(7) In scheduling, how much of our resources do we allot to any particular job? For example, how much storage and how many tape units are scheduled when sorting is able to take a variable number, and other users are competing for space and time?

(8) What records do we need to keep in the system dynamically, so that we can carry out diagnostics or rebinding that may be necessary?

Mealy noted that the impact of the new operating systems on programming languages raises the following questions:

(1) The problem of naming. Most programming languages think they know all the names in the world. We need to have programming languages that can talk to programs written in other languages, and that can talk about data that has been constructed by programs in other languages. They need to be able to talk about and with operating systems and programs written in other languages.

(2) Languages need to be able to recognize many more data types. Most of them think that there are only five data types in the world. The range of data types should be open-ended.

(3) Users need to be able to make control statements, in the language, to the operating system.

(4) Operating systems should be able to deal with data descriptions that are held with the data instead of having been absorbed at translation time.

(5) Not only do programming languages need to be environment-free; but the programs also need to be environment-free. Most of the time they are not.

Following this summary there were questions, the most interesting of which were as follows:

Steel: Should the operating system be constructed in tasks or in special structures of its own?

Mealy: We do not know yet. Both ways have problems and these have not been solved.

(*Unknown*): Why should we multitask our jobs?

Mealy: There are three basic reasons: (1) for a neater, simpler organization; (2) to take advantage of overlapping of resources; and (3) to take advantage of such things as multiprocessors. Note, though, that if we program to take advantage of these things and then do not have them, we can still run.

Green: Do we need any special language changes for multitasking?

Mealy: No, we do not need any major changes in the language. Multitasking can be handled in the same way as subroutine calls. Some of the parameters of the subroutine call will include the environment in which a task must run.

Mitchell: A present problem is that of separate and incompatible languages for the operating system and the translators.

Mealy: This is a big problem. In the construction of large systems, many individuals develop parts separately, and achieving a complete, consistent system is a problem still not solved.

Naur: When there is the problem of allocating resources, either the programmer can build it into his program or it can be built by the systems designer into the operating system. These alternatives are surely too rigid. What we really need is a way for the programmer to make an environment inquiry from his program and then ask the system for a chosen selection of the resources.

Mealy: Yes, this kind of technique would be valuable, but at the moment it does raise problems which occur when the environment changes and the requests cannot be honored.

Orchard-Hays: It is essential to know what resources you may expect from the system, such as the number of tapes and words of core storage.

Mealy: I think the way to tackle this problem is to be able to put a hold on resources for particular programs.

Evolution of the Meta-Assembly Program

David E. Ferguson

Programmatics, Inc., Los Angeles, California

A generalized assembler called a "meta-assembler" is described. The meta-assembler is defined and factors which contributed to its evolution are presented. How a meta-assembler is made to function as an assembly program is described. Finally, the implication of meta-assemblers on compiler design is discussed.

Introduction

A meta-assembly program is a machine-independent assembly program. It is machine independent in the sense that both the assembly language which it is to accept (the source language) and the machine code which it is to generate (the object language) are supplied as part of the program to be assembled. That is, both the source language and the object language are input parameters to the meta-assembler.

The name meta-assembler was coined by analogy with the word meta-language. A meta-language describes a language; a meta-assembler describes an assembler. The term is not directly related to the term meta-compiler, which has recently come into use. A meta-assembler is probably more closely allied to what has recently been called a macro expander.

Meta-assembly is a proven technique. Several meta-assembly programs have been implemented and are operating on many different computers. They have been used to create assembly programs for these computers and for a number of other computers, some with characteristics very different from those of the base computers.

The meta-assembly process is briefly described here, and a few remarks are offered on existing meta-assembly programs together with developments that distinguish each from its predecessors. The paper concludes with a brief description of METAPLAN, a computer-independent systems programming language based upon the meta-assembly technique, which is translated by way of meta-assembly programs.

Presented at an ACM Programming Languages Conference, San Dimas, California, August 1965.

The Meta-Assembly Process

Most symbolic assembly programs have characteristics in common. Among these are the ability to read and identify constants and symbols, to convert numbers from one radix to another, to build and search symbol tables, to compute the values of arithmetic expressions involving constants and symbols, to allocate storage at object time for the object program, and to assemble related partial words into full words for output.

