The Relevance of Some Compiler Construction Techniques to the Description and Translation Of Languages

by

Steven I. Laszlo

Western Union Telegraph Co.*

The framework is machine-translation. Compiler-building can for a variety of reasons be considered as a special case of machine-translation. It is the purpose of this paper to explicate some techniques used in compiler-building, and to relate these to linguistic theory and to the practice of machine-translation.

The generally observed machine-translation procedure could be schematized as in FIGURE 1, or to put it another way,

1. Parsing the source-text.
2. Translation from source to object-language.
3. Synthesis of grammatically correct object-text.

FIGURE 1.

break-down, translation, and recomposition. The translation usually occurs on the level of some simplified, cannonical form (that is not necessarily the kernel-form) of both languages, such that the source-text is decomposed, and the object-text recomposed from this form. The translation algorithm usually requires a statement of the structure of both the source and the object-language, as well as the statement of some primitive-to-primitive

* Currently at Decision Systems, Inc.

correspondence paradigm for both syntactic and lexical primitives. Compilers on the other hand work on the bases of only the first two steps of FIGURE 1.: breakdown, and translation. Consequently, the processor requires only statements of the structure of the source-language and of the correspondence paradigm. That does not imply that the structure of the object-language is irrelevant to the process of translation, but that it is implicit in the correspondence paradigm, and in the selection of what is a primitive or terminal in the description of the source-language.

Through the use of examples it will be shown that BNF and similar language-description devices (8) are -- by themselves -- both analytically and generatively inadequate and depend on other devices, implicit in the translation algorithm. It will be shown that by some extensions of the notion of P-rules and some applications of the concept of T-rules (4), a description that is both analytically and generatively adequate may be constructed for programming languages. The programming language P.O.L.2 (12), (13) was selected for the examples because an adequate, fully explicit description does exist for it; furthermore, the language contains most syntactically problematic features of other programming languages as well as presenting a few unique problems in description that are worthy of attention.

The failure to come to grips with the identity problem is sufficient to demonstrate the inadequacy of BNF and similar devices (8). The simplified program-segments in FIGURE 2. serve to illustrate

EXAMPLE 1.

```
1.  Let A be variable.
2.  Let B be = "7".
3.  Let C be = "9.5".
4.  Let D be = ".072".
5.  A = B + C / D.
6.  Print A.
```

EXAMPLE 2.

```
Define Funct (A, B) = (C).
        .
        .
        .
   End.
      ---and elsewhere---
 Funct (Q, R) = (Z).
 V = D + K * Funct (P, T).
```

FIGURE 2.

this problem. BNF and similar devices would generate a parse designating "A, "B", etc. in EXAMPLE 1. as identifier (a syntactic word-class) but would fail to indicate that the various occurrences of a given identifier (e.g., "A" in statements 1., 5., and 6.) are that of the same lexical token or semantic object. Related to the identity problem is the restriction that each identifier occurring in a program statement must also occur in one and only one definition. This restriction may be called the definition problem. BNF, etc., do not handle the definition problem. Other manifestations of the identity and definition problems are associated with the use of macro- or compound functions (see EXAMPLE 2., FIGURE 2.), subscript expressions, etc.

Since there exists a demonstrable necessity for establishing the above mentioned identities and restriction (3), compilers contain -- implicit in the translation algorithm -- an elaborate table-building/table-searching/identity-testing procedure. Without such procedures, the syntactic description is inadequate, full analysis and translation impossible. In order to deal with these problems explicitly, it was decided to incorporate a transformational component along with the BNF-like phrase-structure component in the description of P.O.L.2. The above reasons for positing a transformational component are in essence the programming-language equivalents of Chomsky's original reasons to use transformations in the description of natural languages.

Rule 1. $M \rightarrow \#, M,_{sel.\,1}, \#$

where "M" is the initial symbol, "#" is the boundary marker, and the subscript will be explained later.

Rule 2. $M_{sel.\,1} \rightarrow$ DEFINE, functmention, program, END

where the convention is used that terminal symbols are all capital letters, and members of the intermediate alphabet are in lower case.

Rule 3. program $\rightarrow$ . . ., placeholder, M, . . .

FIGURE 3.

