INTRODUCTION

AMBUSH is a language in which the user can describe a materials-processing/transportation network in an allegedly readable form. The AMBUSH Compiler transforms this description into a set of linear equations and inequalities and edits this set into an input file for a linear programming package.

Although the language is powerful enough to handle both materials-processing and transportation, its design is heavily biased toward processing, with transportation expected to appear only occasionally, around the “edges” of the network. Thus, a typical subject for AMBUSH description might be a very complex oil refinery with a few arcs representing transport of crude oils into and final products out of the refinery.

AMBUSH was designed by Applied Data Research for the Shell Oil Company, to facilitate the preparation of very large linear programming problems. This project followed a long history of experimental language design by Shell and its contractors, which had yielded a very valuable crop: Shell was able to summarize, fairly confidently and economically, the set of meanings to be expressed. The present language is the result of collusion (and sometimes collision) between Shell’s body of experience and ADR’s background in language and compiler design.

AMBUSH, as actually designed and implemented, represents a conscious compromise between two styles of language design: the “template” style, with a large set of different statement types, corresponding roughly one-to-one to the kinds of utterances the user is accustomed to; and the recursive style, in which the designer hunts for a minimal number of logical primitives sufficient to express all meanings and devises the shortest (in number of types) and most highly recursive grammar possible for combining the primitives. This stylistic tension and, indeed, the historical sequence from pure template languages toward more recursive ones exhibited in the AMBUSH project are, we believe, common characteristics in the development of languages for new problem areas.

Whenever any body of informal discourse about a subject is analyzed in order to create a formal language for saying the same things, the primary effort is concentrated on verifying that the formal language under creation is complete—that one can express every meaning—with little concern for ease of use or its close relative, grammatical simplicity. The first requirement is met by letting the language arise from, and comparing it to, a large body of utterances in the informal discourse. The loom of this body of utterances generally causes the second criterion—essentially that of linguistic “goodness”—to be quickly reinterpreted to mean superficial similarity to the informal language. Forms in the slowly developing formal language are adjudged good insofar as they “look like” the corresponding forms in the informal one. This perfectly natural criterion generally leads to first-cut designs which, although very valuable (for they effect the desired formalization), usually exhibit certain characteristic properties:

a. A syntax which is “broad” rather than “deep.”
   By this we mean that they tend to have a large variety of different, special syntactic forms—each aping a familiar English one—rather than a small number of forms and a small set of rules for recursively combining them.

b. A proliferation of reserved identifiers, each of which once again apes a commonly used word in the informal language, even when several of these identifiers may be logically identical in function.

c. Extremely rigid format requirements, which mechanically copy the forms in which informal information is currently expressed.

These properties tend to characterize languages which, however easy to read, are very difficult to learn to write in, since each type of utterance has its own
special construction rules and its own reserved identifiers; difficult to compile, since the grammars are so broad; and difficult to extend, since the grammars are so non-uniform and format-controlled.

It has been our general experience that a first-cut language design should be viewed as a formal characterization of the class of required utterances, rather than as an end product. What is required next, of course, is a redesign embodying deliberate infusion of the recursive style. The ever-present danger is the obvious one of going too far: users are willing to change their habits only so much, and a language which demands a complete reorganization of their styles of thought and expression sets an unacceptably high price, for perfectly understandable reasons.

In this paper we do eventually present a skeletal grammar of the AMBUSH language, as it was actually implemented. Our primary purpose, however, is not to describe a language. Rather we intend to treat the AMBUSH project as a case history of the language design process, which may give substance to our general remarks about the shift from template to recursive style.

We begin by summarizing how an engineer thinks about his model of an oil refinery: the appearance of his graph, the kinds of assertions he makes about flow in the graph, the packages of information he thinks of as single elements of model description. We then describe, in general terms, the structure of languages in the template style for expressing the engineer's meanings. Next we reexamine the engineer's utterances from the point of view of a designer in the recursive style, concerned with logical minimality and searching for opportunities to shorten the grammar by the use of recursion. In this last section, the general outline of the final grammar of AMBUSH takes shape. Then we present that grammar, so that the reader can verify our claim that AMBUSH is indeed a compromise: a language—basically in the recursive style, but with elements of the template style still remaining, to render it more familiar-looking, and thus more palatable, to the user.

HOW THE ENGINEER DESCRIBES HIS MODEL

The directed graph

An engineer conceives of his model as a directed graph (with no self-loops), with labels on the nodes and arcs. [The bracketed examples in this paragraph refer to the figure in Appendix I.] The nodes of the graph are labeled with the names of the processing units of a plant or refinery, or the staging points of a transport-network. The arcs of the graph represent the paths of flow of the various materials in the model. Each arc is a path for exactly one material and is labeled with the name of that material ["Exotic Liquid," "Unexciting Prod."]; if several materials flow from one node to another, there will be several "parallel" arcs (arcs with a common head node and a common tail node) between the nodes [Universal Solvent and Residue both flow from ICE.PLANT to BLEND.UNEX]. Node names are unique (no two nodes bear the same name), while arc names are not (several arcs may bear the same name: the same material may flow in numerous arcs of the graph [Residue flows from ICE.PLANT to BLEND.UNEX, to TRUCK, and to DEPOT1, and from DEPOT1 to DEPOT2]). Parallel arcs must have different labels: that is, there can be but one arc carrying a given material from a given node to a given other node.

