DOCUMENT RESUME

ED 161 692 SE 025 112

AUTHOR Jarvis, John J.; Papaconstadopoulos, Chris
TITLE A Methodology for Designing High Level Computer Input Systems for Mathematical Programming Models. Industrial and Systems Engineering Report Series No. J-78-16.
INSTITUTION Georgia Inst. of Tech., Atlanta. School of Industrial and Systems Engineering.
SPONS AGENCY National Science Foundation, Washington, D.C.
BUREAU NO NSF-SED-75-17476
PUB DATE May 78
NOTE 45p.; Not available in hard copy due to marginal legibility of original document

EDRS PRICE MF-$0.83 Plus Postage. HC Not Available from EDRS.
DESCRIPTORS *Computer Programs; *Computers; *Linear Programming; *Mathematical Applications; *Operations Research; Post Secondary Education; *Programing; Systems Development

ABSTRACT

Developed and demonstrated here is the design of interface systems for cost-effective communication of the user with the computerized operations research procedures. The concentration is in the area of interfacing methods for implementing the optimization techniques of mathematical programing. Emphasis is given to flexibility of accessing the algorithm, cost-effectiveness, and pedagogical or self-teaching aspects of the interface systems. Methods which take advantage of the characteristics of the input model, for soliciting, storing, and analyzing the input information, are investigated. Concepts in higher level interfacing systems are also explored. Linear programing is the vehicle for experimental development of interface systems. The results are applicable to other mathematical programing procedures. (Author/MP)

Industrial and Systems Engineering
Report Series No. J-78-16
May 1978

A METHODOLOGY FOR DESIGNING HIGH LEVEL COMPUTER INPUT SYSTEMS FOR MATHEMATICAL PROGRAMMING MODELS

by

John J. Jarvis
and
Chris Papaconstadopoulos
Georgia Institute of Technology
Atlanta, Georgia 30332

ABSTRACT

This paper presents a methodology for the design of cost-effective computer input systems for mathematical programming models. It illustrates how the input language can be defined through a formal grammar and how the cost-effectiveness of the input features can be determined by observing their effect on the grammatical structure. A methodology for the design of the input parsing algorithm is presented, based on the graph of the grammar. The graph can be transformed to provide effective error detection at the lexical and syntatical levels. The automatic deletion of edges and vertices, through a syntax-directed type input, provides increased semantical analysis.

INTRODUCTION

Operations research has evolved into a science of a high degree of sophistication. Algorithms have been developed for solving a variety of problems. Many of these algorithms are recognized for efficiency and universal application. The evolution paralleled the growth of computer science and technology, since most operations research methods require a digital computer for efficient cost-effective utilization.

There is, however, a difficulty that causes most of these sophisticated computer algorithms to be underutilized: the barrier associated with communication between man and machine. Since access to these algorithms has required considerable computer skills and special knowledge of program structure, potential users tend to avoid sophisticated computer codes for operations research methods.

The purpose of this paper is to develop and demonstrate the design of interface systems for cost-effective communication of the user with the computerized operations research procedures. These interfacing systems should provide trade-offs between costs incurred by the user and the machine. The concentration is in the area of interfacing methods for implementing the optimization techniques of mathematical programming. Emphasis is given to flexibility of accessing the algorithm, cost-effectiveness, and pedagogical or self teaching aspects of the interface systems. Methods which take advantage of the characteristics of the input model, for soliciting, storing, and analyzing the input information, are investigated. Concepts in higher level interfacing systems are also explored.

Linear programming is the vehicle for experimental development of interface systems in this paper. The results are applicable to other mathematical programming procedures (e.g., nonlinear and dynamic programming), since their data requirements are quite similar.

The spectrum of computer systems for mathematical programming begins with the commercial production packages and ends with the high-level input system. The former represents the communication at the machine level and the latter the communication of the problem environment level. The following list is a representative feature-wise tracing of the development in this area: UNIVAC 1108 - LP, CLP [8], UHELP [3], MPOS [2], EZLP [6]. The last on the list is discussed in detail in a later section of this paper.

Definitions and Basic Concepts

- A metalanguage is a language describing another language (object language).
- Terminal symbols are the symbols of the object language.
- Nonterminal symbols are the symbols of the metalanguage. (In this paper nonterminals will be denoted by capital letters; e.g., DIGIT is a nonterminal and can be used to denote any digit).
- An alphabet A is a finite set of terminal symbols.
- A language is an infinite set of character strings on some alphabet A.
- A production H is a string transformation rule or grammatical rule. H is denoted as $X \rightarrow Y$ where Y is the transformed X.
- A grammar G is the nonempty set of productions over a given set of terminal and nonterminal symbols.

Any element of a language is "produced" through a finite recursive transformation of strings. The starting point of this recursion is called the starting symbol of the grammar. In Operations Research languages, this starting symbol is the nonterminal MODEL. This is the most structured nonterminal.

Consider the grammar: G = (N, A, EXPRESSION, P)

where: N = {B,C} : set of nonterminals

A = {b,c} : set of terminals

P = {EXPRESSION → BC

B → bB, B → b,

C → cC, C → c} : set of producitons

e.g., EXPRESSION → BC → bB → bbBc → bbbC → bbbc

The string bbbc is an element of the language generated by the grammar G. However, consider the string cbbc. A typical translation process would attempt, say, by left to right scanning, to reach the starting symbol of the grammar by successively applying legal transformation rules,

Exhaustive Tree Search:

cbbc → CBBC ; terminated with error since no further transformation is possible

cbbc → CBBC → CBC : terminated with error since no further transformation is possible

Hence, the string cbbc would be rejected as an illegal construct.

- The syntax of a language is the set of rules specifying legal constructs of the language.
- The semantics of a language is the assignment of meaning to the constructions of the language.
- A grammar is nonlinear if at least one of the grammatical rules have, on the right side, more than one nonterminal symbol. All the rules must have a single nonterminal in the left side. e.g., The rule VARIABLE → LETTER. DIGIT is a nonlinear one and so is characterized by the grammar containing it.
- A grammar is linear if all the rules have a single nonterminal on the left side and at most one nonterminal on the right side. e.g.,

  OPERATOR → + : linear rule

Now we have enough formalism to make a speculation about the expected cost

of recognizing (accepting or rejecting) input character strings. We will refer to this process of recognition as parsing.

The parsing of a string generated by a simple nonlinear production rule will be more expensive than parsing a string generated by a linear rule. The reason is that the nonlinear rule contains at least two nonterminals on the right side and the resulting syntactical tree (parsing actually is a tree search) will be more complex than any tree implied by a linear rule. Furthermore, the greater the number of nonlinear rules in the grammar, the higher the expected parsing cost. Also, the higher the degree of nonlinearity, the higher the parsing cost. These two aspects of nonlinear grammar, say density of nonlinearity and degree of nonlinearity, are controllable by the designer of the grammar. Since the rules of the grammar are "models" of the language features, the designer can weight the various input features by their importance and their parsing cost.

The above concepts provide a foundation upon which a cost-effective methodology can be devised. Before that, however, we shall discuss a metalanguage which adequately describes grammatical specifications.

Syntax Specifications

A metalanguage widely used to specify syntax is the Backus-Naur Form (BNF). It precisely specifies syntactical rules, but lacks the power to specify semantical rules.

In BNF, the following notational rules exist:

1. Nonterminals are written in brackets (<>); i.e., <X> means the class of nonterminals named X.
2. The sign of production (→) is replaced by :: = and is read as "is replaced by".

3. Multiple ways of transforming a nonterminal (alternative productions) are written on the same line separated by the ORing operator "|".

EXAMPLES:

In BNF, the following is written:

<DIGIT> :: = 1 | 2 | 3 |...| 9 | 0

The recursive character of a set of productions can be easily specified, as in:

<LIST> :: = <VARIABLE> | <LIST>, <VARIABLE>

An extension of the above feature is:

<LIST> :: = VARIABLE | <LIST>, <VARIABLE>
(n)

where (n) assigns an upper bound to the recursion.

The following example could be a statement in a user oriented mathematical programming system. Consider: LET X1, X2, LINE3, POWER > = 10.5.

In the context of formal languages, this is a phrase of a language with a specified grammar. Part of this grammar is the following segment of syntactical rules related to the above phrase.

<LIST CONSTRAINT> :: = LET <NULL STRING><LIST><REL. OP><NUM>

<LIST> :: = <VARIABLE>|<LIST>, <VARIABLE>

<VARIABLE> :: = <LETTER> | <LETTER><LITERAL>
(k-1)

<LITERAL> :: = <DIGIT>|<DIGIT><LITERAL>|<LETTER>|<LETTER><LITERAL>
(K-2) (k-2)

<NULL STRING> :: = SPACE | SPACE <NULL STRING>
(4)

<NUM> :: = <INTEGER>|<INTEGER>.<INTEGER>

<INTEGER> :: = <DIGIT>|<DIGIT><INTEGER>

<LETTER> :: = A | B |....| Z

<DIGIT>:: = 1 | 2 | ... 9 | 0

It is observed that this grammar specifies a maximal length of K characters for variables. Also, it allows up to five spaces after the key word LET.

In BNF, user oriented features can be included or excluded by a quick transformation of the grammar. Hence, the language designer can visualize the effect of certain features. The effect has two components:

1. The user-response and attitude towards certain syntactical rules
2. The cost associated with the design and utilization of a parsing algorithm

As a final example of this section, a complete mathematical model and its grammar will be presented. Consider the following linear model:

$$\text{OPT} \sum_{I=1}^{N} C_I X_I$$

$$\text{ST} \sum_{I=1}^{N} A_{IJ} X_I = R_J \qquad J = 1,M$$

The corresponding grammar is:

G = (N, A, MODEL, P)

A = {A, B,..., Z, 1, 2,..., 9, =,-, ST,≠, MIN, MAX, NULL}

N = {MODEL, CONST. SET, CONST, OPT. AE, NUM, SIGN, ARITHM. OP, VAR, AE2, LETTER, DIGIT, INTEGER, LITERAL}

Where AE denotes Arithmetic Expression and CONST denotes CONSTRAINT.

