Reprinted from JOURNAL OF THE ASSOCIATION FOR COMPUTING MACHINERY
Volume 10, Number 4, October 1963
Printed in U.S.A.

Experiments with a Heuristic Compiler*

HERBERT A. SIMON

Carnegie Institute of Technology, Pittsburgh, Pennsylvania

AND

The RAND Corporation, Santa Monica, California

Introduction

This report describes some experiments in constructing a compiler that makes use of heuristic problem-solving techniques such as those incorporated in the General Problem Solver (GPS) [1]. The experiments were aimed at the dual objectives of throwing light on some of the problems of constructing more powerful programming languages and compilers, and of testing whether the task of writing a computer program can be regarded as a "problem" in the sense in which that term is used in GPS. The present paper is concerned primarily with the second objective—with analyzing some of the problem-solving processes that are involved in writing computer programs. At the present stage of their development, no claims will be made for the heuristic programming procedures described here as practical approaches to the construction of compilers. Their interest lies in what they teach us about the nature of the programming task.

Theory of Problem Solving

The motivation for the compiler is supplied by a theory of problem solving that also provides the basic framework for GPS [1]. By a problem, a situation of the following kind is meant:

1. We are given a (partial) description of a *present situation* and a *desired situation*. These situations are described in a language that we may call the *state language*. The state lgnguage is sufficiently rich to permit us to describe situations (we shall call such descriptions *objects*) and to describe *differences* between pairs of situations.

2. We are given a list of *operators* which can be applied to situations to transform them into new situations. Operators are named in a language that we may call the *process language*. Any sequence of operators in the process language also is an operator—the compound operator corresponding to the application, in order, of the elementary operators belonging to the sequence.

3. A problem *solution* is a (compound) operator in the process language which will transform the object describing the present situation into the object describing the desired situation.

EXAMPLE. Take as the objects in the state language the integers, 1, 2, ... Take as the elementary operator the successor operation, which we shall desig-

* Received September, 1962; revised February, 1963.

nate as $'$ in the process language. Then $'''$ and $'''''$ are examples of compound operators. Consider the problem of transforming the present object 5 into the goal object 8. The solution is the operator $'''$, for $5''' = 8$. More generally, $'''$ is the operator that removes the difference $+3$ between any two objects, x and y; for if $y - x = +3$, then $x''' = y$. Here $+3$ is a difference, in the state language; $'''$ is the operator *relevant to that difference*, in the process language. We may construct a *table of connections* to associate with each difference the operator or operators relevant to it.

The distinction between state language and process language derives from the problem solver's dual relation with his environment. On the one hand, he perceives objects in the environment and represents them internally—in the state language. On the other hand, he acts upon the environment, and needs a language, the process language, to represent his actions. There may be more than one way of representing environmental objects—more than one state language. There also may be several process languages. In this paper we introduce several state languages in which programming problems may be expressed. (The most important of these will be called the "state description" language and "functional description" language, respectively.) Our process language will be a particular interpretive language, IPL-V.[1]

With this explication of the concept of "problem," many techniques of problem solving can be subsumed under the following general paradigm:

Means-End Analysis. Given the present and desired objects, find a difference between them. Next, find an operator relevant to the difference; determine if the operator can be applied to the present object. If so, apply it. (If not, describe the objects to which it would apply and transform the present object into an object of that kind—a new "desired object.") Take the new object thus obtained as the present object and repeat the process.

The General Problem Solver is a program which uses this scheme of means-end analysis to attempt the solution of any problem cast into the form described.[2]

Program Writing as Problem Solving

The task of proving a theorem can be formulated as a problem for GPS. The desired object is the theorem to be proved. The present object is the set of axioms and already-proved theorems. The operators are the legitimate processes for transforming a subset of axioms and/or theorems into a new theorem. We have a proof when we have a sequence of operators that transforms the present object into the desired object. (What we call a proof here is usually regarded as the *justification* for the proof steps; the proof as usually written out consists of the sequence of successive transformations of the axioms and given theorems.)

The sequence of operators constituting a proof can also be interpreted as a

[1] IPL-V is described only to the extent necessary for this exposition. A complete description will be found in [3].