Traditionally, whenever a new computer model has been manufactured, one or more assembly programs have been written for it. Each of these assembly programs performs all the functions mentioned above. The objective in producing a meta-assembler was to make this redundant effort unnecessary. That is to say, when one has a meta-assembler operating on computer X, one can create assembly programs for computers X, Y, Z, etc., *on* computer X, almost as quickly as one can write down the mnemonic commands and their corresponding computer codes, and the lengths and ordering of the partial words which make up a machine instruction.

One can think of several situations in which computer installations can benefit by assembling on one computer for another, particularly when creating the new assembly program requires less than a day's work. However, there is still a need to assemble for the new computer *on* the new computer. This problem is solved by coding the meta-assembly program itself in the machine-independent language METAPLAN, which is discussed at the end of the paper.

Syntactically, a meta-assembly program consists of one or more logical lines, each of which occupies one or more physical records. Each line contains a label field, a command field, and an operand field. Comments may appear following the line.

The label field is ordinarily blank. If, however, it contains a symbol (identifier), this symbol ordinarily takes on the current value of the location counter, as in a conventional assembler.

The command field contains a symbol. This symbol may be the name of one of the directives built into the meta-assembly language, or the name of an entity previously defined in the current program. The directives built into the meta-assembly language are machine-independent. They merely provide a way of describing the desired assembly program. Several of them are discussed below.

The operand field may be blank or may contain one or more operands separated by commas. Each operand may be an arithmetic expression or a list, where a list is composed of one or more operands separated by commas. Thus, in the general case the operand field contains a list, which may contain lists, etc. That is, the definition is recursive, but ultimately each operand is an arithmetic expression. Arithmetic expressions consist of symbols and constants connected by (1) arithmetic operations of addition, subtraction, multiplication and division; (2) relational operators, less than, greater than and equal to; and (3) the logical operators AND and OR.

One of the directives in any meta-assembly language states the word size of the computer in bits or characters. Another directive gives the algorithm for representing a negative quantity in the object computer in terms of positive quantities. Clearly, different algorithms are required for two's complement, one's complement, and sign magnitude computers. Another directive is the familiar EQU directive, which is present in many conventional assembly programs.

None of the meta-assembly directives produce code. The programmer produces code by means of the FORMAT directive. This directive specifies that the values of one or more arithmetic expressions are to be assembled into words and output as part of the object program, and it specifies the length and ordering of the expressions. Each format has a name. To reference the format the programmer writes its name in the command field. In the operand field he writes the expressions which are to be assembled into this format.

For example, suppose that we wish to assemble the constant -5 for a sign magnitude 36-bit computer. The coding for this might be written as follows. The first line would define the word size as 36 bits. The second line would define the representation of $-N$ as the logical union of N and a sign bit. The third line would contain the definition of a format which stated that one field of 36 bits was to be assembled. The name of the format (say, DATA) would appear in the label field of the third line. The programmer could now write a line containing DATA in the command field and -5 in the operand field, and at assembly time this line would produce a 36-bit -5 in the object program.

To generate an instruction, the programmer might define a format with 3 fields, where the fields represented the command, the address and the index register. He could then reference this format by name, giving as arguments an octal command code, a symbolic address and a symbolic index register; at assembly time this line would produce a computer instruction in the object program.

This example should serve to demonstrate that machine-independent code generation is possible. Obviously, referencing the same format repeatedly to generate instructions would be impractical. Accordingly, several other directives exist within the meta-assembly language to facilitate creating an assembly language more familiar to the user and more concise.

It has been mentioned that several kinds of entities may be defined by the programmer. One of these is the format; another is the procedure. A procedure has one or more names and is referenced by writing any of its names in the command field of a line. One simple procedure might be one to generate instructions. The programmer calls the procedure by writing the appropriate mnemonic command in the command field of a line and the address and index in the operand field, separated by commas. Within the procedure itself is a format reference which generates the actual bits of the command code and assembles the address and index field into the proper places in the output word.

One can see that turning the meta-assembly into an assembly program is simply a matter of writing procedures corresponding to the desired instructions and data formats of the assembly language. Since most computers have few different formats for their instructions and data, few procedures are ordinarily required. The creation of a new assembly program can be accomplished quickly.