In FIGURE 3., in a simplified form it is shown that the phrase-structure component generates function definitions (17), (18) embedded in others (see Rule 3.), and that the form of the function is generated in the definition -- as the expansion of the symbol

"functmention" -- generating place-holders for instances of use of the function. Transformations replace the place-holders with the appropriate form of the function generated in the definition, thus accounting for both the identity and the definition problems. Other transformations exist to handle other instances of these problems e.g., labels, identifiers, subscript expressions. The method is identical: the form is generated in the relevant definition, place-holders are generated for instances of use, and the place-holders are replaced transformationally with the correct form generated in the definition.

Other transformations deal with additional notational restrictions of P.O.L.2. One such restriction is that a function definition may reference other functions but a definition may not be embedded in another. Definitions (see FIGURE 3.) are in fact generated embedded, and it becomes necessary to posit some exbedding transformation (7), moving the nested definitions outside the "parent" definition. There exist several proofs in the literature establishing the equivalence between languages generated by grammars with and without the use of boundary markers (5), (10). The exbedding transformation may be expressed more simply if boundary markers are used (see FIGURE 4.).

#, . . . , #, M, #, . . . , # => #, M, #, #, . . . , #

or

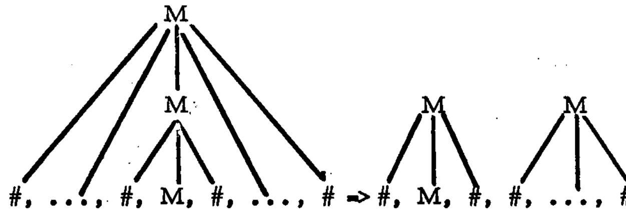


FIGURE 4.

The boundary-markers may be deleted later by another transformation, or they may rewrite as carriage-returns on some keyboard, depending on the orthography of the particular implementation and medium. The T-rules may be generated by positing a set of elementary transformations (i.e., single node operations) and a set of formation and combination rules over the set of elementary transformations, producing some set of compound or complex transformations. This is not significantly different from having locally ordered subsets of a set of elementary

transformations (11), (12).

Syntactic descriptions of programming languages published in the past -- e.g., (1), (9), (19) -- generally took a program-statement to correspond to the basic unit of grammar, denoted by the initial symbol of the phrase-structure grammar. The grammer discussed here takes a function definition (see FIGURE 3.) as its basic unit. Program-statements are elements of the intermediate alphabet and have no other theoretical standing or significance. The natural language correlates of program-statements are sentences, and function definitions correspond to some larger-than-sentence units of discourse (e.g., paragraphs or chapters). This procedure may lead to some syntactic or at least linguistic method of distinguishing between "meaningful" and "meaningless" programs. Using a syntax of programs, or functions also yields an intuitively more pleasing set of relationships among elements of the described language.

The present grammar makes no effort to distinguish between "elegant" and inelegant" programming, but does distinguish both from "ungrammatical" code. Declaring arguments or variables never referenced is inelegant; referencing undeclared operands is ungrammatical. To return momentarily to the identity and definition problems: it is possible to generate a definition such that there are no corresponding place-holders; but each place-holder must be replaced by some definition-generated form of the appropriate nature. In describing the definition and use of functions, separate place-holders accomodate recursive use and the general case of usage.

It is customary to give descriptions of programming languages such that -- with the exception of some small set of key words such as arithmetic operators, delimiters of definitions, etc. -- the phrase-structure grammar generates character-strings for the lexical items. In natural languages the vocabulary is fixed. There is a stable, limited set of vocabulary elements that correspond to each syntactic word-class. In programming languages that is not the case: a small set of word-classes rewrite each as a set of one or more key-words; others will expand -- through the use of some phrase-structure rules -- as any string of characters. In the description of P. O. L. 2 it was decided to separate the lexicon-generation rules from the phrase-structure rules. Though they are the same shape that BNF rules of the same purpose would be, it was determined that separating the rules generating lexical items -- even as morphophonemic rules of natural languages represent a separate class of rules -- is more intuitively acceptable: a class of orthographic rules. FIGURE 5. indicates what some of these rules might look like.

In the text of FIGURE 3., Rule 1., the explanation of the subscript was deferred. Functions and operators used in programming languages are two notational variants of the same concept (17). Depending on the notation of the system, any operation may be expressed either as an operator or a function. Since in

Rule 1. identifier → alpha <, characterstring>

where "< ... >" enclose optional items.

