure 6 permits this. The expression CREATE A CHAIRMAN creates the Activity CHAIRMAN, sets the name to be the word "CHAIRMAN", sets the status to "NOT STARTED", and leaves the remainder of the Activity's structure temporarily undefined. It also makes the Activity just created the value of CUR_TITLE (current title). The instruction WITH modifies the Activity that is the CUR_TITLE by inserting a name at the beginning of the MEMBERS list so that the name will be treated as the Activity head.

The function MEMBERS allows specification of the members other than the head. The function MAKE operates on two arguments and can insert a given name or value into any given position of an Activity. The last function, REORGANIZE, changes the Activity that is the CUR_TITLE and, therefore, the Activity that is changed by the WITH and MEMBERS instructions. With this handful of definitions, the user can write any of the instructions shown at the bottom of Figure 6, as well as an extensive number of others made up of various combinations of the newly created instructions.

"Top-Down" vs. "Bottom-Up" Approaches. The above is an example of "bottom-up" system design. Basic expressions were defined, and, out of these expressions, successively higher-level, user-oriented functions were defined. We will turn now to the capability of reversing this procedure in UAL.

ACTIVITY

```
     {NAME DESC STAT {SUP SUB LAT} MBRS DUAR RESP INPT OUTPT DATA}
NAME←[(OF)ACT]:   1 ST ACT
DESC←[(OF)ACT]:   2 ND ACT
STATUS←[(OF) ACT]:3 RD ACT
SUPERIOR←[(TO) ACT]:  1 ST 4 TH ACT
PARENT←[(OF) ACT]:    1 ST 1 ST 4 TH ACT
SUBORDINATE←[(OF) ACT]:  2 ND 4 TH ACT
ASSOCIATE←[(OF) ACT]:    3 RD 4 TH ACT
HEAD←[(OF) ACT]:  1 ST 5 TH ACT
MBRS←[(OF) ACT]:  2 ON 5 TH ACT
CMTT←[(OF) ACT]:  5 TH ACT
DUAR • • •
RESPR
RESPE
TASK_INPUT
TASK_OUTPUT
DATA
```

Figure 5

```
   CREATE ← [(A) "TITLE"]: CUR-TITLE ← TITLE, Ø, "NOT STARTED"

   WITH ← ["NAME" (AS HEAD)]:  (HEAD OF CUR_TITLE) ← NAME

   MEMBERS ← ["MEMB"]:  (MBRS OF CUR_TITLE) ←← MEMB

   MAKE ← ["NAME" (A) POSITION]:  (POSITION) ←← NAME

   REORGANIZE ← [(THE) "ACT"]:  CUR_TITLE ← ACT

*  CREATE A CHAIRMAN

*  MAKE MR_ K HEAD CHAIRMAN

*  CREATE A CHAIR_COMMITTEE WITH MR_ A AS HEAD    MEMBERS MR_ B, MR_ C

*  MAKE CONTROLLER SUPERIOR TO PARKING

*  REORGANIZE THE PUBLICITY_COMMITTEE WITH MR_D AS HEAD
```

Figure 6

"Top-down" design begins with the final desired expression form and then breaks that down into its constituent parts until there are no undefined parts. The user can indicate, for example, that he would like to be able to say "WHAT IS THE VALUE OF THE PUBLICATION "TOTAL-EXPENSES", and get an automatic response from the system, if the PUBLICATION group has defined its total expense in some form (see figure 7). The designer can start out by defining WHAT, a function that will be expected to find the definition that PUBLICATION has created for its total expenses. After writing the definition for WHAT, the designer can later complete the definition of any of the undefined parts of the expression. In this case, since EXPRESSION is undefined, a definition for it is provided next. Finally, the function DEFINE is defined. This last function can be used to add values or value-expressions to the TASK_OUTPUT of an Activity where they can be automatically interrogated by a superior Activity. An example of this is shown at the bottom of Figure 7.