Procedures are, in effect, a generalization of the macro concept, and it may already be apparent that procedures may be used for much more elaborate purposes than assembling one-for-one machine instructions. An example of the power accorded by procedures is given in certain library programs written by SDS which generate code for the 920 computer or for the 9300 computer based on the value of a single variable within the program. If this variable is set equal to zero a 920 computer program is developed; if it is set equal to 1, a 9300 program is developed. The programs are of different size and contain different instructions.

The code within a procedure may be any code permissible outside a procedure. Procedures may call each other to any depth and may pass their arguments down. There is also provision for making a symbol defined within a procedure available to the next higher level procedure. The number of lines of code generated by a procedure may be zero or greater and may vary from one call of the procedure to the next. This is made possible by the DO directive which permits conditional skipping of one or more lines following it.

It is straightforward to develop a compiler level language for a particular application which consists of nothing but calls on procedures. One such language is the business language developed by SDS which in certain applications calls procedures to 8 levels deep. Procedures at the lower level may be concerned with code optimization and do as good a job of optimization as the source language permits or as good a job as the programmer who writes the procedures cares to build in. A good example might be the problem of moving *m* characters starting in the *n*th character position of location *L* to another location, in a fixed-word-length computer. Optimum code to perform this function varies widely from one computer to the next. It also depends heavily on where the word divisions fall within the character string being moved. If one expresses this operation as a procedure call with 5 arguments, however, the procedure can generate optimum code for whichever computer is to be used for this particular run.

Meta-Assembler Programs

The first program which could qualify as a meta-assembler was UTMOST, for the UNIVAC III computer. It was completed in 1962. UTMOST introduced most of the features which distinguish the meta-assembler: formats and procedures in particular, generalized arithmetic expressions, lists, etc. UTMOST introduced subscripted symbols as a notation for addressing procedure arguments from within the procedure. It was also perhaps the first assembly program to utilize an ALGOL-like block structure.

The syntax of the meta-assembler source line is much like that of FAP, the FORTRAN Assembly Program for the IBM 7000 series. From FAP, the meta-assemblers also gained the following features: the use of a relocatable location counter; the DUP directive in FAP, which is the basis of the DO directive in the meta-assemblers; and the convention for making a symbol external to the current program. (In meta-assemblers, a symbol may be made external to the current procedure as well.)

SLEUTH, for the UNIVAC 1107, followed UTMOST. SLEUTH introduced the notion of the function, which is essentially a procedure which returns a value and is referenced in the operand field rather than in the command field.

META-SYMBOL, for the SDS 900 series computers, generalized the concept of a list in several ways beyond that of either UTMOST or SLEUTH. META-SYMBOL also introduced a new "squoze" scheme for encoding symbolic information into binary. This reduced the volume of information by a factor of about 10 to 1.

A meta-assembler was developed as part of a FORTRAN II compiler for the militarized version of the UNIVAC 490. It had both extensions to and deletions from the customary syntax, for this special purpose. The program was syntax-directed and was able to translate two different assembly languages as well as FORTRAN.

Meta-assemblers have also been completed for the RCA 110A computer and for the Spectra 70 series. With the Spectra 70 meta-assembler, the concept of the many-to-many macro was introduced, probably the most significant development in assembly programs since the introduction of the standard, or one-to-many macro. It is through the use of the many-to-many macro that the Spectra 70 series meta-assembler is able to produce highly efficient code from the METAPLAN source language.

METAPLAN

METAPLAN stands for META Programming LANguage, and denotes a computer-independent language at the compiler level. METAPLAN statements are of two types: declarative and imperative. Declarative statements declare data of several kinds, including: data in tabular form, whole word and partial word data, and programmable switches. Imperative statements specify the program flow, moving and testing of the data, and arithmetic operations.

A typical imperative statement is:

```
IF F(A) PLUS 5 EQ G(B) GOTO L
```

In this example, F and G are previously declared fields. Let's assume the F field is the 3rd to the 6th bits of the argument (in this case, A), and that the G field is the last 15 bits of the second word following the argument (in this case, B). Optimum code is generated to load the contents of A, isolate the 3 bits of the F field, add 5, and compare the result with the last 15 bits of the word at B+2. What actually happens here is that IF, PLUS, EQ and GOTO are procedure calls, each with an argument. The four calls together constitute a many-to-many macro, many procedure calls producing many instructions in the object code.

