A Generalized Approach to Interpretation Machines

Ross W. Goodell
Codex Corporation
15 Riverdale Avenue
Newton, Massachusetts 02195

Introduction

There is a growing awareness of the importance of structure to programming[1,2,3,4]. The structuring of programs, however, is made difficult by the lack of structure in the mechanisms that the programs are written for. Structured programming by almost anyone's definition requires the formalization into discrete processing blocks which pass information primarily through explicit arguments[5,6]. Yet compiler facilities are almost always developed in an ad hoc manner which prevents the structuring of compile time functions; operation systems provide loosely related sets of execution time facilities which often require implicit information passing; and computer instruction sets provide an inflexible set of computational short cuts. External data storage and transmission structures are also very eclectic.

In this paper, the possibility of integrating such diverse components of a system into a single architecture will be explored. A generalized structure is developed and formalized to an extent. The relationships between syntax, semantics, and context are investigated. Techniques for machine and human representation of these structures are discussed. The implications of generalization on error checking are then touched on. And, finally, the economics of such a generalized approach are considered.

Information Syntax

In many higher level computer languages there is a functional semantic unit

$$f(x)$$

which is used to represent an action and to return a value. The returning of a value might be represented as

$$y = f(x).$$

This representation of information can be generalized. The first generalization, available in many higher level computer languagues, is to allow the argument to be a vector, as

$$y = f(x_1, x_2, \ldots).$$

Further, in some languages (e.g. AED [7] and LISP [8]), vectors or lists may be named and referred to in functional notation. Thus "f" might represent a vector. Let us allow the "function" to be explicitly represented as a vector also,

$$y = (f_1, f_2, \ldots)\ (x_1, x_2, \ldots).$$

The final generalization, allowing the value or return value to be a vector, is used in APL [9] and could be represented by

$$(y_1, y_2, \ldots) = (f_1, f_2, \ldots)\ (x_1, x_2, \ldots).$$

This provides a very general means of structuring the reduction of information. With these generalizations, a unit of information, say a "node", can be represented by a vector, such as

(r, s, t),

whether it is being viewed as input, output or function. Single element nodes, such as (f) can be expressed without parentheses. Also null nodes will be needed.

Such general purpose nodes afford some interesting capabilities. First, the argument may itself be a function. For example, in

(ℓ, m) (n, o, p) (q)

the function (q) would be evaluated first (its argument is null) to produce a return value which is the argument for the function (n, o, p) which would in turn produce the argument for the function (ℓ, m). In effect we have a right-to-left parse between nodes[1] and a left-to-right parse within nodes.

Second, the return value or output can later be considered as a function and further reduced. The execution of a structure may therefore occur over a series of execution times. For instance in the above example the evaluation of (n, o, p) (q) might produce the vector (r, s). The evaluation of (ℓ, m)(r, s) might produce (ℓ, m)(r, s) which would be evaluated in a later execution pass.

Third, the elements of a node, such as "n", "o" and "p" in (n, o, p) are themselves special cases of node representation, so it is reasonable to allow any form of node as an element, allowing information to be represented in tree structures.

Fourth, since a sequence of nodes can be reduced to a single node, it follows that such sequences, which will be called nests, should be allowed as elements of more complex nodes.

Finally, it is natural to represent a complex node with a single element node, i.e., a name. For example, if "f" were defined as (r, s)

then

(r, s) (x, y, z)

and

f (x, y, z)

would be equivalent and could be used interchangeably. The existence of such definitions determines the context in which this structure is reduced. This will be pursued further in the section on semantics.

[1]NOTE: A slight modification to this will be introduced later.

It should be noted that these nodes are amenable to generalized handling.[2] They can be saved, moved, or transformed without disturbing their internal structure. For instance, they may be queued or stacked. The representation

(ℓ, m) (n, o, p) (q)

used above, is a visual stack. Given the proper representation these nodes can be handled by humans, computers, magnetic storage, and transmission lines.

Information Semantics

The syntax outlined above includes no semantic information. The two aspects of information can therefore be completely separated to allow greater flexibility in reducing the information. The meaning of a single element node(name) depends on what definition (if any) has been made for that name. These definitions constitute the context for the information. With semantics separated from syntax, context can be limited to a set of explicit definitions such as an association list in LISP [8].

The context with the structure comprises the meaning or the information. The meaning of a structure may be represented as a 2-tuple

[nest, context]

where "nest" is a function and its argument, and "context" is an association list or tree. "Interpretation" can be defined as the process of reducing a structure within a context to produce specified actions and a value.