Use of Expressions Defined by Other Activities. The chairman requests PUBLICATION to set up and keep current a best estimate of total costs. PUBLICATION does this and defines it based on its costs and the best cost estimate of the PRINTING subactivity. Now, when the Chairman interrogates PUBLICATION for total costs, not only will PUBLICATION calculate its costs and automatically respond, but PUBLICATION will automatically ask PRINTING for its current best estimate of costs.

MEMO Communications. The last area the system designer will deal with is communication. There are three types of communications or MEMO's: instructions, notes, and reports. Instructions are MEMO's from a superior to a subordinate; reports are MEMO's from subordinate to superior; notes are lateral flows of information. If not otherwise specified, MEMO's are of the same type as the relationship. For example, if an Activity sends a MEMO to a subordinate, it is an instruction unless otherwise indicated. Reports are automatically forwarded upward through those Activities designated by the user. Similarly, instructions are distributed downward automatically under the control of forwarding instructions. Although the report is the default MEMO from a subordinate, the subordinate may, if he wishes, specifically send a note to his superior in order to directly communicate information that he did not wish to be forwarded. In a similar fashion, notes or even reports can be sent.

Setting Up Basic Communication Functions. An example of the type of definitions needed for communication is shown at the top if Figure 8. The head of an Activity may wish to have an instruction that will give him all his messages when he checks with the system. Such a function is CHECK_IN. The second definition for CHECK_IN (shown after "of:"), instead of printing all MEMO's received to date, prints those now in the TASK_INPUT, and then transfers these to a MEMO-list for a record of past MEMO's. He could even use a MEMO-checking function, which automatically checks for MEMO's at regular hours of

TOP DOWN

* WHAT IS THE VALUE OF THE PUBLICATION "TOTAL_EXPENSES"

WHAT ← [(IS THE VALUE OF THE) ACTIVITY LABEL]:

EXPRESSION FOR LABEL OF THE TASK_OUTPUT OF ACTIVITY

EXPRESSION ← [(FOR) LABEL (OF THE) LIST]:

FORALL ITEMS IN LIST IF (NAME OF ITEMS)=LABEL THEN RETURNR 2 ND ITEMS

DEFINE ← ["LABEL" (TO BE THE) VALUE]:

(DATA OF MY_ACTIVITY)←← LABEL, VALUE

CHAIRMAN

* MEMO FOR PUBLICATION_COMMITTEE : "SET UP TOTAL_COSTS COMPUTATION"

PUBLICATION

* DEFINE TOTAL_COSTS TO BE THE SUM OF COST_OF_MATERIALS, COST_OF_PREPARATION, COST_OF_DISTRIBUTION + WHAT PRINTING TOTAL_COSTS

CHAIRMAN

* WHAT IS THE VALUE OF THE PUBLICATION "TOTAL_EXPENSES"

AUTOMATIC SYSTEM RESPONSE

2137

Figure 7

```
MEMO ← [(FOR) ACT (:) TEXT]:  (TASK_INPUT OF ACT) ← APPEND TEXT AND TASK_INPUT OF ACT

CHECK_IN ← :  FORALL MEMOS IN TASK_INPUT OF MY_ACTIVITY PRINT MEMOS

        or :  IF (TASK_INPUT OF MY_ACTIVITY) ≠ ∅
              THEN FORALL MEMOS IN TASK_INPUT OF MY_ACTIVITY
                                   (PRINT MEMOS    APPEND MEMO AND MEMOS_LIST)

REPORT ← [(TO) ACTIVITY (:) TEXT]:  . . . .

INSTRUCT ← [(TO) ACTIVITY (:) TEXT]:  . . .

NOTE ← [(TO) ACTIVITY (:) TEXT]:  . . . .
```

Figure 8

the day and, based on the anticipated contents, takes appropriate action--all without direct intervention.

The foregoing is intended to be not an exhaustive development of the PO&M system but an indication of (1) the type of system that could be developed, (2) the instructions needed to establish such a system, and (3) the power and flexibility of UAL in building other, quite different systems on top of itself, complete with a modified supervisor, new vocabulary, and altered language syntax.