Those of you who have dealt with the problem of generating optimum code in this kind of situation will recognize that treating IF, PLUS and EQ as unrelated one-to-many macros, and generating code from each in turn, would not in general produce optimum code. The many-to-many macro, however, gives the translator the ability to look ahead to future requirements (or back) and thus hold shift operations to a minimum, take advantage of particular computer commands which load and store address fields, etc. Furthermore, the actual code to be generated by the procedures is expressed in assembly language, whereas in a conventional compiler it is hidden within the compiler program itself.

What we have here is a way of exhibiting the semantic content of a compiler statement, just as recently developed syntax-directed techniques have made it possible to exhibit the syntax of the compiler language. METAPLAN is essentially a *semantics-directed* compiler.

It seems likely that *semantics-directed* compilation techniques will have an impact on the compiler technology of the next few years comparable to the impact that syntax-directed techniques have had in the last few years. By combining syntax-directed and semantics-directed techniques, implementors can adapt an existing compiler or meta-assembler to a new computer in weeks, rather than coding an entirely new program over a period of months. In fact, the bulk of the task consists in programming I/O routines and a loader for the new computer.

The introduction of the many-to-many macro, and the resulting development of METAPLAN and the concept of the semantics-directed compiler, have brought the meta-assembly program to its current state of evolution.

DISCUSSION

Gorn: To trace the evolution facts we could, for instance, go at least as far back as 1954 when there was a report by the Navy on a conference just like this, on automatic programming. You'll find in there a number of sources for many of these ideas. It isn't hard to see where they were using meta-assembly concepts, and the one I remember best is by a peculiar coincidence my own presentation (laughter) on a universal coding experiment. Of course, universal coding was a typical mathematician's name for what is now called common programming language.

One of the things that occurred in the universal coding experiment was the separation of assembly and translation, and the characteristic thing about the assembly job was that it was considering generalized linkage of what is being called here many-to-many macros. In fact, the universal code itself has a numerical code giving the number of entrances, the number of inputs, the number of outputs, the number of exits. The fact that it was meta-assembly that was involved was clearly indicated by the fact that the assembly program tried was written in U-code itself, and was therefore, in that respect, machine independent. Going out from there, even in Sperry Rand there was a development up to that point, and you might look at the work by Holt and Turanski on GP and GPX, and that evolves to the extended machine concept that we've been hearing about. Also, even as recently as two years ago, there was Sibley's SLANG language in IBM, which handled a typical meta-assembly process.

Ferguson: This sounds interesting. I'd like to know more about it, and perhaps Holt will tell us a little more about it in a minute. Maybe I should have used the term "parallel evolution."

Carr: It's interesting to try to find what one might mean by assembly. My definition, your definition, may disagree. Nevertheless, I think there are some basic assumptions which could be useful.

The first is that an assembler is some sort of growing system that inputs one language, or many languages, and outputs a second language which is close to machine language. That this second part is close to machine language is most important.

Now I wrote down some things I thought were wrong assumptions. The first is that an assembly program should be separated from other portions of the overall system. This is purely historical in that in any machine you generally have to write the assembly program first. If you are bootstrapping then you go back later on and redo the job. Most people are willing to do this because you get a far more powerful product than the one started with.

The second assumption which I think is implicit in most of the discussion here is that the assembly program should be special purpose. I don't know why it should be special purpose. I can see no reason why one should insist that it always has to be written on one line per card, or the result to be in one certain form. I think this again is just historical. The purpose as I understand it is to produce good code by means of programmer decisions. And these programmer decisions could be decisions of his own or they could be meta decisions which are programmed into the axioms of the particular problem.