Physical actions (such as tape movement) which result from functions may have impact on the meaning of subsequent functions. Such side-effects will not be explored in this paper. There is another class of side effects which appears to be intrinsic to information handling. That is the modification of context, the creation of definitions and their scopes. We could therefore view an interpretation as also producing a 2-tuple

[value, new context]

Now when a nest is interpreted, its return value nodes would be a nest or structure which is determined by the structure and context of the parent nest. For instance, if function f reduced to value r in context A and function g of argument (m, n) reduced to value (s, t, u) in context A.

Then the interpretation of

f g (m, n)

in context A, would produce

r (s, t, u).

[2]This entire development is actually an extension of the "common referent notation" approach in AED [7].

This output could be final or could be reduced in another context. The return values produced in interpreting a nest could be generally viewed as another nest which may or may not be re-interpreted. The process of interpreting or reducing information could therefore be viewed as mapping a 2-tuple (nest and context) to another 2-tuple (nest and context) while producing certain un-mapped physical actions, symbolically

$$\text{interpretation } [\text{nest}_1, \text{context}_1] \rightarrow [\text{nest}_2, \text{context}_2] + \text{actions}$$

where, for the purposes of this paper the actions are assumed to have no effects on subsequent interpretations.

Interpretation Machines

A mechanism which produces such a mapping might be called an interpretation machine. Note that output from one interpretation machine may be input to another interpretation machine, ad infinitum. Also, the input medium for an interpretation machine may be different from its output. Thus a series of machines may greatly alter the physical nature of the information.

There are many forms of such "machines" in use. For instance, a human might read a series of symbols (nest) which he knew (context) represented a FORTRAN program and punch cards to produce a source deck (nest) and control cards to run the FORTRAN compiler (context). The cards would then be read by the operating system and compiler to produce a disk representation which would eventually be put in computer storage (nest) and executed by a given machine running under a given operating system (context)[3].

An example of a context being altered at execution time is any interactive system which provides commands to alter some characters and parameters, such as backspace and line length[4]. In these examples, context is handled in an ad hoc manner and meaning may even be embedded in the syntax being used. It is possible however, to separate semantics from syntax, and to organize context information just as any other data.

Machine Representation

Machine representations of nests and explicit contexts can be easily devised and implemented, although endless refinement is possible. Many explicit contexts have been implemented, notably symbol tables and external symbol directories.

Nests such as

(ℓ, m) (n, o, p) (q)

[3]This sketch is of course greatly simplified. I prefer to view the meaning itself as the physical actions which occur in transforming the information nest and context. Thus, the human translates optical marks and stored knowledge to keystrokes; the keypunch translates keystrokes to punch actions within the context of the switches; the card reader is a machine which, given the right control signals, translates card holes into electric signals. And so forth.

[4]Many time-sharing editors have such capabilities, for instance the Stanford Wylbur system [10].

can be parenthesized to show their heirarchical nature

((ℓ, m) ((n, o, p) (q))).

In this form, nests have a well defined machine representation in LISP [8]. In general, all we need is two distinguishable delimitable units, where a delimitable unit is a variable-length field with a recognizable type indication. One type of unit would represent elements separated by commas. The second type would represent elements separated by spaces. For instance, the example

(ℓ, m) (n, o, p) q

would consist of:

```
[type 2

     [type 1 ℓ]
     [type 1 m] ]

[type 2

     [type 1 n]
     [type 1 o]
     [type 1 p] ]

[type 2 q].
```

Human Representation

Representing these information structures in a humanly readable form is a more interesting problem. As LISP has shown, complex parenthesizing can become quite tedious and confusing. Many improvements to the LISP syntax have been devised, such as the A-Language of Henneman [11]. This paper will approach human readability without reviewing LISP modification schemes, primarily because the notions generally come from the more common higher-level languages, such as FORTRAN, Algol, and PL/I.[6] Some semantics will be injected into this syntax but only where this is judged important to human usability, and the semantics are always separable by an initial parse.

NESTING

We have already seen one simplification from LISP parenthesizing by introducing two field delimiter types (comma for sequential separation and no delimiter for nested or heirarchical separation) while viewing parentheses as bracketing symbols only and not field type indicators. Thus, what would be

(A (B (C (D E))))

[6] The human representation of this syntax is being explored in an experimental language, tentatively named HOLOGO.

in LISP at the EVAL level, could be

```
A B C (D E)
```

in the syntax proposed above.

