medium relationship, is properly an operating system function. The language compiler need only make the internal names available to the operating system for reference. This in turn implies an appropriate job-control language to allow the run time specification of the desired assignments.

The implementation problems, although not discussed here, have been considered both from the viewpoint of syntactical and semantical consistency and possible ambiguity, and impact on the operating system. The study has indicated no ambiguity, and a structure for implementation of the operating system has been tentatively prepared. Details of the latter will be published when an implementation is completed.

VIII. Conclusions

The extensions developed to provide a process-control language within the context of PL/I have been described. These extensions provide a great deal of flexibility in the areas of real-time input/output, interrupt handling, and program segment control, yet they have been accomplished with relatively few additions and no contradictions of existing PL/I syntax/semantics. The end result, to anyone versatile in PL/I, is a very natural and flexible extension of the language. This design experience has led to the conclusion that the original choice of PL/I was well-founded and that a further effort to implement PL/I as a real-time language is justified.

Finally, it is interesting to note that some of the proposed features, such as the EXCLUDE and EXCLUSIVE prefixes, have intriguing capabilities within standard PL/I programs as well as in real-time problems.

References


Synthesis and Analysis: A Flexible Technique for Processing Command Language

ISAAC N. SAMUEL

Abstract—This paper describes a simple, efficient, and highly flexible technique for interpreting and processing commands entered by operators and other users of a computer.

The technique consists of two major operations: synthesis and analysis. Synthesis is the process of rearranging the elements of the input statement into a Polish string to facilitate later processing. Analysis is the process of using the Polish string to build parameters and to perform the actions called for by the command. The technique can be applied to processing the statements in any command language in which the commands are essentially imperative in structure. An imperative command is one in which the action verb is the central element (for example, DISPLAY A + B), even though performance of that action may be controlled by a conditional or location clause (for example, IF C > 5, or AT location 1000).

Finally, it is interesting to note that some of the proposed features, such as the EXCLUDE and EXCLUSIVE prefixes, have intriguing capabilities within standard PL/I programs as well as in real-time problems.

The flexibility of the technique makes it a useful programming tool for processing a command language, especially when the action for a particular verb may change or when verbs are to be added to the command language.

The flexibility of the technique makes it a useful programming tool for processing a command language, especially when the action for a particular verb may change or when verbs are to be added to the command language.

Index Terms—Command processing, Polish notation, synthesis and analysis.

Introduction

As the need for communication between man and the computer increases, the use of an effective man–machine language and the efficient processing of that language become more important.

At the same time, designers of a program to process the language must be concerned with future growth of the language. Frequently, the initial set of commands provided for system operators and terminal users is
only a subset of full range of commands to be provided in the future. With this in mind, the designer of the command processor must allow for changes in the functions of some commands and for the addition of other commands. The initial design of the processor must be flexible if additions to the language are not to produce more and more modules and more and more code.

The command processing technique described in this paper fits the criteria stated above. The technique is suitable for handling imperative commands in which the action verb precedes the elements on which the action is to be taken. The processing technique combines efficiency and flexibility, making it a useful programming technique that program designers should apply to other command-processing situations.

The technique described here has been successfully applied to the processing of a command language used for debugging in a time-sharing environment.

**Elements of the Language**

For a language to be useful, both the person using it and the program with which it communicates must be able to understand it.

The person who is using the command language is familiar with a spoken and written language. In America, that language is English. Obviously, the program cannot be designed to handle the full spectrum of the English vocabulary, grammar, and syntax. But it can be designed to handle a limited part of that language, keeping the number of rules to a minimum and making the commands fit a natural imperative format. This was the nature of the command language for which the processor was developed.

The following is a summary of the basic elements of that language.

Each command must contain a verb.

A semicolon denotes the end of a command, and an end-of-block character (Δ) denotes the end of a statement.

A command must begin with a verb or one of two words signifying a conditional clause.

Any second verb within a command is not recognized as a verb; instead, it is assumed to be a symbol.

The placement of a comma in a command not protected by a pair of parentheses causes the effect of the verb to be distributed equally on both sides of the comma.

The placement of a comma inside a pair of parentheses causes the item that preceded the comma to be regarded as an operand on which action is to be taken.

For the processing examples in this paper, a sample language is used, consisting of the following elements.

