The design and implementation of a table driven compiler system

by C. L. LIU and G.D. CHANG*
Massachusetts Institute of Technology
Cambridge, Massachusetts

and

R. E. MARKS
University of Illinois
Urbana, Illinois

INTRODUCTION

The broader application of digital computers to various areas of studies has prompted the design and usage of special purpose problem-oriented programming languages. Although designing and writing a compiler for a special purpose language is no more a mysterious task as it was ten years ago, it is still, in most cases, a very tedious task that might take a large number of man-months to perform. The purpose of designing a generalized table-driven compiler system is to allow a user to write his own compiler for his special language at a reduction of the time currently required in the implementation of a compiler for a new source language. It should be pointed out that not only can a user design a compiler of his own but he can also make modifications, large or slight, to an existing compiler developed with the system. This, therefore, also provides an ideal simulation environment in connection with the implementation of new ideas in translation process. This particular line of development in translation systems naturally lends itself to the increasing use of digital computers in a time-sharing environment.

The notion of a "table-driven compiler" is an extension of the notion of a "syntax-directed compiler" first studied by Irons. The difference between a conventional compiler and a syntax-directed compiler is that in the former, the syntax of the source language is essentially buried in the coding of the compiler itself. The compiler is, therefore, rigidly bound to a fixed source language, and the slightest deviation from the original syntax is, of course, forbidden. In a syntax-directed compiler, the encodement of the syntax structure of the source language, usually in some form of tabular data structure, is separated from the other portions of the compiler and is used to control the operation of the compiler. It is, therefore, possible to replace the encodement of the syntax structure of one source language with that of another when the compiler is used to translate programs written in the other source language. The idea of using replaceable tables to control the operation of a compiler is extended in a table-driven compiler system. Besides having a syntax table to control the syntactic analysis, we shall also have tables to control the allocation of storage space as well as the assembly of binary machine codes. It follows that not only can we modify the syntax of the language that is to be translated by the compiler, but we can also specify the ways in which the compiler allocates storage space and generates object programs. Therefore, to design a compiler for a new source language, we have only to design these tables that control the correct operation of the compiler system. This is a much easier task than that of writing a new compiler for the source language.

General organization

In this paper, we describe a "Table-driven Compiler System" which is designed and implemented on the IBM 7094 Computer. Our goal is to provide the users of the system with an environment within which they can freely design and produce their compilers. The primary design criterion is generality so that the users can define a large class of input languages oriented toward any kind of problem solving purposes, and can also define a large class of object programs to be executed on different computer systems. Therefore, in our system we do not limit the users to specific ways of doing syntactic analysis, or doing storage allocation, or producing binary pro-
grams of a specific format for a particular computer system. What we provide are mechanisms that are general enough for whichever way a user desires to build his compiler.

The Table-driven Compiler System consists of a base program, two fixed higher level languages: the Table Declaration and Manipulation Language and the Marco Interpretation Language together with the corresponding translators which generate the control tables according to the user's specification. A third higher level language: the Syntax Defining Language and its corresponding translator are also needed. However, their definitions are left to the users for the reason of providing them with greater flexibility in specifying the method of syntactic analysis. The base program is controlled by the control tables to perform the task of translating source programs into object machine codes. It is a general program which is independent of the particular source language being translated as well as the method of translation. The control tables contain an encodement of the syntax of the source language, an encodement of the method of translation and an encodement of the characteristics of the target machine.

The base program in the system is divided into three segments; the syntactic analyzer, the table processor, and the assembler. Each segment is controlled by one or more control tables. Their organization is shown schematically in Figure 1. The syntactic analyzer accepts programs written in the source language, recognized syntactic types, and transmits information that will be used for storage allocation to the table processor. After syntactic types have been recognized, the syntactic analyzer also generates a list of macros which will be interpreted by the assembler at assembly time. The syntactic analyzer is controlled by three tables, the lexical table, the test table, and the action table. The table processor can be divided into two parts. The first part accepts information from the syntactic analyzer and enters them into a set of tables called information tables. The second part manipulates all the information tables. After the entire source program is scanned by the syntactic analyzer, this part will sort, merge these tables, and ultimately assign addresses to all the symbols and literals defined in the program so that storage allocation information will become available to the assembler. The table processor is controlled by two tables: the main directory, and the table-manipulation control table. The assembler accepts the list of macros from the syntactic analyzer and generates the binary object program. The assembler is controlled by a macro interpretation table. In addition, the assembler uses another table which contains the binary representation of machine instructions, but is not controlled by it. Figure 2 is a schematic diagram showing how a source program is processed by the compiler system to produce the final object program.

To design a compiler for a particular source language, a user must prepare a set of control tables that are appropriate for this source language. Using the Syntax Defining Language, he shall specify the syntactic rules of the source language and the way macros are generated upon the recognition of syn-
The syntactic analyzer

The syntactic analyzer operates on the input string to generate a corresponding list of macros. It carries out a series of comparisons between sections of the input string and sections of the encodement of the syntax of the language. When a syntactic pattern is recognized, a set of operations will be performed.