[2] This is a bare-bone description of GPS, but it will suffice for present purposes. [1] gives a fuller description.

program which generates the desired object from the given object; for if we apply the operators of the proof, in sequence, to the present object (the axioms and previous theorems), we obtain precisely the desired object—the theorem to be proved. Thus a theorem-proving system can be regarded, at least formally, as a program-writing system. Conversely, if we can formulate a programming goal as a difference between a present and a desired object, we can presumably use the same processes, which in the other context will generate the proof of a theorem, to generate a program.

Outline of a Heuristic Compiler for IPL-V

In the remainder of this paper we describe a number of routines for compiling programs in Information Processing Language V (IPL-V), an interpretive list processing language. What is common to all of these compiling procedures is that they embody the problem-solving notions discussed in the preceding paragraphs. That is, each of the compiling routines accepts the task of writing programs in IPL-V on the basis of certain information provided to it. The task is accomplished by the application of the means-end analysis described. The several compiling routines differ with respect to their methods of formulating or representing the problem—that is, each operates with a different state language. At present, there are three compiling routines:

1. *State Description Compiler.* This routine takes as its input a description (state description) of the contents of the relevant computer cells before and after the routine to be compiled has been executed. It produces an IPL-V routine that will transform the input state description into the output state description.
2. *Functional Description Compiler.* This routine takes as its input a verbal definition (in the form of an imperative sentence) of the routine to be compiled. It produces an IPL-V routine that is the translation, in the interpretive language, of that definition.
3. *General Compiler.* This is an executive routine that can use the state description compiler, the functional description compiler, and others as subroutines. It takes as its input information about the routine to be compiled; the information can be stated in any one of several representations (e.g. those appropriate to either of the component compilers). The routine then selects subroutines that can use this information to produce the desired IPL-V code.

From a logical standpoint, we could describe the Heuristic Compiler as a single program whose executive routine is the General Compiler, and which contains the State Description Compiler and the Functional Description Compiler as subroutines. For clarity of exposition, it will be better to describe the two parts first as independent programs, and then show how they are imbedded in the General Compiler.

Some Characteristics of IPL-V

Before we begin, it will be useful to mention a few of the features of IPL-V that will be referred to in our discussion. In IPL-V, cells may have lists (pushdown lists) associated with them. The primitive processes of IPL-V find their operands in a *communication cell* and its pushdown list. We shall call this cell the "accumulator," because it has many of the functions of the accumulator in a standard computer. Processes (except tests) put their outputs in the accumulator and its pushdown list. Tests in IPL record their result by placing a PLUS or a MINUS in a special cell called the Signal Cell.

Lists in the IPL memory may have *description lists* associated with them. A description list is simply a list having a special format. It consists of pairs of symbols; the first symbol of each pair designating an attribute, the second symbol designating the value of that attribute. The value may be a simple symbol or it may itself be a list. Thus, we might have in memory a representation of a class of objects called "apples." The description list associated with this class might contain the attribute "color" with the value "red." Another attribute of "apple" could be "type," having the list of values "Winesap," "Delicious," and so on.

Values of attributes of objects may themselves have descriptions. Thus, in the compiler we shall have occasion to store representations of *routines* or programs in memory. These representations will take the form of description lists, each routine having one or more of the attributes, "IPL name," "IPL-V definition," "Functional Description," "State Description," and "Flow Diagram." The values of these attributes will themselves be described—will have description lists associated with them. Thus, for example, the state description of a routine will be given by a description list having the attribute "list of affected cells." The value of that attribute will be a list, each item of which will again have a description list associated with it.

State Description Compiler

A computer routine can be defined by specifying the changes it produces in the contents of the storage locations it affects, or, what amounts to almost the same thing, by specifying the before-and-after conditions of these storage locations. A definition of this kind is not, of course, univocal, for programming is a synthetic, not an analytic task; generally there will be many programs (not all equally efficient or elegant) that will do the same work. As presently constituted, the State Description Compiler attempts to find one routine to accomplish a given task.

EXAMPLE. In IPL-V there is a process, "Put symbol MINUS in Signal Cell," which affects a single memory location, the Signal Cell. This process has the following state description: *before* the process is executed, the Signal Cell contains a symbol, call it SYMB1, followed by an indeterminate list of symbols, PUSHDOWN1 (call this the pushdown list associated with the Signal Cell); *after* the process has been executed, the Signal Cell contains the symbol MINUS followed by the same list of symbols PUSHDOWN1, as before. The token of symbol SYMB1 previously in the Signal Cell has been destroyed.