**Verbs**:

| DISPLAY | PATCH |

**Words introducing conditional clauses**:

| if | and |

**Modifiers**:

| + | - | * | $ | [ | ] |

**Names**:

| Symbolic names or literals (A literal is indicated by a type prefix and enclosed in single quotation marks; for example, X'01' signifies a literal of a hexadecimal 1.) |

In the examples in this paper, the term *operand* applies to names only. The term *operator*, therefore, applies to all else: modifiers, separators, and verbs.

The functions of the verbs and conditional words are:

**Verb**

| Meaning |

**DISPLAY**

specifies that the contents of a symbolic location or actual address are to be written on an output device.

**PATCH**

specifies that the contents of a specified data field are to be changed as indicated in the command.

**AT**

specifies a symbolic location or actual address at which an action is to be taken.

**IF**

specifies an arithmetic or logical condition under which the remainder of the command is to be executed.

Given these elements and functions, let us examine some of the ways in which they can be employed. In the following examples and throughout this paper, the examples will reflect a hexadecimal computer. Each storage location consists of eight bits or two hexadecimal digits. To examine the language, we assume the contents of a small portion of main storage are as follows.

<table>
<thead>
<tr>
<th>Actual Hexadecimal Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 0 1 2 3 4 5 6 7 8 9 A B C D E F</td>
</tr>
<tr>
<td>Contents: 00 00 00 05 00 00 02 11 22 33 44 55 66 77 88 99 AA BB CC</td>
</tr>
<tr>
<td>Symbolic Location A B C D</td>
</tr>
</tbody>
</table>

**IEEE TRANSACTIONS ON COMPUTERS, NOVEMBER 1969**
The effects of command operations on this portion of main storage are as follows.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Modifier Symbol</th>
<th>Command</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>+</td>
<td>DISPLAY A + B</td>
<td>7</td>
</tr>
<tr>
<td>Subtraction</td>
<td>-</td>
<td>DISPLAY A - B</td>
<td>3</td>
</tr>
<tr>
<td>Negation</td>
<td>-</td>
<td>DISPLAY -(A - B)</td>
<td>-3</td>
</tr>
<tr>
<td>Multiplication</td>
<td>*</td>
<td>DISPLAY A*(B - 5)</td>
<td>-15</td>
</tr>
<tr>
<td>Logical or Multiple Conditions</td>
<td>(N/A)</td>
<td>DISPLAY X'1010'</td>
<td>X'1111'</td>
</tr>
<tr>
<td>Offset</td>
<td>.</td>
<td>Use of this symbol modifies the location at which the operation is performed. It allows the user to specify at least two parameters. The first parameter provides a+ or − displacement from the base address; the second parameter specifies the number of storage locations to be used in the operation.</td>
<td></td>
</tr>
</tbody>
</table>

Starting three locations after symbolic location C, the contents of four locations are displayed, producing the result shown.

DISPLAY C. (3, 4) 44556677.

Starting eight locations before symbolic location C, the contents of twelve locations are displayed, producing the result shown.

DISPLAY C. (-8, 12) 000000050000000211223344.

PATCH

PATCH A = X'7777', B = X'4444'

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents Before</td>
<td>00000005</td>
</tr>
<tr>
<td>Contents After</td>
<td>77770005</td>
</tr>
</tbody>
</table>

The IF and AT words enable the user to designate the condition or location for an operation. The condition specified in an IF clause must be met before an operation in a command is performed. An AT clause enables the user to specify an address at which an operation is to be performed. Once the format and rules of the language are established, the program designer turns to the problem of processing the language.

**Basic Structure of Command Processing**

The command processing technique described in this paper involves two steps: synthesis and analysis. An overview of the processing flow is shown in Fig. 1. Synthesis is indicated by the number 1 in the figure, and analysis is indicated by the number 2.

Tracing the flow, the command enters the system from a terminal or an operator console and is routed by communication routines to the synthesis routines. After synthesis processing, control is passed to the analysis routines which prepare operands for processing, link to appropriate routines to process modifiers and verbs, and control input/output operations.

The synthesis phase takes an incoming command, and using a system of weights and measures, rearranges the input into a Polish string. This places the elements in a sequence so that the next phase—analysis—can process the elements in a single left-to-right sweep over the string.

The analysis phase does the actual step-by-step decoding of the Polish string, finding the parameters when parameters are needed, and delivering those parameters to the appropriate routines when necessary.