P is the set of productions.

The grammar of the model is shown below in BNF:

<MODEL> :: = <OPT><AE>ST<CONST. SET>

<CONST. SET> :: = <CONST.>|<CONST.><CONST. SET>
(M-1)

<CONST.> :: = <AE> := NUM

<AE> :: = <SIGN><NUM><VAR>|<SIGN><NUM><VAR><AE2>
(N-1)

<AE2> :: = <ARITHM OP><NUM><VAR>|<ARITH. OP><NUM><VAR><AE2>
(N-2)

<VAR> :: = <LETTER>|<LETTER><ALFANUM>

<ALFANUM> :: = <DIGIT>|<DIGIT><ALFANUM>|<LETTER>|<LETTER><ALFANUM>
(k-2)

<NUM> :: = <INTEGER>|<INTEGER>.<INTEGER>

<INTEGER> :: = <DIGIT>|<DIGIT><INTEGER>

<DIGIT> :: = 1 | 2 | ... 9 | 0

<LETTER> :: = A | B | ... | Z

<OPT> :: = MAX | MIN

<SIGN> :: = + | - | null

<ARITM. OP> :: = + | -

A DESIGN METHODOLOGY

The methodology for processing (recognition and interpreation) of languages for operations research (OR) models consists of a series of processing phases. These phases represent a decomposition of the processing into sub-processing phases so that each phase will be responsible for processing certain grammatical constructs of the language in such a way that:

i. the information content will not be altered

ii. a grammatically illegal construct will eventually be detected

iii. the input string will be transformed in a way that will assist the processing by the next coming phase

iv. the system can be modularly designed.

This approach is language independent.

A natural decompostion, used in most compilers, is divided into three phases: lexical analysis, syntactical analysis, and semantical analysis.

Lexical analysis is the process in which the basic grammatical constructs (variables, coefficients, etc.) are identified and their grammatical correctness is checked.

Syntactical analysis is the process in which grammatical constructs of higher order (arithmetic expressions, constraints, etc.) are identified and checked.

Semantical analysis is the process in which a meaning is assigned to each construct. This process checks the meaning of a specific construct with respect to other constructs, assuming syntactical correctness. For instance, in a syntactically correct double bounded constraint, the lower bound is greater than the upper bound.

The first two phases represent the recognition process; the last one is the interpretation process. These phases will be presented in more detail in the following sections.

The design of the interfacing system should be initiated with the design and analysis of the dialogue of the system. As a first step, the designer of the interfacing system should define the level of communication by identifying various ways of entering the model. As well, the input features should be selected and explicitly stated.

As a second step of the design, the grammatical restrictions on the input should be described. This implies syntactical and semantical specifications. The syntax can be specified by formally describing the grammar. The semantics attached to this grammar should be clearly stated.

The third step of the design should be the analysis of the grammar in order

A natural decompostion, used in most compilers, is divided into three phases: lexical analysis, syntactical analysis, and semantical analysis.

Lexical analysis is the process in which the basic grammatical constructs (variables, coefficients, etc.) are identified and their grammatical correctness is checked.

Syntactical analysis is the process in which grammatical constructs of higher order (arithmetic expressions, constraints, etc.) are identified and checked.

Semantical analysis is the process in which a meaning is assigned to each construct. This process checks the meaning of a specific construct with respect to other constructs, assuming syntactical correctness. For instance, in a syntactically correct double bounded constraint, the lower bound is greater than the upper bound.

The first two phases represent the recognition process; the last one is the interpretation process. These phases will be presented in more detail in the following sections.

The design of the interfacing system should be initiated with the design and analysis of the dialogue of the system. As a first step, the designer of the interfacing system should define the level of communication by identifying various ways of entering the model. As well, the input features should be selected and explicitly stated.

As a second step of the design, the grammatical restrictions on the input should be described. This implies syntactical and semantical specifications. The syntax can be specified by formally describing the grammar. The semantics attached to this grammar should be clearly stated.

The third step of the design should be the analysis of the grammar in order

to allow an estimation of the expected cost associated with the grammar. This analysis identifies potential transformations and trade-offs of flexibility. The final version of the grammar will be the input to the fourth step.

The fourth step of the design should be the selection and design of the supporting software. Algorithms and the mode of their operation for lexical and syntactical analysis should be defined. The proper cost-effective data structure should also be designed considering user features and the internal OR technique. The data structure is the only part of the interfacing system that will be affected by the OR technique.

Lexical Analyis

The basic function of the lexical analysis is the left-to-right scanning of the input string and the grouping of characters in units which represent the basic classes of the language. The following is the result of the lexcial analysis on a typical linear programming objective function:

MAXIMIZE	3	X1	+	2	X2	-	15

According to the grammar of the language, these basic syntactic classes (tokens) are nonterminal symbols and hence, they have names. Therefore, [X1] is a VARIABLE, [+] is an ARITHMETIC OPERATOR, and so on. Blanks are suppressed.

A nonterminal might be mapped into a variety of terminals. Therefore, another function of the lexical analysis is to find the exact mapping element and establish the proper pointers so that the information can be passed to the next phase.

How efficiently the information is processed is a problem of data structures and will not be discussed here. For simplicity, a simple table is used

which is usually called the Uniform Symbol Table (UST). It consists of two entries: the class name (nonterminal), and the pointer to the appropriate table.

Various support tables will be used by a lexical analyzer for variable names, numeric entries, etc. The UST for the above example will be as follows:

Table 1. The Uniform Symbol Table

Token	Type	Pointer in Tables
MAXIMIZE	OPTIMIZE	1
3	NUMERIC	1
X1	VARIABLE	1
+	ARITHM. OP.	1
2	NUMERIC	2
X2	VARIABLE	2
-	ARITHM. OP.	2
15	NUMERIC	3

This table, along with the updated support tables, will be the input to the syntactical analysis phases. The support tables are needed to preserve information which will be used by the parser and semantic analyzer. Hence, the lexical analyzer should correctly update these tables. For instance, if a variable name is scanned by the lexical analyzer, it should be entered in the table for variable names only it if is not already entered.

Through lexical analysis, the information content is preserved and the

string is transformed into symbols of unit length, but of higher grammatical order. This will make the syntactical anlysis easier since the parser will not be working on characters, but on tokens.

Lexical analysis, more formally defined, is the analysis of an input string through certain linear productions or "linear-like" productions, such as:

<VARIABLE> :: = <LETTER>|<VARIABLE><LETTER>

By scanning the terminal symbols, which are the characters, the nonterminals (tokens) are constructed. At the same time, certain violations of the rules are detected (e.g., illegal characters in an identifier, alphabetic characters in numeric field, etc.).

At the presense of an error, the lexical analyzer should be able to recover to scan the remaining string, and to create a UST so that the parser will detect any syntactical errors of higher order.

The implementation of the lexical analyzer can be accomplished by coding a state diagram where each state represents a nonterminal symbol of the grammar and each arc represents a terminal symbol. For instance, the productions

<VARIABLE> :: = letter|<VARIABLE> letter|<VARIABLE> digit

<INTEGER> :: = digit|<INTEGER> digit

can be interpreted as the graph of Figure 1.

The state LINK represents a starting state and the remaining of the grammatical rules. To increase the flexibility, an abstract symbol is introduced; this is the "break" symbol which could be anything that differentiates syntactical constructs including the null. That is to say, the break symbol terminates a syntactical construct and introduces a new one. By introducing the "break" symbol into Figure 1, we obtain the graph of Figure 2.

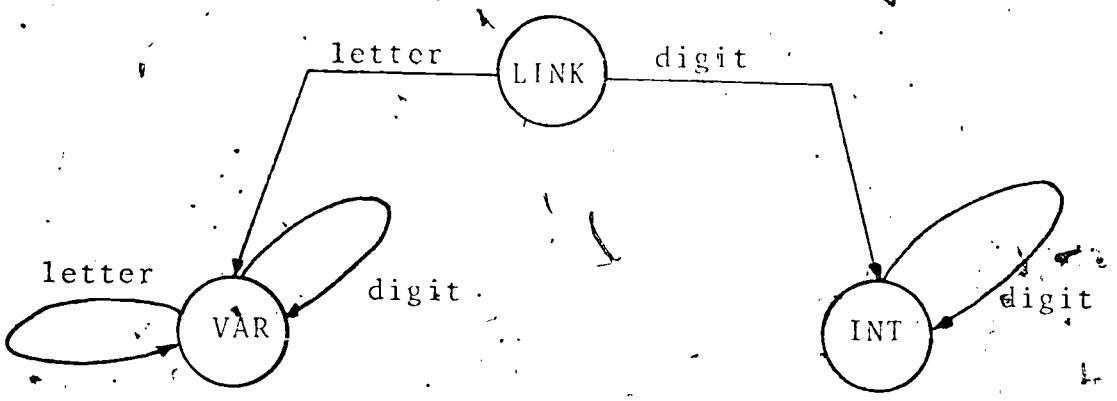


Figure 1. The State Graph of Grammatical Definitions

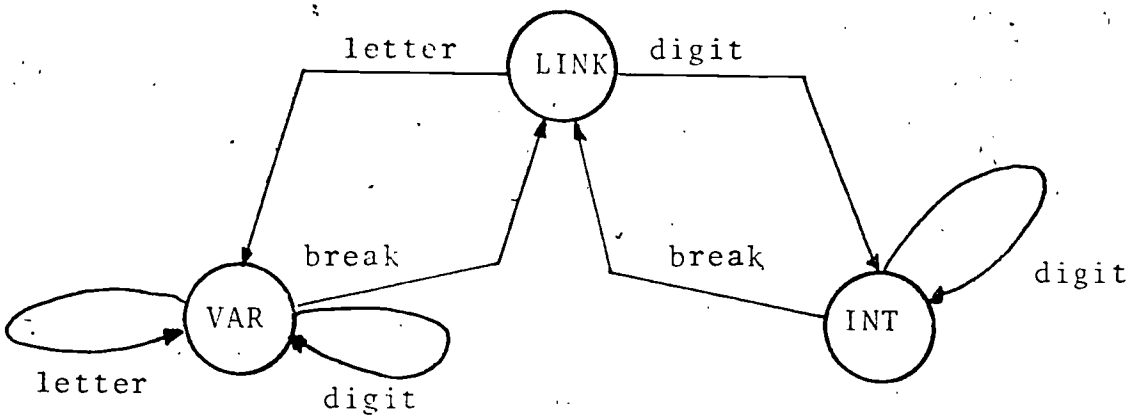


Figure 2. The State Graph of Grammatical Definitions Including Transitions under the Break Symbol