SEPARATING

As was indicated above, nodes would be either nested, i.e. have no separators between them, or they would be separated. For instance, an argument vector is nested to its function while the elements of the argument vector are separated from each other. Explicit control of parallel processing might be desirable. One very readable way of approaching this is to allow two separators: the semicolon to separate serial elements and the comma to separate parallel elements. For instance,

```
DO  (20;

     READ  (A, B);

     WRITE SUM (A, B)  )
```

would indicate that READ and SUM can evaluate their arguments in parallel but DO must evaluate its arguments in series.[7] This approach introduces semantics into syntax. The separation of semantics could be accomplished for machine representation by the transformation

```
(a, b, ...) → fork (a; b; ...).
```

Alternatively "," and ";" might be used as terminators rather than separators ([12], p.14).

BRACKETING

An obvious step toward readability is to use BEGIN and END as bracketing symbols like "(" and ")". Various forms of labeled and unlabeled brackets have been devised.[8] One form of bracketing that might be profitably used as an extensibility function would be bracketing by separators. The separators THEN and ALSO would be used similarly to semicolon and comma respectively, but would imply a bracketing also. For example,

```
WRITE (A; B; C)
```

could be expressed as

```
WRITE A THEN B THEN C
```

[7]It would be a descendant function of DO which would actually evaluate the arguments and use this information if the program were being computed.

[8]See for instance Wegner [13, p.40]

and

```
WRITE (A, B, C)
```

could be expressed as

```
WRITE A ALSO B ALSO C.
```

Note that IF A THEN B is simply the function: IF (A; B). Other special purpose separators could be user defined to aid readability of individual functions.

LABELING

Labeling is a specialized form of definition and could be left to definition functions. Most languages, however, allow the convenience of a syntactic definition (i.e., labels) for functions which have no explicit arguments (i.e., are used as addresses). A label syntax like that used in PL/I could be used, letting the label be separated from the node being named by a semicolon.

PARAMETERIZING

The definition processes require some definition-time functions to specify binding and return values for a function.

One possible approach to binding is specifying the arguments of a function along with the label. Using an image of the execution structure left of the colon as in

```
F (arg1, arg2, arg3)  :  definition
```

introduces an awkward look-ahead. It is also desirable to save the nesting relation left of the colon for labeling hierarchies[9].

Although the LISP LAMBDA could be used to specify the argument variables in a definition, a slightly different function, say ARGS, would be more useful. This function would be used to evaluate and define arguments. The first nodes nested under ARGS would be a vector of the argument names. The remaining nest would be the portion of the definition within the scope of the argument definitions. For example, a function, FCN, might be defined as

```
FCN:  ARGS(name 1, name 2, ...)
      definition
```

[9]Nested labeling is beyond the scope of this paper.

In many languages there are special functions[10] which are intended for unevaluated arguments. This could be handled by passing the argument nest unevaluated to the invoked function and leaving the evaluation to the definition function[11]. It should be noted that the arguments themselves would be in the function's execution time nest, while the arguments of ARGS would be evaluated at the funtion's definition time.

The individual unevaluated arguments (elements of the first nested vector) might be referred to outside of (i.e. before) the normal definition function.

VALUING

In many higher level languages there is a RETURN function for attaching a value to a subroutine and returning control to the caller. In the syntax discussed here the return valuation must be a vector of values. To aid readability, the return valuation could be specified near the top of a procedure by a function called RETURNS. The arguments of RETURNS would specify temporary fields to hold the return values. The remainder of the definition vector under RETURNS would specify the code to be executed within the scope of those field definitions. Execution out of the bottom or execution of a RETURN (no "S") function would cause the actual return of control.

The following is an example of a complete definition, using a label, the ARGS function and the RETURNS function[12].

```
ADD1:  ARGS NUMBER
       RETURNS TOTAL
       ASSIGN (TOTAL, SUM (NUMBER, 1) )
```

ATTRIBUTING

In most languages, expected attributes may apply to certain kinds of functions. In LISP a user may arbitrarily associate an attribute with the definition of an atom via the property list. In a generalized or extensible information language it would be useful to allow a special context to be associated with each function definition which would list that function's "attributes". This context would be concatenated with the execution context whenever the function was invoked.

[10]For example the MAP... functions in LISP [8].

