An experimental general purpose compiler

by RICHARD S. BANDAT and ROBERT L. WILKINS

Western Union Telegraph Company
Paramus, New Jersey

INTRODUCTION

With the advent of numerous programming languages, both special and general purpose, much interest has been generated in developing newer and higher level programming languages. Two notable approaches\(^1\)\(^2\) have been taken in the attempt to provide language processors for the development of new programming languages with a minimum investment in programmer time and effort. One method is to provide, in an existing programming language, facilities for list processing, stack maintenance, and character manipulation, which can then be used to write a compiler for the language under development. Another approach is the now classic "Meta-Compiler" method typified by the "Meta" series of programs by Val Schorre et al. In these programs, a description of the syntax of the source language is given, together with a paraform code which contains transformational rules. The output is the desired compiler.

This paper describes a third approach which is sufficiently different so as not to be categorized as a subset of the above methods in design, but a union of their favorable characteristics. Its aim is to facilitate defining the syntax of new programming languages and to parse them so that there need be only one output routine for each operator in the new programming language.

Discussion

Our approach was as follows: First, implement a parsing program which would only require the hierarchal level of an item as input and yield an operator and its operands as output; second, design a generic method for determining the hierarchy and syntactic legality of each input character.

The parsing program can be viewed as a modified stacker and is covered later in the paper. An initial solution for the remaining problem was a matrix similar to the type used in a table-driven compiler.\(^3\) However, it was noted that a very large matrix would be required for most useful languages because of their numerous syntactic types. In addition, it was also noted that there were two undesirable characteristics of this method; namely, large amounts of core were wasted because the matrix was sparse, and no efficient way could be devised to include transformational rules. The need for the latter capability directed our efforts toward another table which was neither sparse nor lacking transformational characteristics, yet which was sufficiently analyzed so as to enable its usage to be understood. The result was a table based on a Turing Machine.\(^4\)

The first attempt towards a boot-strap compiler was made by filling in the Turing Table numerically by hand so that at least a rudimentary symbolic Meta-language program could be read in as source statements. For testing purposes, the symbolic Meta-language program described itself, or actually a Turing Table which would allow the syntax/solidus parser to read in its own meta-language. Naturally, the semantics routines were written so that their output would be the Turing Table for the language which was described by the meta-language. After "several" attempts a clean table was obtained which would read its description and generate an exact replica. This procedure was used to check out each version of the Meta-language, which then underwent continual revision in an attempt to increase its readability and raise its scope above the single input character level.

The procedure then for using the program as a compiler consists of two steps; namely, to describe the programming language and to write for each operator in that language a routine to accomplish the task for the type of compiler being written, i.e. interpretive or batch.

The overall size of this program, which is implemented in GE-235 Time Sharing FORTRAN, can be determined from the amount of code generated by its FORTRAN compiler. For the meta-compiler
version (i.e. semantics routines generating a Turning Table), 2.5K is used for instructions and about .5K minimum is required for all the tables. Coding the program in assembly language would surely reduce the figure by a significant amount.

Detailed description

The program consists of a small driver, Figure 1, three functional programs, and four utility routines, which include an internal symbol table. The driver initializes the routines and passes information between the main routines and the symbol table.

Utility routines

The Input Program, Figure 2, simply reads one character at a time from the input buffer, refilling the buffer as needed.

The Classing Function, Figure 3, compares the input character with the table of legal input symbols and assigns to the character the class number which was assigned to it when the language was defined.

Since our present computer is the GE-235 Time Sharing System, it has been convenient to have a pack routine, Figure 4, which packs up to three input characters per GE-235 computer word.

Syntax

The syntax routine, Figure 5, effects the syntax checking, controls parsing, and does whatever transformations may be required. The present implementation is a Turing Table and its associated driver.
The table entries are quintuples of the following form (state, symbol, move, new symbol, and new state). The state is determined by position rather than by name. A pointer to the next alternate state is provided to avoid table searches. Moves are presently restricted to “no move” and “move right,” with “lookback” being delegated to the input routine. Table self-modification has been provided for, but use of this feature depends upon the meta-language.