Notice that it is implicit in this definition of SET SIGNAL MINUS that the content of no cell other than the Signal Cell has been altered by the routine. We can depict the state description diagrammatically as follows:

State Description of SET SIGNAL MINUS

Affected Cells:	Signal Cell
Input	SYMB1, PUSHDOWN1
Output	MINUS, PUSHDOWN1

Generalizing, the state description of a routine consists of a *list of affected cells*. For each affected cell on the list, the state description specifies its *input state* and its *output state*.

To compile the IPL-V code for SET SIGNAL MINUS, the Compiler proceeds as follows:

1. It matches the input states with the output states of the affected cells until it *finds a difference*. In the example cited, the difference between the input and output states of the Signal Cell may be called a *replacement in the Signal Cell.*

2. It searches a *table of connections* which associates with each difference a list of *operators* (compiled IPL-V routines) relevant to that difference. In the example, the table of connections contains, associated with the replacement difference, the IPL-V routine REPLACE [(CELL)] BY (ACCUMULATOR).[3]

REPLACE [(CELL1)] BY (ACCUMULATOR) replaces the symbol in cell CELL1, a variable, with the symbol in the accumulator. Thus, the "Replace" process has the following state description:

State Description of REPLACE [(CELL1)] BY (ACCUMULATOR)

Affected Cells:	Accumulator	Cell1
Input	SYMB2, PUSHDOWN1	SYMB1, PUSHDOWN2
Output	PUSHDOWN1	SYMB2, PUSHDOWN2

3. It tentatively applies the relevant operator it has found to the input state of the state description to be compiled, and determines the resulting output state. In applying the operator, it makes appropriate substitutions for the variables in the operator. Thus, applying REPLACE[(CELL1)] to the input of SET SIGNAL MINUS, we find, by matching, that we should set CELL1 = SIGNAL and SYMB2 = MINUS, giving:

State Description of REPLACE (SIGNAL) BY (ACCUMULATOR)

Affected Cells:	Accumulator	Signal
Input	MINUS, PUSHDOWN1	SYMB1, PUSHDOWN2
Output	PUSHDOWN1	MINUS, PUSHDOWN2

4. The application of the operator creates two new subproblems: Let I_a represent the input state of the routine to be compiled, O_a its output state, I_b the input state of the operator, and O_b its output state. The original problem was to transform I_a into O_a . The new problems are: (1) to transform I_a into I_b (i.e. to establish the input conditions for application of the operator), and (2) to transform O_b into O_a (i.e. to transform the output state of the operator into the de-

[3] We adopt the usual convention that parentheses mean "the contents of."

sired output state of the routine to be compiled). Either of these new problems may reduce to the identity transformation, in which case that part of the problem is solved. If this reduction does not occur, then the same steps 1, 2, 3 are applied to the new subproblem.

In the example at hand O_b is identical with O_a ; hence the remaining subproblem is to transform I_a into I_b , that is, to compile a routine with state description:

	Accumulator
Input	PUSHDOWN1
Output	MINUS, PUSHDOWN1

The repetition of step 1 for this subproblem discovers a new difference, an addition to the contents of the accumulator. Step 2 finds the relevant operator, LOAD [S] INTO ACCUMULATOR, which adds to the symbol list in the accumulator the symbol s. Applying, in step 3, the operator LOAD [MINUS], the input state of the accumulator is transformed into the desired output state. Hence the solution to the original problem of compiling SET SIGNAL MINUS is obtained by the sequence LOAD MINUS, REPLACE (SIGNAL) or, in the usual IPL-V format:[4]

Set signal minus	J3	10J3		Load MINUS
		20H5	0.	Replace (SIGNAL), Terminate

We see that for the state description compiler to operate, it must be provided with a set of differences and matching tests for noticing differences, a set of already-compiled operators, and a table of connections between differences and operators. Further, when it has compiled a new routine, the compiler can annex this routine to its set of available operators and use it in compiling subsequent routines.