Thus, an arc of the graph is completely specified by an ordered triple of names: (node name, arc name, node name).

The variables of the model correspond to the arcs of the graph or, more precisely, to the flow along the arcs. That is, the engineer states his restrictions on the model and his costs in terms of flows on arcs, and seeks an optimal solution stated in terms of flows on arcs.

The arcs of the graph will become the columns of a conventional LP matrix; the restrictions on the model will become rows of the matrix; and the various costs incurred in the model will become the objective function(s) of the matrix.

Restrictions on flow in the graph

Material balance

In general, at every node of the graph which possesses both inputs and outputs, the engineer wishes to relate the various flows entering to the flows leaving the node. These relations are conventionally expressed, in the LP matrix, as a set of "material balance equations" (rows). Each equation expresses the total amount of a single material flowing out of a node as a linear expression in the flows of the materials entering the node. The coefficients of such an expression are called "yield factors," and the engineer thinks in terms of these yield factors. It is important to note that a single material balance equation, while a natural element (a single row) of the LP matrix, does not necessarily correspond to a natural single utterance for the engineer. He may, for example, refer to a handbook of standard processing units which contains an array (inputs vs.
outputs) of yield factors, each of whose columns corresponds to a single equation; or an array of outputs vs. inputs, with rows corresponding to equations; here the entire array corresponds to a single utterance for the engineer. On the other hand, the engineer may assemble his description of material balance at a node from several sources, each of which supplies a few yield factors—each factor associated with an input/output pair—in no particular order. In sum, the engineer's natural utterance is an arbitrary subset of the yield factors at a single node, or a complete array of these yield factors organized in either of two ways.

**Quantity restrictions**

The engineer may set a limit (maximum, minimum, or fixed amount) on the total flow through a set of arcs. Each such restriction becomes a row of the LP matrix. It is noteworthy that the set of arcs in a single restriction is not at all arbitrarily chosen, but is always either a subset of the arcs entering a single node or a subset of those leaving a single node. This limitation imposes no logical restriction on the engineer, for he can always introduce further arcs and nodes so as to create a set of input arcs to a single node (or even a single arc) whose flow will be precisely equal to the sum of the flows in an arbitrary set of arcs. The point here is not that a certain kind of arc set is logically sufficient, but rather that it is the only kind that the engineer in fact uses as a domain for quantity restrictions.

**“Quality” restrictions**

The engineer may set a limit on the value of some physical property for the mixture of the flows on several arcs. For example, three arcs might carry three different materials each with a certain specific gravity. The engineer wishes to restrict the relative flows on the three arcs so that the specific gravity of the total flow satisfies some limit (maximum, minimum, or fixed value). If we denote the flows in the different arcs by \( f_i \) and the respective specific gravities by \( g_i \), the engineer is setting a limit on the value of \( \frac{\sum_i f_i g_i}{\sum_i f_i} \). Once again, as with the quantity restrictions, the set of arcs is always either a subset of the arcs entering, or a subset of the arcs leaving, a single node.

**Ratio restrictions**

The engineer may set a limit on the ratio of the total flow in one set of arcs to the total flow in another set of arcs. Again, it is a feature of user behavior that the union of the two sets of arcs referenced in a single ratio restriction is always a subset of the arcs entering or a subset of the arcs leaving a single node.

**Cost and income**

The engineer may wish to state that flow along certain arcs incurs cost or generates income. Eventually, all his remarks of this kind will become the objective function of the LP problem, a linear expression in flows on arcs which is to be minimized. The engineer's typical utterance does not correspond to an entire objective function, but rather to the supply of a single coefficient value for the objective function; such a coefficient value may apply to one arc or to a set of arcs: thus, the engineer may say, in effect, "flow on any of this set of arcs costs $3.50 per unit flow." It is no serious distortion of user behavior to say that the set of arcs to which a single cost/income coefficient applies is always either a subset of the arcs entering, or a subset of the arcs leaving, a single node.

