LA-UR -73-890

TITLE: Algorithmic Languages and Machine-Oriented Tasks

AUTHOR(S): Mark B. Wells

SUBMITTED TO: IFIP/TC-2 Working Conference on Machine-Oriented Higher Level Languages, Trondheim, Norway

los alamos
scientific laboratory
of the University of California
LOS ALAMOS, NEW MEXICO 87544

Form No. 836
St. No. 2629
1/73

UNITED STATES
ATOMIC ENERGY COMMISSION
CONTRACT W-7405-ENG. 36

Algorithmic Languages and Machine-Oriented Tasks*

by

Mark B. Wells

This paper presents the point of view that very high level languages can and should be applied for the specification and programming of machine-oriented tasks. The essential concept is that the language and a compiler for the language must not be confused. Features that such a language should have are discussed.

*This work supported in part by the United States Atomic Energy Commission and in part by the National Science Foundation.

1. Introduction

The thesis of this paper is that the use of a very high level, general purpose, machine-independent programming language for machine-oriented tasks is both feasible and desirable. (These languages are called here *algorithmic languages*. No attempt is made to define precisely what an algorithmic language is, but the ensueing discussion, especially Section 5, does indicate some properties that the author feels such languages should possess.) The contention is that significant short term as well as long term gains can result from more effort being spent on both the development and the use of very high level languages for systems programming. Jean Sammet is probably right when she says [8] "... it is futile to try to develop a 'universal programming language' for all applications." Nevertheless it is probably also true that one of the better ways to uncover general underlying principles of programming and programming language design is to attempt such development. Furthermore, this author believes that there is much less that is peculiar to systems programming than many people believe.

To begin to appreciate the point of view of this paper one must first brush aside the myth that "high level" necessarily implies "massiveness". The most general language, in the sense of being able to express precisely the details of execution of any algorithm, is absolute machine code. As we move progressively through assembly languages Fortran, Algol, etc. towards more and more machine-independent higher level languages (i.e. toward problem oriented languages) we yield more and more of the details of implementation to the compiler. It is only when we try simultaneously to incorporate high level constructs and maintain control over implementation details that complexity, massiveness, and baroqueness of the language enter the picture. A well designed algorithmic language, while being general purpose and having considerable expressive power, may greatly restrict the programmer relative to many precise details of algorithmic execution in order for itself to remain simple and elegant. The basic question is how much control are we willing to abdicate in order to reap the significant benefits of the

use of a high level language. In this author's opinion, simplicity, or machine-independence, or some other important characteristic of an algorithmic language has all too often been sacrificed merely to simplify compiler writing or to allow the user control over execution detail, sometimes unnecessarily.

The basic conclusion is that there is a great deal to be gained by studying the application of high level languages--PL/1, Algol 68, Madcap--to machine-oriented tasks. We might do better by investigating the proper design of compilers for algorithmic languages rather than by designing new machine-oriented languages.

Apart from this introduction there are four sections to this paper. Section 2 gives a general view of the role of algorithmic languages. It briefly discusses advantages and disadvantages to their use, the importance of the separation of the concepts of a language and its compiler, and, what might be considered as a fundamental function of algorithmic languages today, their use in the design of large systems. Section 3 discusses inter-algorithm communication and suggests that storage allocation is not a concern at the level of algorithm generation. Section 4 is concerned with data structures and their implementation as storage structures. It is pointed out, for instance, how the way in which a data structure is used can help the compiler choose a proper internal representation for it. In the concluding Section 5, the author summarizes the features of algorithmic languages which he feels are necessary and desirable for machine-oriented tasks as well as for more general application.

