Compiler generation using formal specification of procedure-oriented and machine languages

by PHILIP GILBERT
Measurement Analysis Corporation
Los Angeles, California

and

WILLIAM G. McLELLAN
Rome Air Development Center
Griffiss Air Force Base, New York

INTRODUCTION
This paper reports on a recently developed compiler generation system which is rigorously based, and which allows formal specification both of source (procedure-oriented) languages (POLs) and of machine languages (MLs). Concepts underlying the system are discussed, an example correlating source language specification with system operation is given, and the status and potentialities of the system are discussed.

The crucial problem of compiler generation is the characterization of procedure-oriented languages; the process is of limited use unless such characterization allows machine-independent processing of programs in these languages (and hence allows invariance of the language itself from machine to machine). Our solution interposes between POL and ML a "buffer" or "intermediate" language, called BASE, thus reducing the required POL→ML transformation to two logically independent subtransformations:

(1) POL→BASE (called compilation)
(2) BASE→ML (called translation)

This arrangement isolates questions of POL characterization within the first transformation, and questions of ML characterization within the second transformation. BASE itself is an expandable set of non-machine-specific operators,* declarators, etc., expressed in a uniform "functional" or "macro" notation; the meaning or intent of such operators is arbitrary insofar as the compilation transformation is concerned. The POL→BASE transformation may then be regarded as a machine-independent conversion, from a grammatically rich format to a simple linear format.

Theoretical basis
Within our system, a POL is characterized principally by a grammar (i.e., set of syntactic productions), and the consequent processing of programs in the POL is syntax-driven. To assure adequacy with respect to completeness, ambiguity, and finiteness of analysis, our syntactic method is rigorously based. A grammatical model (the analytic grammar) was developed, which provides a rigorous description of syntactic analysis via formalization of the notion of a scan. Within this model, the selection process of a scanning procedure can be precisely stated, and thus made amenable to theoretical investigation. Some characteristics of this model are:

- all analytic languages are recursive
- all recursive sets are analytic languages
- all phrase structure grammars are analytic grammars
- there is a simple sufficient condition under which an analytic grammar provides unique analyses for all strings.

The grammar in a POL specification permits certain abbreviations and orderings of productions (for convenience, brevity, and efficiency), but is neve-
theless equivalent to a grammar using the simple scan $S_4$ of reference 8. (An equivalent grammar using $S_4$ is obtainable via a simple construction) Context-sensitive productions may be used. Our method guarantees uniqueness of analysis—it is impossible to embed syntactic ambiguity in a language specification. A simple test ensures finite analyses of all strings. Such a grammar is at least as inclusive as the context-sensitive phrase structure grammar, and there does not appear to be any grammatical structure which cannot be accommodated (grammars of ALGOL, JOVIAL, and FORTRAN were obtained without difficulty).

In fact, such grammars are sufficiently powerful to accommodate the notions of “definition” and “counting” (cf. 7 and the examples of $o_2$), but to actually do so is neither efficient nor expedient. Therefore, a POL characterization includes description of pertinent “internal operations” (see the example in this paper).

**Compilation system data base**

Processing of input strings (POL programs) by a generated compiler is intended to occur in two parts:

(a) preliminary conversion of “raw” input symbols to yield a “syntactic” or “construct” string, which represents the raw input for all further processing, and then

(b) step-by-step syntactic analysis, and (at each analysis step) performance of prescribed sets of internal operations, prescribed output of “code blocks,” output of diagnostic messages, and (if desired) performance of additional auxiliary processes.