REFERENCES

Bell, J. "Transformations: The Extension Facility of Proteus," in Christensen, C. and C. J. Shaw (editors), Proceedings of the Sigplan Extensible Languages Symposium, May 1969, pp. 27-31.

Bennet, R. K. "The Design of Computer Languages and Software Systems: A Basic Approach," Computers and Automation, Vol. 18, No. 2 (1969), pp. 28-33.

Christensen, C., and J. S. Christopher (Eds.). Proceedings of the Extensible Languages Symposium, sponsored by the Special Interest Group on Programming Languages (SIGPLAN), Association of Computing Machinery, Boston Massachusetts, 1969.

Dahl, O. J. and K. Nygaard. "SIMULA - An ALGOL-Based Simulation Language," Communications of the ACM, Vol. 9, No. 9 (1966), pp. 671-678.

Gauthier, R. L. "PL/1 Compile Time Facilities," Datamation," Vol. 14, No. 12 (1968), pp. 32-34.

Hawkinson, L., S. L. Kameny, C. Weissman, et al. LISP 2. A series of documents produced in performance of contract AF19(628)-5166 with the Electronics Systems Division, Air Force Systems Command, 1966.

Hormann, A. M. "Programs for Machine Learning, Part I," Information and Control, Vol. 5, No. 4 (1962), pp. 347-367.

Hormann, A. M. "Programs for Machine Learning, Part II," Information and Control, Vol. 7, No. 1 (1964), pp. 55-77.

Hormann, A. M. "How a Computer System Can Learn," IEEE Spectrum, Vol. 1, No. 7 (1964), pp. 110-119.

Hormann, A. M. "Gaku: An Artificial Student," Behavioral Science, Vol. 10, No. 1 (1965), pp. 88-107.

Hormann, A. M. Designing a Machine Partner--Prospects and Problems, SDC document TM-2311/003/01, 1965.

Hormann, A. M. A New Task Environment for Gaku Teamed with a Man, SDC document TM-2311/003/00, 1966.

Hormann, A. M. Problem Solving and Learning by Man-Machine Teams (Summary of Current and Projected Work), SDC document SP(L)-3336/000/02.

Hormann, A. M. Application Problems of Man-Machine Techniques, SDC document TM(L)-4452, 1969.*

Hormann, A. M. Planning by Man-Machine Synergism: A Characterization of Processes and Environment, SDC document SP-3484, 1970.

Irons, E. T. "The Extension Facilities of IMP," in Christensen, C. and C. J. Shaw (editors), Proceedings of the Sigplan Extensible Languages Symposium, May 1969, pp. 18-19.

Iverson, K. E. A Programming Language, New York: John Wiley and Sons, 1962.

Kleine, H., and R. L. Citrenbaum. An On-Line Interactive Hierarchical Organization and Management System for Planning, SDC document SP-3482, 1970.

Knowlton, K. C. "A Programmers Description of L^6," Communications of the ACM, Vol. 9, No. 8 (1966), pp. 616-625.

Knuth, D. E., and J. L. McNeley. "SOL - A Symbolic Language for General Purpose Systems Simulation," IEEE Transactions on Electronic Computers, Vol. EC-13, No. 4 (1964), pp. 401-408.

Linde, R. R., C. Weissman, and C. E. Fox. The ADEPT-50 time-sharing system, System Development Corporation, Santa Monica, California, 1969.

Landin, P. J. "The Mechanical Evaluation of Expressions," The Computer Journal, Vol. 6, No. 2 (1963), pp. 134-143.

Markowitz, H., B. Hausner, and H. Karr. SIMSCRIPT, A Simulation Programming Language, Englewood Cliffs, New Jersey: Prentice-Hall Inc., 1963.

Marschak, Jacob. "Economics of Language," Behavioral Science, Vol. 10, No. 2 (1965), pp. 135-140.