**TEMPLATE-STYLE LANGUAGES**

Even from the rather brief picture given above of the engineer's view of his model, it should be clear that he has a fairly large number of different—or what he believes are different—kinds of things to say. He must be able to describe a graph, supply a set of yield factors, supply an array of yield factors in input/output form, supply the array in output/input form, state quantity restrictions on the output side of a node, or the input side, assign cost to an arc, assign income to an arc, and so forth. It is important to note that, although the difference between some two elements of this list may seem quite trivial to the reader (input/output form versus output/input form of an array of yield factors, for example), the difference may look enormous to the engineer, who obtains the two kinds of information from different sources, uses them in different modeling contexts, and discusses them in a different vocabulary. In our effort to summarize the engineer's utterances concisely, we have made use of logical similarities between utterances which might, to him, look quite different. Thus, in our description of quantity restrictions, we took advantage of the fact that to set a quantity maximum is not unlike setting a quantity minimum. In some real problem, however, all the maxima may come from physical limitations (pipeline capacities, say), while all the minima may reflect business obligations (requirements to buy at least so much from various suppliers). The engineer in this
situation is not working with maxima and minima (which even sound similar) but with capacities and obligations, which he may customarily treat with quite dissimilar vocabularies.

A common characteristic of the early language designs for the description of these models was a strong tendency to preserve, more or less uncritically, the many logically empty distinctions between “different kinds” of information. Thus, such a language would typically consist of a large number of dissimilar-looking templates, composed of reserved words; each template would correspond to one of the different kinds of information, and the reserved words would guarantee a familiar-looking quasi-sentence appropriate to that kind. The holes in the templates were to be filled with nothing more complex than lists of numbers or lists of names.

It would be grossly unfair to early workers in the area to suggest that the number of templates in any real language design was anywhere near as large as completely uncritical acceptance of the engineer’s discriminations would have implied. Considerable, and quite fruitful, effort was devoted to the reduction of this number to even more manageable limits. Nonetheless, it is fair to say that even the latest of the pre-AMBUSH languages retained all the stigmata of the template approach:

a. An extremely broad and shallow grammar: a very large number of syntactic types at the “statement” level, with almost no phrases in common (above the level of name list or number list).
b. A very large number of reserved words.
c. Inconsistencies of lexical style: the use of punctuation marks as separators when the templates were to look sentence-like, the use of spaces or position-on-card to delimit symbols if the templates were to look like arrays.
d. Prefix characters to signal statement type: since the templates possessed little or no common phrase structure, the analyzer was likely to consist of many essentially independent recognizers entered through a common switch and thus a prefix character was required to control the switch.

ANOTHER LOOK AT THE ENGINEER’S UTTERANCES, BY DESIGNERS IN THE RECURSIVE STYLE

Reduction in variety of user’s statement forms

Let us set aside for the moment the problem of specifying the structure of the graph itself, and confine our attention to those of the engineer’s utterances which include numerical information—these are the statements of restrictions or remarks about cost and income. The most striking fact about these numerical assertions is that each concerns an extremely small portion of the whole graph. Thus, a remark about material balance always talks about a subset of the arcs entering or leaving a single node; and the other numerical assertions only refer either to a subset of the arcs entering, or a subset of the arcs leaving, a single node.

This immediately suggested the possibility of turning the language “inside out” in the following sense: instead of having one statement type per “kind of information,” each containing its subgraph specification after its own fashion, we could have a far smaller number of statement types, one per kind of subgraph, each containing the kind-of-information specification in a more uniform fashion. This inversion had several attractions:

a. It dropped to a lower syntactic level the vast array of special forms, with their reserved words and need for special recognizers; above this level, phrase structure could be quite simple and uniform.
b. It simplified the job of persuading the user of the logical emptiness of many of his discriminations; if the template for pipeline capacities looks completely different from that for purchase obligations, with reserved words scattered analogously throughout the two forms, it is hard to convince him that they are logically similar; if, however, he sees the differences entirely embodied—reserved words and all—in two short phrases which fit in exactly the same larger structure, he becomes far more willing to drop the reserved words for his 10 different kinds of quantity restriction and simply say “maximum” or “minimum.”
c. It became trivial (indeed, almost inevitable) in grammar design, and very palatable to the engineer, to take advantage of the symmetry between the inputs of a node and its outputs. The syntactic type for discussing the one could look identical to that for discussing the other, to within a reserved word or two.
d. It suggested some interesting possibilities of nested information. The engineer frequently had several things to say about the same subgraph, or about two subgraphs, one of which was a proper subgraph of the other. We could perhaps arrange our grammar to permit him to nest his subgraph specifications, and to associate with each nesting level a set of (say) restrictions.
The redundancy of graph specification

A notable feature of the numerical assertions described above (the restrictions and income/cost utterances) is that, whether they be written in engineer's jargon, some template language, or in some new, recursive language, they inevitably contain information about the structure of the graph. This is perhaps obvious, for—as we have earlier indicated—every numerical assertion must contain a specification of the subgraph to which the assertion applies.

Thus, to say that the maximum amount of M flowing out of A is less than five surely suggests that the graph contains a node named A with at least one arc labeled M leaving it. Every numerical assertion contains some such topological information, and the set of such information collected from all assertions appeared to yield a fairly good picture of the overall shape of the graph. Initially, this redundancy was expected to provide a check: the topological implications of each assertion might be verified against a previously given graph description.