The internal operations in a POL specification assume a set of data entities (the “date base”), which are later manipulated as prescribed by a generated compiler. Each entry of the construct string (which represents the raw input during processing) contains a construct (or syntactic type or token) and an associated datum, which is originally derived from the raw input, but may be internally altered. The use of appropriate string handling routines allows effectively a construct string of unbounded length. Other data entities are:

(a) a set of function registers $F_i$, for storage and manipulation of “temporary” numeric data

(b) a set of symbol registers $S_i$, for manipulation of symbol strings

(c) a property table of integer properties $P(J)$, for storage and manipulation of numeric data (e.g., number of dimensions) associated with “variables” in the input string. “Names” (i.e., contents of symbol registers) can be “defined” to the table to reserve table entries for associated data, and the table can be “searched.” Defined names are placed in a property table index. The $j$th table entry consists of four properties $P_1(J)$, $P_2(J)$, $P_3(J)$, $P_4(J)$. By convention, $P_4(J)$ is the syntactic class of the corresponding defined name.

See Figure 2 for further details.

**POL specification and compilation system operation**

The relation between a POL specification and the consequent compilation system processing is best shown via an example. Figure 3 shows a specification* of the language LEMA2 (first exhibited in Lema 2,7 which consists of sentences having the form $\text{A}^n\text{B}^m\text{A}^n\text{B}^m\text{C}^c\text{C}^c$ where $X^a$ signifies a sequence of $k$ $X$'s. Some sen-


The specification contains five sections:

1. **Symbols** — specifies the preliminary conversion of input symbols and "reserved words" to construct string entities.

2. **Syntax** — a set of syntactic productions for use in syntactic analysis.

3. **Internal Functions** — the internal processing to be carried out at each analysis step.

4. **Code** — the sequences of codes to be output at each analysis step.

5. **Diagnostic Messages** — a set of messages for output.

The sections containing internal functions, codes, and diagnostic messages are unnecessary in defining the language structure, but have been added to illustrate these mechanisms. The codes BEG, PWR, AAA, and BBB appearing in the code section were invented expressly for this example; arbitrary BASE operation codes may be designated at will, since these codes are merely transmitted during compilation. The following discussion can be correlated with Figure 4, which shows the compilation analysis trace for a LEMMA2 program, together with resulting values of function registers and code output at each analysis step.

The conversion specified in the Symbols section, of raw input symbols to construct string format, is performed specifically to eliminate dependency of processing on particular machine character sets and hollerith codes. A construct string entry containing a construct and an associated datum replaces each input symbol (or symbol sequence constituting a reserved word); Figure 5 illustrates this process. An arbitrary numeric or hollerith datum may be specified.

The syntactic productions in a specification's Syntax section are applied (as determined by the compiler model's scan) to "rewrite" the construct string, in a step-by-step fashion (see Figure 6). The succession of these rewritings constitutes the syntactic analysis of the construct string. In selective productions from the set of Figure 6, the compiler model uses the "leftmost" scan $S_1$ of [8], i.e., at each step the production chosen is the one whose "left side" occurs first (leftmost) in the construct string. Thus at the first analysis step, the substring chosen is BAA;
at the second, ABK; and so on. To allow explicit reference to the data which accompany the constructs of the substring chosen, a scan position is defined (at each step) to occur at the last (rightmost) construct of the selected substring (see Figure 6).

At each analysis step, internal operations associated with the selected production are performed: function registers or properties within the property table may be set, used, or arithmetically manipulated; character strings may be placed in, prefixed to, or suffixed to symbol registers, and so on. The Internal Functions section (see Figure 7) consists of sequences of internal functions operations. The first operation of each sequence has the label of the production for which action is taken. Thus the sequence RTV F3 -2, etc., is performed each time production 001 is selected.

Care has been taken in formulating the internal operations to achieve economy of means—simple operations, a minimum of system data entities, and a minimum of compiler model machinery. Such a formulation allows a simple compiler model program, while language complexities must be expressed within the language specification. Some anomalies of notation still remain from our earlier efforts, but it is planned to revise and clarify notation.

Operation sequences pertaining to different productions are independent of each other, since there is no "GOTO" operation (a "skip forward" is sometimes permitted). Thus a finite sequence of operations is performed at any analysis step.

Code may be output at any analysis step. Operation codes and operand type specifiers given in the Code section (see Figure 8) are merely transferred to the output, while operands are inserted as specified.