[11]This is the modification to the right-to-left parse mentioned in Footnote[1]. The inter-node parse would be two phase: left-to-right initialization, followed by right-to-left reduction.

[12]A definition with nothing nested below the argument of RETURNS would correspond to a LISP DEFINE. The nest below the RETURNS argument introduces a PROG type of feature [8].

ASSIGNMENT

As was assumed in the last example (under VALUING), the assignment function, may be a normal prefix function. An infix assignment operator, however, greatly aids human readability. Any symbol, such as "=" or "←" or ":=" might be used as an infix assignment operator. Infix operators could be easily converted to the normal prefix notation during the initial parse.

The semantics of assignment should correspond to familiar usage. Assignment is traditionally the redefinition of the value of a function while preserving its form and attributes. In most languages, the function being assigned must be a scalar or array variable. In PL/I, there are built-in pseudo variables which appear as functions being assigned [14]. In AED, user defined functions may appear to be "assigned" by a mechanism which translates the assigned value into an extra argument for the function being "assigned" [7].

One way of achieving a familiar but versatile assignment function is to associate a value attribute with each function. Variables would be functions which simply return their value attributes. Arrays would select from value vectors according to the calling arguments. User defined functions (i.e. data types) could use the value attribute just as any other variable in their context.

ARITHMETIC OPERATORS

The conventional infix operators for arithmetic would be important for human readability.

POSSIBLE OTHER SYNTACTIC LEVEL FUNCTIONS

Functions implemented at the syntactic level should be as restricted as possible to ease human learning as well as implementation. The system outlined here could easily accomodate most capabilities as explicit functions and be reasonably readable. Extension at the syntactic level should be limited to capabilities which would greatly affect readability and which would be very basic to the system.

Delaying execution is one possibility. In a system of interpretation machines there is a series of execution times: Macro time, compute time, assemble time, link time, JCL time, initialization time, etc. Having a capability of explicitly specifying the context under which a function is to be reduced might become very important. A function, such as "DELAY n nest" might be used to delay the reduction of "nest" for n passes. This concept has not yet been well explored, but relative or absolute specification of activation context might become important enough for syntactic expression. One possible representation would be an optional infix operator such as "@" which could be used to attach the specification to a function, for instance[13]

[13]COMPILATION would, of course, be a function defined in the interpreting context to delay reduction until the compilation pass.

BUILDTABLE @ COMPILATION (2,256)

Another candidate for syntactic implementation in a few situations might be the specification of a function's units. So far, the development in this paper has assumed that all functions are dealing in the same units. Most effects of functions have been moved to explicit argument passing or explicit context modification. The units used, however, have been left to implicit agreement between functions. An infix operator such as a "#" might be used to allow explicit unit specification, for example[14]:

DISTANCE # INCHES = 35 # MPH * 3 # SEC

Error Checking and Redundancy

The approach suggested here increases generality at the expense of redundancy. Lack of redundancy facilitates implementation, learning and use but eliminates error detection information. Dropping a comma, for instance, would always lead to another valid construct. Even excess arguments could be passed along to the next higher level without taking note. Such flexibility is essential to the generality of the system. In actual use, however, some redundancy would be required to help detect errors.

Formalized redundancy has been developed in such disciplines as communications and accounting. It would also be possible in interpretation machines. A user might include with a new function definition an example of an input vector and the expected output. The function could then be "parity checked" whenever a modification was made to the system which might affect it. The user could certainly specify limits such as number of arguments when they were known. In short, redundancy can be implemented in explicit extension functions as effectively as when it is imbedded in the basic language. The difference is that with limitations imbedded in the nuclear language, it is difficult to generalize and explore what might be possible in programming tools.

Economics

The development of a formal interpretation machine system would be top down to the point of apparently defying the economics of computation. A table lookup, for instance, would be required for each pass of each function in each context[15].

[14]Again the units would be functions defined in the interpreting context.

[15]This is actually no more than is done in current systems being interpreted by micro-coded machines.

On the other hand, the possibilities for higher-level economies are greatly improved. Eliminating redundancy saves memory and transmission time. Optimization logic would apply equally to all uses. Changes could be implemented as easily as original code. There is also the all-eggs-in-one-basket effect, since the localization of active logic to a fairly simple mechanism would allow better control of resource usage so that a system could be tuned to the economics of the resources being used. For instance, multiple processors and hierarchical memories (e.g. cache, high speed, low speed, disk) could be used quite effectively. Even hierarchical bus (channel) structures could be supported to spread usage of that critical resource. It would appear at least possible that a more formally integrated interpretation machine architecture would be cheaper when all the costs were considered.