Then a further observation was made. A very large number of the nodes in a graph did not represent either processing units or staging points, but were essentially formal: they represented “pools” of intermediate products. Within a model, the engineer deals with two kinds of materials: the “pooled” ones, like intermediate products in a refinery, such that any consuming node can obtain them from any producing node; and the non-“pooled” ones, like raw materials and final products, which flow from each producing node to certain consuming nodes, and no others. For each pooled material, the engineer establishes a single pool represented by a node, with arcs to the node for each producer and arcs from the node for each consumer.

Moreover, since our user population modeled very large refineries with only a small amount of transportation at the edges of the graph, it turned out that virtually all materials in a typical model were of pooled type. This implied that, if we required the user to declare explicitly his non-pooled materials (which should be but a small burden, for they were not numerous), the Compiler could draw far stronger topological inferences from his numerical assertions than we had originally thought. Thus, from the example given earlier in this section, one can deduce not only that at least one arc carries M out of A, but—if M is pooled—that the arc must go to the pool for M and must be unique (since parallel arcs must have different labels).

This analysis led to the conclusion that the set of topological implications contained in all the engineer’s numerical utterances amounted to a virtually complete description of the graph, and thus separate statement types designed for graph specification were quite unnecessary. True, there might be an occasional piece of graph left ill-defined after the numerical statements were written, but our statements for making numerical assertions were going to incorporate the specification of little subgraphs to which a set of numerical remarks would apply. If we simply permitted these types to contain an empty set of numerical remarks, we would have a mechanism for describing little subgraphs, thereby filling in the holes in the graph definition. (The fact that this degenerate form would be inappropriate for describing a whole graph was entirely irrelevant.)

THE GRAMMAR OF AMBUSH

The grammar of AMBUSH derives directly from the observations of the preceding section. There is no statement type intended exclusively for graph description; there are three statement types for making numerical assertions, differentiated semantically by the kinds of arc set (subgraph) they discuss:

a. The YIELDS statement, for discussing the inputs and outputs of a node; this is used exclusively for the supply of yield factors.

b. The SENDS statement, for discussing subsets of the arcs leaving a node; this is used for all numerical assertions except the supply of yield factors.

c. The TAKES statement, for discussing subsets of the arcs entering a node.

(Note: In this section, we have permitted ourselves the liberty, in stating our rules of grammar, of using, without definition, the types (identifier) and (numex), which latter type corresponds to “numeric expression.”)

The YIELDS statement

Grammar

\[
\begin{align*}
\langle \text{row} \rangle &::= \langle \text{identifier} \rangle, \langle \text{numex} \rangle | \langle \text{row} \rangle, \\
\langle \text{numex} \rangle &::= \langle \text{numex} \rangle | \langle \text{row list} \rangle, \\
\langle \text{row list} \rangle &::= \langle \text{row} \rangle | \langle \text{row list} \rangle, \langle \text{row} \rangle, \\
\langle \text{name list} \rangle &::= \langle \text{identifier} \rangle | \langle \text{name list} \rangle, \\
\langle \text{identifier} \rangle &::= \langle \text{identifier} \rangle \text{ RUNNING} \\
\langle \text{yields statement} \rangle &::= \langle \text{identifier} \rangle \text{ YIELDS} \\
\langle \text{row list} \rangle &::= \langle \text{row list} \rangle | \langle \text{row} \rangle.
\end{align*}
\]
Examples

a. U YIELDS \( S_1, S_2, S_3 \)
RUNNING \( S_4, .1, .4, .5, S_5, .6, .1, .1 \)

b. U RUNNING \( S_4, S_5 \)
YIELDS \( S_1, .1, .6, S_2, .4, .1, S_3, .5, .1 \)

c. U RUNNING \( S_1 \) YIELDS \( S_2, .4, S_3, .5 \)

Discussion

It should be clear from the first two examples (which are identical in meaning) that the grammar has been designed to facilitate supply of a large array of yield factors in either input/output or output/input form. The more "linear" style, exhibited in Example c, for the supply of a few yield factors, is identical grammatically to the array style.

It should be noted that the grammar demands that the set of yield factors supplied in a single statement always correspond to some array (the cross product of a set of inputs and a set of outputs), however small. This requirement is made less troublesome than might appear, for the Compiler systematically fails to distinguish between a yield factor of zero and no yield factor at all. Thus, the engineer can, for example, write one statement to cover all but one of the yield factors at a node (representing the missing one by zero) and a second statement to supply the omission.