The Polish notation was chosen because it allows simple, effective, and straightforward design: one pass, left-to-right, over the incoming command yields the Polish string; one pass, left-to-right, through the Polish string yields the results requested in the command.
Fig. 1. Overview of communication and command processing.

A further and important reason for choosing Polish notation is the fact that it facilitates modification or enlargement of the command language. This advantage is discussed in a later description of the Action Code Matrix—the focal point for control of synthesis.

**The Process of Synthesis**

In synthesis, a given command is introduced into the program as an EBCDIC character string, and rearranged to provide a *Polish string*. A Polish string, sometimes called Polish notation, is a method of representation that places the elements of an input string in the order in which they are to be employed rather than in the order in which they were introduced into the program.

While the source string is being rearranged to form the Polish string, use is made of two preliminary work areas. The first of the two is the *name string* into which each symbol or literal is placed as soon as it is encountered in the source string. The other is the *pushdown stack* which is charged with the function of actively participating in the rearrangement. As soon as an operator is encountered in the source string, it is compared with the last item placed in the pushdown stack, and as a result of the comparison, one of the following actions is taken.

The incoming source item may be placed at the top of the pushdown stack, thus becoming the most recent item, eligible to participate in the next comparison.

The incoming item may cause the most recent item to be removed from the pushdown stack and attached to the tail-end of the name string. This exposes the previous item in the pushdown stack, which becomes eligible to participate in the next comparison.

The incoming item may cause the most recent item in the pushdown stack to be copied into the name string, while the item remains in the pushdown stack.

The actions mentioned above are only the three most common actions which aid and abet the rearrangement.

In all, there are ten actions presently in use, and each action is assigned a number from 0 through 9. A grid is shown in Fig. 2 containing 13 rows and 13 columns, and filled with the numbers 0 through 9. This grid is called the *action code matrix*. At each intersection of a row and a column, there is a number that corresponds to one of the codes. The meaning of each *action code* is provided in Fig. 3.

The action code descriptions refer to **STEP** and **WORK**, defined below.

**STEP**: Get the next incoming source item, and decide whether it is an operator or an operand (name). Locate the horizontal row on which it resides.

**WORK**: Get the operator on the top of the pushdown stack, and locate the vertical column in which it resides. Where row and column meet, there is an action code. Perform the action indicated by the code.

The reader may have noticed terms *italicized* in the preceding paragraphs. Those are the terms which are of utmost importance in understanding synthesis. The reader would do well to keep in mind the terms, restated here.

**Polish string**: a string of input items arranged in the order in which they are to be processed. The string is the result of synthesis and is input to analysis.

**Name string**: a string of symbols and literals. (In later diagrams this string is built from left to right with the most recent name being added to the right-hand side.)

**Pushdown stack**: a stack of operators. (In later dia-
grams this stack is built from left to right, with items being added to and removed from the right-hand side.)

Action Code Matrix: the grid that shows all action codes. By selecting the intersection of a row and column, one can find out which action should be applied at any given time.

**STEP:** Get the next source item, and select a row.

**WORK:** get the last operator in the stack and find a column. Where row and column meet, find an action code and perform the indicated action.

Inspection of the Action Code Matrix will reveal weight assignments on the right-hand side and on the bottom (see Fig. 2). Names, as can be seen, have the lowest weight and verbs have the highest. The weights are primarily used for determining the order of operations in arithmetic and Boolean combinations.

The weight assignments are (from low to high):

- **Names**
- **Multiply/Divide**
- **Plus/Minus**
- **High/Low/Equal**
- **AND/OR**
- **Not**
- **Verbs**
- **End-of-Transmission**

At this point, let us take an example to illustrate the use of the Action Code Matrix and to show the effects of action codes. Referring to the definitions of **STEP** and **WORK**, we proceed.

**Example A:** Suppose that the input is:

\[ \text{DISPLAY } A + B \Delta \ (\Delta = \text{end-of-block character}) \]

The actions for each item in the command are as follows.