**The meta-language**

As a general-purpose transformation program, one use is the generation of compilers; that is, since the program parses and transforms input on the basis of the syntax table and produces output on the basis of the semantics routines, it can (given a description of a language in some meta-language) produce a Turing Table for that language. This table may then be used with the program to compile the language described. The availability of a satisfactory meta-language greatly affects the utility of the program. Construction of a Turing Table on a line-by-line basis becomes a tedious task if one wished to give the syntactical and transformational rules for a language, like ALGOL or FORTRAN. BNF as a meta-language seems satisfactory for syntactical rules, but will not handle transformational rules. Currently, the program is being used to explore various notations which will combine syntactic and transformational rules as well as allow self-modification of the Turing Table, giving variable syntax and transforms. The ability to bootstrap from one experimental meta-language to another has proved valuable and timesaving, although no entirely satisfactory notation has yet been found.

**Symbol table**

Once an item is deemed syntactically correct and is ready to be passed to the treeing (parsing) program it first must be logged into the symbol table, Figure 6, which looks to see if that symbol is already present. If it is present, the location is returned; otherwise, it records the symbol at the next available spot, and likewise, returns its location. There is also an entry for returning a symbol given its location.

**Parser**

The parsing program, Figure 7, manipulates a tree which can be viewed as a two-dimensional stack, with linkages corresponding to the way the items are placed within the tree. It is convenient to link input items into a tree and to remove a node with its corresponding leaves. Also resultant “temp” cells can be allocated and optimized dynamically. Flexibility and control are afforded in a tree structure with regard to what is passed on to the semantics...
routines. We use a tree, since much of the literature today uses a tree to depict the syntactic structure of a language. Inherent in the tree structure is considerable power which readily lends itself to such ideas as cross-linked leaves and nodes, and also restructuring of the tree links.

Input to the routine is the location of the syntactic input item from the internal symbol table and its associated hierarchal level. Output is a 4-tuple of the form (node, leaf, leaf, resultant), corresponding to an operator or verb, two operands or objects, and a “temp” result. The mechanics of the routine are simply two programmatical questions which determine where the latest syntactic item should be placed in the tree: Question 1, “Is the new item higher in the hierarchy than the last item in the tree?” If yes, the new item should link up to the last item in the tree; otherwise, Question 2, “Is the new item’s level higher than the item to which the last item links?” If yes, swap links; otherwise, either continue asking the above two questions after having first changed the pointer from the last item to its up-link; or pass the last item (which must be a leaf), the item to which the last item is up-linked (which is a node), and any other leaves up-linked to the node, to the semantics routines. The node can be replaced with the location of a temp cell which has been optimized by noting the level of the node and its position.

Semantics routine

The semantics routine receives four arguments which are a representation of the input operator (verb), two of its associated operands, and a resultant temp cell. At this point the user should have a rough idea of how his input statements will look when parsed so that he will know which of the operands and/or temp cell will be useful for each operator; secondly, he must know what he expects to accomplish in the way of a target language (e.g. FORTRAN, Assembly Code, Turing Table, and so forth). This routine is written by the user for his particular application.

SUMMARY

In conclusion, the program can perhaps best be described as a “skeletal compiler,” i.e., a set of routines which function either as a compiler or as a meta-compiler. Syntax checking, character transformation, character grouping, and parsing are under control of a modified Turing Table. The output may be another table in which case the program functions as a meta-compiler. If this new table replaces the original table, the program may be used as a compiler, the output now being assembly code, or calls to functional routines serving as an interpreter.
REFERENCES

1  J E PLASKOW S A SCHUMAN
   The Tranen System on the M460 Computer
   Massachusetts Computer Associates Inc
   Wakefield Massachusetts

2  D V SCHORRE
   Meta II A Syntax Oriented Compiler Writing Language
   Proceedings 19th National ACM Conference 1964

3  S WARSHALL R M SHAPIRO
   A General Purpose Table Driven Compiler
   Massachusetts Computer Associates Inc
   Wakefield Massachusetts

4  A M TURING
   On Computable Numbers with Application to the
   Entscheidungsproblem
   Proceedings of the London Mathematical Society
   sec 2 vol 42 pp 230 265 1936 1957

5  M DAVIS
   Computability and Unsolvability