The ideas presented in this paper are primarily based on the author's experience with the Madcap 5 and Madcap 6 languages implemented on the Maniac II computer at Los Alamos. Madcap 5 [9,12] has been in continuous use as a general purpose programming language since 1964. Its chief relevant features are the availability of a specific set-theoretic data structure--sets of natural numbers implemented as bit patterns--and the capability of embedding assembler language in a source program. It has been used to write a precursive time sharing operating system for the

Maniac and a compiler for Madcap 6. Madcap 6 is an interactive language and system currently available on the Maniac; work is underway to write compilers for it on the IBM 360/91 at UCLA (University of California at Los Angeles), the R-2 computer at Rice University in Houston, Texas, and the CDC 7600 at Los Alamos. Its chief relevant features are powerful control structures [6], a unified data structuring capability including arbitrary (unordered) sets [13], recursion, and an effective block structure-filing system scheme. No assembler language embedding is allowed. It has been used as a specification language for major parts of its Maniac compiler and is currently being used as a design tool for a new Maniac time sharing operating system.

The author makes no claim to originality or profoundness of the ideas put forth in this paper; however, perhaps the point of view expressed here by these ideas is more radical (idealistic?) than has been expounded elsewhere.

2. The role of algorithmic languages

An excellent discussion of the advantages and disadvantages of the use of high level languages for systems programming is given by Sammet [8]. In summary, the advantages are

(a) the efficiency of program preparation;

(b) the ease of program documentation, maintenance, and understanding; and

(c) the potential for program portability.

The chief disadvantage is

(d) the possible time and space inefficiency of object code.

The advantages stem of course from the expressive power, naturalness, and machine independence of the language, while the disadvantage occurs because of the increased difficulty of producing an adequate compiler for the language. As a language becomes more and more algorithmic its advantages become more and more pronounced, yet so apparently does the difficulty of compiler construction. To this author, this does not suggest that we lower our sights on high level language design. On the contrary, this suggests

(1) that we attack the basic problems of compiler and hardware design and (2) that we restrict ourselves to algorithmic language subsets that are useful for systems programming and for which effective compilers can be written.

Now, it is very important from the outset to appreciate the difference between a language and one of its compilers. To achieve true machine independence in a language, the concept of using the language to express algorithms must be kept quite separate from that of executing an algorithm written in that language in the environment of a particular computer system. (Further, properties of programs written in a language must not be confused with properties of the language itself. A machine independent language can of course be used to write a highly machine dependent piece of software.) A language is concerned with the specification of algorithms, a compiler with its computer is concerned with the efficient execution of these algorithms. Of course, during language design there is always give and take between incorporation in the language of various facilities and development of a practical compiler. In fact, it often happens that particular language features are suggested or discarded on the basis of the ease or difficulty of their implementation. Furthermore, in practice, choosing a particular compiler oftentimes effectively selects a language subset. Nevertheless it is still important to keep the concepts independent. By so doing one obtains a clearer view of the problems and how they can be attacked. For instance, the problem of packing data for efficient memory utilization is chiefly a compiler and hardware problem and is of no concern at the algorithm level since at that level referencing of the data depends only on its inherent ordering (or its values) and not on its precise representation. An appreciation of this greatly enhances a programmer's ability to organize his programs in a systematic top-down manner (e.g. see [15]).

A language itself may be machine independent but of course its compiler and the system in which it is embedded is not. Thus, while algorithms written in the language are portable between systems, one system

or another may be more effective for a certain class of algorithms. In actuality, any particular system not only effectively defines a subset of the language, but also provides new capabilities such as a filing system, editing facilities, and debugging tools which greatly facilitate the preparation and execution of algorithms.

The aspect of a system which most closely interacts with language and compiler design and use is perhaps the filing system (with which we include libraries of debugged subroutines). To be truly effective for machine-oriented tasks (or any other particular area for that matter), an algorithmic language requires access to a library of appropriate existing algorithms. Perhaps, as in Algol, this is a block global to every running program. Two generalizations of this Algol concept of an "environment" are worth considering. One is to allow more structure in the environment, i.e. more levels of nesting external to a running program. This allows a more precise taxonomy of library programs. The second is to allow data other than executable procedures (constants of data type "expression") into this environment. The concept of having real numbers, Boolean quantities, sequences, sets, and other language constants accessible from a running program but not specifically declared within it has considerable merit. With this concept the filing system is much like the activation record of a partially executed supra-program. An application of this idea to a machine-oriented task is given in Section 3.