The "break" symbol should be used only when needed; if one knows exactly where to go from the current state, then going gack to LINK might duplicate the effort. Thus, the "break" symbol should violate the syntactical rules, as represented by the current location on the graph. This means that the break symbol does not drive the parsing back to the current state nor to any other state connected to the current one. The break symbol causes a transition back to the LINK state where it will either be defined as the starting symbol of another syntactical construct, or it will cause an error; in the latter case, it will be ignored to enable the process to continue.

The graphical representation should be able to handle certain low-order non-linear productions since there is an implicit memory in the graph. This is illustrated by Figure 3, as a modification of the graph of Figure 2, so that the production <NUMERIC> :: = <INTEGER>|<INTEGER>.<INTEGER> can be handled at the level of the lexical analyzer.

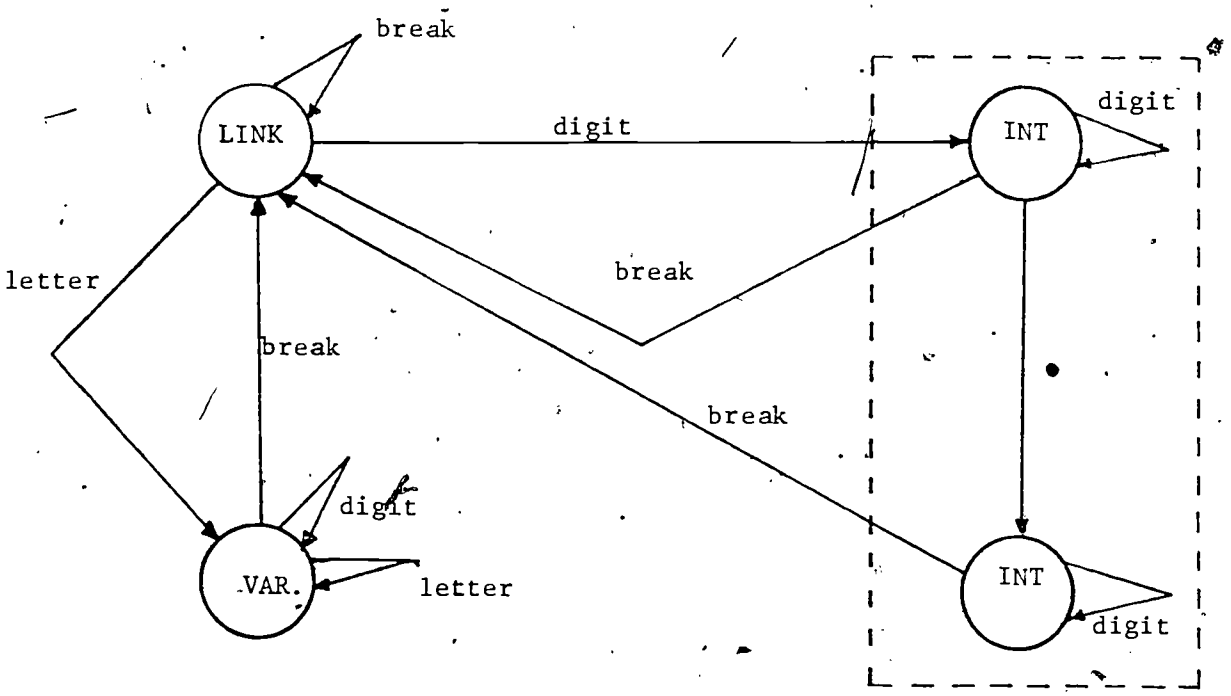


Figure 3. The State Graph of a Grammatical Definition Illustrating the Graphical Representation of Nonlinear Definitions

From Figure 3 we observe that:

i. The construct NUMERIC is represented implicitly by the two states in the square with dotted lines.

ii. The "break" arcs are reduced to those needed. The graphical representation presented above can serve as the framework for a lexical analyzer. For coding, each state of the graph must be represented with a separate portion of code. This approach will make the lexical analyzer highly modular since states and arcs can be easily added or deleted. Hence, grammar modifications can be performed at a low designing cost. Since all the grammatical constructs will be defined on the graph, uncertainty will be decreased, and thus the time of processing will probably decrease in comparison to a table driven parsing.

Syntactical Analysis

The syntactical analyzer operates on the UST (Uniform Symbol Table). Its basic function is to recognize the syntactical constructs and check the correctness of their construciton as specified by the rules of the grammar. At the phase of parsing, the rules which were utilized by the linear analyzer are eliminated from the grammar and the terminal symbols of the grammar at the current stage are the symbols of the UST (first entry). The second entry of the table is not used by the parser, but is used for the interpretation process which might be incorporated with the parser.

An error recovery facility should exist in the syntax analyzer for economical reasons since it would enable the parser to locate most of the errors without reprocessing the string. At worst, an error recovery facility enables the parser to locate further errors not related to the first error encountered in a string, while perhaps identifying legal syntactical elements wrongly as errors

due to assuming a state after an error which is different from the intended. At best, an error recovery facility whose assumed state sufficiently matches the intended state often indicates specific alternative syntaxes that can be redefined as legal in the redesign of the syntax analyzer.

Consider the model:

$$\text{OPTIMIZE} \quad \sum_{i=1}^{N} c_i X_i$$

$$\text{ST} \quad \sum_{i=1}^{N} a_{ij} X_i \geq R_j \qquad j = 1,M$$

Assuming that the input has been processed by the lexical analyzer as described in the previous section, all the linear or "linear-like" productions have been eliminated and the UST has been created. The entries of the UST are the terminal sumbols for the reduced grammar of the model. This grammar is of reduced nonlinearity since certain nonterminals were converted to terminals. An algebraic analogy to the reduction to nonlinearity is the conversion of variables to constants. Thus, the grammar of the model could be rewritten in reduced form where entries in small letters denote terminal symbols and the underline is used to separate them. Let us call this the "UST grammar." The UST grammar in BNF is as follows:

<MODEL> :: = OPT__ <AE>__st__<CONST. SET>

<CONST. SET> :: = <CONST.>|<CONST.><CONST. SET>

<CONST> :: = <AE> > = num

<AE> :: = sign __num __var | sign __num __ var __<AE2>

<AE2> :: = ao__num__var__ | ao__num__var__<AE2>

where

opt denotes optimize

st denotes subject to

num denotes numeric

var denotes variable

ao denotes arithmetic operator

In the case of a state graph, a linear production, $A \rightarrow xB$ is converted to a transition from state A to state B through the arc x. At the syntax analysis level, the productions are nonlinear with degree on nonlinearity at least two.

Previously, the implicit memory in the state graph was used to handle certain nonlinear productions at the lexical analysis level. The definition of a nonterminal (i.e., NUMERIC) was decomposed to its elements; each nonterminal was represented by an arc. The same concept can be applied at the syntax analysis level. However, at this level, the productions might have more than one terminal associated with each nonterminal and hence, the terminals cannot be represented by arcs. To solve this problem, a special assumption will be made.

All the symbols of the grammar are represented as states in the graph. A neutral symbol (e) will be used to denote the arcs. It is assumed that e precedes all the symbols of the grammar. (The UST grammar is assumed.) The introduction of this neutral symbol can be viewed as a linearization of the grammar. Assuming symbols S_1, S_2, S_3, S_4, the production

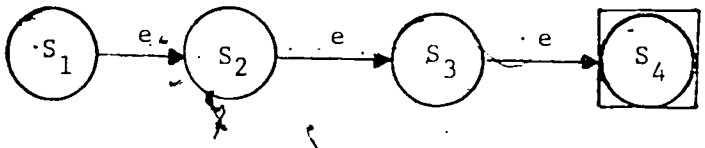
$S_1 \rightarrow S_2\ S_3\ S_4$ is written as

$S_1 \rightarrow eS_2\ eS_3\ eS_4$

The following is a notation where the depth of the structure is illustrated:

$$S_1 \xrightarrow{e} (S_2 \xrightarrow{e} (S_3 \xrightarrow{e} (S_4)))$$

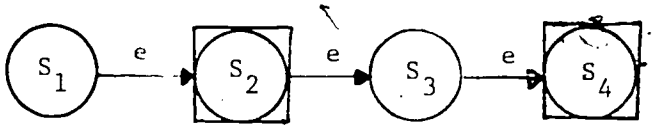
This in state graph notation is:



The S_i's are the states of the graph and the state in the square is the terminal state. There may be more than one terminal state.

The state graph of the production

$S_1 \rightarrow S_2 \mid S_2S_3S_4$ is



Consider the following UST grammar:

$$S_0 \rightarrow S_1 \mid S_0S_1 \mid S_1S_2$$

$$S_2 \rightarrow S_1 \mid S_3S_2$$

The "e-equivalent" of this grammar is

$$S_0 \rightarrow eS_1 \mid eS_0eS_1 \mid eS_1eS_2$$

$$S_2 \rightarrow eS_1 \mid eS_3eS_2$$

The state graph of this grammar is presented in Figure 4.

