U2648

ENGINEERING RESEARCH INSTITUTE

# AN INTERMEDIATE PROGRAM LANGUAGE AS AN AID IN PROGRAM SYNTHESIS

By

Arthur W. Burks

PROJECT M 828

BURROUGHS ADDING MACHINE COMPANY

DETROIT, MICHIGAN

JANUARY 25, 1951

UNIVERSITY OF MICHIGAN • ANN ARBO

## TABLE OF CONTENTS

# AN INTERMEDIATE PROGRAM LANGUAGE AS AN AID IN PROGRAM SYNTHESIS

## 1. An Approach to Program Synthesis

As a practical matter it is necessary to begin programming with a formulation of the accounting problem in ordinary business language. The end result of the process of programming is a definite program stated in the internal (machine) language (in the narrower sense).[†] Thus the task of the programmer is to translate statements from ordinary business English to the internal program language. We propose to simplify this task by means of intermediary languages so that the translation can be done in smaller steps.

We suggest the following four program languages and present them in the order of their use:

1) Ordinary Business English (as it now exists): This language contains such commands as "post the ledger", "modify the inventory record in accordance with this sale", etc.

2) Operator Program Language: This is to be a restricted form of 1) designed from the operator's point of view (not the machine's) — in it he is to express accounting problems in

[†] For a definition of the terms used here see Arthur W. Burks, "The Logic of Programming Electronic Digital Computers", forthcoming in the first issue of the Journal of Industrial Mathematics, Industrial Mathematics Society, Detroit, Michigan, Section 3. We will refer to this article hereafter as "Logic of Programming".

The essential point here is that the program must ultimately be expressed in a language that consists only of numbers (arbitrarily assigned to represent operations and memory locations) and does not contain variables.

a sufficiently precise form to make mechanization feasible.[†]

3) Intermediate Program Language: An abbreviated form of 4), to be discussed in detail subsequently.

4) Internal Program Language: This is designed from the point of view of the machine, with efficiency of computation and communication the main consideration.

Only the conception of 3) is new; it is the main purpose of this memorandum to elaborate on it. The use of the complete hierarchy of languages is discussed in Section 4.

Hereafter "Program Language" will often be abbreviated by "PL".

## 2. A Proposed Intermediate Program Language

The actual locations of quantities in the memory are arbitrary. Hence, to assign them when the program is first being constructed is to add to the complexity of writing the program and to decrease the perspicuity of the representation of the program. Indeed, in many cases definite addresses are not assigned but only free variables; e.g., instead of

```
7. i
8. 1
9. +(7,8,7)
```

we write

```
E.6) i
E.7) 1
I.3) +(E.6, E.7, E.6) .
```

It is therefore proposed that the variable quantities themselves be used in

[†] Cf. Arthur W. Burks, Frank D. Faulkner, Janet C. Wahr, Don W. Warren, Jesse B. Wright, The Design of the Languages for an Electronic Accounting System, hereafter referred to as Design of Languages. As remarked on pages 9 and 11 of this report, the Operator Program Language presented therein makes no claim to achieving this ideal.

the Intermediate PL instead of their (variable) locations. Thus the above is replaced by

I.3') $i + 1 \rightarrow i$ .

The fundamental point here is that though the variables and constants represent quantities rather than memory locations, they are to be chosen so that there exists a certain kind of correspondence (to be discussed later) between the atomic symbols used and the bins needed. Hence, when the program is translated from the Intermediate PL to the Internal PL the assignment of memory locations is a routine step, easily automatized.

It should be noticed that because of the rule of one symbol to one bin, the assignments of variables in the Intermediate PL will sometimes be different from the assignments now used. Sometimes fewer variables must be employed. For example, whereas we may now use different variables to distinguish the contents of the same bin at different times (e.g., "$g_i$" and "$g_{i-1}$" in the example presented in the next paragraph), in the Intermediate PL only one variable can be used. On the other hand, whereas we may now use the same variable to describe a single quantity at different points of the memory, in the Intermediate PL as many variables as there are bins occupied by this quantity must be employed. Consider, for example, the problem of meshing the monotonic sequences $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_m$ . Let us suppose that $a_1 < b_1$ so that the first term of the resultant sequence is $a_1$ . In the symbolism of the Intermediate PL this term is no longer $a_1$ after the transfer. Rather, we must symbolize the resultant monotonic sequence in some such way as $c_1, c_2, \ldots, c_{n+m}$ and indicate the transfer by

$$a_1 \rightarrow c_1 \ .$$

It should be emphasized that this is a natural way of using variables to describe processes and that it should facilitate the process of programming, not hinder it.†

It is sometimes desirable in the Intermediate PL to refer to memory locations rather than to their contents; this can be done by means of brackets. Consider, for example, the following sequence in the Internal PL:††

L+i. $a_i$

1. $g_i$

2. +(1,L+i,1) $\quad g_i = g_{i-1} + a_i$ .