Now, one of the points being made in this paper is that a good algorithmic language should be useful at all stages of design of a large system. This of course implies that the language be very high level (recall the earlier remarks that high level need not imply massiveness). Realistically, it may also imply that the language is not a programming language at all in the sense that algorithms written in it can be directly compiled for execution on a computer; that is, it may be necessary to have a "human compiler" perform a translation. Nevertheless, the clarity, uniformity, and discipline allowed and imposed by an algorithmic language make its use invaluable at all levels of design. The existence of a compiler is an extremely worthwhile convenience but is certainly not a prerequisite.

Thus, perhaps the chief role of algorithmic languages today is in the specification of large systems. Properly designed itself, an algorithmic language can serve as an important quide to well structured design and programming of systems. Furthermore, once compiler and machine design do catch up, we will have made a truly giant step towards solving the "software crisis". Even until they do though, the translation from algorithmic language to programming language by humans is probably less burdensome than programming as it exists today. It is only during this part of the development process, the translation phase, that detailed efficiency considerations come into play. In addition and more importantly, performing this translation gives us insight into how compilers and machines should be designed themselves. A very real danger in concentrating on the development of machine dependent languages with primary emphasis on efficiency of the object code is that we learn very little about the "proper" way to design a computer. It seems to this author that proper hardware and compiler design must follow an understanding of computational processes and their expression as algorithms. The development of the Burroughs 5000/6000 series of computers [7] is one of the few commercial examples where this top-down philosophy has been pursued in practice.

3. Processes and their intercommunication

A large program (the words "program", "algorithm", "process", "procedure", "subroutine", etc. are essentially interchangeable in this discussion) is subdivided into smaller programs themselves subdivided, and so forth, until manageable units appear. There are many questions concerning these processes, their calling and sequencing, their intercommunication, storage allocation of their variables, etc. of relevance to system programming as well as general purpose programming. It is not the aim here to give a thorough account of these questions, only to point out how various machine-oriented tasks can fit under the general algorithmic language framework being advocated.

Consider first the communication question. Algorithmic languages essentially allow two types of communication between procedures: an actual

dynamic passing of parameters and a static referencing of global or external variable values. Except possibly for the extent and sufficiency of its use, there is really very little controversy surrounding parameter passing as a legitimate means of interalgorithm communication. Most language designers agree that by one means or another passing arguments by value, by reference (pointer), and by name are all important. On the other hand, there is more active debate concerning static communication [16], or as it might better be called, *environmental communication*.

To illustrate an application of environmental communication to systems programming, consider the buddy storage allocation scheme given in the Appendix. (This algorithm is written in the Madcap 6 language. A manual and/or formal definition of this language does not yet exist. However, since the object here is to discuss concepts and not notation it is felt that very little will be lost by not describing Madcap, or for that matter the buddy program itself, in detail.) The routines that do the work are get and free, which respectively find an available block of storage or return a block to the free storage pool. These routines exist and are executed in an environment which presumably is part of an operating system. That part of the environment which is referenced from within get and free appears in Figure 1. It contains definitions of the special data types and operations used in the scheme (see later discussion), the assignment of appropriate expression values to the identifiers get and free themselves, the initialization of the basic data structure Ablocks which is to be a sequence with $h^{\underline{th}}$ component the set of available blocks of length 2^h whose buddy is unavailable, and finally the calls to the routines themselves.

The ability to establish such an environment for executing algorithms is extremely useful for systems work. Such an environment is very much like the outer system library block of an Algol 60 program, although values other than procedures are accessible here. More involved examples show the need for a hierarchy of environments. This seems to suggest a block structured-like filing system wherein the outer blocks are essentially activation records for load time evaluated programs. Such a filing system is in use with Madcap.