Rule 2. characterstring → $\left\{\begin{matrix}\text{alpha}\\ \text{numeral}\end{matrix}\right\}$ <, characterstring>

where "$\{\dots\}$" enclose alternative options such that one and only one of the options enumerated must be selected.

Rule 3. alpha → $\left\{\begin{matrix}A\\ B\\ .\\ .\\ .\\ Z\end{matrix}\right\}$

Rule 4. numeral → $\left\{\begin{matrix}1\\ 2\\ .\\ .\\ 0\end{matrix}\right\}$

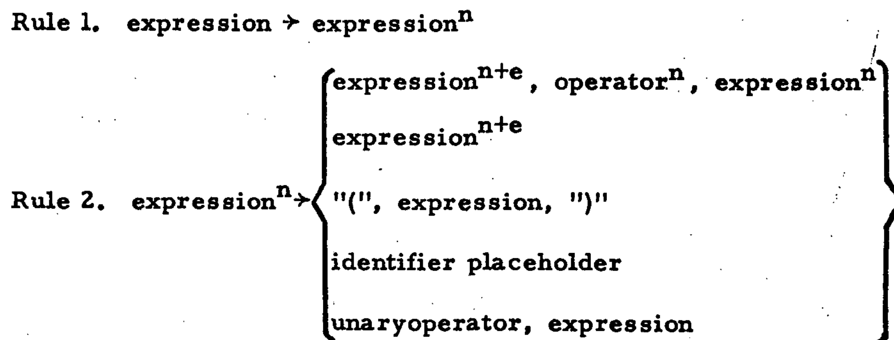
FIGURE 5.

P. O. L. 2 there are both functions and operators, depending on notational convenience, newly defined operations may be defined as either. Being defined as one or the other, however, restricts their distribution or "embeddability" to certain contexts. This phenomenon is accounted for by the use of a device similar to the notation of complex symbol theory (4), (11), (12), (15). The P. O. L. 2 notation is such that functions (i. e., defined macros) may occur as functions, coordinate transformations (linear or otherwise) or as operands (denoting their value for a particular set of arguments) and operators may appear as arithmetic, relational or logical operators, depending on range and/or domain as well as distributional restrictions. In P. O. L. 2 every program - however simple or complex -- must have an "outermost" function, one into which all others are embedded by the P-rules. The first rule of the grammar (see FIGURE 3., Rule 1.) expands the "outermost" function. Elsewhere in the phrase-structure component, depending on context, other.

"$M_{sel.1}$s" are introduced, as well as "$M_{sel.2}$s", "$M_{sel.3}$s", "$M_{sel.4}$s", and "$M_{sel.5}$s".

These correspond to the various embedded occurrences of functions and operators. The rewrites or expansions of the several versions of "M" are almost identical except for the string denoting the left bracket delimiting the definition. Alternative solutions exist but the above one appears most intuitively satisfying.

There are proofs and demonstrations in the literature to the effect that full, left, or right parenthesis notation is context-free, but not much on elided parenthesis notation. We have in the past constructed several context-sensitive grammars generating elided parenthesis notation, but they did not seem very satisfactory. Adding a device not heretofore associated with production-rules, a set of rules was produced to generate the elided parentheses notation such that the rules look and process very much like context-free rules (see FIGURE 6.).



where for one cycle (11) $\underline{n}$ remains the same integer between subrules 1 and 2 and $\underline{e}$ remains the same integer increment.

FIGURE 6.

Though the "counter" $\underline{n}$ and the "increment" $\underline{e}$ are not part of a known system of production rules, their nature and the reason for their use can be clearly stated. Their use permits a simpler scanner for the syntax than context-restricted rules do.

A similar counter is used to handle the concatenations of n-tuples.. In P. O. L. 2 an item of data may be declared as a pair, triple, or n-tuple, and operations may be performed over n-tuples of identical $\underline{n}$-s (see FIGURE 7.).

Rule 1. n-tuple-expression → n-tuple, operator, n-tuple

where n = n = n. Any of the n- tuples may however be concatenates of two or more n-tuples of smaller n-s such that:

Rule 2. n-tuple → (m) - tuple, concatenator, (n-m)-tuple

where n and m are positive integers and the arithmetic relationship designated obtains.

FIGURE 7.