The third assumption which I think is false is that assembly programs should be written in machine language or something like it. This is partially implicit here, although, of course, the idea of machine independence is very much there. Why can't the assembly program be written in JOVIAL (which is a very good language for doing this sort of thing), not only in JOVIAL but as part of JOVIAL, so that any programmer can go along and at any stage he can write a JOVIAL line or he can write a line which gives him an assembled instruction? I would like to propose that one should have a central unified system, that all languages could be intermixed. I know that the author does point out that FORTRAN and assembly language have been mixed and I think it is a good thing. But I see no reason why if one planned correctly that all the languages one has could not be able to refer to each other at almost any stage. It's a much more difficult planning job but it's one where the end result would seem to be more powerful. There should certainly be a common naming structure, and finally, the assembler should be written in the most powerful language around, rather than the most trivial language around.

Ferguson: I'd be glad to comment, but I'm not sure I got the question. So far as I'm aware, this type of format is unusual to me in assembly languages, although I don't think it's a particularly important point. As far as efficiency goes, I don't think of that as a function of the assembler; that is, I think of the assembler as assembling the code the user wrote. On coding the assembler in JOVIAL, that's another computer-independent language that it could be written in, and probably pretty well, although I don't think it would have the property of being able to move itself onto a new computer just by changing procedures.

Opler: May functions and procedures be nested within other procedure and function definitions?

Ferguson: Yes.

Opler: May calls to functions and procedures be recursive?

Ferguson: Yes.

Holt: Re: Dr. Gorn's remark, it seems to me that systems are complicated things and the knowledge of them consists in some totality of growth possibility and a particular way of combining various technical inventions.

I think that Ferguson's remark about UTMOST being the first meta-assembler in that sense is highly justified, no matter what separate parts of technique that are found there were repeatedly found in other peoples suggestions.

Then, I'd like to give my own translation, short translation, of Dr. Carr's comments, which is that it seems to me he has said he is not interested in assembly.

Leavenworth: I think somebody else has already said that the many-to-many macro tries to imitate a portion of a compiler. It seems to me that the many-to-many macro has a large effect on syntax. I don't see why the macro in general should be restricted to a functional prefix format. I don't see why you can't couple a macro concept to a syntax compiler that handles general syntax structures, so that you could define new constructs in terms of those previously recorded. A few people have mentioned the macro system implemented at Bell Telephone Laboratories. I'd like to refer to the paper by McIlroy in 1960 in the *Communications*, which discussed what those macro concepts were.

Ferguson: I agree with you. The purpose here was to achieve a particular goal, namely, the goal of developing a language in which we can express a meta-assembly process and translate it into the meta-assembly language with procedures. That, of course, doesn't conclude our interest in the subject, but it's all that was necessary to achieve this result.

Irons: One of the things which was said, and I think one of the important things, about this technique is the ability to bootstrap a compiler onto a new machine. This will save a great deal of programming time, I am sure, but I wonder how much.

Some of the things that strike me that have to be redone or accounted for in some way are, obviously, some concerned with input/output. But more importantly than that, two things I think of are: (1) that a new machine is likely to have facilities different enough such that we might have to rewrite things to take good advantage of the capabilities of the machine; and (2) that one of the important features in the assembly program seems to be interaction with the library. Many assemblers do generate something that is relocatable and as I think they should, have facilities for getting the decks in the library.

Ferguson: Yes, let me first of all say that I didn't tell the whole truth. When you produce an assembly program or another assembler for a new computer you still have to write the input/output if somebody else hasn't done it, and you also have to write a loader if you're going to use it. Your remarks about a new computer and the possibility of not having the capability to handle it is a good one. We've encountered it before and as Jean [Sammet] pointed out, perhaps the B5000 is an example. All I can say is that until we're confronted with this situation, we'd have to not make any comments. The system outputs binary, which might be important if you have a decimal computer. This says that you would have to do some hand coding in the binary output package. It wouldn't affect anything else.

Irons: Is it bound code with addresses assigned?

Ferguson: The binary output can also include binary heading information and so, depending on the system that it is to operate in, you can define this control information. You can test for the relocatability of symbols, you can make a decision as to whether these should be put in as some kind of header information; you can test for the presence of the definition of the symbol which you might decide would constitute an external reference if it weren't otherwise declared. So you do have this capability. There's no ability to communicate with the library at assembly time except for the PROC's, but you do have symbolic linkage at load time.

Green: In the paper it mentions "semantics-directed." I would like to comment that you did the same thing that syntax-directed compilers do.