It is possible for various machine resources to be made available in this environment. Identifiers such as keyboard and time, when not declared locally, can reference a terminal or the machine real time clock. Even hardware (index) registers can be made available in this manner by utilizing a friendly filing system and compiler. Of course in a sense this makes the language machine dependent, but only as used in a particular necessarily machine dependent environment. For instance, it never makes sense to write a program to ask what time it is unless facilities for answering that question actually exist within the particular system in which the program is to be executed.

It is natural to ask the question about storing into, i.e. changing, the environment. As the reader is certainly well aware, allowing programs arbitrarily to change the environment of other programs only invites disaster. The question is part of the general question of side effects. This author now feels that whereas using values of global variables is harmless and naturally beneficial, assigning of values to global variables can be dangerous and must be controlled. Strongly impinging on this issue are the questions of pointers, call-by-reference, etc. In Madcap, primitive parameters--reals, Booleans, etc.--are always passed by value, while structures--sets and sequences--are passed by codeword (there is no "pointer data type") [13]. Furthermore, global stores are not allowed. (This involves compile time "global attribute" checking which is quite similar to data type checking--see Section 4.) Thus it is impossible for a nested procedure to change the environment unless it is specifically allowed to do so. Note, for example, in the buddy program that Ablocks is passed as a parameter to get and free so that they may modify the component sets within that structure. This is just one of the many possible solutions of course, but it does illustrate the author's philosophy that system's programmers should have no more freedom to abuse good programming practice than anyone else. The algorithmic language should encourage good habits and disallow bad habits and the compiler should be smart enough so as to pacify the programmer for his resulting loss of control.

A closely related subject is the matter of storage allocation. To this author, storage allocation matters ideally belong solely in the province of the compiler. Unfortunately, practical considerations of today seem to require compromise. For instance, the need for independent procedure compilation seems to call for some form of compiler assisting language such as a declaration of the use of a procedure as recursive or reentrant. This information, which would not be difficult to gather by the compiler had the compiler knowledge of the total program, is of course useful in determining whether storage must be allocated dynamically at run time (the recursive case) or could be allocated statically at compile time (the non-recursive case).

In some cases, the systems programmer is called on to utilize his intimate knowledge of the system and of the computational model of the language in order to regain some measure of control over storage allocation. In Madcap for instance, it is possible to allocate a large sequence, call it M, whose components are of arbitrary type:

$$M \leftarrow \langle 0 \text{ @ UNIVERSE} : 1000 \text{ items} \rangle.$$

This sequence can then be used to store arbitrary values, even other sequence values, provided the user is aware of the implementation scheme for those values. (A cell based computational model [4] is used for Madcap. A cell containing a "value"--this is a pointer for a sequence--occupies at most one word of the target machine.) For instance, the assignments

$$M_{10 \text{ to } 15} \leftarrow \langle 2.1, 7.6, 3.8, 1.9 \rangle$$

$$M_9 \leftarrow M_{10 \text{ to } 15}$$

will embed the given sequence of reals within the M sequence and store its codeword in M_9. Of course to make use of this technique the programmer must know that a sequence has two words of header information attached to it and also generally must know the exact form and type of this information (see later discussion on packing of storage structures).

The symbolism and technique of this last example represent even more of an undesirable concession to practicality than that of the compiler assisting declaration of a recursive procedure. In both cases, however, the inelegance arises because we are trying to put into the language capabilities which properly belong in the compiler. Surely computer science will eventually develop general and efficient storage allocation schemes that on the average will be better than what the typical systems programmer can devise for himself.