<table>
<thead>
<tr>
<th>Contents of Pushdown Stack</th>
<th>Source Item</th>
<th>Action Taken</th>
<th>Action Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Δ</td>
<td>DISPLAY Δ</td>
<td>placed on the pushdown stack</td>
<td>2</td>
</tr>
<tr>
<td>Δ</td>
<td>DISPLAY A</td>
<td>placed in the name string</td>
<td>1</td>
</tr>
<tr>
<td>Δ</td>
<td>DISPLAY +</td>
<td>placed on the pushdown stack</td>
<td>2</td>
</tr>
<tr>
<td>Δ</td>
<td>DISPLAY +</td>
<td>placed in the name string</td>
<td>1</td>
</tr>
<tr>
<td>Δ</td>
<td>DISPLAY Δ</td>
<td>+ is taken from the pushdown stack and added to the name string. (This is the beginning of the formation of the Polish string.)</td>
<td>4</td>
</tr>
<tr>
<td>Δ</td>
<td>DISPLAY</td>
<td>Δ is taken from the pushdown stack and added to the name string.</td>
<td>4</td>
</tr>
<tr>
<td>Δ</td>
<td>Seeing the initial Δ, the synthesis routines perform cleanup housekeeping</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
The Polish string at the end of synthesis contains:

\[ AB + \text{DISPLAY} \Delta. \]

In other examples in this section, we will use the following notation.

Capital letters at the top of the notation guide will indicate incoming source item.
Parentheses will be used to enclose the contents of the name string and the pushdown stack as successive items are processed.
Name string items will be in lower case italics in the upper tier within the parentheses.
Pushdown stack items will be in lower case letters in the lower tier within the parentheses.
Action codes will be shown beneath each parenthesized group of items.
Using this notation, try the same example shown above.

**Example B (using the notation):**

<table>
<thead>
<tr>
<th>INCOMING SOURCE ITEM</th>
<th>NAME STRING</th>
<th>PUSHDOWN STACK</th>
<th>ACTION CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>(name string)</td>
<td></td>
<td>(pushdown stack)</td>
<td>init EOB</td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DISPLAY A</td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(ab + display)</td>
<td>(ab + display)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The result is the Polish string: \( ab + \text{display} \).

This Polish string becomes input to the analysis phase. During analysis, the string is interpreted as follows:

find \( a \); find \( b \); get the sum; display the sum.

As another example, let us consider an error condition caused by the misuse of the offset modifier.

**Example C:** The input is:

\[ \text{DISPLAY} A. (2; 3). \]

The synthesis action is as follows until the error condition is discovered.

The error code 0 is found in the Action Code Matrix because a semicolon should never find a left parenthesis at the top of the pushdown stack. The error message SYNTAX ERROR is issued, synthesis is stopped, and the operator is invited to enter a different statement.

With these examples, we have attempted to illustrate the basic use of the action code matrix, the use of the notation, and the manner in which an action code can signal an error. In the discussion of analysis later in this paper, we will show analysis of the Polish string that was completed in Example B.

**A Prelude to Understanding Analysis**

Each verb in the language is processed by a routine. These routines need parameters upon which they can operate. Control blocks are used to describe the parameters so that a verb routine will know how to handle a field upon which it must operate. Among other things,

the control block for a parameter provides the information shown in Fig. 4.

The reader is reminded that the term operand stands for parameters such as symbols and literals; the term operator stands for verbs, modifiers, and punctuation.

**The Process of Analysis**

Analysis consists of processing a completed Polish string. Starting at the left side of the string and proceeding to the right, control blocks are built to describe each operand. Upon meeting an operator, the indicated operation is performed, using as many as or few control blocks as the operation requires.

(Code indicates an impermissible operation.)
Taking the familiar case of

\[ \text{DISPLAY } A + B \Delta \]

which became the Polish string

\[ ab + \text{display } \Delta \]

in Example A, we will now show analysis of the completed string.

*Example D:* The string is analyzed in one left-to-right pass as follows.

<table>
<thead>
<tr>
<th>Incoming Item</th>
<th>Action Taken</th>
<th>Preliminary Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>A control block is built.</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>A second control block is built.</td>
<td>a  _  _</td>
</tr>
<tr>
<td>+</td>
<td>The control blocks are used in computing the sum of the two symbols. The sum is placed in another control block.</td>
<td>(a  _  _  +) (a + b)</td>
</tr>
<tr>
<td>display</td>
<td>The sum is displayed at the terminal in integer format.</td>
<td>(a + b  _  _  _ display) +00000005</td>
</tr>
</tbody>
</table>

Notice that the notation used to illustrate analysis is similar to the notation used for synthesis. For analysis, all items are internal to the computer and are shown in lower case letters. The significant additions to the notation are:

- \( \square \) = a control block
- \( \bigcirc \) = the operation taking place.