Ferguson: The idea here is that information is supplied to the meta-assembler which has the purpose of extracting and exhibiting separately the semantic mapping and is certainly not in conflict with syntax-directed techniques. The important difference is that the semantic contents can be supplied or modified by the user; whereas, whether the assembler operates as a syntax-directed processor or not is really immaterial.

Green: You mean by semantics, the macro definition?

Ferguson: Right. I mean what is defined as the result of a particular source language representation.

Floyd: Does it allow for multiple program counters and, if so, are these treated in exactly the same way as general symbols; that is, is there a distinction between a location counter name and a symbolic address, and how do you control them?

Ferguson: You have several questions. To the first one, yes and no: we have produced systems which do and systems which do not have multiple location counters. Symbolic address was effected by writing a dollar sign in front of the location counter whatever it might be. Dollar sign and some expression referred to location counters and then when you define the symbol it was defined under control of a particular location counter and had the relocatability of that location counter.

Floyd: It seems to me that it would be plausible in assembly to have any identifier act as a location counter.

Ferguson: I agree it would be desirable.

Floyd: How about addressing relative to the present location? If one writes $\$ + n$, are you talking about n locations down in the assembly listings or n locations down in the object program? Do you deal with both of those?

Ferguson: We deal with both of those. We deal with the first one because of history, not because we like it. My personal objection is that it's sad news, but the second is a very useful thing, particularly within procedure declarations. One of the attributes that we have is the line number attribute and with the line number attribute you can ask what is the line number of such and such a label. There are some rather strong constraints here. The label must be a forward reference, must be inside of the procedure, and you'd better find it before you find the end of the procedure. The result you get is the relative line number of the label.

Floyd: How complex, how big, and how fast is a meta-assembler? How long does it take to implement?

Ferguson: As far as efficiency goes, I guess the answer would be which one. The best one is equivalent in speed or comparable in speed to a conventional two-pass assembly program, the FAP, SAP type assembly program. As far as how long it takes to implement one, on a fixed time scale we've coded and produced them in less than two months. Now obviously, if you're going to code it from scratch it's more difficult. I've spent most of the morning trying to prove we don't have to code it from scratch, but obviously every time we do it we think of new things we want to do next time.

Gorn: How big is "we"? You said two months?

Ferguson: We can have one programmer put an existing meta-assembler on a new computer in two months. Now as far as one assembler assuming a working meta-assembler, assembling for another machine, one man-day is a better figure.

Floyd: In short, the question is: why not have all assemblers look alike? I think we should include the proposal, in talking about the matter.

Ferguson: I think that eventually we will. I think that the point is a good one. We had a lot of headaches because in the Spectra 70 we had to be compatible with the IBM language, which means instead of having a very clean syntax we have to do things like the DC constant of 360 and, if you're familiar with the syntax of the DC constant, you'd see the problem. The syntax is completely foreign to the rest of the syntax in the meta-assembly language for no apparent reason. It may be as good but there is really no necessity, as I see it, for it to be different.

At this point, A. Holt talked at length about the facilities provided in the GP and GPX assembly systems for generalizing the concepts of subroutine structure and linkage and the extensive exploitation of library mechanisms. The discussion subsequent to his presentation follows.

Woodger: Generally, one learns by experiencing something one shouldn't have done. I'd like to know if you would indicate what is the main lesson in this work, in this sense.

Holt: Well, I think first of all that I have, since the time that that work was done, seen a variety of other inventions which seem to contribute to the same objective in ways we had not thought of, and I think a prime example of this was presented to us by Dave Ferguson today. There are a lot of techniques that I see there, and even though he faces basic motivation differently from what I presented to you, I see a lot of techniques there which would have been very helpful to us.

I would say another thing that we did not do right was to understand that this assembly system really had to be a part of a larger operating environment. That was something that we very, very insufficiently appreciated in the beginning and gradually came to appreciate. So there was built up a whole complex of, you might say, subsidiary functions or related functions which had to do with library maintenance. Library maintenance was a very important feature here because the whole idea was for people to form many libraries, and the assembler had the capability of referring to many libraries during a single compilation. A problem that we never satisfactorily solved was the following: we thought it would be nice if people would be able to invent formats for writing instructional type information and we thought it would be nice to be able to embed in the same framework stored structures which amount to certain varieties of translators which would help to interpret those special formats in the context of everything else that is going on. We never satisfactorily solved that problem.