4. Data structures and their implementation as storage structures

Part of the distinction between a language and its compilers is the distinction between "data structures"--complex numbers, graphs, queues, etc.--at the algorithmic level and "storage structures"--pointers and memory blocks, bit patterns, floating and fixed point numbers (not necessarily respectively), etc.--at the hardware representation level. The programmer is concerned with and chooses the data structures according to the requirements of the algorithm, while the compiler selects storage structures to represent those data structures which are consistent with the capabilities of the target computer. (In practice, the storage structures associated with many machine-oriented tasks are often predetermined by specific object code requirements such as an established control word format. In these cases the data structures are chosen so that the compiler will translate them into the existing form. This does imply that the compiler was especially designed for the machine-oriented tasks and that the systems programmer understands the translation algorithm, but does not affect our view of language-compiler separation.) In order properly to select these storage structures the compiler needs to know the domains of possible values for the variables of the program and the operations that are to be applied to those variables. This information is customarily contained in the source program declarations of data type. For instance, when we say a variable b is of type BOOLEAN, we are saying that the domain

of possible values for b is {true, false} and that the operations $\neg$ (not), $\wedge$ (and), and $\vee$ (or), at least, are defined upon values in that space.

The allowable form of declarations in many high level languages does not impart type information sufficiently precise for the compiler to utilize straightaway. For instance, merely knowing that a variable is integral does not immediately determine whether or not the compiler can have it stored in an index register. The variable may assume values too large for index registers of the particular target machine or the variable may be involved in multiplication or division operations for which it is inconveniently accessible from a register. Thus, effective compilers for these (insufficiently) high level languages must scan a source program for more precise type information. Certain scans are not difficult. In fact, in a compiler which does type checking, e.g. the compiler for the language of project Sue [3] or in Madcap, there is little additional overhead involved. For type checking, one must see if an operation is consistent with the types of the operands, while for the information gathering scan one records the fact, for possible later analysis, that the variables appearing as operands have had the given operation applied to them.

Scanning to discover the domain of a variable is more difficult. This author believes that the answer to this problem lies with extensible languages whereby precise algorithm-oriented data types (value spaces) can easily be defined where needed. Again, consider the buddy system storage allocation scheme. Three new data types are defined for the scheme. First is ADDRESS.SPACE which is the set of natural numbers less than 2^k, the size of the memory. This is a subdomain of the base language space REAL, the set of all rational approximations representable in one word of the target machine. Permissible real number operations are allowed on ADDRESS.SPACE elements although the composition of the restricted set that are actually applied (here only {+}) can be used to influence the compiler's decision on a representing storage type. The second is Ablocks.COMPONENT.SPACE which is the set of subsets of ADDRESS.SPACE. These are the possible component values for the (k+1)-tuple ℓ which exists inside the procedures get and

free. The tuple (i.e. sequence) ℓ is the basic data structure for the entire scheme. At any point in time during execution, ℓ_h (or Ablock_h) is the set of addresses of available blocks of length 2^h where buddy is unavailable. It is important to note that in this set-theoretic formulation, the storage structure of ℓ_h is not specified. It may be a linked list, as implied by Knuth's scheme [5], or possibly a bit pattern, often an efficient representation for a set of natural numbers [12]. (In actuality, our specification of Ablocks.COMPONENT.SPACE is still not sufficiently precise; the allowable values of each component should be made to depend on h). The final new data type is INDICATOR. This is just the set {-1, +1} and presumably only the equality operation is applied to its elements. Thus, an intelligent compiler could determine that only one bit is required to implement variables of this type.