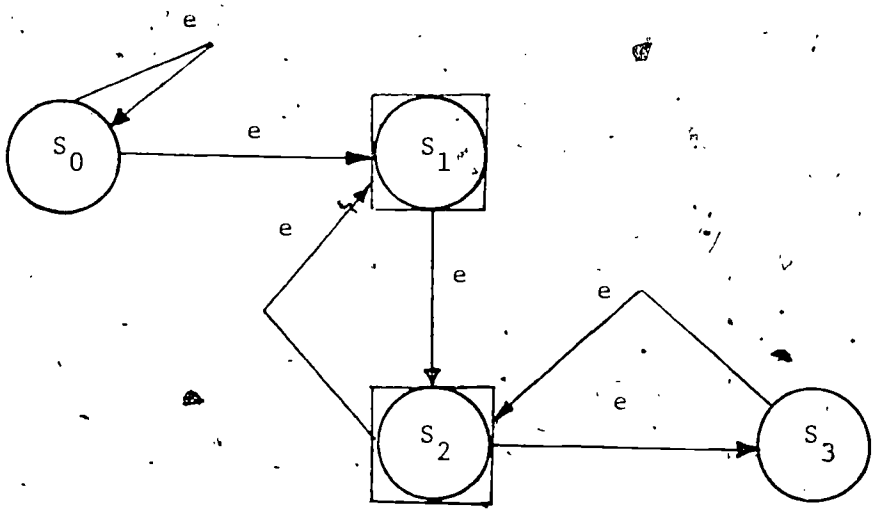


Figure 4. The State Graph of an "e-equivalent" Grammar

Indeed, the string $S_1S_2S_3S_2S_1$ is a legal one since the "flow" reaches the terminal state S_1. However, the string $S_1S_2S_3S_1$ is not legal since there is not path from state S_3 to state S_1.

The amount of memory, implicitly embedded in the above state graph, is exactly the same amount of memory embedded in a 0-1 table representation of the graph as shown in Table 2.

Table 2. The Matrix Equivalent to the State Graph of Figure 4

	S_0	S_1	S_2	S_3
S_0	1	1	0	0
S_1	0	0	1	0
S_2	0	1	0	1
S_3	0	0	1	0

That is to say that the parsing algorithm "remembers" only what is the current and the previous state of parsing. This can be considered as the major disadvantage of a table driven parsing. It would be desirable to extend the memory of the state graph because it would facilitate features such as error recovery or diagnostics. The cause of the problem of limited memory is the recursion through the same state. For instance, in Figure 4 there is recursion flow through states S_1 and S_2. A way of extending the memory of the graph would be to consider each state as a family of states, sot that state S_j is a type of states and not a single state. Then, all the recursions over state S_j would be eliminated by introducing an additional state of type S_j each time there is a recursive transition over state S_j. That is to say, in Figure 4 a new state, S_1 and S_2, should be introduced to extend the memory of the graph. This is illustrated in the state graph of Figure 5, where the points indicate an infinite repetition of the same state sub-graphs. The state graph of Figure 5 has an infinite memory.

By extending the memory of the graph:

i. the locational relationship of each state to other states is explicitly defined, and

ii. at each point in time it is known exactly where the parsing flow is located.

However, the above approach could be considered as extreme. Certain recursions can be preserved and still have the above two advantages if the following rule is used: A recursion is eliminated by duplicating states if the state on which the recursion occurs belongs to two or more syntactical constructs.

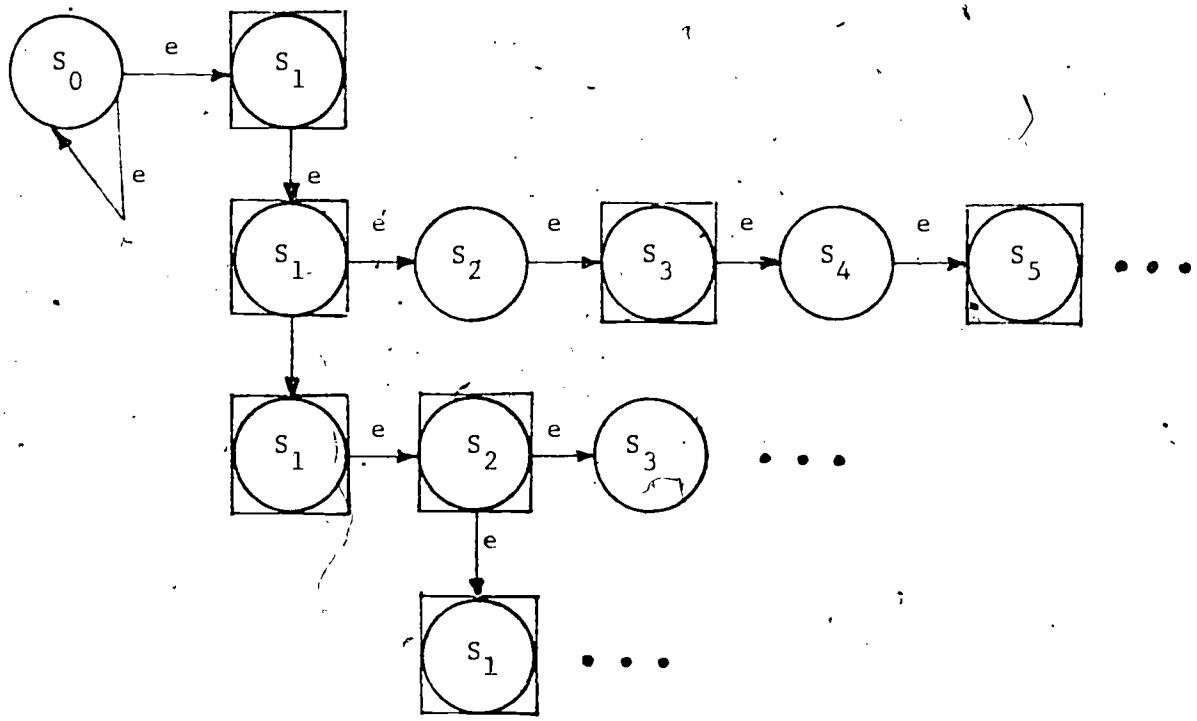
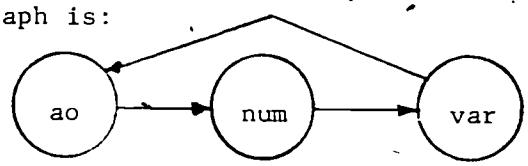


Figure 5. The Extreme Case of Memory Extension of the State Graph of Figure 4

Consdier the production:

<AE> :: = ao__num__var | ao__num__var__<AE>

The corresponding graph is:



The state at which the recursion occurs is (ao). However, it belongs only to the construct AE and the recursion should be preserved.

The above methodology is illustrated with a complete example. Figure 6 represents a UST grammar similar to the one presented at the beginning of this section. Figure 7 is the extended memory state graph of the same grammar, but with the recursions eliminated as needed. All the transitions are under the neutral symbol e.

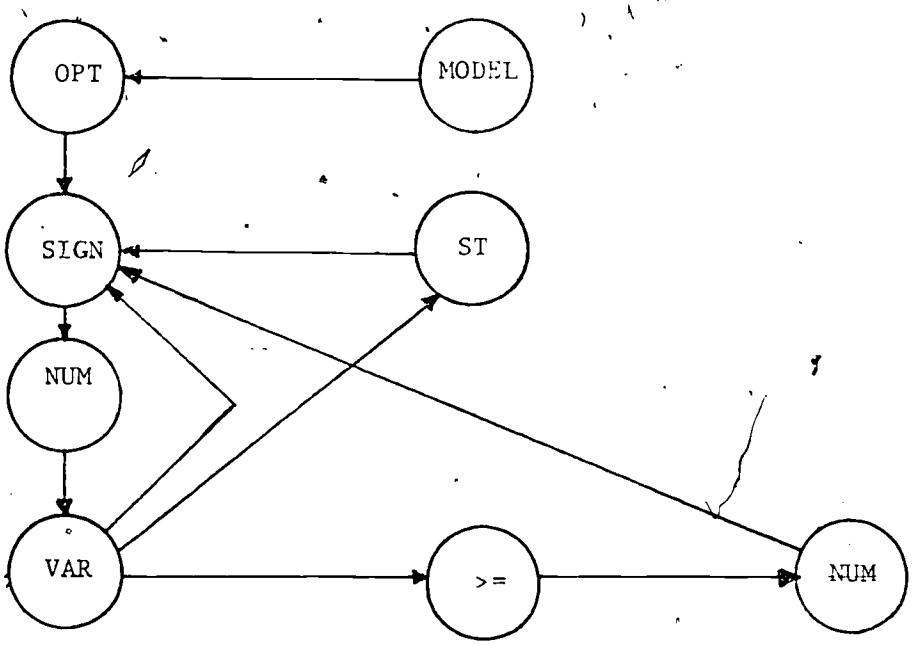


Figure 6. The State Graph of a Grammar of a Simple Linear Model. The memory of the graph has not been extended.

In the extended memory state graph, certain syntactical constructs are implicitly represented in the sense that no explicity state was required. These are AF, AE2, CONST. and CONST. SET.

The concept of extended memory can be viewed as an elementary learning process since the more input that is processed the more can be said about the following portion of the input. Hence, this learning increases from left to right. This, combined with the fact that there is prior restriction on the expected input (e.g., linear expressions in linear programming) provides the foundation for efficient error recovery and explicit diagnoistics. The efficiency of error recovery will similarly increase from left to right. This is important since the chance of an occurrence of a human error increases from left to right. An additional advantage of the state graph is that any changes in the grammar can be easily implemented by properly modifying the state graph.