Naur: I would like to get a clarification on the last point, regarding the distinction between using facilities for achieving certain ends and say, on writing algorithms.

Holt: To look at the majority of algorithmic languages one gets the impression that the problems of good utilization of facilities are made very hard. I mean it's sort of washed out of view as much as possible in a certain sense and you are even prevented very often from addressing yourself to the problem of utilizing the facilities of the computing device at the bottom end, hoping always that by some sort of entirely automatic means the problem of so-called efficiency can be solved by some system that lies in between, of which, in fact, programming intelligence is demanded.

Naur: I agree with you that programming languages and other higher level languages act as a sort of cushion between you the user and the machine behind. I made these points at the IFIP Conference. It is certainly true that this great danger is something we should be well aware of all the time. But you could look at a higher level language and more or less self-impose or restrict yourself to viewing things from the angle of that language. There you have the same problem of utilizing the facilities behind that language.

Holt: In that sense I certainly agree with you.

Graham: I think most of us agree what translation is. When speaking about programs, this means to take a program in language A and transform it into a program in language B in such a way that the transformed program does approximately what the original one did, providing the meaning of the two languages is known. And of course one type of translation is into machine language, which is what most translators of languages like ALGOL and FORTRAN do. Certainly most assemblers as we know them today are translators.

My idea of a *useful* definition of assembly is the following: We have always associated assembly or assemblers in some way with machine code. So I propose that assembly is a process which takes as input, machine code with something I will call "binding data." Now this machine code is certainly machine-like; it isn't arithmetic statements of the form found in ALGOL. It's essentially machine instructions, with some of the addresses unspecified, unbound. Now the object of assembly seems to be to generate executable code.

The production of final machine addresses I like to call "binding," the implication being that originally this address started out in some symbolic form. The assembly process, then, is to take one or more chunks of machine code (in which some of the addresses are yet unspecified) and the binding data that goes along with each chunk and put them together. In general, this process may not completely bind, although the binding has been carried further. Complete binding of the address may occur in a number of steps in which the address, in some sense, is bound tighter and tighter on each step.

The BSS loader also does some of this. It interprets directions written in a particular language. It's funny language, it consists of certain binary bits in certain places on binary cards which it reads. The binding it does is very simple, it consists of what we call relocation.

Today, the assembler does not in fact do the entire assembly process and hasn't for a long time. Only absolute assemblers ever completely assemble a program. In today's systems the assembly, in general, is in two places: at the tail end of the assembler, and in the loader. In future systems the assembly process is certainly going to occur frequently in still another place, and that is at execution time upon first reference to a symbol that will then be bound. Until that time it will not be bound. An executing program in no sense will be completely assembled.

Holt: I do not want to raise any new arguments about the proposed definition of assembly that has been given, although I disagree with it. I think that all efforts of definition of this sort are going to remain fruitless occupations for considerable time to come. Dr. Strachey suggested that we need some way of understanding what we mean by address and similar fundamental terms that deal with programming. That is a very difficult undertaking in my opinion and until that is solved, really technically satisfactory definitions of functional processes are simply not going to be perceptible.

Green: I maintain that a good deal of the confusion is the lack of distinction between the assembly function and language. I think the assembly function is something that everybody has been talking about as binding. However, the assembly language, or the languages which we call or declare as assembly languages, have a great difference from the higher level languages in the fashion in which they use names. That is that a name in an assembly language is not a variable in the real sense. When we say in assembly language CLA A, what we are referring to as what the name A represents is something which I call the machine equivalent address; that is, the value which you would assign in a linkage function at operation time. When we say A+1 [in assembly language] the value we use for A is this machine equivalent address. It represents, in the class of machines being used right now, an integer. Now one of the properties that an assembly language has which is given to it because of this use of name, is that it has the ability to be introspective; that is, it can treat instructions and data as the same. It can manipulate instructions.