Two other data structure-storage structure distinctions pertinent to machine-oriented tasks are procedures versus macros and sequential structures implemented as packed strings of data. Consider the macro question. From the high level viewpoint there is only the concept of procedure--an algorithm which accepts certain input values and produces specified output values. Whether or not such a procedure is implemented as an "in line macro", a set of instructions inserted directly into the code at the point used, or as a "closed subroutine", a separate package called (jumped to) when needed, is really a matter of implementation and does not affect the correct functioning of the master algorithm. Thus, in a truly algorithmic language the macro concept does not exist; the choice of implementation schemes is the compiler's. This choice will depend on whether or not the procedure is used as a constant, the number of times it is used, its length, whether or not it contains only local references, etc. In the buddy example, the constant procedure defined in the extended language for calculating the buddy (c.f. $\ominus \oplus \ominus \equiv \ll \_ \gg$), most likely qualifies as a macro. By the way, the algorithm used here for this calculation is but one of many reasonable possibilities. A general bitwise exclusive-or of two addresses could even be expressed in set-theoretic terms, e.g.

```
⊖ ● ⊖ ■
« (x,y): address
S ←
{b : let(z←x,b←0) next(z←⌊z/2⌋,b←b+1)› z mod 2 = 1 until z = 0)
&          ‡This is the set-theoretic symmetric difference‡
{b : let(z←y,b←0) next(z←⌊z/2⌋,b←b+1)› z mod 2 = 1 until z = 0)
Σ_{m∈S} 2^m
»
```

so that the compiler (perhaps a human in this case) could know to take advantage of an exclusive-or machine instruction, if one exists. (Of course, examples such as this where the high level specification of an operation is many times more long-winded than the equivalent machine instruction have been used to argue against the use of high level languages. These arguments are unfair. The precise, possibly long-winded definition of every operation must exist somewhere. If its definition is well known enough to be taken for granted, then it probably belongs as a base operation in the high level language.)

Now consider the question of sequential structures. An example taken from the Madcap 6 compiler itself is the sequence

```
‹(c,t,s,ℓ): @ BOOLEAN; Operator: @ OPERATORS;
    Nodepointer: @ NODE.LIST; Operands: @ CODE.LIST›
```

which is the data structure for an operator of the intermediate language. This sequence has seven fields named c, t, s, ℓ, Operator, Nodepointer, and Operands. The precise meaning of these fields is of no concern here, but the way this sequence can be represented in the machine is. The data type environment for this structure is

```
BOOLEAN = {true, false}
OPERATORS = {i : 0 ≤ i < 256}
NODE.LIST = Ω‹(some sequence form)›
CODE.LIST = Ω‹(some sequence form)›
```

This information is sufficient for the compiler to deduce that the structure can be packed in one forty-eight bit Maniac word. Specifically, for instance, C, t, s, and & being Boolean quantities can each be represented in one bit; the Operator field has a universe of 256 natural numbers hence can occupy eight bits; and Nodepointer and Operands are each sequences whose codewords can be eighteen bit pointers. Of course, it is true that the choice of this data structure was guided by an intimate knowledge of the Maniac computer and its compiler for Madcap. However, the fact remains that the data structure is independent of both the particular hardware and particular compiler. The Madcap program containing this data structure can be executed under any system which has a faithful Madcap compiler. Once again it is the concept of language-compiler separation which makes this possible.

5. Conclusion

The primary conclusion of this paper is that an algorithmic language of "sufficiently high level" can be of good use in accomplishing (perhaps, for now, only specifying) machine-oriented tasks. Now, what features should such a language possess? (This question is also discussed in Bergeron, et al [1] at a more detailed and less idealistic level.)

First, a *data structuring mechanism* which allows tree-like or, more generally, graph-like structures is needed because this apparently is the natural form of the language-related data processed by compilers, editors, and the like. Facilities for either *copying* or merely *referencing* all or part of a structure also seem to be essential. (We have been very happy with our decision to keep pointers per se out of Madcap [13].) *Recursive procedures* are a very convenient tool for processing such graph-like data. An *environment concept*, perhaps as occurs inherently in block structured languages, helps a user sort out natural levels of parameterization and thereby assists the compiler in determining effective storage allocation. Some form of *extensibility* (of data types at least) is almost a necessity because in systems programming as elsewhere it is a virtual impossibility

to anticipate all of the data forms and the operations upon them that will be needed [1]. Incidently, in a *typed language* as is being advocated here, the handling of structures such as memories and stacks which can contain elements of various different types requires some form of type combination. Algol 68 allows unions of types; Madcap has a type called UNIVERSE which is equivalent to the set of values associated with all possible constants of the source language.