The "break" symbol introduced in the state graphs for lexical analyzers is meaningless in the graphs for syntactical analyzers. In an actual design,

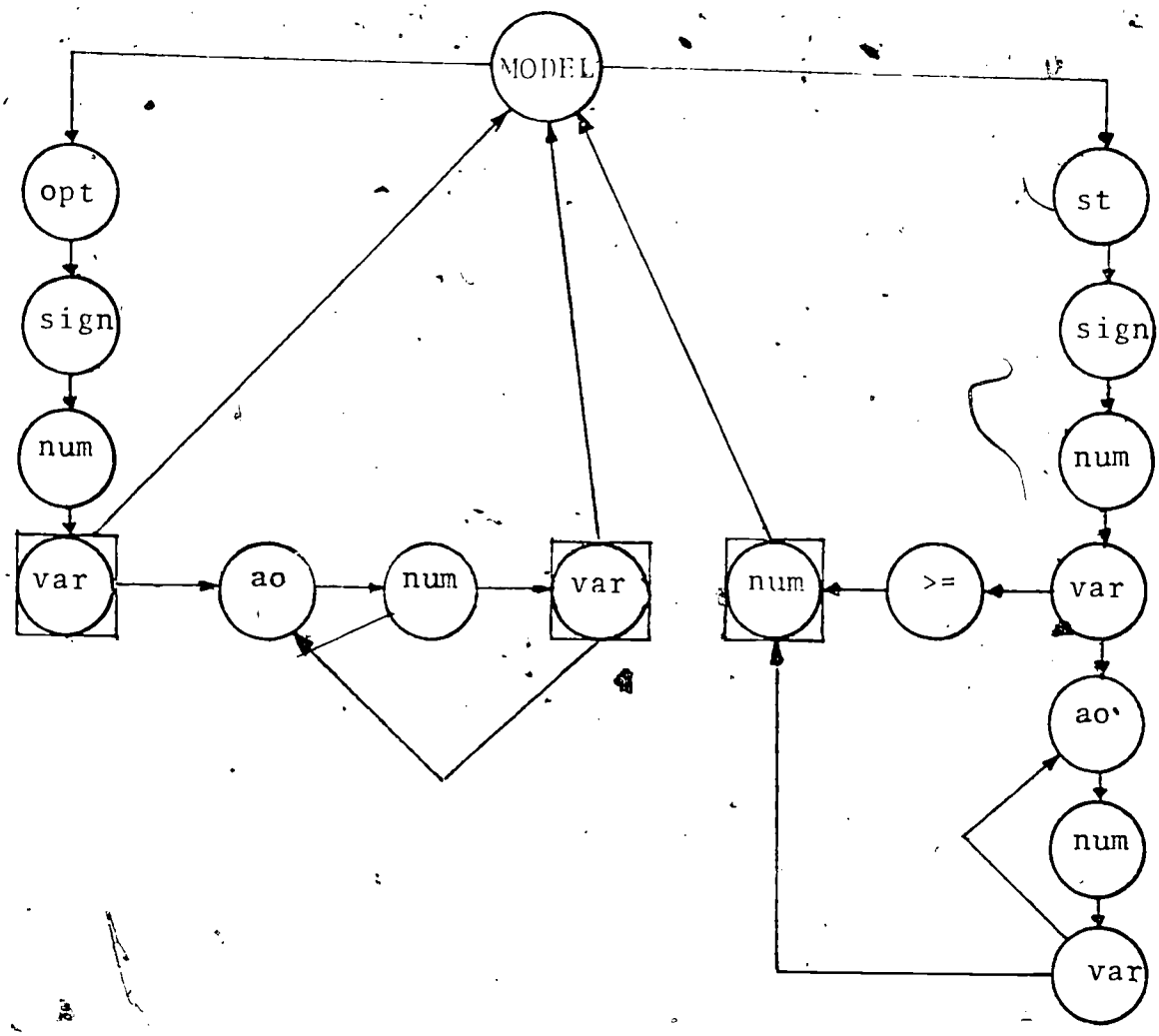


Figure 7. The Partially Extended-Memory State Graph of Figure 6.

each state of the graph should be represented by a separate set of computer language statements. This increases the modularity of the system. The length of the code and the coding process appears to be the only disadvantages of the graphical simulation. The advantages of the graphical representation are summarized below:

1. The memory can be expanded up to any desired degree.
2. Certain syntactical constructs are implicitly represented.
3. Efficiency exists with respect to:
   i. grammar expansion or reduction
   ii. error recovery
4. Explicit diagnostics exist.
5. Modular design exists.
6. No supporting tables are required.

Dynamic Syntax Adjustment

The extended memory state graph provides a means for an indirect semantical analysis. Given the type of a problem there exists some prior knowledge of its syntax at parsing time. It there is a way of explicitly specifying the type of the model, the system could properly adjust the state graph by deactivating certain states.

Let us consider the example of the transportation problem. The corresponding model is a linear one with the restriction that the matrix coefficients be 1 (zeros would not be entered). The proper adjustment of the graph should be the bypassing of the two nonterminal states -num- in the constraint section of Figure 7. If a coefficient different than the implicit 1 is entered, the system will detect this as a semantical error. In fact, the restriction of "implicit 1" could be relaxed at the lexical level by suppressing all the co-

efficients equal to 1. This concept could be implemented by representing the characteristics of a specific type of model by a set of active states. Hence, a group of types of linear problems (i.e., networks, machine scheduling, etc.) can be represented by a 0-1 matrix where the columns are the states.

A modular design of the system with the proper labeling for the transfer of the execution flow would facilitate the activation-deactivation of the states at a dynamic mode, i.e., at the end of the execution the graph could be reset to its initial configuration. In addition, the concept of dynamically adjusting the state graph of the system provides a means for detecting certain model characteristics. Given a set of active states for a specific type of model, and an initial configuration of the state graph, it would then be possible to determine the type of the model. Moreover, this could be expanded since the linear model, at least grammatically, is considered as a special case (i.e., a set of active states) of the nonlinear model.

Semantic Analysis

The semantic analysis which operates on the UST concentrates on the second entry of the table. The basic function of this analyzer is to assign the proper meaning to an entry and execute the appropriate information handling. For instance, if a numeric entry is located in front of a variable, it should be interpreted as a coefficient.

The coding of the semantic analyzer is usually an ad hoc procedure; most often it is implicitly included in the parser.

In languages for OR models, semantic analysis is important since it can detect certain irregularities before the optimization routine is activated and computer time is wasted. For instance, bound inconsistencies can be easily detected. However, the state of the art of semantic analysis does not provide a theory upon which a semantic analysis routine can be designed.