Functional Description Compiler

Let us now consider an alternative compiling scheme for the same routine, SET SIGNAL MINUS. Instead of specifying the before-and-after condition of the computer cells, we define the routine in terms of the function it performs: "Replace the symbol in the Signal Cell by MINUS". This definition (functional description) resembles more closely than the previous one the manner in which routines are defined for conventional compilers like FORTRAN or LISP. What distinguishes the present scheme from these is the use of heuristic means-end analysis for working from the definition to the compiled routine.

The first step in the Functional Description Compiler is to search a list of available (compiled) routines to find one whose functional description is as similar as possible to the functional description of the routine to be compiled. In the case at hand, we would find the routine REPLACE (CELL1): "Replace the symbol in CELL1 by the top symbol of the pushdown list in the accumulator."

[4] The standard IPL-V notation for the routine is shown in the center, with its translation in the left- and right-hand columns. Thus J3 is the IPL name for SET SIGNAL MINUS and also for the symbol MINUS. 10 means LOAD, 20 means REPLACE. H5 is the name for SIGNAL, 0 for TERMINATE.

At the second step, means-end analysis is performed to transform the compiled routine into the new routine. The transformations are performed on the functional descriptions. Thus, in the present example there are two differences between REPLACE [(CELL1)] BY (ACCUMULATOR) and REPLACE (SIGNAL) BY MINUS. The former refers to the cell, CELL1, the latter to the Signal Cell; the former refers to the symbol that is contained in the accumulator; the latter to the symbol MINUS.

The compiler notices these differences (in a sequence), and searches for an operator relevant to removing the differences. In this case, CELL1 can be transformed to SIGNAL by a *substitution* operator. (ACCUMULATOR) can be changed to MINUS by an *addition* operator ("Make (ACCUMULATOR) equal to MINUS by addition"). The application of these operators to the functional description of REPLACE [(CELL1)] would compile the desired routine in the following stages:

REPLACE [(CELL1)]	Replace the symbol in CELL1 by (ACCUMULATOR)
Apply *substitution*	Replace the symbol in SIGNAL by (ACCUMULATOR)
Apply *addition*	Replace the symbol in ACCUMULATOR by MINUS

The resulting program in this case is identical with that obtained by the State Description Compiler.

A somewhat more complex routine compiled by the Functional Description Compiler is[5] INSERT 1ACCUMULATOR AT THE END OF (THE VALUE OF ATTRIBUTE 0ACCUMULATOR OF 2ACCUMULATOR). The list of available IPL routines includes INSERT 0ACCUMULATOR AT THE END OF 1ACCUMULATOR.

The differences between these two functional descriptions are in their arguments. The latter has the argument 0ACCUMULATOR where the former has the argument 1ACCUMULATOR; the latter has the argument 1ACCUMULATOR where the former has the argument THE VALUE OF ATTRIBUTE 0ACCUMULATOR OF 2ACCUMULATOR. Since it is not easy in IPL-V to rearrange arguments located in the pushdown list of the accumulator, the compiler facilitates matters by incorporating in the compiled routine an algorithm that moves the inputs of the routine to be compiled into known working storage locations, then puts these inputs back into the accumulator pushdown list in the order in which they are needed for the subprocesses. That is, the compiler first transforms INSERT AT END OF VALUE LIST into another routine, which it then compiles. The functional description of this intermediary routine[6] is INSERT 1WORKING AT THE END OF THE VALUE OF ATTRIBUTE 0WORKING OF 2WORKING. The code for INSERT AT END OF VALUE LIST may be written as:

```
Insert at end of value list:
                         J13    J52              Put symbols in working storage
                                K13              Execute intermediary routine
                                J32        0.    Restore working storage, Ter-
                                                   minate
```

[5] We shall use, from now on, the following abbreviations: 0ACCUMULATOR is the symbol in the accumulator, 1ACCUMULATOR is the first symbol of the push-down list of the accumulator, 2ACCUMULATOR the second symbol, and so on. In IPL-V, the operands for processes are held in the accumulator and its push-down list.

[6] 0WORKING, 1WORKING, etc., are abbreviations for the contents of a set of working cells available in IPL-V.