The most important control structures are *sequential execution*, *iteration*, and *if-then-else conditionals*. Less used, but important is the *case* construction (in Madcap this is an indexed sequence of expressions). Although this author has had little experience with them, it is quite likely that *decision tables* [10], in some form, would be very useful in certain contexts, say in building a machine simulator. Neither go-to's nor escapes are necessary nor desirable [6]. (This author now feels that no branch instructions of any kind are desirable in an algorithmic language. Even procedure returns or exits at places other than the end can too easily be misused hence should not be made available.) Some *coroutine* and/or *parallel processing* mechanism is needed, although it is this author's opinion that associated language constructs which are sufficiently natural and which satisfactorily hide implementation concerns do not yet exist.

Restrained use of *comments* and *descriptive identifiers* can greatly aid the production of self-documenting programs. However, too many comments or too long identifiers can require an excessive amount of page or line continuations and can destroy the important visible structure of a program or expression.

Finally, while perhaps not essential, the inclusion of *set-theoretic notions* in an algorithmic language seems to have considerable merit [2,11, 13,14]. This idea carries us still further in the direction of machine-independent language. No longer is a machine imposed ordering essential to the construction of our algorithms. For instance, at the algorithmic level, a hashed symbol table is often merely an unordered set of symbols, to which elements are adjoined or deleted, serving as a repository of items to be processed later in some arbitrary order.

In scanning this list, one quickly notices that while all the items do have application to machine oriented tasks none of them has application only to such tasks. They indeed are worthwhile characteristics of a general purpose language. In summary, this author feels that we must begin exclusively to associate algorithms with language, object code efficiency with compilers and computers, and machine-dependence with particular programs. Mixing them only produces "mud".

REFERENCES

[1] Bergeron, R. D., J. D. Gannon, F. W. Tompa, D. P. Shecter, and A. van Dam, "Systems programming languages", pp. 176-283 in *Advances in Computers*, Vol. 12, Academic Press, New York, 1972.

[2] Childs, D. L., "Description of a set-theoretic data structure", pp. 557-564 (Part 1) in *Proc. 1968 FJCC*, Vol. 33, Thompson Books, Washington, D.C.

[3] Clark, B. L. and J. J. Horning, "The systems language for project SUE", pp. 79-88 in Proc. of Symposium on Languages for Systems Implementation, *SIGPLAN Notices*, Vol. 3, No. 9 (October 1971).

[4] Johnston, J. B., "The contour model of block structured processes", pp. 55-82 in Proc. of Symposium on Data Structures in Programming Languages, *SIGPLAN Notices*, Vol. 6, No. 2 (February 1971).

[5] Knuth, D. E., *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, Addison-Wesley, 1968.

[6] Morris, J. B. Jr. and M. B. Wells, "The specification of program flow in Madcap 6", pp. 28-35 in Special Issue on Control Structures in Programming Languages, *SIGPLAN Notices*, Vol. 7, No. 11 (November, 1972).

[7] Organick, E. I. and J. G. Cleary, "A data structure model of the B6700 computer system", pp. 83-145 in Proc. of Symposium on Data Structures in Programming Languages, *SIGPLAN Notices* Vol. 6, No. 2 (February 1971).

[8] Sammet, J. E., "Brief survey of languages used in systems implementation", pp. 2-19 in Proc. of Symposium on Languages for Systems Implementation, *SIGPLAN Notices*, Vol. 3, No. 9 (October 1971).

[9] Sammet, J. E., "Madcap", pp. 271-281 in *Programming Languages: History and Fundamentals*, Prentice-Hall, Englewood Cliffs, N.J., 1969.

[10] Silberg, B. editor, Special Issue on Decision Tables, *SIGPLAN Notices*, Vol. 6, No. 8 (September 1971).