In accordance with the abbreviations used previously, this could be written as

$$1'. \quad g + a_i \longrightarrow g \ .$$

However, this way of writing the program does not make it explicit that the address of $a_i$ must be computed from a constant and the index $i$ and then substituted into a command. To make all this evident we write the following in the Intermediate PL:

$$\text{L+i.} \quad a_i$$

$$1''. \quad g + [\text{L+i}] \longrightarrow g \ .$$

The expression "[L+i]" means <u>the quantity located at L+i</u> ; in contrast, the expression "g" means <u>the quantity g</u> . Note that the subscript is omitted from "g" at 1" ; as we pointed out before, this is necessary to keep a correspondence between the atomic symbols used and the bins needed.

† Jesse Wright, who brought this problem to my attention, suggests that there are often natural ways to symbolize the variables needed in the Intermediate PL to describe accounting problems. For example, "$\text{price}_{\text{before}}$" and "$\text{price}_{\text{after}}$" could be used to symbolize the price of an item before and after an order has been converted into an invoice.

†† Strictly speaking, the language used here is not the Internal Program Language but a metalanguage for it, and hence is already an intermediate language. See "Logic of Programming", Section 3.

(Note that for our purposes a symbol like "$C_3$" is an atomic symbol.) In general, subscripts showing variations in the contents of a bin with respect to time must be omitted.

Variables with such subscripts may, however, be employed on the right. The following is a more complete expression of the above program:

L+i. $a_i$

1". $g + [L+i] \longrightarrow g$ $\qquad g_i = \sum_{j=1}^{i} a_j, \quad g_0 = 0$ .

In the notation used for the Internal PL the explanatory comments on the right tell the reader more than the commands (written on the left) do. The reason for this is that the information on the right is recorded in a significant notation (one belonging to the operator PL) rather than the arbitrary notation of addresses (whether $g_i$ is stored at address 1 or address 23 doesn't matter until we are ready to give the machine a definite program). What we should do in constructing the Intermediate PL is to employ as much of this significant notation on the left as is compatible with indicating the details of the computation and the memory requirements.

The planning of a program is simplified if the main course of the computation can be charted without concern for the fine details. Hence, it is an advantage to abbreviate short sequences of commands by single commands. As an example consider this sequence of words expressed in the Internal Program Language:[†]

| | |
|---|---|
| 60. X(39,53,57,61) | 57: $v_{n-1}\Delta$ |
| 63. X(40,23,40,64) | 40: $F(hy_{n-1})G(s_{n-1})$ |

[†] "Logic of Programming", Section 4.

| | | |
|---|---|---|
| 64. | +(40,54,40,65) | 40: $E_{n-1} \left\{ = \frac{F(hy_{n-1})G(s_{n-1})}{C} \right\}$ |
| 67. | X(40,57,41,68) | 41: $E_{n-1}v_{n-1}\Delta$ |
| 68. | X(55,53,40,69) | 40: $g\Delta$ |
| 69. | +(40,41,41,70) | 41: $E_{n-1}v_{n-1}\Delta + g\Delta$ |
| 70. | -(39,41,39,72) | $v_n \longrightarrow v_{n-1}$ . |

When one is making a logical analysis of the problem and deciding on the method of computation, it is easier to design the main flow of commands if all of the above is expressed in the Intermediate PL by a single command

$$55'. \quad v - \frac{FG}{C} v\Delta - g\Delta \longrightarrow v \ .$$

There is one point of importance which the sequence starting with 60 shows that its abbreviation 55' does not, namely, the order in which the intermediary quantities are computed. This order is not particularly relevant when we consider only the single command 55', but it is when we consider the complete abbreviation of words 51 through 70 (of which the above sequence is a part):

| | | |
|---|---|---|
| 51'. | $t + \Delta \longrightarrow t$ | $t_n = t_{n-1} + \Delta$ |
| 52'. | $x + u\Delta \longrightarrow x$ | $x_n = x_{n-1} + u_{n-1}\Delta$ |
| 53'. | $y + v\Delta \longrightarrow y$ | $y_n = y_{n-1} + v_{n-1}\Delta$ |
| 54'. | $u - \frac{FG}{C} u\Delta \longrightarrow u$ | $u_n = u_{n-1} - E_{n-1}u_{n-1}\Delta$ |
| 55'. | $v - \frac{FG}{C} v\Delta - g\Delta \longrightarrow v$ | $v_n = v_{n-1} - E_{n-1}v_{n-1}\Delta - g\Delta$ . |