Mooers, C. N. "TRAC A Procedure-Describing Language for the Reactive Typewriter," Communications of the ACM, Vol. 9, No. 3 (1966), pp. 215-219.

Shaw, J. C. "JOSS: A Designer's View of an Experimental On-Line Computing System," AFIPS Conference Proceedings, Vol. 26, Fall Joint Computer Conference (1965), pp. 455-464.

Smith, D. C. MLISP, Stanford Artificial Intelligence Project, MEMO AIM-135, Report Number CS-179, Computer Science Department, Stanford University, 1970.

---

* This SDC document is not available for distribution outside the corporation

Strachey, C., et al. "The Main Features of CPL," The Computer Journal, Vol. 6, No. 4 (1964), pp. 308-320.

Thompson, F. B., et al. "REL: A Rapidly Extensible Language System," Proceedings of the 1969 ACM Conference.

Van Wijngaarden, A., B. J. Mailloux, J. E. L. Peck and C. H. A. Koster. "Report on the Algorithmic Language Algol 68," The Mathematical Centre, Vol. 49, No. 2e: Boerhaave-straat, Amsterdam, MR101, Jan. 1968.

Weissman, C. LISP 1.5 Primer, Belmont, California: Dickenson Publishing Company, 1967.

APPENDIX A

A UAL TERMINAL SESSION

The following is taken from a demonstration of UAL on the IBM 360/67 ADEPT Time Sharing System using an Execuport 300 terminal. (See Appendix B for minor character conversions.) The session covers expression formation, character string manipulation, assignment, list manipulation, operationals, evaluation and inhibiting, extensibility and protection, function definition and argument maps, conditionals, iteration, argument forcing, input/output, state conditions, the control character #, and function defining functions. User inputs occur after the preset lineprompt of #. Computer replys are indented except in the case of a non-formatted print.

```
# 21+45                                    expression formation
     66

# $+2                                      $ contains value of last expression
     68

# 5↑3
     125

# 5.8-3.74
     2.06

# 50/2*5
     5.0

# 4<7                                      true=1 false=0
     1

# 5>5
     0

# 5>=5
     1


# "HELLO"                                  character strings
     HELLO

# CONCAT "HELLO" " THERE."
     HELLO THERE.
```

```
# ""
      EMPTY STRING

# "A""A"                          getting " character into a string
     A"A

# "5"+1                           automatic conversion
     6

# "5 FEET"+1
     6


# A←7                             assignment
     7

# A
     7

# B←7
     7

# A=B
     1

# A==B
     0

# B←A
     7

# A==B
     1

# A←←8
     8

# B
     8
```

lists

```
# {1 2 3}
    LIST
         1
         2
         3
    END

# 2 ND {2 4 6}
    4

# {1 2 {4 5} 6}
    LIST
         1
         2
         LIST
              4
              5
         END
         6
    END

# 2 ND 3 RD $
    5

4 TH {7 8}
    NO VALUE

# {}
    EMPTY LIST

# 3 ON {1 2 3 4}
    LIST
         3
         4
    END

# APPEND {1} {5 6}
    LIST
         1
         5
         6
    END
```

```
# {2 4 ...  10}
     LIST
           2
           4
           6
           8
          10
     END

# L←{1 3 5 7}
     LIST
          1
          3
          5
          7
     END

# E←3 RD L
     5

# E←←6
     6

# L
     LIST
          1
          3
          6
          7
     END

# (3 RDN L)←←10
     10

# L
     LIST
          1
          3
          10
          7
     END

# E
     10
```

```
# K←(PRINT "A" PRINT "B" PRINT "C")
     A
     B
     C
     A

# K
     A

# K←(PRINT "A" PRINT "B" SETOP "C" PRINT "D" RET PRINT "E")
     A
     B
     D
     C

# K
     C
```

operationals

last A is value of entire expression

```
# D1←5
     5

# D2←'D1'
      EXPRESSION

# D2
     5

# D3←:D2
      EXPRESSION

# D3
     5

# D4←::D3
      EXPRESSION

# D4
        D3

# EVAL D4
     5
```

evaluation and inhibiting

The word EXPRESSION is substituted for the printout of the value when it would be a simple repeat of the input.