To describe the analysis notation more fully:

\[ (\text{Notation Guide}) \]

\[
\begin{array}{c}
\text{Incoming Polish string item:} \\
\text{Operands } \square \text{ enter into operations } \bigcirc \text{ (upper tier) to yield} \\
\text{residual operands (on the lower tier)}
\end{array}
\]

\[ (a  \_  \_  +) (a + b) \]

To demonstrate the notation, we will take the example shown above and repeat it in a different form as Example E.

*Example E (using analysis notation):*

Incoming string: \( ab + \text{display} \Delta \).

The analysis:

\[
\begin{array}{c}
(a  \_  \_  +) (a + b  \_  \_  \_ \text{display}) +00000005 \bigcirc \end{array}
\]
We can say it in words as follows.

1) The operand $a$ is sent to a symbol resolution routine which returns a control block indicating that $a$ is a decimal 3. The control block is placed on the control block list.

2) The operand $b$ is sent to a symbol resolution routine which returns a control block indicating that $b$ is a decimal 2. The control block is added to the control block list.

3) The operator $+$ causes a link to be made to a plus-processing routine which needs two control blocks and operates on the two most recent control blocks on the list. Finding two control blocks, the routine computes the sum of $a$ and $b$ and produces one control block representing that sum.

4) The operator `DISPLAY` causes a link to the display routine which seeks to find one and only one control block and to print the value, formatted as requested, at the terminal. The control block is discarded.

5) The operator $\Delta$ causes execution of that portion of the analysis routines which checks accuracy by ensuring that there are no control blocks left over. The presence of a nonzero control block count denotes an error condition.

---

**Example F (showing order of operations):**

\[ \text{DISPLAY } A * B - 5 \Delta \]

The synthesis is:

\[
\begin{array}{cccccccc}
 & a & b & \circ & - & 5 & \text{display} & \Delta \\
\text{init} & 2 & 1 & 2 & 1 & 4 & 2 & \\
EOB & 5 & \Delta & & & & & \\
\end{array}
\]

The completed Polish string is: $ab*5 \text{-- display } \Delta$ (multiply $a$ times $b$; then subtract 5; display result).

The analysis is:

\[
\begin{array}{cccccccc}
 & a & b & \circ & 5 & \text{display} & \Delta \\
\text{init} & 2 & 1 & 2 & & & & \\
EOB & 5 & \Delta & & & & & \\
\end{array}
\]

Contrast the above with the following:

**DISPLAY $A*(B-5)$ $\Delta$**

\[
\begin{array}{cccccccc}
 & a & b & \circ & 5 & \text{display} & \Delta \\
\text{init} & 2 & 1 & 2 & & & & \\
EOB & 5 & \Delta & & & & & \\
\end{array}
\]

The reader will notice that in one left-to-right pass, the Polish string provided the analysis routine with one operand at a time to use as input to the appropriate resolution routine. A control block resulted from each such linkout.

Whenever the next item in the Polish string was an operator, the corresponding routine was called into action, receiving the complete control block string as a parameter string to use in the operation. Once the specified operation had been performed, the unused control blocks (if any) were returned.

The central idea of analysis has been demonstrated. Every routine needs parameters. As a matter of pure logistics, the language delivered to each routine the parameters that were needed and carried away the residual products.

Now that we know what synthesis and analysis are, let us consider some of the additional features of the language.

---

**DISPLAY $a \times b \& B = 5 \Delta$**

The synthesis is:

\[
\begin{array}{cccccccc}
 & a & b & \circ & - & 5 & \text{display} & \Delta \\
\text{init} & 2 & 1 & 2 & 1 & 4 & 2 & \\
EOB & 5 & \Delta & & & & & \\
\end{array}
\]

The completed Polish string is: $ab*5 \text{-- display } \Delta$ (multiply $a$ times $b$; then subtract 5; display result).

The analysis is:

\[
\begin{array}{cccccccc}
 & a & b & \circ & 5 & \text{display} & \Delta \\
\text{init} & 2 & 1 & 2 & & & & \\
EOB & 5 & \Delta & & & & & \\
\end{array}
\]

Contrast the above with the following:

**DISPLAY $A*(B-5)$ $\Delta$**

\[
\begin{array}{cccccccc}
 & a & b & \circ & - & 5 & \text{display} & \Delta \\
\text{init} & 2 & 1 & 2 & & & & \\
EOB & 5 & \Delta & & & & & \\
\end{array}
\]

The reader will notice that in one left-to-right pass, the Polish string provided the analysis routine with one operand at a time to use as input to the appropriate resolution routine. A control block resulted from each such linkout.