The Diagnostic Message section contains a set of messages, which are output by PRN internal operations. The operation PRN1 S1, which is executed for production 006, prints message 001 and the contents of S1.

**Figure 4—Compilation of a LEMMA2 program**

- Each step of the analysis trace shows the string in the vicinity of the scan position, after application of the production, performance of internal functions, and code output.
- The code output for production p precedes the trace line for production p.
- Values of the first 20 function registers are shown at each analysis step: on line with construct F0 through F9, on line with datum F10 through F19.

From the collection of the Computer History Museum (www.computerhistory.org)
The number in parentheses on the left indicates the number of characters comprising the reserved word. The symbols of the reserved word follow.

A construct (e.g., (END)) is specified for each symbol or reserved word. Use of the construct (NULL) specifies that no construct string entry is to be made; thus "blanks" are ignored above.

A datum is specified for each symbol or reserved word. Either a numeric datum (e.g., (3)) or a hollerith datum (e.g., (h), where h is the desired hollerith datum) may be specified.

The special notation ((EOC)) denotes the "end of card symbol", which in many languages is regarded as a punctuation mark. A representation of ((EOC)) must be given in every Symbols section.

Figure 5 - Preliminary symbol conversion

Figure 6 - Syntactic analysis
duces, from an input program of BASE operations, an 
equivalent program in the target assembly language, in 
a format acceptable to the target assembler. The pro-
duction of assembly language guarantees compatibili-
ty of the object program with the machine’s monitor 
system, and allows the assumption in translation of 
system subroutines and macros.

A BASE program contains generalized item declar-
ators, array declarators, etc., and generalized computa-
tion operators (e.g., ADD, SUB). Since data definition 
is explicit, the BASE computation operators do 
not take account of the data types involved in the oper-
ations. Thus for each computation operation, there is 
an equivalent set of standard suboperations; e.g., cor-
responding to ADD are the standard suboperations 
“add a fixed item to a fixed item” 
“add a floating item to a fixed item” 
and so on. Determination of the specific sub-
operation required for a given BASE operation, taking 
into account the data types involved, is performed 
within the translator.

Translation thus occurs in two parts: 
(a) analysis of BASE operations by an analysis 
section, to derive equivalent sequences of 
standard suboperations, followed by 
(b) expansion of the standard suboperations by a 
macro-processor section, to produce assembly 
code.

A machine specification defines expansion of the 
standard suboperations. In other words, it defines 
for each standard suboperation an equivalent se-
quence of assembly language instructions. Embedded 
in these expansions are format specifiers, which cause 
the appropriate format to be generated. A machine 
specification is processed by the translator genera-
tion system to produce corresponding data tables, 
which are combined with the translator model program 
to form the desired translator. These data tables 
direct the expansions performed by the translator’s 
macro-processor.

Parameters required by the expansions are fur-
ished by the translator’s analysis section via a 
communication table, from which they are retrieved 
as necessary by the macro-processor section. Within 
a machine specification, parameters are specified via 
position in this table.

Our present machine specification notation is pro-
cessor-oriented, and not easily readable; however, it 
is planned to formalize this notation. Some typical 
macro definitions are shown in Figure 9, in a contem-
plated notation, as an illustration of the features 
provided in a machine specification.

The translator model program, except possibly for 
one output procedure, is machine-independent. The 
analysis of BASE operations is dependent only on 
the operator, accumulator data type, and operand data 
type involved, while macro expansion is table-driven. 
All dependency on the target machine is isolated with-
in the data tables used to direct expansions. Assembly 
code is output in the form of 80 column card images, 
which are almost universally acceptable by target as-
semblers. Unusual cases might require simple modifi-
cation of the output procedure.

CONCLUSIONS

Using the syntactic model* we have developed a 
system to formally characterize languages which are 
rich in grammatical structure, and to subsequently 
process strings in such languages. Such processing 
can produce linear code (BASE language). The BASE 
language contains computation and data declaration 
operations sufficient to accommodate the functions of 
ALGOL, FORTRAN and JOVIAL. BASE is ex-
pandable, so that more convenient or efficient opera-
tions may be introduced when these are desirable. We 
have shown the feasibility of formally characterizing 
machine (assembly) language, and of machine-inde-
pendent translation (BASE→ML). In sum, we have 
presented a rigorously based, machine-independent 
compiler generation system.