extensibility and protection

```
# P←VALUE +
     UAL PRIMITIVE

# 3 P 4
     7

# +←17
     ← ASSIGNMENT ATTEMPTED ON A PROTECTED VARIABLE.
     17

# UNPROTECT +
     NO VALUE

# +←17
     17

# +
     17

# + P 2
     19

# +←VALUE P
     UAL PRIMITIVE

# 1+1
     2

# PROTECT +
     NO VALUE
```

functions

```
# COMBINE←[X (AND) Y]:X↑2+Y↑2
     EXPRESSION

# COMBINE 3 4
     25

# COMBINE 3 AND 5
     34

# COMBINED←[X @ (WITH) Y]:X+Y+10
     EXPRESSION
```

```
# 2  COMBINED WITH 3
     15

# COMBINATION←[X Y @]:X*Y*2
    EXPRESSION

# 4 5 COMBINATION
    40
```

indirect assignment

```
# N←4
    4

# M←"N"
    N

# (M)←44
    44

# M
    N

# N
    44
```

argument map saving

```
# EXPR←::X*Y*Z
    EXPRESSION

# ARGM←[: X Y Z]
    ARGUMENT MAP

# MULT3←ARGM EXPR
       [X Y Z]:X*Y*Z

# MULT3 2 3 4
    24
```

conditionals

```
# IF 2<4 THEN "OK"
    OK

# IF 4<2 THEN "OK"
    NO VALUE
```

```
# IF 2<4 THEN (N←45 PRINT "DONE")
     DONE
     45

# N
     45

# IFE 2<4 THEN "OK" ELSE "NO GOOD"
     OK

# IFE 4<2 "OK" "NO GOOD"
     NO GOOD
```

forall

```
# FORALL X IN {2 4 6 8}  X↑2
     LIST
          4
          16
          36
          64
     END

# FORALL X IN {5 12 7 20} WHENEVER X>10   2↑X
     LIST
          4096
          1048576
     END
```

argument forcing

```
# G←[X Y;Z]:X*Y+Z
     EXPRESSION

# G 2 4
     8

# G;3 2 4 1
     9
```

input/output

```
# PRINTOFF

# 2+2
```

```
# PRINT $
     4

# SPECIFY S
     PLEASE SPECIFY S.

# 145

# PRINT S
     145

# OUTPUT S
145

# STASH "       S="

# STASH S

# OUTPUT "."
       S=145.

# OUTPUT {1 2 3 7}
1237

# PRINTON
     NO VALUE

# D←READ

# 4                                   This input is under the control of READ
     4

# D
     4

# READ

# 2+2                                 This input is under the control of READ
     4

# READQ

# 2+2
     2+2

# EVAL $
     4
```