Now the intermediary routine is to be compiled with the aid of INSERT AT END OF LIST. Comparing the corresponding arguments of the two routines, we see that this involves finding the value of attribute 0WORKING of 2WORKING, placing the value in the accumulator, bringing 1WORKING into the accumulator, and then performing INSERT AT END OF LIST. That is to say the intermediary routine will have the general form:

```
Insert at end of value list:
                    K13     Find V(0W, 2W)   Find value list of attribute
                                               0WORKING OF 2WORKING
                            11W1             Add 1WORKING
                             J65             Insert 0ACCUMULATOR AT END OF
                                               1ACCUMULATOR
```

In the list of available routines, the compiler finds FIND THE VALUE OF ATTRIBUTE 0ACCUMULATOR OF 1ACCUMULATOR, which may be abbreviated, "Find V(0A, 1A)". Comparing its arguments with those of V(0W, 2W), we see that 1ACCUMULATOR must be set equal to 2WORKING and 0ACCUMULATOR to 0WORKING. Hence V(0W, 2W) is equivalent to

```
                            11W2             Add 2WORKING
                            11W0             Add 0WORKING
                             J10             Find value of 0ACCUMULATOR of
                                               1ACCUMULATOR
```

Hence, the complete code for the intermediary routine is

```
                            11W2             Add 2W
                            11W0             Add 0W
                             J10             Find value of 0A of 1A
                            11W1             Add 1W
                             J65             Insert 0A at end of 1A
```

and the complete code for the desired routine[7] is

```
Insert 0A at end of value list of 1A of 2A:
                    J13      J52             Put symbols in working storage
                            11W2             Add 2W
                            11W0             Add W0
                             J10             Find value of 0A of 1A
                            11W1             Add 1W
                             J65             Insert 0A at end of 1A
                             J32        0.   Restore working storage, Ter-
                                               minate
```

General Compiler

The General Compiler is an executive routine whose task is to compile a routine from information in any of the forms already discussed (state description or

[7] Readers familiar with IPL-V will see that we are simplifying for purposes of illustration. The routine as written does not take care of the case where the attribute value in question does not exist.

functional description) or in other forms that may be described. It takes as its input the internal name of the routine to be compiled. Associated with this routine (on its description list) is the information to be used in the compilation. More formally:

A *routine* is a description list containing values of some subset of the following attributes:

1. *IPL name.* The value of this attribute is a description list naming a region and a location in the region—e.g. J60, R149, J3.
2. *IPL-V definition.* The value of this attribute is a list of IPL-V instructions, each in the form of a description list describing the corresponding IPL-V word, defining an IPL-V routine with the specified name. For example, the routine with IPL name J3 might have the following IPL-V definition:

```
J3    10J3
      20H5    0.
```

3. *Functional description.* The value of this attribute is an imperative sentence (encoded as a list structure) describing the process defined by the IPL-V definition. For example, the routine with name J3 has, as already explained, the functional description: REPLACE THE SYMBOL IN SIGNAL BY MINUS.
4. *State description.* The value of this attribute is a list structure describing the state of the IPL computer before and after the routine in question has been executed. Only changes are mentioned explicitly. Thus the state description of J3, SET SIGNAL MINUS, is: Affected cell, SIGNAL; Input, SYMB1, PUSHDOWN1; Output, MINUS, PUSHDOWN1.
5. *Flow diagram.* The value of this attribute is a list structure giving the flow diagram corresponding to the IPL-V definition. This list structure will be described in more detail later.

A compiled routine is a routine having an IPL-V definition. Now we can state the problem of compiling a routine as follows: given a routine without a definition (the present object), find the corresponding routine with a definition (the goal object). "Corresponding" means that the compiled routine has the same state description or functional description as the given routine. Figure 1 presents the flow diagram of a compiler using means-end analysis to accomplish this compilation.

Let us translate this flow diagram into the language of means-end analysis.

1. *Test whether the routine has an IPL-V definition.* This test determines whether the present object has the characteristics of the desired object. If so, the compilation is complete.