Summary

A general purpose syntax for information handling has been developed by generalizing the relation $y = f(x)$ to $(y_1, y_2, \ldots) = (f_1, f_2, \ldots)\ (x_1, x_2, \ldots)$. This generalization introduces a flexible unit of information, the node, which may be null, a scalar, a vector, or a tree in structure and which may be interpreted to produce another structure and side-effects.

A formal approach to interpreting information has been suggested, viewing interpretation as the reduction of a nest of nodes within an alterable context; i.e., interpretation is viewed as a 2-tuple mapping: interpretation $[nest_1, context_1]$ $[nest_2, context_2]$ + action. This approach allows the generalized interpretation of many different forms of information in many different contexts. It also allows the interpretation process to be nested. A mechanism which performs such a mapping was termed an "interpretation machine" and examples of currently existing, ad hoc interpretation machines were given.

Machine representation of information and context for interpretation machines were explored lightly. The more difficult problem of human representation was explored at greater length, outlining some suggestions for readability.

The suggested generalizations necessarily involve an elimination of accidental redundancy which could have been useful in detecting errors. Experience in other fields, however, such as communication and accounting, indicates that formal redundancy can be introduced and used as effectively.

The formalized implementation of an interpretation machine capability would involve an unconventional approach to the economics of computation. Despite apparent inefficiencies a system based on such a model could turn out to be more economical than ad hoc systems.

REFERENCES

1. Gries, David. "On Structured Programming - A Reply to Smoliar". Comm.ACM, 17,11 (Nov. 1974), 655-657.

2. Abrahams, Paul. ' "Structured Programming" Considered Harmful'. SIGPLAN Notices 10,4 (April, 1975) 13-24.

3. Wegner, Eberhard. "Control Constructs for Programming Languages". SIGPLAN Notices 10,2 (Feb. 1975), 34-41.

4. Robinson, L. "Design and Implementation of a Multi-level System Using Software Modules". Carneigie-Mellon University Report, 1973.

5. Liskov, B. H. "A Design Methodology for Reliable Software Systems". Proceedings FJCC, 1972, 191-199.

6. Parnas, D. L. "On the Criteria To Be Used in Decomposing Systems into Modules". Comm.ACM 15,12 (Dec. 1972), 1053-1058.

7. Introduction to AED Programming, 1973, SofTech Inc., Waltham, Mass.

8. McCarthy, John, et al. LISP 1.5 Programmer's Manual, The M.I.T. Press, M.I.T., Cambridge, Mass.

9. Elson, Mark. Concepts of Programming Languages, Science Research Associates, Inc., 1973.

10. Wylbur Manual, 1973, Systematic Data Processing Services, Inc., Waltham, Mass.

11. Henneman, William. "An Auxiliary Language for More Natural Expression-- the A-Language". The Programming Language LISP: Its Operation and Applications, Berkeley, Edmond C. and Daniel G. Bobrow eds., 1974, 239-248.

12. Gannon, J.D. and Horning, J.J. "The Impact of Language Design on the Production of Reliable Software". SIGPLAN Notices 10,6 (June, 1975), 10-22.

13. Wegner, Eberhard. "Control Constructs for Programming Languages". SIGPLAN Notices 10,2 (Feb. 1975), 34-41.

14. IBM System/360 Operating System PL/I (F) Language Reference Manual (Dec. 1972), IBM Order No. GC28-8201-4.

15. Dijkstra, Edsger G. "Guarded Commands, Non-determinancy and a Calculus for the Derivation of Programs". SIGPLAN Notices 10,6 (June, 1975), 2-13.

16. Reifer, Donald J. "Automatic Aids for Reliable Software". SIGPLAN Notices 10,6 (June 1975), 131-142.

17. Winograd, Terry. "Breaking the Complexity Barrier Again". SIGPLAN Notices 10,1 (Jan. 1975), 13-22.

18. Parnas, David L., et al. "On the Need for Fewer Restrictions in Changing Compute-time Environments". SIGPLAN Notices 10,5 (May 1975) 29-36.

19. Parnas, D.L. and Siewiorek, D.P. "Use of the Concept of Transparency in the Design of Hierarchically Structured Systems." Comm.ACM 18,7 (July 1975) 401-408.