A consequence of these results is that language in-
variance can be maintained from machine to machine.
It is possible to have a standard version of each pro-
cedure-oriented language, rather than machine-depen-
dent variants.

The system is presently running on the CDC 1604 
computer. Specifications of ALGOL, FORTRAN 
and JOVIAL have been written, as has machine speci-
cification for the CDC 1604. The ALGO and FORT-
RAN specifications have undergone tentative check-
out and modification, as has the CDC 1604 specifi-
cation. Preliminary comparisons of operating char-
acteristics have been made. For a small number of 
short programs, our system produces object programs 
about the same size as do the manufacturer-supplied 
compilers, and requires between twice and three times 
the computer time. Since our system is a prototype, 
these results indicate that it may be possible to gen-
erate compiler/translator systems which have com-
petitive efficiencies. We contemplate major opera-
tional changes, without the sacrifice of theoretical 
rigor, which should increase system speed by a factor 
of between 3 and 5.

The compiler (POL→BASE) portion of this system 
has other uses. The ability to formally characterize 
grammarically rich languages and to subsequently pro-
cess strings in such languages is of importance where-
ever string-structure-dependent processing is re-
quired.
- SET F5 1 places the value 1 in the function register F5.
- PUT S1 V0(-1) places the datum (regarded as hollerith) from construct string position (-1) - relative to the scan position - into the symbol register S1. All previous contents of S1 are deleted.
- SUF S1 V0(0) suffixes to the string in S1 the datum from construct string position 0.
- DEF S1 ((A)) "defines" the string in S1 to the property table: a property table entry (say the n<sup>th</sup>) is reserved, the string in S1 is entered into the property table index, together with the entry number n. The number representing the construct (A) is placed in P0(n), and n is placed in F0.
- ASO 0 F0 "associates" the value in F0 with the construct in string position 0: the value F0 is placed in the datum of position 0.
- SET P1 (F0) 1 places the value 1 in P1(F0), i.e., in P1(n).

Figure 7 – Performance of internal function
ACKNOWLEDGMENT
The authors are indebted to Donald M. Gunn and Craig L. Schager, both for their significant contributions to this work and for their valuable suggestions regarding this paper.

REFERENCES
1. N CHOMSKY
   Syntactic Structures
   Mouton & Company The Hague 1957
2. N CHOMSKY
   On certain formal properties of grammars
   Inform Contr 2 137-167 1959
3. S GINSBURG and H G RICH
   Two families of languages related to ALGOL
   J ACM 9 350-371 July 1962
4. PNAUR (Ed)
   Report on the algorithmic language ALGOL 60
   Comm ACM 4 299-314 May 1960
5. E T IRONS
   A syntax-directed compiler for ALGOL 60
   Comm ACM 5 51-55 Jan 1961
6. D G CANTOR
   On the ambiguity problem of Backus Systems
   J ACM 9 477-479 Oct 1962
7. A C DIFORINO
   Some remarks on the syntax of symbolic programming languages
   Comm ACM 6 456-460 Aug 1963
8. P GILBERT
   On the syntax of algorithmic languages
   J ACM 13 90-107 Jan 1966
9. GILBERT, HOSLER and SCHAGER
   Automatic programming techniques

From the collection of the Computer History Museum (www.computerhistory.org)
RADC-TDR-62-632 Final Report for contract AF30(602)-2400

10 GILBERT, GUNN and SCHAGER
Automatic programming techniques

RADC-TR-66-54 Final Report for contract AF30(602)-3330

11 GILBERT, GUNN, SCHAGER and TESTERMAN
Automatic programming techniques

RADC-TR-66-665 Supplemental Report for contract AF30(602)-3330