2. *Find the closest definition.* This process corresponds to finding a difference between the present and desired objects. However, we generalize this notion to mean: look for a *characteristic* of the present object that will suggest a relevant operator. If the object possesses a functional description, then an attempt could be made to compile the IPL-V definition from the functional description; if it possesses a state description, an attempt could be made to compile the IPL-V definition from the state description. The attributes the routine could possess are listed in an order reflecting the relative ease of compiling an IPL-V definition

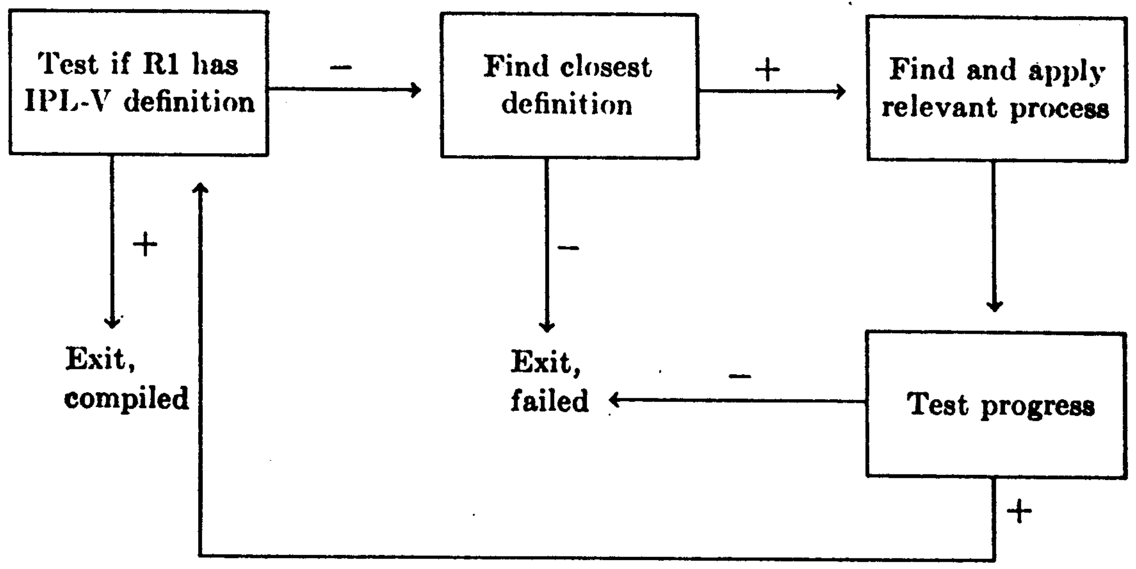

FIG. 1. Flow diagram for Compile Routine R1

from them. The process then finds the first attribute on this list possessed by the routine to be compiled. In the present form of the compiler, it is assumed that it is easier to compile from a functional description than from a state description; hence the attributes are listed in this order. If the routine possesses no attribute that could be used as a basis for compilation, the compiler reports a failure.

3. *Find and apply the relevant process.* The input to this process is the "closest definition" just found. A table of connections is searched to find a process relevant to compiling the IPL-V definition from the closest definition. If one is found, it is applied (in a manner to be described later).

4. *Test progress.* If the operator has been applied successfully, the routine will now possess at least one attribute (an IPL-V definition or another) not previously possessed. If the progress test detects that it now has a definition closer to the IPL-V definition than any it had previously, it initiates a new compilation cycle; if not, it reports a failure and quits.

The present list of "closest definitions" is very short, consisting only of the functional description and state description. The present table of connections is also brief:

1. If the routine possesses a functional description, apply the operator, COMPILE IPL-V DEFINITION FROM FUNCTIONAL DESCRIPTION.
2. If the routine possesses only a state description, apply the operator, COMPILE IPL-V DEFINITION FROM STATE DESCRIPTION.

Relation of the Heuristic Compiler to the General Problem Solver

Since each of the major components of the Heuristic Compiler is a system of means-end analysis, each of these components can be viewed as a rudimentary General Problem Solver. It should therefore be feasible, by modifying the top level programs, to bring the Heuristic Compiler into a form which would allow its problem-solving processes to be governed by GPS. The programs for detecting differences, the tables of connections, and the operators would provide definitions of task environments for GPS. To accomplish this, GPS would have to be arranged so that a subproblem could involve applying GPS to a new task environment. That is, GPS would first be applied to the task environment of the General Compiler; applying an operator in this environment would consist in applying GPS to the task environment of the functional descriptions or the state descriptions, as the case may be.