[11] Schwartz, J. T., "Set theory as a language for program specification and programming", Courant Institute of Mathematical Sciences, New York University, 1970.

[12] Wells, M. B., *Elements of Combinatorial Computing*, Pergamon, Oxford, 1971.

[13] Wells, M. B. and J. B. Jorris, "The unified data structure capability in Madcap 6", pp. 193-208 in *International J. of Computer and Information Sciences*, Vol. 1, No. 3 (September 1972).

[14] Wirth, N., "The programming language Pascal", pp. 35-63 in *Acta Informatica*, Vol. 1 (1971).

[15] Woodger, M., "On semantic levels in programming", TA3 pp. 79-83 in *Proc. IFIP Congress 1971*, North Holland, 1972.

[16] Wulf, W. and Mary Shaw, "Global variable considered harmful", pp. 28-34 in *SIGPLAN Notices*, Vol. 8, No. 3 (February 1973).

```
¢2^k is the memory size--k exists from a higher environment.¢
ADDRESS.SPACE ≡ {h: 0≤h<2^k}
Ablocks.COMPONENT.SPACE ≡ 2^ADDRESS.SPACE
INDICATOR ≡ {-1, +1}
⊖ ⊕ ⊖ ≡ « POWER ≡ {2^h: 0≤h≤k+1}; x: @ ADDRESS.SPACE; y: @ POWER;
          x + (if x mod 2y = 0: y else: 0) »

¢Assign expression values to get and free.¢
get ← « §0823 »
free ← « §0824 »

¢Initialize the Ablocks sequence.¢
Ablocks ← ⟨{} @ Ablocks.COMPONENT.SPACE: 0≤h≤k⟩
Ablocks_k ← {0 @ ADDRESS.SPACE}

¢Call get.¢
¢n indicates the size, 2^n, of a desired block.¢
get.output ← get(n, Ablocks, "OK")
i ← get.output_0
if i = -1: "handle error"
else: "i is the address of the available block"

¢Call free.¢
¢i is the address of the block to be freed.¢
free.output ← free(i, n, Ablocks)
if free.output_0 = -1: "handle error"
else: "the block has been freed"
```

Figure 1: Environment for get and free

```
get ← ≪
n: @ ADDRESS.SPACE; ℓ: @ ΩAblocks; Message: @ STRING
i ← -1
     if n < 0: Message ← "n<0"
     else: if n > k: Message ← "n>k"
     else:
          H ← @ ADDRESS.SPACE
          for n ≤ h ≤ k • ℓ_h ≠ {} until H exists:
               H ← h
          if ¬ H exists:
               Message ← "no area of size 2^n allocatable"
          else:
               i ← MIN ℓ_H; ℓ_H ← ℓ_H ~ {i}
               for n ≤ j < H: ℓ_j ← ℓ_j ∪ {i + 2^j}
⟨i, Message⟩
≫
```

Figure 2: The get procedure

```
free ← ≪
(i,n): @ ADDRESS.SPACE; ℓ: @ ΩAblocks; Out: Ω⟨@ INDICATOR, @ STRING⟩
Out ← ⟨+1, "OK"⟩
    if n < 0: Out ← ⟨-1, "n<0"⟩
    else: if n > k: Out ← ⟨-1, "n>k"⟩
    else:
        for n ≤ h ≤ k as for t ← false until t:
            if i ∈ ℓ_h:
                Out ← ⟨-1, "area already free"⟩; t ← true
            else:
                ℓ_h ← ℓ_h ∪ {i}
                if i mod 2^(h+1) ∉ {0, 2^h}:
                    Out ← ⟨-1, "i ≠ q · 2^n(some q)"⟩; t ← true
                else:
                    b ← i ⊕ 2^h          ¢b is the buddy of i¢
                    if b ∈ ℓ_h:
                        ℓ_h ← ℓ_h ~ {i, b}
                        i ← MIN(i, b)
                    else:
                        t ← true
OUT
≫
```

Figure 3: The free procedure