state conditions

```
# M←10
     10

# |M<12|
     1

# M←5
     5

# M←15
     ATTEMPTED VIOLATION OF STATE CONDITION 1.
     15

# M
     5

# STATES
     CURRENT STATE CONDITIONS:
     (1) M<12
     DONE
     NO VALUE

# N←13
     13

# |N>M|
     1

# N←7
     7

# N←1
     ATTEMPTED VIOLATION OF STATE CONDITION 2.
     1

# N
     7

# STATES
     CURRENT STATE CONDITIONS:
     (1) M<12
     (2) N>M
     DONE
     NO VALUE

# DELSTATE 1
     NO VALUE
```

```
# STATES
     CURRENT STATE CONDITIONS:
     (1) CANCELLED
     (2) N>M
     DONE
     NO VALUE

# N←24
     24

# M←15
     15


# 1+2+3+4#D5+6                          control character #
     11

# 1+2+3+4#C

# +5
     15

# #C"ABCDE

# FGH"
     ABCDEFGH

# "ABCDE#3FGH"
     ABFGH

# #W"ABCDE

# FGH

# IJK

# LMN"#E
     ABCDEFGHIJKLMN

# "AA#!AA"
     AA#AA

# "AA#[YES THIS IS A COMMENT]#AA"
     AAAA
```

```
# N#[IN A NAME]#N←5#[IN A NUMBER]#5
     55

# NN
     55

# MN←15 #("N","M")
     15

# MN
     NO VALUE

# MM
     15

# M←15*(12+8) #P
  M←15*(12+8)
# #("1","2")
    ATTEMPTED VIOLATION OF STATE CONDITION 2.
    500

# "ABCD##24EFG"                          "24" is the hexadecimal code for
     ABCD                                line feed.
         EFG

# "1234##OF567"                          "OF" is the code for carriage return.
     1234
567


# LINEPROMPT "##OF> "                    miscellaneous functions
     NO VALUE


> 1+1
     2

> LINEPROMPT "##24#! "
     NO VALUE

# SQRT 2
     1.414213562373094
```

```
# EVALST "2+2"
     4

# FREESPACE
     35201

# (2*5 8<4 D←{1} 2000 STACKSTATUS RETR 0)
     LIST
          2000
          LIST
               1
          END
          0
          10
     END
     0


# R1←[X]: R2←[Y]: Y↑X                    function defining functions
     EXPRESSION

# R2 5
     NO VALUE

# R1 3
     [Y]:Y↑3

# R2 5
     125

# R1 4
     [Y]:Y↑4

# R2 5
     625
```

APPENDIX B

PRECEDENCE

The before-and-after precedence for predefined functions is as follows:

BEFORE		AFTER
0	←	14
0	← ←	14
1	ST	2
3	%	--
3	↑	4
5	*	5
6	/	6
7	+	7
7	-	7
8	<	8
8	≤	8
8	=	8
8	≠	8
8	==	8
8	≥	8
8	>	8
--	~	9
10	&	10
11	v	11
12	→	12
13	≡	13
13	≠	13
14	,	14
15	functions	16
--	:	16

The lower-numbered functions are combined first, that is, have the highest precedence. In cases of equal precedence, combination done from left to right.

APPENDIX C

CHARACTER CONVERSIONS

	UAL	TELETYPE	EXECUPORT 300
Opening Grouping Parenthesis	(	(	(
Close Grouping Parenthesis	)	)	)
Open Operational	(	(	(
Close Operational	)	)	)
Open List	{	<<	{
Close List	}	>>	}
Open Character String	"	"	"
Close Character String	"	"	"
Open Argument Map	[	[	[
Close Argument Map	]	]	]
Open Quoted Expression	`	'	'
Close Quoted Expression	´	'	'
Open State Condition	|	\	|
Close State Condition	|	\	|
Open Quoted Argument*	`	'	'
Close Quoted Argument*	´	'	'
Open String Argument*	"	"	"
Close String Argument*	"	"	"
Blank Space	␣	␣	␣
Line Prompt	#	#	#
Logical Implication	→	none	none
Logical And	&	&	&
Logical Or	V	V	V
Logical Negation	~	none	~
Assignment (Definition)	←	←	←
Assignment (Change)	←←	←←	←←
Exponentiation	↑	↑	↑

APPENDIX C (Cont'd)

	UAL	TELETYPE	EXECUPORT 300
Integer Division	÷	//	//
Division	/	/	/
Multiplication	*	*	*
Subtraction	-	-	-
Addition	+	+	+
Function Name Positioner*	∧	@	@
Less Than	<	<	<
Less Than or Equal to	≤	<=	<=
Equal to	=	=	=
Not Equal to	≠	/=	~=
Greater Than or Equal to	≥	>=	>=
Greater Than	>	>	>
A Member of	ε	IN	IN
List Formation	,	,	,
Argument Forcing	;	;	;
Open Local Variable List*	;	;	;
Decimal Point	.	.	.
Expression Quote	:	:	:
Argument Map Quote*	:	:	:
Identically Equal to	==	==	==
List Continuation	...	...	...
Line Entry	*cr*	*cr*	*cr*
Character Delete	*#dd*	*#dd* or *rubout*	*#dd* or *bs*
Line Delete	*#D*	*#D* or *break*	*#D* or *break*
Global Variable	!	!	!
Percent	%	%	%
Value of Previous Expression	$	$	$
Unused	?	?	?