Clearly, the recomputation of $v\Delta$ and $FG/C$ at 55' would be wasteful and must not be permitted. There are two alternatives here: on the one hand, we can leave it to the person or machine making the translation from the Intermediate PL to the Internal PL to decide on the order of computation; and on the other hand, we can indicate this order in the Intermediate PL program by means of an appropriate symbolism. Since the former alternative makes the Intermediate PL to Internal PL translation more complicated then we desire, we choose the latter alternative.

In the original program, 51-70, permanent addresses were assigned to the quantities $u\Delta$ and $v\Delta$ while $FG/C$ was stored in a temporary storage bin. We will assign all these quantities to permanent storage;[†] "permanent storage" here means "storage occupied by only this quantity throughout a given program." Previously, we had assigned only atomic symbols ( "t", "x", "$c_3$", etc.) to (permanent) storage; we must now extend this assignment to include some compound symbols. To differentiate these compounds from others, we place the former in parentheses. Further, we prime the first occurrence of each in a program to indicate that at this point the parenthesized quantity is to be computed and stored in its assigned bin.[††] In accordance with these rules 51'-55' is rewritten as follows:

51". $t + \Delta \longrightarrow t$

52". $x + (u\Delta)' \longrightarrow x$

53". $y + (v\Delta)' \longrightarrow y$

54". $u - (FG/C)'(u\Delta) \longrightarrow u$

55". $v - (FG/C)(v\Delta) - g\Delta \longrightarrow v$ .

We are now in a position to discuss the correspondence of Intermediate PL symbols to Internal PL storage addresses. Let us first distinguish

[†] This may sometimes cause an inefficient use of storage space. To obviate this we could employ a concept of "relatively temporary storage" (cf. the concept of "temporary storage" introduced below) or "storage over a few lines of computation". Such a device raises problems concerning the automatic translation of a program from the Intermediate PL to the Internal PL, and we will not consider it further here.

[††] This technique was suggested by Janet Wahr, Don Warren, and Jesse Wright, who are also responsible for many of the ideas concerning storage and the one-one correspondence principle formulated below.

storage of commands from storage of noncommand words and consider only the latter. We have already stated that the atomic symbols are to be assigned on the principle that each such symbol requires exactly one storage bin, and the same holds for parenthesized compound symbols. (This generalization covers constants as well as variables.) The foregoing remarks cover all the storage requirements except the "working storage", and we propose to handle that in the following manner.

Let us define line of computation as the sequence of steps required to execute a single line of instructions stated in the Intermediate PL (e.g., line 53" or 55" in the above example). Further, define temporary storage (in contrast to permanent storage) as the storage required during (at most) one line of computation (e.g., the storage required for $(FG/C)(v\Delta) + g\Delta$ while the machine is on its way to produce $v$ ). Clearly, the same temporary storage bins can be used for each line of computation, and the total number of such bins required is the largest number (call it "$n$" ) required by any line of computation. These temporary storage bins may be represented in the Intermediate PL by the atomic symbols "$t_1$", "$t_2$", ..., "$t_n$" . It is important that these symbols are merely introduced for convenience in formulating the one-one correspondence principle given below; they would not appear in Intermediate PL programs (but only in the list of primitive symbols).

The preceding rules of symbolism for the Intermediate PL have been constructed so as to guarantee an isomorphism or one-one correspondence between certain of the expressions used in the Intermediate PL and the storage required for noncommand words in the Internal PL. This correspondence may be formulated as follows: There exists a one-one correspondence between the memory bins needed for noncommand words in the Internal PL and the atomic symbols (including the $t$'s ) plus the parenthesized compound symbols in the Intermediate PL. The purpose of imposing this rule on the Intermediate

PL is, as we remarked previously, to insure that a program in the Intermediate PL will cover the essential details of the final program even though actual bin numbers are not employed.

There are problems that need to be investigated here. For example, when two programs are combined to make a compound program, these two programs might initially have some symbols in common which do not have the same meanings and hence do not represent identical bins. In this case the symbols would need to be redefined in order to preserve the one-one correspondence principle. The general situation here is similar to that resolved in logic by means of quantifiers of specified scope, and perhaps an analogous theory would need to be developed.