Of course, the (m)-tuple or the (n-m)-tuple may be further broken down by the same rule into further concatenates.

The above are selected examples rather than an exhaustive list of the transformations in the syntax of P.O.L.2. A rigorous statement of the transformations is available, stated as mappings of structural descriptions into structural descriptions, accounting for the attachment and detachment of nodes. Presenting the selection of transformations here in a descriptive rather than a rigorous form offers an idea of the general approach.

Constructing the phrase structure component, many alternative solutions or approaches came up at every juncture; in specifying the transformational component, the alternatives quickly multiplied beyond manageable proportions. It is certainly the case that throughout its brief but exciting history, one of the aims of transformational theory has been to describe language in terms of the most restricted -- hence simplest -- system possible. But one may well regard the sets of devices sofar advanced as parts of transformational theory, as algorithmic alphabets (in the A.A. Markov/Martin Davis (5), (15) sense). Specific algorithmic alphabets are more or less arbitrary selections from some universe of elementary and compound algorithms bound by formation and combination rules. This paper is not a proposal toward the modification, extension or restriction of transformational theory, merely an indication that an overlapping set of algorithms may be selected to deal with a similar but not identical problem: the structural description of some formal notation systems such as programming languages.

Beyond doubt, substantial simplification and sophistication may be achieved over the model described here. The effort here has been toward the application of linguistic techniques to artificial languages, conforming to the linguist's notion of what it means to "give an account of the data", rather than to the laxer standards of the methods used to describe programming languages.

BIBLIOGRAPHY:

1. Backus, J.W. "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference", Information Processing; Proceedings of the International Conference on Information Processing. Paris: UNESCO, 1960.
2. Cheatham, Jr., T.E. The Introduction of Definitional Facilities into Higher Level Programming Languages. Draft Report CA-6605-0611.; Wakefield, Mass.: Computer Associates, Inc., 1966.
3. ________ The Theory and Construction of Compilers Draft Report CA-6606-0111.; Wakefield, Mass.: Computer Associates, Inc., 1966.
4. Chomsky, Noam. Aspects of the Theory of Syntax. Cambridge, Mass.: MIT Press, 1965.
5. ________ "On Certain Formal Properties of Grammars", Information and Control, 2, (1959), pp. 137-167.
6. Davis, Martin. Computability and Unsolvability. New York; McGraw-Hill, 1958.
7. Filmore. C.J. "The Position of Embedding Transformations in a Grammar", Word, 19, 2, (1963).
8. Gorn, Saul. "Specification Languages for Mechanical Languages and their Processors -- A Baker's Dozen", Communications of the ACM, 7, 12, (1961).
9. Heising, W.P. "History and Summary of FORTRAN Standardization Development for the ASA", Communications of the ACM, 7, 10, (1964).
10. Landweber, P.S. "Three Theorems on Phrase Structure Grammars of Type 1", Information and Control, 6, (1963), pp. 131-136.
11. Lakoff, G.P. Cycles and Complex Symbols in English Syntax. Unpublished Manuscript, Indiana University, 1963.
12. ________ Some Constraints On Transformations. Unpublished manuscript, Indiana University, 1964.
13. Laszlo, S.I. "Report on a Proposed General Purpose Procedure Oriented Computer Programming Language". Report of the Institute of Educational Research, Bloomington: Indiana University, 1965.
14. ________ "P.O.L., A General Purpose, Procedure Oriented Computer Programming Language" Report of the Institute of Educational Research, Bloomington: Indianan University, 1965.
15. Matthews, P.H., "Problems of Selection in Transformational Grammar", Journal of Linguistics, 1, (1965).
16. Markov, A.A., Theory of Algorithms. Washington, D.C.: U.S. Printing Office, 1965.
17. McCarthy, John "A Basis for a Mathematical Theory of Computation", in Computer Programming and Formal Systems. P. Braffort & D. Hirschberg (ed.), Amsterdam: N. Holland

Publishing Co. 1963.
18. ____________ et al., LISP 1.5 Programmer's Manual; Cambridge, Mass.: MIT Press, 1962.
19. Naur, Peter (ed.) "Revised Report on the Algorithmic Language ALGOL 60", Communications of the ACM, reprinted in E.W. Dijkstra, A Primer of ALGOL Programming. New York: Academic Press, 1964.