Flow Diagrams

Up to this point we have considered only very simple programs requiring no branches or loops. Each program is a list of instructions; each instruction, an IPL word represented as a description list with these attributes: type, name, sign, P, Q, symbol, and link.

To represent a program with branches and loops, we divide the program into segments. Each entry point to a loop (an instruction with a local name) begins a new segment; each branch instruction (branches are indicated by P=7) ends a segment. Each segment has the same attributes as an IPL word—specifically: name, P, symbol, link, and an additional attribute, IPL-V definition, whose value is the list of IPL instructions for the segment. The name of the first instruction of the segment is assigned as the name of the segment; if the segment ends in a branch instruction, it is assigned P=7, and its symbol and link are set equal to the symbol and link of the branch instruction. If the segment does not end in a branch, it is assigned P = 0 and SYMB = 0, and its LINK is set equal to the link of its last instruction. Under these conventions, the list of segments is a flow diagram of the routine with the detail of the routine segments appended.

To illustrate the format of a flow diagram, we show below the code for the IPL routine named J77 followed by its flow diagram. The functional description of J77 is: TEST WHETHER THERE IS A SYMBOL EQUAL TO 0ACCUMULATOR ON LIST 1ACCUMULATOR.

IPL-V CODE FOR J77

```
  Name       P Q Symb     Link
---------------------------------
  J77            J50       90     Segment I: Put 0ACCUM in 0WORKING
---------------------------------
  90             J60              Segment II: Find next location on list 1ACCUM;
             7 0 91        92       if end of list, go to 91
---------------------------------
  92         1 2 H0               Segment III: Test if symbol at location is
             1 1 W0                 equal to 0ACCUM. If equal, go to 91; if not,
                 J2                 go to 90
             7 0 90        91
---------------------------------
  91         3 0 H0               Segment IV: Clean up and exit
                 J30       0
---------------------------------
```

FLOW DIAGRAM FOR J77

Name	*P*	*Symb*	*Link*
J77	0		90
90	7	91	92
92	7	90	91
91	0		0

From the description of the flow diagram, it is easy to provide a program that will construct a flow diagram from an IPL routine, and a program that will compile an IPL routine from the flow diagram and appended code segments. In this way the task of compiling an IPL routine is reduced to the problem of compiling its flow diagram and compiling the code for each of the segments of the flow diagram.

The program for compiling such a routine from its functional description has has not yet been written, but examination of the structure of the routine itself shows what is involved. The test involves a quantifier—whether *there exists* on a particular list of symbols a symbol having a certain property. In IPL-V, such existence tests are performed by means of a loop or a generator; the members of the set in question are produced one by one and tested for their possession of the property. If a test result is positive, the process stops and the symbol PLUS is stored in the signal cell. If the set is exhausted, the symbol MINUS is stored in the signal cell. Thus a standard flow diagram can be used for all routines of this kind:

```
A  Perform required setup
B  Locate another member of set
     (If none, exit, via D)
C  Perform test on member
     (If it succeeds, exit via E;
     if it fails, return to B)
D  Exit with signal minus
E  Exit with signal plus
```

Except for the provision of two distinct exits, this is identical with the flow diagram previously shown for J77 (Set A = J77, B = 90, C = 92, D = E = 91). Now, we can compile for each segment of the flow diagram a routine that corresponds to the functional description of that segment. For example, FIND ANOTHER MEMBER OF 1ACCUMULATOR becomes LOCATE NEXT, J60, (after appropriate recognition of the changed location of 1ACCUMULATOR); PERFORM TEST ON MEMBER becomes:

```
12H0
11W0
 J2          Test 0ACCUM and 1ACCUM for
               equality
```

The only complications lie in moving the inputs for the various processes (J60, LOCATE NEXT, and J2, TEST IF EQUAL) in appropriate ways. The compiler can do this in a straightforward, if inefficient, way by using the working storages. Thus, an unedited compiled version of J77 might look like this:

```
Test if 0A is on 1A     J77     J51    90   Put inputs in working storage
Locate next on 1A       90      11W1        Bring in 1W (list location)
                                 J60        Locate next
                                20W1        Store location again
If none, exit                   7091   92
Compare next symbol
  with 0A               92      12W1        Bring in located symbol [(1W)]
                                11W0        Bring in comparison symbol, 0W
                                 J2         Test for equality
Branch on test                  7090   93
                        91       J31   0    Clean up working storage and termi-
                        93       J30   0.     nate
```