It will be seen from the foregoing remarks that the essential merit of the Intermediate PL is that it is an abbreviated form of the Internal PL in which the essential features at the planning stage are shown but the details at this stage are not. Hence, other kinds of abbreviations besides those illustrated in 51" through 55" should be made. We shall mention a few others just as examples. First, the computation of an address and the substitution of it into a command may be conveniently combined. Thus the sequence of words 21 through 26[†] reduces to[††]

| | | |
|---|---|---|
| 21. | $M + z \longrightarrow 22$ | $M + z$: $f(z)$ |
| 22. | $[M + z] \longrightarrow f\ ;\ \beta$ | Look up $f(z)$ and transfer it to a bin; shift control to $\beta$ . |

Second, unconditional switches of control may be indicated by a symbol to the

† "Logic of Programming", Section 4.

†† It might seem that special brackets should be placed around "22" at bin 21 to show two things: first, that the contents of 22 (rather than the number 22 ) are being replaced, and second, that only part of these contents are replaced (since this is a partial substitution). However, no ambiguity can result from omitting these brackets, and it is simpler to write the program without them.

right of a semicolon, as in 22 . Third, the transfer of a phrase from one area to another[+] involves a simple cycle which may be represented by a single command. Finally, a phrase-comparison operation which normally requires several commands[++] may be represented by one.

## 3. Comments on Flow Diagrams

Flow diagrams have been proposed as an intermediate language.[+++] Most people have not found them adequate for this purpose, however; the diagrams are cumbersome to draw and their formulation requires that certain details not represented on the diagram be settled. Hence, they are generally used either as convenient accessories in the process of writing a program or as summaries to be prepared after the program has been written.

As summaries, flow diagrams do show clearly something that an ordinary program does not: the path of the control through the commands. Part of this superiority of flow diagrams over programs is not avoidable, but part of it is. A program is naturally one-dimensional whereas a flow diagram is two-dimensional, and on this account it is impossible for the former to show the path of the control as clearly as the latter. But it is frequently difficult when one is reading a program to determine how the control gets to a certain command, and this information can be easily incorporated into the Intermediate PL representation of the program. The following translation of the 48 words of Programs III and IV[++++] shows how this can be done.

---

[+] Design of Languages, p. 56.

[++] Ibid., p. 51.

[+++] By H. H. Goldstine and John von Neumann, Planning and Coding of Problems for an Electronic Computing Instrument, Institute for Advanced Study, Princeton, New Jersey, Vol. I, 1947. They do not present the diagrams as a language, but they do hold that programming is best accomplished in two distinct steps: preparing the flow diagrams and doing the detailed coding.

[++++] "Logic of Programming", Figures 2 and 3 and Section 4.

## Function Table Subroutine:†

From 33, 41

| | | |
|---|---|---|
| 21. | $M+z \longrightarrow 22$ | $M+z$: $f(z)$ |
| 22. | $[M+z] \longrightarrow f\ ;\ \beta$ | Look up $f(z)$ and transfer to storage. |

To 38, 51

## Main Routine:

| | | |
|---|---|---|
| 31. | $hy \longrightarrow z$ | Transfer argument of $F(hy_{n-1})$ |
| 32. | $100 \longrightarrow M$ | $100 + z$: $F(z)$ |
| 33. | $\beta_1 \longrightarrow \beta$ | $\beta_1 = 38$ |

To 21

From 22 ($\beta_1$)

| | | |
|---|---|---|
| 38. | $(u^2 + v^2)^{1/2} \longrightarrow z$ | Transfer argument for $G([u^2 + v^2]^{1/2})$ |
| 39. | $f \longrightarrow F$ | Transfer $F(z)$ to storage |
| 40. | $200 \longrightarrow M$ | $200 + z$: $G(z)$ |
| 41. | $\beta_2 \longrightarrow \beta$ | $\beta_2 = 51$ |

To 21

From 22 ($\beta_2$)

| | | |
|---|---|---|
| 51. | $t + \Delta \longrightarrow t$ | $t_n = t_{n-1} + \Delta$ |
| 52. | $x + (u\Delta)' \longrightarrow x$ | $x_n = x_{n-1} + u_{n-1}\Delta$ |
| 53. | $y + (v\Delta)' \longrightarrow y$ | $y_n = y_{n-1} + v_{n-1}\Delta$ |
| 54. | $u - (FG/C)'(u\Delta) \longrightarrow u$ | $u_n = u_{n-1} - E_{n-1}u_{n-1}\Delta$ |
| 55. | $v - (FG/C)(v\Delta) - g\Delta \longrightarrow v$ | $v_n = v_{n-1} - E_{n-1}v_{n-1}\Delta - g\Delta$ |

† Don Warren and Jesse Wright have suggested that the term "maximal linear sub-sequence" may be appropriately and usefully applied to sections between "From" and "To".