The SENDS statement

Grammar

a. Modifier

\( \langle \text{limit} \rangle :: = \text{MAX} | \text{MIN} | \text{FIX} \)
\( \langle \text{qlimit} \rangle :: = \text{QMAX} | \text{QMIN} | \text{QFIX} \)
\( \langle \text{cost} \rangle :: = \text{COST} | \text{COST0} | \text{COST1} \)
\( \ldots | \text{COST9} | \text{INCOME} | \text{INCOME0} | \text{INCOME1} | \ldots | \text{INCOME9} \)
\( \langle \text{quantity phrase} \rangle :: = \langle \text{limit} \rangle \langle \text{numex} \rangle \)
\( \langle \text{quality phrase} \rangle :: = \langle \text{identifier} \rangle \langle \text{qlimit} \rangle \langle \text{numex} \rangle \)
\( \langle \text{ratio phrase} \rangle :: = \langle \text{limit} \rangle \langle \text{numex} \rangle \langle \text{stream/node factor} \rangle \)
\( \langle \text{cost phrase} \rangle :: = \langle \text{cost} \rangle \langle \text{numex} \rangle \langle \text{modifier} \rangle \langle \text{quantity phrase} \rangle | \langle \text{cost phrase} \rangle | \langle \text{quality phrase} \rangle | \langle \text{ratio phrase} \rangle \)

b. SENDS Statement

\( \langle \text{stream/node factor} \rangle :: = \langle \text{identifier} \rangle | \langle \text{stream/node expression} \rangle \)
\( \langle \text{stream/node term} \rangle :: = \langle \text{stream/node factor} \rangle | \langle \text{stream/node term} \rangle, \langle \text{modifier} \rangle \)
\( \langle \text{stream/node expression} \rangle :: = \langle \text{stream/node term} \rangle | \langle \text{stream/node expression} \rangle, \langle \text{stream/node term} \rangle \)
\( \langle \text{ sends clause} \rangle :: = \text{SENDS} \langle \text{stream/node expression} \rangle \text{ TO } \langle \text{stream/node expression} \rangle | \text{ SENDS} \langle \text{stream/node expression} \rangle \)
\( \langle \text{ sends statement} \rangle :: = \langle \text{identifier} \rangle \langle \text{ sends clause} \rangle \)

Examples

a. Modifier

1. MAX 50
2. SPEC.GRAV QFIX .75
3. MAX .5 (M_1, M_2)
4. COST3 -3.50
5. INCOME3 3.50

b. SENDS Statement

1. U SENDS A, MAX 50
   The maximum quantity of A output by U is 50.
2. U SENDS (A, MAX 50, MIN 40, B, MAX 50), COST 3.50
   The maximum quantity of A output by U is 50.
The minimum quantity of A output by U is 40.
The maximum quantity of B output by U is 50.
A cost of 3.50 is incurred per unit of either A or B flowing out of U.
3. U SENDS (A, MAX 50, MIN 40, B, MAX 50), COST 3.50 TO X, MAX 10, Y, MIN 10
   All the meanings included in Example 2, plus:
The total flow of A or B from U to X has a maximum of 10.
The total flow of A or B from U to Y has a minimum of 10.

4. U SENDS A, B TO X
This example includes no numerical assertion, and is thus "degenerate." It informs the Compiler that there is one arc carrying A, and another carrying B, from node U to node X.

5. U SENDS A, MAX .5 (B, C, MIN 10)
The minimum quantity of C output by U is 10.
The quantity of A output by U is no more than .5 times the total quantity of B and C output by U.

Discussion

The four varieties of modifier correspond to the four kinds of numerical assertion (other than supply of yield factors) covered in earlier sections: quantity, quality, and ratio restrictions and income/cost assertions. The large number of reserved words in the definition of (cost) derives from the desire of the user to define several objective functions and from his preference not to treat an income as a negative cost (modifier Examples 4 and 5 are identical in meaning).

Each modifier must apply to a certain subgraph (set of arcs) of the graph and the only difficulty in understanding the SENDS statement is that of learning the rules which determine to which subgraph a given modifier applies.

First it should be noted that every SENDS statement begins with an identifier; this names a node of the graph, which we will—in this section—call the "subject node" of the statement. The word SENDS indicates that we are discussing arcs leaving the subject node. Next follows a stream/node expression in which appear identifiers which are the names of physical materials. The large number of reserved words in the definition of (cost) derives from the desire of the user to define several objective functions and from his preference not to treat an income as a negative cost (modifier Examples 4 and 5 are identical in meaning).

Each modifier must apply to a certain subgraph (set of arcs) of the graph and the only difficulty in understanding the SENDS statement is that of learning the rules which determine to which subgraph a given modifier applies.

The TAKES statement

Grammar

(takes clause) := TAKES (stream/node expression) FROM (stream/node expression) | TAKES (stream/node expression)

Examples and Discussion

The TAKES statement is completely analogous to the SENDS statement, both grammatically and semantically. With appropriate substitution of TAKES and FROM for SENDS and TO, the examples of the previous section will be correct. With similar substitutions in the discussion section, that section will work also.

Final notes on the grammar

The basic skeleton of AMBUSH consists of the three statement types described above and a few declarations: notably, one to list non-pooled materials and another for the supply of physical property values for materials which participate in quality restrictions.