The same flow diagram would be used in the compilation of LOCATE ON 1ACCULATOR AN X SUCH THAT (X) = 0ACCUMULATOR. In fact, this routine is identical with the one just discussed, except that it requires 11W1 (LOAD 1WORKING INTO 0ACCUMULATOR) before the exit. It should be observed that the indefinite article, "an," plays the same role in the functional description of this routine as the quantifier, "there is a," in the previous one. The compiler, therefore, would be provided with the knowledge that the above flow diagram, using J60 (LOCATE NEXT SYMBOL) in the second segment, is the appropriate means for translating this quantifier.

Declarative and interrogative sentences in a functional description correspond to tests in the compiled routine. Thus, the phrase "such that SYMB1 = (ACCUMULATOR)" leads to the question, "Does SYMB1 equal (ACCUMULATOR)?" and thence to the test J2[SYMB1, (ACCUMULATOR)].

Summary

The experiments described in this paper demonstrate that compiling tasks—at least simple compiling tasks—can fruitfully be viewed as problem-solving tasks, and can be performed by a program having the general organization and capabilities of the General Problem Solver.[8]

The explorations have followed two main lines. The state description compiler illustrates how the state of a computer before and after the execution of a process can be described, and how this description can be used to define and solve a compiling problem. The compiler would require substantial further development before it could be used as a practical compiling device, but, besides providing guideposts for such development, it casts light on two important topics: (1) the nature of the problem-solving processes involved in programming—particularly

[8] Some further discussion of language and representations in the Heuristic Compiler, together with a listing of the program as of June 1961, will be found in [2]. The discussion of language and representations has been omitted here because another report will be made on these topics, on the basis of more recent investigations.

machine-language programming; (2) the use of description lists in IPL-V as an expository language having expressive capabilities not readily available in more usual computer command languages. In the state description compiler the declarative mode, rather than the imperative mode, bears the main burden of information storage and communication.

The functional description compiler follows more traditional lines. Apart from its heuristic organization, it resembles closely compilers like FORTRAN, ALGOL, and LISP. Its language is primarily a language of imperatives, of processes. The functional description compiler illustrates how means-end analysis, like that employed in the General Problem Solver, can be used to translate requests for general, functionally-defined processes into programs. Like the state description compiler, it gives us a great deal of information about the problem-solving processes required for programming. Further experiments, to be reported subsequently, suggest that it can be a tool in exploring the relation between problem statements in natural language and in formalized computer languages.

The investigation described here stimulated a certain amount of introspection about my own programming processes and inquiries about the processes of my fellow programmers. From these informal inquiries I am persuaded that programmers employ, in their problem-solving processes, both a language like that of the state description compiler and a language like that of the functional description compiler. The former becomes essential in designating machine representations and in writing programs depending on the detail of machine representation—processes of packing and unpacking words that are intimately dependent on word structure. On the other hand, at the higher levels often represented by flow diagrams, a language of functions appears generally the more natural and powerful. In general, the mappings from functional descriptions to process language are simpler and more direct than the mappings from state descriptions—a possible explanation of why most existing compilers use a functional source language. The Heuristic Compiler is an exploratory tool that is beginning to reveal to us something of the role played by these and other representations in the programming process.

ACKNOWLEDGMENT. The work described here is part of the author's research on complex information processing. He is indebted to his colleagues at The RAND Corporation for discussion and criticism of the ideas presented here, but particularly to H. S. Kelly, who developed some linguistic components for the compiler which are omitted from this report.

REFERENCES

1. NEWELL, A., SHAW, J. C., AND SIMON, H. A. Report on a general problem-solving program for a computer. *Information Processing*: Proc. Internat. Conf. Inform. Processing, pp. 256–264. Paris: UNESCO, 1960.
2. SIMON, H. A. Experiments with a heuristic compiler. The RAND Corp., Report P-2349, June 30, 1961.
3. NEWELL, A. (Ed.) *Information Processing Language-V Manual*. Prentice-Hall, 1961.