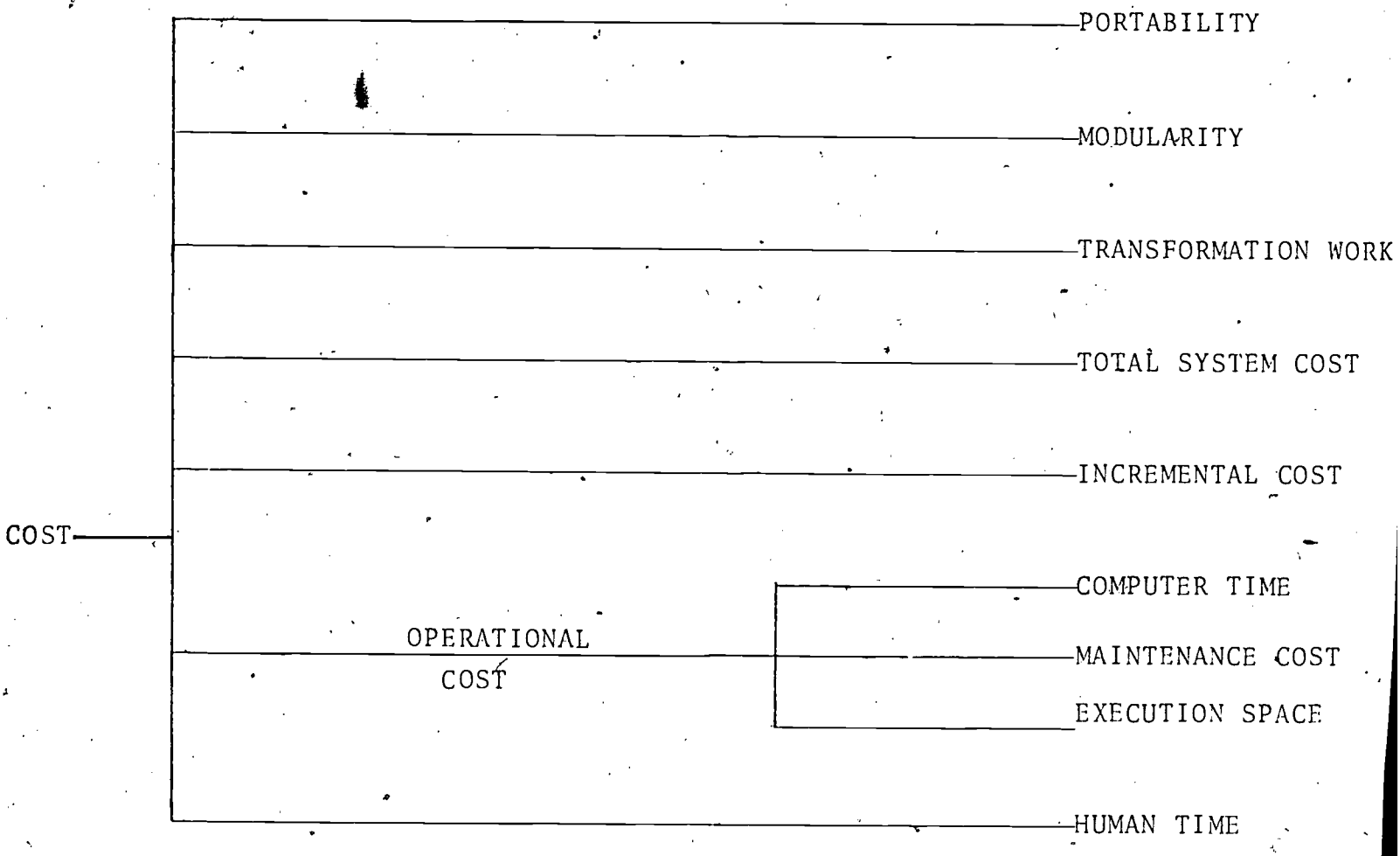


Figure 8. Cost Attributes of an Interfacing System

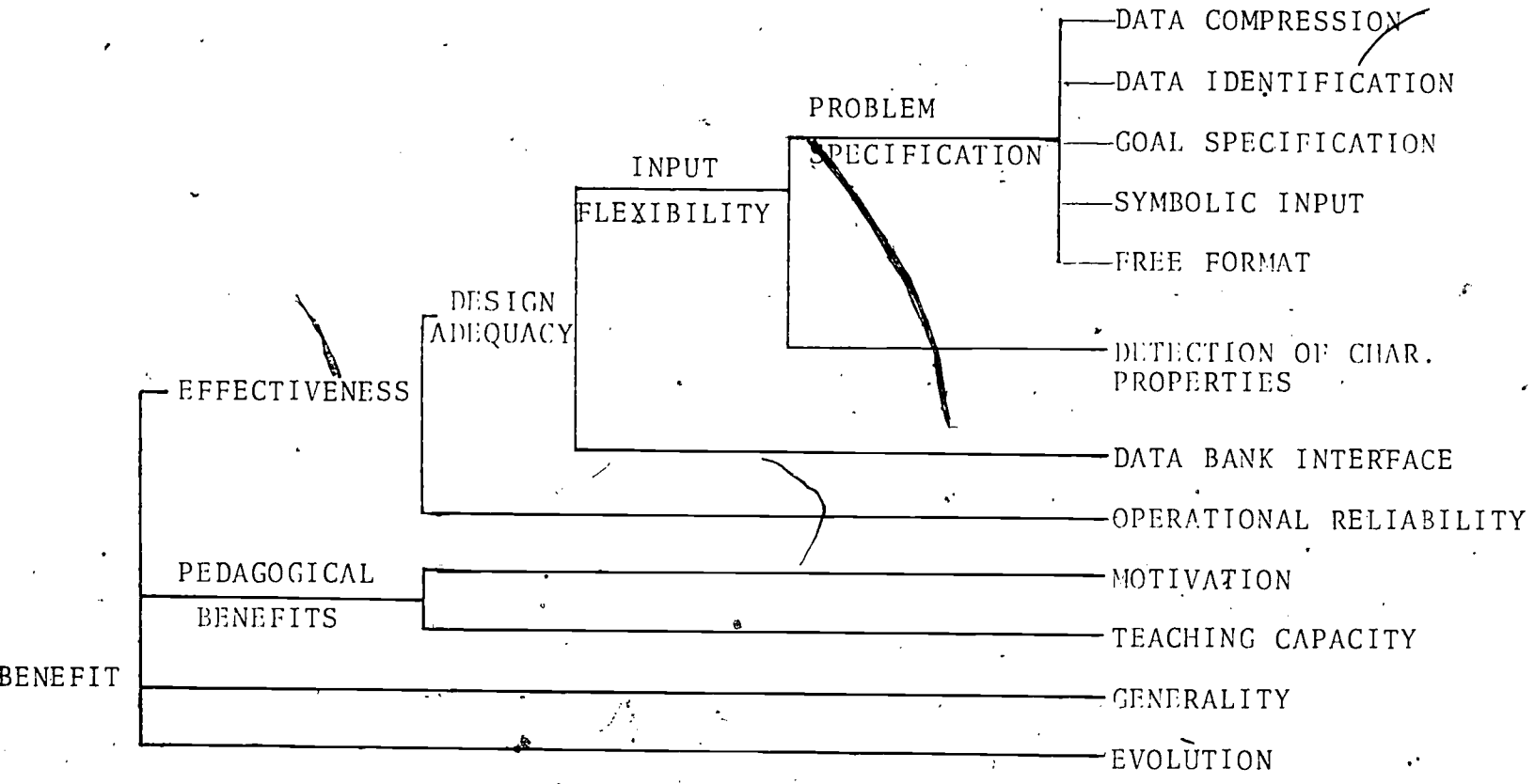


Figure 9. Benefit Attributes of an Interfacing System

ECONOMIC CONSIDERATIONS

A major difficulty in the area of software economic analysis is the lack of a closed-form metric system; Gilb [4] presents progress towards that direction and some of his metrics will be presented here.

With respect to the interfacing systems, the economic attributes can be divided into two general classes: cost and benefit. Figures 8 and 9 illustrate the breakdown of various attributes; the terminal attributes are the ones that have real economic meaning, and the nonterminal ones are used to illustrate the hierarchy and thus clarify the effect of the terminals on the system so that the analyst can estimate associated costs and benefits. This classification provides a cost/benefit decision framework. The decision, depending on the application environment, can be reached through an assignment of weights of importance to the attributes attaining a small cost/benefit ration, the design of the system can be scheduled according to the optimal weight distribution.

USER PSYCHOLOGY

User psychology is an important part of the design of an interfacing system, but because the mechanics of the system absorb most of the designer resources, little attendtion is paid to it. Concentrating exclusively on the mechanism of the interfacing can be viewed as only a partial design; a complete design should take into consideration the mechanics of the human element of the man-machine system. The objective is not to minimize the present worth of the total cost, but rather to maximize the present worth of the total gain.

User psychology considers the human interface with computer systems, its objective being to identify and analyze behavioral patterns that increase the effectiveness of the system. As such, it falls under human engineering and deals with rather complex environments.

User psychology provides the designer with information about the potential user, i.e., capabilities, intelligence, amount of time acceptable to learn a syntax, psychological effects of the design (motivation, boredom, etc,). The overall effectiveness of the system will be increased if this information is properly utilized during the designing process.

There are "intelligence gaps" in current generation computers, and this becomes more apparent in the problem solving area. A successful man-machine system should utilize human intelligence available on the terminal to fill the gaps of the machine. This would also substantially contribute to the effectiveness of the system. Hence, the interfacing system should have the capabilities of retrieving from the user not only information but intelligence as well. User psychology provides an insight in this area.

Martin [7] indicates that the user psychology should be studied at three levels: The first one is the functional considerations which studies the distribution of functions to the man and to the machine (i.e., which functions the user performs and which ones the machine performs). It is at this level that the user's intelligence will be utilized. An important consideration of this level is the ability of the interfacing system to retrieve human experience which cannot be quantified, and programmed explicitly in the machine. This is very important in the environment of operations research where one "channel of flow" of human experience and intelligence is the mathematical model. The capacity of this channel is increased if the interfacing system provides a high degree of input flexibility.

Human judgment and experience are further effectively utilized if the system provides editing capabilities. In the OR environment, it is a functional consideration of user psychology to decide whether the model will be formulated

by the user or by the machine. In other words, the user could enter an explicit formulation or certain key characteristics of this problem.

The second level is that of procedural considerations which organize the operation into a sequence of procedures, e.g., when is the model entered? When are error messages displayed? When can the user edit the input? These are the questions to be answered at the procedural level. A good design should avoid the "human channel overload," or user's boredom. On the other hand, attention should be paid to creating to the user motivation for using the system. For instance, motivation can be increased by a comprehensive output. With respect to user's boredom, an important procedural consideration is the flexibility of the system to accept input through alternative sequences of instruction. Experience indicates that the user becomes bored by a system that requires a fixed input sequence, especially if its length is not controlled by the user. He should be able to effectively utilize his knowledge of the system.

The third level is that of syntactical considerations, which primarily deals with the effect of the syntax on the user. Experimental psychology provides insight into how a particular syntactical feature is accepted by the user. The syntax can affect:

i. the time nexessary for a system to be learned, and

ii. the chance of an error occurrence.

Through statistical investigation, researchers have concluded that certain conventions of natural language should not be used. For instance, abbreviations as ISN'T or DON'T should be written as IS NOT or DO NOT. Also words such as ILLEGAL or UNKNOWN should be written as NOT LEGAL or NOT KNOWN.

To summarize, the designer should consider the following:

1. Minimize the time necessary for the user to become familiar with the system.
2. The design should promote user motivation for using the system.
3. The design should try to minimize the chance of user frustration due to diagnostics or other system behavior.
4. The human time on the keyboard should be minimized.

APPLICATIONS TO LINEAR PROGRAMMING

EZLP is an interactive computer system for linear programming problems which was developed in the School of Industrial and Systems Engineering of the Georgia Institute of Technology [6]. The goal of the system was to increase the compuater share in formulating and solving a linear programming problem. In that respect, EZLP is a step towards narrowing the gap between the OR analyst and the computer.

An informal characterization of the EZLP syntax is that it provides a free formation with respect to the mathematical model. Hence, all the restrictions of a typical LP system are relaxed. Usual restrictions are:

i. The linear expression must be entered before the relational operator. Usually, only a numeric entry is allowed to the right of the relational operator.

ii. The variables should be entered in the same sequence.

iii. The dimensions of the problem should be specified in advance.

EZLP relaxes the above restrictions and provides certain additional features as summarized below:

1. There is no restriction to the number of alternative objective functions.

2. Single variable constraints with the same right hand side

can be grouped into a single constraint (explicit list constraint) e.g., X1, X2 X2, X3 >= 10.

3. A bound can be assigned to all the variables e.g., ALL VARS >= 5.

4. A bound can be assigned to all the variables not assigned a bound up to the point of entrance of the constraint, e. g. ALL OTHER VARS >= 10.