The language has a full macro capability, including the insertion, at macro-expansion time, of actual parameters for formal parameters in the macro definit-
tion. Macros are, of course, used for abbreviation and to reduce copying errors. But they also serve as a way to parametrize a model: an identifier may be used in place of any numeric quantity, and its value may be changed from run to run. Other methods of parametrization, without recompilation, are also available.

For the sake of completeness, it should be added that:

a. The Compiler treats the AMBUSH program as a string, and consumes 72-column card images stringwise without format requirements.

b. AMBUSH (like many languages without a statement terminator) has a statement initiator (the # character).

c. An AMBUSH program is bracketed by the reserved strings BEGIN and END.

d. An AMBUSH identifier is built of alphanumeric and the period character; the initial character must be alphabetic and the character limit is twelve.

e. An AMBUSH number is a string of up to 14 digits among which at most one decimal point may appear.

f. All arithmetic is performed in floating point.

g. Numeric expressions are built in the customary way out of numbers, the four arithmetic operators, parentheses, and the special functions EXP and LOG. Unary minus is permitted. The evaluation rule is left-to-right and gives equal precedence to * and / and equal precedence to + and −.

h. A comment may be inserted between any two tokens of the language (roughly, in any place a space would be allowable). The comment is delimited by < and >; the # sign is prohibited within comments so that, when the programmer neglects to close his comment, it will be closed automatically by the initiator of the next statement.

e. There are relatively few reserved words.

d. Lexical strategy is uniform; the input is treated as a string.

On the other hand, there still remain certain elements of template style; for example:

a. The two forms of YIELDS statement (input vs. output and output vs. input) are a logical redundancy.

b. The reserved words QMIN, QMAX, and QFIX could be replaced by MIN, MAX and FIX. The (qlimit) forms remain as a reflection of the user's habits of thought: a quantity limit is somehow different from a quality limit.

Finally, it should be noted that the grammar we have given above is incomplete. Most of the differences between that grammar and AMBUSH as actually implemented come from the deliberate introduction of a few totally redundant templates to reduce the language's unfamiliarity. The interested reader will find several examples of these templates in the sample problem in Appendix I.

The reader will probably have noticed a feature which makes AMBUSH rather different from most languages which programmers use: there are no imperative statements in the language. All the sentences the programmer writes are in the indicative mood: they are either declarations, giving global information about the materials which flow in the network, or are local statements about individual nodes of the network. Thus the language is not algorithmic—there is no flow of control, no modification of values in the course of computation. This implies that the order of inputting sentences to the compiler is irrelevant, and this is essentially true. (The exception is that, if conflicting numerical values are given, the last one input is used.)

STATUS

The AMBUSH Compiler was written for the IBM 360/65 and has since been moved to the 360/85; it was designed to interface with the IBM LP package (MPS and MPSX). By the Fall of 1971, it had also been moved to the UNIVAC 1108, and was being shaken down on that hardware.

The AMBUSH Compiler was delivered near the end of 1969, and has been in regular use by Shell since that time. We understand that, as of late 1971, there were approximately 35 qualified AMBUSH programmers at Shell. The language is, apparently, fairly easy to learn, since fully half of the 35 received no formal instruction: they were given a language manual and a certain amount of hand-holding by more experienced users.
The language appears to permit the expression of most meanings fairly naturally, although occasional user models have exhibited structures which were unanticipated at language design time (and thus expressible only clumsily). The frequency of the need for tricks and clumsy expressions tends to reduce with time, since the need arises less from the nature of the application than from the habits of thought of the user. Experience with a new language inevitably alters the style of thought. Clear understanding of the problem comes to mean clear expressibility of the problem in the programming language.

Applied Data Research has been licensed by Shell to market the AMBUSH package in the United States and Canada. Initial explorations of the suitability of the language for various classes of LP users were under way at the end of 1971.

ACKNOWLEDGMENTS

We wish to express our appreciation to the many employees of the Shell Oil Company who gave unstintingly of their time and expertise in the course of this language design. We further wish to thank Shell for the use of the sample problem in Appendix I.

The Compiler for AMBUSH was actually built by a team led by Naomi Rotenberg; the other members of this crew were Barbara Butler, Margaret Carlin, and David Goldberg.

Finally, the author wishes to acknowledge the assistance of his colleague, Kirk Sattley, who undertook overseeing the final preparation of the manuscript when the author was out of the country.

APPENDIX I: EXAMPLE PROBLEM

A simple problem in processing/transportation typical of the process industries follows. While only a small problem, it permits invoking most of the AMBUSH syntactic forms. The AMBUSH description (a listing of the input to the AMBUSH Compiler) follows the statement of the problem itself.

A few statement forms are used in this example which are not described in detail in the paper, or in the skeleton grammar. The reader will probably understand them immediately from the problem description, but in any case, some explanatory notes are given following the listing.