Whenever the next item in the Polish string was an operator, the corresponding routine was called into action, receiving the complete control block string as a parameter string to use in the operation. Once the specified operation had been performed, the unused control blocks (if any) were returned.

The central idea of analysis has been demonstrated. Every routine needs parameters. As a matter of pure logistics, the language delivered to each routine the parameters that were needed and carried away the residual products.

Now that we know what synthesis and analysis are, let us consider some of the additional features of the language.

---

**DISPLAY $A * B - 5 \Delta$**

The synthesis is:

\[
\begin{array}{cccccccc}
 & a & b & \circ & - & 5 & \text{display} & \Delta \\
\text{init} & 2 & 1 & 2 & 1 & 4 & 2 & \\
EOB & 5 & \Delta & & & & & \\
\end{array}
\]

The completed Polish string is: $ab*5 \text{-- display } \Delta$ (multiply $a$ times $b$; then subtract 5; display result).

The analysis is:

\[
\begin{array}{cccccccc}
 & a & b & \circ & 5 & \text{display} & \Delta \\
\text{init} & 2 & 1 & 2 & & & & \\
EOB & 5 & \Delta & & & & & \\
\end{array}
\]

Contrast the above with the following:

**DISPLAY $A*(B-5)$ $\Delta$**

\[
\begin{array}{cccccccc}
 & a & b & \circ & - & 5 & \text{display} & \Delta \\
\text{init} & 2 & 1 & 2 & & & & \\
EOB & 5 & \Delta & & & & & \\
\end{array}
\]
The Approach to Error Detection

We have attempted to explain some of the principles of an imperative command processing, done through the use of Polish notation.

A very important feature in the design was the decision to perform only minimal error-checking during synthesis. This allows almost any statement to go through synthesis unscathed. It is the job of the verb-processing and modifier-processing routines to do detailed error-checking.

\[
\text{DISPLAY } A + G \Delta
\]
\[
a \ g \ + \ \text{display } \Delta
\]

Before attempting to perform addition, the plus operator checks control blocks. Finding that \( G \) is undefined, the plus-processing routine notifies the user of the error. Allowing “imperfect” statements to be treated as “normal” permits maximum input to the verb- and modifier-processing routines, and thereby as much flexibility as possible. It is those verb- and modifier-processors that screen out improper number, type or form of incoming operands.

On Flexibility and Adaptability

The original design covered the immediate foreseeable objectives. It also left openings to handle the unforeseen under the same structure. Just what can be implemented under the existing system is limited only by the needs of the system and the ingenuity of the programmer.

If an unforeseen need arises to add a new verb, for example, it is simple. Why? Verb-processing routines need to be passed parameters and must be linked to at a timely moment. Analysis already does that. To enable synthesis to recognize a new modifier and to act on it, it is only necessary to change the Action Code Matrix or perhaps redefine a particular action code. For example, the addition of dual modifier capability such as:

\[
\text{IF } A = > B \text{ DISPLAY } A\Delta
\]

would cause slight impact because the = and > modifier-processing routines are already in existence. To provide for proper synthesis, the action for action code 9 would need to be expanded to cause a “look-ahead” to determine whether there was a second condition, and if so, to combine the conditions in the Polish string. For analysis, the only change would be to expand the high, low, and equal processors to handle the dual conditions and to return proper codes to control verb processing.

We could continue to provide examples, but suffice it to say that if a language designed to drive a system is based on sound well-defined principles, if the language is easily understood and used, if it is processed with a cohesive but open-ended technique, it can be adopted as is, adapted to other uses, and raised to cover situations yet unforeseen.

Acknowledgment

The author is grateful to R. L. Bean whose technical writing ability was invaluable in making this paper possible, to W. A. Bernstein whose knowledge of the field of programming languages made it possible to spot the feasibility of such an approach to our particular needs, and to J. Lukasiewicz who originally developed the idea which has become known as “Polish notation.” The author also wishes to thank J. T. Owens who recognized the ultimate capabilities of the language design as it related to our project as a whole, and who was instrumental in integrating the analysis portion into the total system, and R. B. Plank whose excellent research into the principles involved and examples of the application of those principles made rapid progress possible on the synthesis portion of the system.