5. Different bounds can be assigned to the same variable in different points in the model but the minimum upper bound and the maximum lower bound will prevail.

6. Arithmetic expressions can be bounded from above and below, for example, 5 = X1 + X2 + X3 <= 10

7. Arithmetic expressions can be entered in both sides of a relational operator, e.g., X1 + X2 <= 10 - X3.

8. There is no restriciton to the order of variables, e.g., X5 + X3 + X4 <= 10 - X2 + X1

9. Indexed variables are allowed, e.g., X1,2,3 X25,3

10. Numeric values can be entered in an arithmetic expression as single arithmetic entities. The summation of all these numerics will form the right hand side of the constraint, e.g., ROW3: 3 POWER - 2 + 6 HEAT + 10 <= 15 is equivalent to Row3: 3 POWER + 6 HEAT <= 7.

11. Prior knowledge of the size of the model is not required.

The design of the interface of EZLP was based on the methodology presented in the previous sections. The state graph approach was used to perform lexical and syntactical analysis. The complexity of the EZLP grammar was such that the design took advantage of all aspects of the state graph.

Semantical analysis, embedded in the process of syntactical analysis, is limited to checking the consistency of bounds of the variables and expressions.

EZLP detects most of the errors in a line. However, in certain instances, it stops at the first error, ignoring the remaining part of the input string. This occurs when it is suspected that several artificial errors would be generated by error propagation should the scanning process continue. The state graph provides the facility of pointing the exact error point with a self-explanatory diagnostic message.

The following is the EZLP nonlinear grammar in BNF:

Notation:

IMPL. LIST CONS.	denotes	Implicit List Constraints
EXPL. LIST CONS.	denotes	Explicit List Constraints
ALT. OBJ. FCN	denotes	Alternative Objective Function
AE	denotes	Arithmetic Expression
INT. AE	denotes	Internal Arithmetic Expression
REL. OP	denotes	Relational Operator
AO	denotes	Arithmetic Operator
NUM	denotes	Numeric
VAR	denotes	Variable
SPECS	denotes	Specifications
COEF	denotes	Coefficient
LP	denotes	Linear Programming

Definitions:

<LP MODEL> :: = <OPT><LINE NAME><AE><MODEL SPECS>

<MODEL SPECS> :: = <MODEL ENTRY>|<MODEL ENTRY><MODEL SPECS>

<MODEL ENTRY> :: = <ARITHM. CONSTRAINT>|<LIST CONSTRAINT>|
<ALT. OBJ. FUN>

<LIST CONSTRAINT> :: = <EXPL. LIST CONS>|<IMPL. LIST CONS>

<ARITHM. CONSTRAINT> :: = AND <LINE NAME><AE><REL. OP><AE>|
AND <LINE NAME><NUM><DEL. OP.><AE>
<REL. OP>

<ALT. OBJ. FUN> :: = ALSO <LINE NAME><AE>

<EXPL. LIST CONS> :: = AND <LINE NAME><NUM><REL. OP.><VAR LIST>|
AND <LINE NAME><VAR LIST><REL. OP><NUM>|
AND <LINE NAME><NUM><REL. OP><VAR. LIST><REL. OP.><NUM>

<IMPL. LIST CONS.> :: = AND <LINE NAME> ALL <OPTION 1> VARS
<OPTION 2>

<OPTION 1> :: = OTHER|null

<OPTION 2> :: =<REL. OP><NUM>|URS

<AE> :: = <SIGN><TERM>|<SIGN><TERM><INT. AE>

<INT. AE> :: =<AO><TERM>|<AO><TERM><INT. AE>

<TERM> :: = <COEF><VAR>|<NUM>

<VAR LIST> :: = <VAR>|<VAR >, <VAR LIST>

<VAR> :: = <LETTER>|<LETTER><VAR STRING>

<VAR STRING> :: = <PREFIX STRING>|<PREFIX STRING><VAR STRING>

<PREFIX STRING> :: = <DIGIT>|<LETTER><INDEX>

<INDEX> :: = <DIGIT>, <DIGIT>|<INDEX>, <DIGIT>

<LINE NAME> :: = <VAR>| Null

<COEF> :: = <NUM>|Null

<NUM> :: = <INTEGER>|<INTEGER>|<INTEGER>.|

<INTEGER>.<INTEGER>

<INTEGER> :: = <DIGIT>|<DIGIT><INTEGER>

<DIGIT> :: = 0|1|2| --- |9

<LETTER> :: = A|B| --- |Z

<OPT> :: = MAX | MIN | MAXIMIZE | MINIMIZE

<SIGN> :: = + | - | Null

<AO> :: = + | -

<REL. OP> :: = <= | = |> = | =< | = >

The above grammar after the lexical analysis is transformed to the following UST grammar.