Statement of problem

Choose the most profitable set of operations for the following situation.

The Archetype Manufacturing Company Inc., operates two processing plants:
1. ICE processing plant
2. EGG processing plant

The ICE plant operates with an inexpensive raw material which is obtainable as follows:

<table>
<thead>
<tr>
<th>SOURCE</th>
<th>COST $/TON</th>
<th>TONS/DAY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arkansas</td>
<td>7.00</td>
<td>120</td>
</tr>
<tr>
<td>Louisiana</td>
<td>7.50</td>
<td>300</td>
</tr>
</tbody>
</table>

Furthermore, Archetype has a long term contract with the Louisiana supplier guaranteeing the purchase of at least 180 tons/day at the $7.50/ton price. The ICE plant can process up to 250 tons per day of raw materials at a cost of $.20 per ton yielding three products as follows:

<table>
<thead>
<tr>
<th>PRODUCT</th>
<th>RAW MATERIAL SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gas</td>
<td>ARKANSAS .10</td>
</tr>
<tr>
<td>Universal Solvent</td>
<td>.70</td>
</tr>
<tr>
<td>Residue</td>
<td>.20</td>
</tr>
</tbody>
</table>

The EGG plant operates with a more expensive raw material obtained as follows:

<table>
<thead>
<tr>
<th>SOURCE</th>
<th>COST $/TON</th>
<th>TONS/DAY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alaska</td>
<td>20.00</td>
<td>100</td>
</tr>
<tr>
<td>Labrador</td>
<td>25.00</td>
<td>150</td>
</tr>
</tbody>
</table>

The EGG plant has capacity to process all the raw material available at a cost of $.50 per ton and produces two streams.

<table>
<thead>
<tr>
<th>PRODUCT</th>
<th>RAW MATERIAL SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exotic Liquid</td>
<td>ALASKA .60</td>
</tr>
<tr>
<td>Rejected</td>
<td>LABRADOR .40</td>
</tr>
</tbody>
</table>

The process requires .11 tons fuel gas per ton of raw material. The fuel gas must be obtained from the ICE plant. Any fuel gas not used in the EGG plant must be burned on site.

The Exotic Liquid can be blended with Universal Solvent to produce Exotic Product which has the following specification requirements.

<table>
<thead>
<tr>
<th>EXOTIC PRODUCT SPECIFICATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blue Color Index</td>
</tr>
<tr>
<td>Specific Gravity</td>
</tr>
</tbody>
</table>
The corresponding properties of the two components are:

<table>
<thead>
<tr>
<th></th>
<th>BLUE COLOR INDEX</th>
<th>SPECIFIC GRAVITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exotic Liquid</td>
<td>11.5</td>
<td>1.2</td>
</tr>
<tr>
<td>Universal Solvent</td>
<td>10.0</td>
<td>0.9</td>
</tr>
</tbody>
</table>

In order to meet the blue color index, Blue Dye must be added. One gram of Blue Dye will raise the blue color index of one ton of Exotic Product .7 units and will have an insignificant effect on specific gravity. Blue Dye costs $.80 per gram.

Universal Solvent can also be blended with Reject from EGG plant and Residue from ICE plant to make Unexciting Product. The blend has the following formula:

- Universal Solvent .25
- Reject .45
- Residue .30

All Reject must be disposed of in this manner. Any remaining Universal Solvent and/or Residue may be sold as produced.

All products leave Archetype by Pipeline except Residue which can also be removed by Truck. Pipeline carries products to Depot 1 and from there to Depot 2. Pipeline pumping limitations for each product and prices and demands at each depot are given below.

### MAX PIPELINE FLOW

<table>
<thead>
<tr>
<th>PRODUCT</th>
<th>TONS/DAY</th>
<th>DEPOT 1/ DEPOT 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exotic Product</td>
<td>500</td>
<td>225</td>
</tr>
<tr>
<td>Universal Solvent</td>
<td>400</td>
<td>175</td>
</tr>
<tr>
<td>Unexciting Product</td>
<td>450</td>
<td>200</td>
</tr>
<tr>
<td>Residue</td>
<td>300</td>
<td>125</td>
</tr>
</tbody>
</table>

### SALES

<table>
<thead>
<tr>
<th>PRODUCT</th>
<th>DEPOT 1</th>
<th>NET PRICE</th>
<th>DEMAND</th>
<th>NET PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exotic</td>
<td>125 Max.</td>
<td>40</td>
<td>120 Max.</td>
<td>35</td>
</tr>
<tr>
<td>Universal Solvent</td>
<td>60 Max.</td>
<td>30</td>
<td>80 Fix.</td>
<td>25</td>
</tr>
<tr>
<td>Unexciting</td>
<td>120 Max.</td>
<td>15</td>
<td>Unlimited</td>
<td>10</td>
</tr>
<tr>
<td>Residue</td>
<td>Unlimited</td>
<td>3</td>
<td>Unlimited</td>
<td>3</td>
</tr>
</tbody>
</table>

Note that the pipeline capacity effects are cumulative. That is, 250 tons/day of Exotic Product plus 150 tons/day of residue would be permissible, but would use all of the capacity.