One of the big difficulties with doing this has been that the language in which you attempted to do this, if you used a higher level language, prevented you from really being introspective. It prevented you from utilizing the machine's facilities efficiently. And the reason that it did that is because it never let you handle a machine address as a value. One of the few things which I have not heard described and which we have done and which I think would be useful in this area, is the development in a higher level language of an operator which enables one to use the actual machine address. In XTRAN we use something we call "name operator." (This was developed by Bob Shapiro in about 1959.) The name operator has the property that any expression behind it is converted to integer and it is used in the place of a name. The contents function does not do that. Adding the name operator to a higher level language can give the same properties for being able to control the machine as an assembly level language, and, therefore, the conclusion I would draw is that, if you want to be able to construct different processors for different machines, then there's no need to go down to an assembly level language to accomplish this.

Mealy: I'm afraid I'm about to be unkind. I think that Bob Graham's discussion only succeeded in drawing an elaborate, putrid, red-herring across the issue and Julien Green's discussion has nothing to do with the issue.

Let us first talk about the assembly function as opposed to assembly programs. I think the dictionary definition of assembly is perfectly fitting and proper and applicable and I think many of us have used it this way in the programming field for some years now. Namely, it is the process by which things get glued together, or bolted together, bound if you wish. I think restricting the notion of binding just to address values is a mistake. Symbols have all kinds of attributes. We have to speak about binding for each attribute a symbol may have. Assembly programs—the large majority that have been in existence for the last fifteen years—have done no more assembly than most of the compilers have. They have proceeded in binding address values to symbols on occasion; so have compilers. The real assemblers, the real assembly programs, have also glued in pieces of code in a one-to-many manner—these we call macros; they have pulled things in from libraries, and they have in general glued things together. I think this is a perfectly adequate description of assembly, although not of most assembly programs.

Orchard-Hays: I think it's been demonstrated that assemblers are not understood, but I would like to point out that George [Mealy] intimated that assemblers do not necessarily produce machine code. Data can be assembled in much the same way as code is; in fact, it's been done for many years and many applications.

Gorn: I agree with Holt and Mealy on a broader concept of assembly and think that Bob Graham's point of view is too narrow. Bind is a concept that comes from logic and mathematics, and is the same concept that I see here when you want to bind data. The binding time, therefore, is the time at which you put something into a certain storage position, and that is all binding means. The linkage function of assembly will be binding entrances and exits, and inputs and outputs, in that way.

Requirements for Real-Time Languages

Ascher Opler

Computer Usage Education, Inc., New York, New York

Real-time languages have different requirements from other programming languages because of the special nature of their applications, the environment in which their object programs are executed and the environment in which they may be compiled. It may not be the language extensions that ultimately advance developments in the field. Progress may be made by attacking the special compiling and executing system problems that must be solved.

It is not easy to delineate those areas of computing which may be correctly termed real-time. It is complicated by the overlap between online computing and real-time computing. In the *online* technology, terminal equipment is directly connected to a computer and may be involved at any time in data transmission. In the *real-time* area, at least six types of computing may be differentiated:

I. *Simulation* in real time. A computer executes a program with the time scale corresponding to that of the process to be studied via simulation (e.g. programs to train or test responses).

II. *Parallel Operation with a Process* in real time. A computer executes a program with a time in close correspondence to a real process (e.g., missile position display programs).

III. *Hybrid Operation with an Analog Computer* in real time. A computer performs its function as part of a total system closely coupled to an analog computer.

IV. *Performing an Operational Function* in real time. A computer serves primarily as an element in an external environment (e.g., controlling an actual process via feedback mechanisms).

V. *Performing a Remote Communications Function* in real time. A computer is connected to and services a multiplicity of remote terminals (e.g., message switching, inquiry/response station network).

VI. *Controlling the Operation of One or More Computers.* The modern control program (supervisor) as used with multiprogramming, multiprocessing and/or time-sharing.

Types I and II can generally be handled without enriching existing languages, although statements establishing a time-reference scale would prove a useful addition.

Type III requires special language enrichment to deal with the specialized environment. Since hybrid computation may be done with varying degrees of task division, the language requirements can be expected to be equipment-oriented and very detailed. In any case, language elements related to analog-digital *conversion* requirements would be heavy.

For Type IV, the relations between system-wide process elements and the specific functions to be performed by the computer will dominate. Input and output may require considerable conversion and normalization. Input requires access to sensors, and control instructions to

Presented at an ACM Programming Languages and Pragmatics Conference, San Dimas, California, August 1965.