56. 0, y; 31, $\delta$ , Repeat cycle until shell strikes the ground.[+]

To 31 if $y > 0$

To $\delta$ if $y \leq 0$

A few tests made by the writer indicate that it is possible to write a program at least for simple scientific problems in the Intermediate PL in a straightforward manner, given a formulation of the problem in the Operator PL.[#] It is easy to translate such a program into the Internal PL; indeed, the Intermediate PL was designed so that this step would be routine. In contrast, it is not generally possible to write a program directly in the Internal PL; at least it has been the experience of the writer that (except for trivial programs) a number of trials are necessary. In the next section we will consider why the Intermediate PL should make this difference.

## 4. The Value of the Intermediate Program Language

The purpose of the proposed system of languages is to enable the programming to be done in smaller steps than are now possible. This has two principal advantages. First, smaller steps can more easily be mechanized than larger ones. Second, different kinds of work can be allocated to different stages of the process and to different specialists. Let us consider in detail how this second point works out with respect to the hierarchy of languages.

The programming process begins with a vague, general statement in Ordinary Business English of the accounting problem.[##] The first step in

[+] "0, y; 31, $\delta$" commands a conditional switch of control: "If $0 < y$, then take command at 31 ; if $0 \geq y$, then take command at $\delta$".

[#] Thus it seems to achieve the purpose for which von Neumann and Goldstine designed their flow diagrams.

[##] Recent work on Symbolisis, Abstraction, etc., is attacking the problem at this level of the hierarchy.

programming is to make the problem definite by specifying precisely the inputs and the outputs (the time dimension must be included here). This is done by expressing it in the Operator PL, which has the requisite symbols for this purpose. It seems impossible (in the present state of the art of planning accounting systems) to separate the formulation of the problem from the choice of the methods for achieving the transformation of input into output (sorting, monotonizing, searching, blending, etc.). But at this stage the operator does not need to concern himself with the details of these methods.

The next step is to write the program in the Intermediate PL. This involves more than a mere translation of what was written in the Operator PL, for the methods of solution must be worked out in considerable (though not complete) detail. The variables needed to describe the computation-communication process are chosen in such a way that each variable requires exactly one storage bin and the steps of computation and the flow of information are described.

The last step is that of translation into the Internal PL. This is a bona fide translation, for methods of assigning addresses and of expanding abbreviated commands into sequences of commands can be worked out in advance. Hence the computer could be instructed to do this work. The main problem here is that of conveniently representing in the Internal PL (which contains no variables) programs written in the Intermediate PL (which contains genuine variables). It should be emphasized, however, that even if it were not efficient to use a computer to make the translation, the Intermediate PL would nevertheless be useful to the human programmer in planning and constructing programs.

We can summarize the process of coding as follows: In the first stage the programmer is concerned mostly with the statement of the problem, in the second with the logical formulation of the method, and in the third with the routine details of coding. Thus the hierarchy of languages makes possible a division of labor and the use of specialists. Even if a single person does the complete job of coding, he needs to be concerned with only one kind of consideration (formulation of problem, logical analysis of method, details of coding) at a time. It is for this reason that one can probably learn to write a program in the Intermediate PL in a direct manner, without having to make several trials.

The Intermediate PL is much easier to use in coding than the Internal PL because the variables used have significance for the operator. (This is true essentially because they are symbols of the Operator PL.) Yet, it is close enough in structure to the Internal PL to enable one to decide on all but the most routine details of the program. Hence, it may be used effectively in testing command languages and studying programs, even to the point of making time studies.

To conclude the discussion of the value of the Intermediate PL we make two comments concerning its value in programming accounting problems.

The use of a library of tapes has often been proposed as a method of mechanizing programming. While this technique is very promising for scientific computation, it has one great drawback for accounting. In general, a program made by compounding already existing programs does not take advantage of the unique features of the problem, and hence, does not give the quickest path of computation. This is not a serious limitation in scientific work, where a given program is used only relatively few times, but it would result in inefficient computation in accounting, since in that field a single program may be used for weeks and months. Hence accounting programs must be tailor-made. Since the Intermediate Program Language is very close in structure to the Internal Program Language, it may be used to facilitate this process.

Further, it should be pointed out that there are questions associated with clerical problems stemming from the use of processes which are mainly combinatorial (as contrasted to arithmetic) and also from the inclusion in the language of such things as variable-length phrases and paragraphs — situations not encountered in scientific problems. Further investigation is required before specific questions of this sort can be answered finally.