Residue removed by truck costs Archetype $.50/ton for hauling it away. In addition, handling problems limit to 30 tons/day the Residue disposed of as Residue.

The AMBUSH statements describing this problem follow. The AMBUSH compiler will convert these statements into a matrix which can be optimized by an LP code to give the combination of operating choices which will maximize profit by Archetype.

---

**Figure 1**—Archetype Manufacturing Company Process flow
#BEGIN
#MACRO: FOUR,PRODUCT=EXOTIC,PROD, UNEXCIT,PROD, XES,SOLVENT, EXCESS,RESID
#NPOOL: FOUR,PRODUCT
#ADDITIVE: BLUE,DYE
< >
< RAW MATERIAL SLATE >
< ARKANSAS SENDS ARKAN,IE,R,M, COST 7.00, MAX 120>
< LOUISIANA SENDS LOUIS,IE,R,M, COST 7.50, MAX 300, MIN 180>
< ALASKA SENDS ALASK,EX,R,M, COST 20.00, MAX 100>
< LABRADOR SENDS LABRA,EX,R,M, COST 25.00, MAX 150>
< PURCHASE SENDS BLUE,DYE, COST .80>
< >
< PROCESSING UNITS >
< ICEPLANT > MAX 250, COST .20
< >
< ICEPLANT RUNNING ARKAN,IE,R,M, LOUIS,IE,R,M>
< >
< FUEL,GAS, C10, C12, C7, C14, C20, C14, C20, C30
< >
< EGGPLANT > COST .50
< >
< EGGPLANT RUNNING ALASK,EX,R,M, LABRA,EX,R,M>
< >
< EXOTIC,LIQ, C60, C70, C40, C30
< >
< REJECT, C45, C30
< >
< EGGPLANT TAKES FUEL,GAS, FIX .11 (ALASK,EX,R,M, LABRA,EX,R,M)
< >
< PRODUCT BLENDS >
< PROP: BCINDEX, SPEC,GRA
< STREAM: EXOTIC,LIQ, 11.5, 1.2, 10.0, 0.9, 7.0
< BLUE,DYE, 7
< >
< BLENDEXOT TAKES (EXOTIC,LIQ, UNI,SOLVENT, BLUE,DYE)
< BCINDEX QMIN 12, SPEC,GRA QMAX 1.1
< >
< BLENDUNEX MAKES UNEXCIT,PROD
< BLENDING: UNI,SOLVENT, C25, C25, C45, C30
< >
< RESIDUE, TRUCKS, COST .50, MAX 30
< >
< ACCUMULATE EXCESSES>
< POOL(RESIDUE) SENDS RESIDUE TO (RESIDUE, EXCESS, TRUCKS, COST .50, MAX 30)
< >
< SOLVENT,XES TAKES UNI,SOLVENT>
< BURNING TAKES FUEL,GAS
< >
< DEPOT1 TAKES EXOTIC,PROD FROM BLENDEXOT
< UNEXCIT,PROD FROM BLENDUNEX
< XES,SOLVENT FROM SOLVENT,XES
< EXCESS,RESID FROM RESID,EXCESS
< >
Notes to the listing

Card numbers:

400,500: These illustrate the form of simple declarations—a declarative reserved word, followed by a colon, followed by a list of identifiers used as names of materials. Note that FOUR.PRODUCT is a list of four material-names by virtue of the MACRO definition on cards 200-300.

400: NOPOOL tells the compiler not to automatically set up a POOL node for the material(s) named in the declaration. This implies that arcs carrying these materials must have both end points specified (see cards 5800-6100 for the FOUR.PRODUCT materials). Compare this to cards 2000, 3100, and 5500 which together describe the handling of FUEL.GAS: ICEPLANT “yields” it (to its pool), and EGGPLANT and BURNING “take” it (from the pool).

500: The material BLUE.DYE is declared to be an additive. This informs the compiler that the quantity of blue dye coming into a node (card 4100) has no effect on material-balance calculations, but only on quality determinations.

1600,2400: This is a shorthand statement for applying modifiers to the total capacity or throughput of a node. Formally, such a statement is interpreted by the compiler as a degenerate form of the TAKES statement, with the modifiers applying to the set of all input arcs to the node.

3500-3900: This declaration gives, for each relevant material, the coefficient to be used in “quality” calculations, described in an earlier section.

4400-4800: This is one of the “totally redundant template statements” retained in the language because of its familiarity; the MAKES ... BLENDING statement is a variant of the YIELDS statement, used to describe a node which produces a single material, by giving the “recipe” for its product in terms of its inputs.