*Found only in the argument map

cr = carriage return

bs = back space

β = blank space

APPENDIX D

UAL SYNTAX

<digit>::=Ø|1|2|3|4|5|6|7|8|9
<letter>::=A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<integer>::=<digit>|<digit><integer>
<real>::=<integer>.|.<integer>|<integer>.<integer>
<number>::=<integer>|<real>
<variable>::=<name>|<symbolic name>
<symbolic name>::=+|-|*|/|÷|↑|%|<|≤|=|≠|==|≥|>|~|∨|
&|→|≡|≠|←|↔|ε|,|@|...|!
<name>::=<letter><alpha-num sequence>
<alpha-num sequence>::=<alpha-num>|<alpha-num><alpha-num sequence>|<empty>
<alpha-num>::=<letter>|<digit>
<character string>::="<string>"
<string>::=<character>|<character><string>|<empty>
<character>::=<letter>|<digit>|<symbolic name>|(|)|[|]|""|{|}
`|´|;|:|.|||∧|⊬
<empty>::=

<expression>::=<function name><argument list>|<list>|<operational>|
<character string>|<variable>|<number>|
<function definition>|<state condition>|<empty>
<state condition>::=|<expression>|
<list>::={<expression sequence>}
<expression sequence>::=<expression>|<expression><expression sequence>
<operational>::=(<expression sequence>)
<function name>::=<variable>|<qualified variable>
<qualified variable>::=<variable>;<integer>
<argument list>::=<expression sequence>
<function definition>::=<argument map><quoted expression>
<quoted expression>::=`<expression>´|:<expression>|<expression>
<argument map>::=[<bound variable list><local variable specification>]|<empty>
<bound variable list>::=<bound variable>|<bound variable><bound variable
list>|<empty>
<bound variable>::=<variable>|(<variable>)|`<variable>´|"<variable>"|
<variable>!|:|∧|<integer>
<local variable specification>::=;<local variable list>|<empty>
<local variable list>::=<local variable>|<local variable><local variable list>
<local variable>::=<variable>|<variable>!

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)	2a. REPORT SECURITY CLASSIFICATION
System Development Corporation Santa Monica, California	Unclassified
	2b. GROUP

3. REPORT TITLE

User Adaptive Language (UAL): A Step Toward Man-Machine Synergism

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

Scientific

5. AUTHOR(S) (First name, middle initial, last name)

Aiko Hormann, Antonio Leal, David Crandell

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
June 28, 1971	60	31

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
DAHC15-67-C-0149 b. PROJECT NO. ARPA Order No. 1327, Amendment 3, c. Program Code ID30 and IP10 d.	TM-4539/000/01
	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report) None

10. DISTRIBUTION STATEMENT

Approved for public release; distribution unlimited

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY

13 ABSTRACT

The User Adaptive Language (UAL) is designed to provide a convenient and flexible means for man-machine communication in cooperative problem-solving/decision-making efforts. The language is extensible and functionally oriented with user control of evaluation and data manipulation. The interactive nature of UAL provides a "conversational" environment conducive to dynamic decision making.

The syntax of UAL is simple and straightforward. The function definition features provide a means of creating new terms and primitives allowing a higher level of sophistication in the communication of ideas. The new functions may be compact and stylized or English-like in their use. Consequently, the problem solver is free to concentrate on what to say rather than how to say it. UAL can be a powerful tool in building systems in which man and machine can work together to complement each other's capabilities.

DD FORM 1 NOV 65 1473

UNCLASSIFIED

Security Classification

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT


SDC