<LP MODEL> :: = Opt. - Line name - <AE><MODEL SPECS>

<MODEL SPECS> :: = <MODEL ENTRY>|<MODEL ENTRY><MODEL SPECS>

<MODEL ENTRY> :: = <ARITHM. CONSTRAINT>|<LIST CONSTRAINT>|

<ALT. OBJ. FUN>

<LIST CONSTRAINT> :: = <EX PL. LIST CONS.><IMPL. LIST CONS.>

<ARITHM. CONSTRAINT> :: = AND - Line name - <AE> Rel. op - <AE>|

AND - Line name - num - rel. op -

<AE> - rel. op. - num

<ALT. OBJ. FUN> :: = ALSO - line name - <AE>

<EXPL. LIST CONS.> :: = AND - line name - num - rel. op -

<VAR LIST>|

AND - line name - <VAR LIST> -

Rel. op. - num |

AND - line name - num - rel. op. -

<VAR LIST>| - rel. op - num

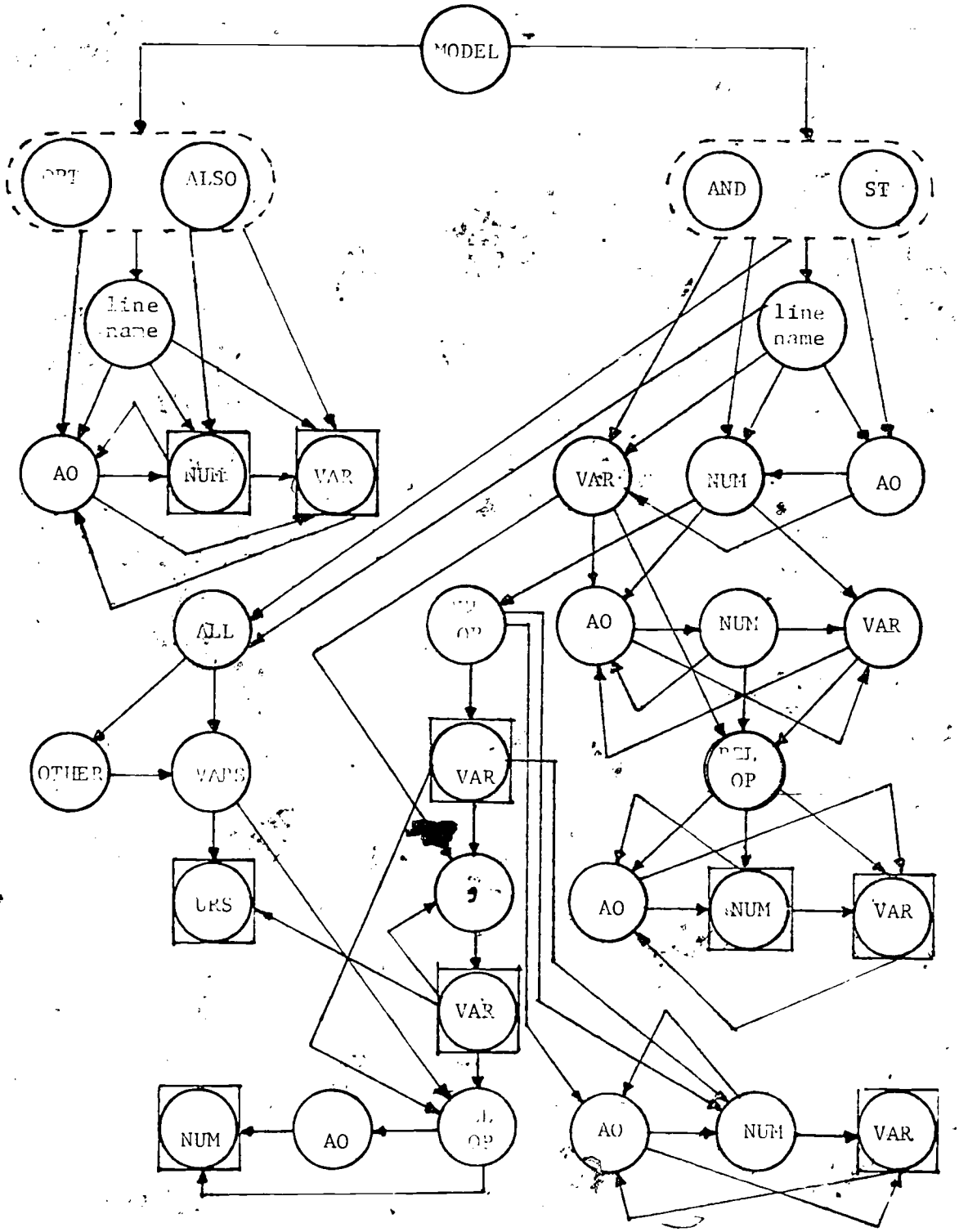


Figure 10. The State Graph of the HELP Grammar

<IMPL. LIST CONS.> :: = AND - line name - ALL - <OPTION 1>. - VARS - <OPTION 2>

<OPTION 1> :: = OTHER | Null

<OPTION 2> :: = rel. op - num URS

<AE> :: = Sign - Term | Sign - <TERM><INT. AE>

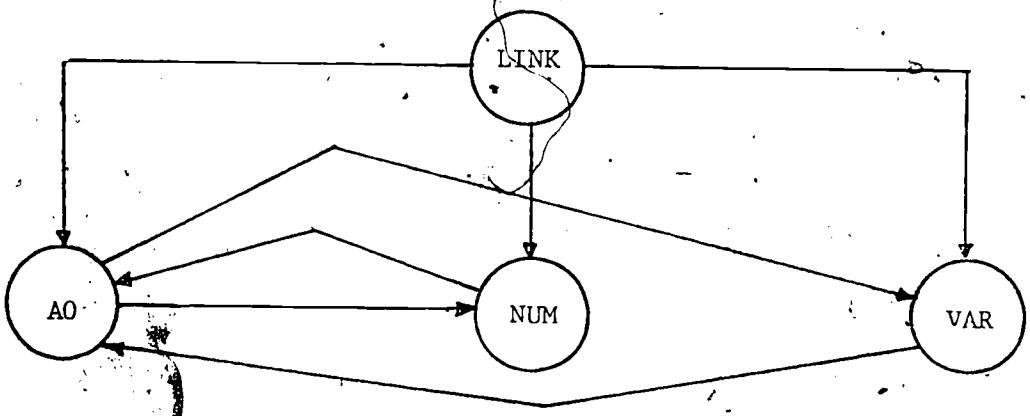
<TERM> :: = Coef - Var | num

<VAR LIST> :: = Var | Var, - <VAR LIST>

The state graph corresponding to the above grammar is shown in Figure 10. In this graph, certain nonterminals are represented implicitly, i.e., by a group of states. These are all the nonterminals at the left of the definitions.

The terminals - sign - and - coef - are omitted since their difference from their counterparts - ao - and - num - is the null element.

The EZLP grammar could be dynamically adjusted at low cost to extend its capabilities. Special forms of linear programming could be inputted through the EZLP interface with the error recovery mechanism reflecting their special characteristics for instance, if EZLP were to be adjusted to handle only models with 0-1 coefficients, the only change should be to redefine TERM as <TERM> :: = <VAR>|<NUM>. In the state graph, this translates into eliminating the arcs leading from the state num to the state var. The generic state graph of AE (Arithmetic Expression) should be as in Figure 11.



The State Graph of an Arithmetic Expression with 0-1 Coefficients

Aspects of user psychology were considered in the design of EZLP. The high flexibility of input reduces the user's boredom and increase his likelihood of using the system, especially in the academic environment. The capacity of the user's attention channel has been increased by increasing the information capacity of the EZLP entries. Constraints as " AND X1, X2, X3, >= 10" or "AND ALL OTHER VARS <= 5" are examples of entries with increased information capacity, for the same reason, the capacity of the user's short term memory has been increased. As a result, human judgment can be exercised in a more effective way in the process of formulating or editing the model.

By comparing EZLP to the other existing interfacing systems for LP it is apparent that there are mainly two advances: the elevation of the communication closer to the user's thinking level, and a higher rate of communication. The first can be considered as a contribution to the task of "retrieving user's experience" and hence a more effective man-machine system. The second is a major economic consideration affecting several economic attributes.

EZLP and its variations represent a man-machine communication at a fixed level of information compression. This implies that the rate of communication of the EZLP variation will be approximately the same, i.e., a grammar transformation will not substantially affect the rate of communication. On the other hand, it may affect the "experience retrieval."

To substantially increase the rate of communication in the linear environment, the system should provide facilities of symbolic input of higher information capacity. For instance, the arithmetic expression

MAX C1X1 + C2X2 + ... + C10X10 could be inputted as:

MAX SUM CIXI I = 1,10 or

MAX SUM CIXI I = 1,2,3,...,10

Note the simplicity of the grammatical definitions:

<OBJ FUN> :: = <OPT> SUM <COEF> X <INDEX><INDEX> = <NUM>, <NUM>

<COEF> :: = <INDEX>|Null

These facilities, if carefully designed considering the potential user psychology, may substantially affect cost effectiveness of the system.

Nonlinear Programming

The concepts discussed in the previous chapters apply to processing nonlinear models. Once the features of the system have been decided, the grammar should be specified in an efficient metalanguage. This specification for nonlinearity would be similar to the general linear model specification with the addition of nonlinear arithmetic expressions. As an example, the grammar of EZLP of the previous section, can be easily converted to a grammar for nonlinear input. All the features of EZLP would be preserved if AE, INT. AE and TERM are the only redefined nonterminals. The new definition should be a broader one able to accept nonlinear entries as well as linear. This could be accomplished by the following definitions:

<AE> :: = <SIGN><TERM>|<SIGN><TERM><INT. AE>

<SIGN><NUM>|<SIGN><NUM><TERM><INT. AE>

<INT. AE> :: = <AO><TERM>|<AO><NUM>|<AO><TERM><INT. AE>

<AO><NUM><TERM><INT. AE>

<TERM> :: = <VAR><EXP>|<VAR><TERM>

<EXP> :: =|↑<NUM>|NULL

With the above definitions the system would be aboe to accept an input as:

Max X + X↑2 - 2 X YZ + 15 - XZ ↑ 2W

Lexical analysis would transform the above definition to the following UST

definitions:

<AE> :: = Sign - <TERM>| Sign - <TERM><INT. AE>

Sign - Num | Sign - Num - <TERM><INT. AE>

<INT. AE> :: = AO - <TERM>| AO - Num |

AO - <TERM>| AO - Num - <TERM><INT AE>

AO - <TERM><INT. AE>

<TERM> :: = Var - ↑ - Num Var - <TERM>

The above translates to the following state graph of Figure 12.

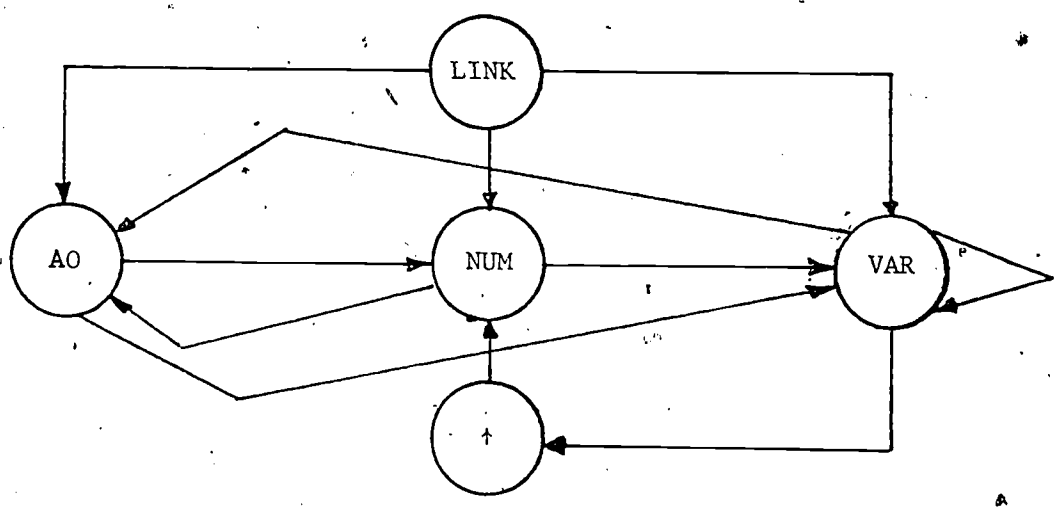


Figure 12. The State Graph of a Nonlinear Arithmetic Expression

Nonlinear programming, with exceptions of some special forms as quadratic programming, is based on function evaluation and search methods rather than on a closed form matrix dirven methodology. The most efficient way of handling this situation is to simply append the nonlinear expressions to a program written in a high level, general purpose language. The associated compiler would check the syntax of the input. So the syntax of a high level, well-known language

would be the primary part of the syntax of the interfacing system. This implies that the potential user would, very possibly, be required to become familiar only with a few key words of the system.

CONCLUSIONS

The goal of this research was to develop a cost-effective methodology for inputting mathematical models of Operations Research into the computer. Elements of mathematical linguistics provided a framework for this methodology. Through grammatical specifications, the designer can visualize the complexity of the system and, hence, estimate the associated development, operational and maintenance "costs." Through grammatical transformations, these costs can be altered and the system, during the analysis stage, can reach a cost-effective state. Given the final form of the grammar, the parsing algorithm can be designed as a flow through legal syntactical states. This leads to the state graph approach.

The above methodology can be seen as a systematic step-by-step transformation of the system features expressed in English to a highly modular code. The most important advantage of this feature is that it provides the basis for a system whose syntax can be altered at input time.

REFERENCES

1. Aho, A. V. and J. D. Ullman, The Theory of Parsing, Translation, and Compiling, V. 1 -- Parsing, Prentice-Hall, Inc., 1972.

2. Cohen, C., A Guide to MPOS Version 2, Multi-Purpose Optimization System, Vogelback Computer Center, Northwestern University, 1975.

3. Donaghey, P. Sewan, D. Singh, A Beginner's Language for LP, Industrial Engineering, p. 17, 1970.

4. Gilb, Tom, Software Metrics, Winthrod Computer Series, 1977.

5. Gross, M. and A. Lentin, Introduction to Formal Grammars, Spring and Verlga, 1967.

6. Jarvis, J. J., F. H. Cullen, C. Papaconstadopoulos, EZLP: An Interactive Computer Program for Solving Linear Programming Problems, School of Industrial and Systems Engineering, Georgia Institute of Technology, 1976.

7. Martin, J., Design of Man-Computer Dialogues, Prentice-Hall, Inc., 1973.

8. Wagner, G. R. and M. M. McCants, Conversational Linear Programming for Experimental Learning, Engineering Education, Vol. 62, No. 7, 1972.

9. Wall, R., Introduction to Mathematical Linguistics, Prentice-Hall, Inc., 1972.