The analyzer consists of three basic routines, called LEXICAL, TEST, and ACTION. These routines are controlled by control tables, respectively called LTAB (Lexical analysis TABle), TTAB (Test TABle), and ATAB (Action TABle). There is another table STAB (System TABle) which is used to store the intermediate results of the syntactic analysis and the system variables. Figure 3 shows schematically the organization of the syntactic analyzer.

Routine LEXICAL, controlled by LTAB, performs the lexical analysis on the input string. When a basic syntactic type (e.g. variable name, literal) is recognized, the result of lexical analysis will be put in one of the information tables in the table processor. A pointer, corresponding to this entry, is returned by the table processor to the syntactic analyzer. This pointer will be stored in the table STAB by the routine LEXICAL and is used to represent the entry in the information table throughout compilation.

Routine TEST performs the comparisons between the pointers in STAB or the table processor entries associated with the STAB pointer and an encodement of the syntax which is stored in the table TTAB. When a successful sequence of tests are performed, that is, when a certain syntactic pattern is found, control is changed to routine ACTION.

Routine ACTION has quite general arithmetic and system control facilities. ACTION, as controlled by the table ATAB, generates a list of macros by manipulating the pointers tested by routine TEST. ACTION is the only routine which can change the STAB-table processor structure; TEST cannot; and LEXICAL may only add entries to it in a very controlled manner. The list of macros will be interpreted by the assembler later to assemble binary machine codes. ACTION also performs many other bookkeeping operations upon the fields of the pointers. For example, a field in each of the entries in the symbol table may be used to keep track of identifier usage—whether a given identifier has been used before and if so, how and when, is its present usage consistent with previous usage (e.g. is a label now being used as an indexed array?). It is routine ACTION with table ATAB that would manipulate and test these fields.

The data structure

The table processor is external to the syntactic analyzer in the sense that all the communication between them are done via interface subroutines. Within the analyzer an entry in an information table in the table processor is represented by a pointer issued by the table processor. Within the table processor there are two values associated with this pointer—a location value and a table value. The table value identifies the information table in which the entry is stored. Moreover, the table value also gives the location within the table processor of a set of packing and fielding information which tells how the information associated with an entry is to be broken up into fields. The location value is the location within the table processor which contains the actual information associated with an entry. This block of information may be broken into many fields, each field giving a single characteristic of the entry. In practice, the size of a field may range from one bit to several computer words. It is intended that information tables such as SYMBOL TABLE, LITERAL TABLE, and TERMINAL SYMBOL TABLE are to be organized and used in the table processor. Even though the elements in these tables are thought of as entirely different objects, they are all represented in the same way—by a pointer—in the syntactic analyzer.

Inside the syntactic analyzer, all the results of the
analysis are stored in the table STAB. Two basic types of variable can be used in the analyzer—“value” and “pointer.” A value simply has a numeric value and is treated as such. A pointer is either one that points to an entry in an information table or one that points to an entry in STAB. An entry in STAB may be a simple unindexed value or pointer, or it may be an indexed element in an array or an element in a pushdown stack. STAB may be organized into different number of pushdown stacks and arrays by the user. Such flexibility in organizing the table STAB is desirable as different methods of syntactic analysis might use different numbers of arrays and stacks.

The routine test

The routine TEST makes a series of comparisons between fields of pointers and numeric constants in order to identify syntactic types. Each comparison made by routine TEST is encoded in sixteen fields which require three to four words of 7094 storage. These fields are stored in table TTAB.

The tests performed in TEST are rather simple; if an extremely complex test is needed, control can be transferred to routine ACTION which operates more slowly than TEST but has a completely general arithmetic—comparison facility. Such a call to ACTION is in a different mode and should not be confused with the return to ACTION after a sequence of TESTing. It is termed a predicate call. The result of a predicate call is a truth value which upon return from ACTION is used in the same manner as is a truth value computed internally in TEST.

The routine action

Routine ACTION performs the various arithmetic and manipulative functions necessary for bookkeeping and for moving the pointers that represent the input elements into a list of macros. Thus TEST checks the input string for the existence of syntactic patterns; when a pattern is found, control is passed to routine ACTION. When ACTION completes its processing sequence, control is again transferred to TEST for more pattern testing. The instructions for ACTION are encoded in table ATAB. Part of an ATAB line either describes a location within the data structure or gives numeric values and part defines the operations ACTION perform.

The routine lexical

Lexical analysis is the process of analyzing the input string and putting the results of this analysis in the table processor. The essential results of lexical analysis are a pointer to the information table entry and a truth value indicating whether or not lexical analysis did in fact find an acceptable BCD sequence. Lexical analysis is actually a test of syntactic patterns in the input string. However, due to the specialized nature of the task, it was thought that a significant decrease in operation time could be achieved if lexical analysis was isolated into a separate routine rather than being made a function of the more general TEST-ACTION syntactic analysis. If LEXICAL finds—an according to the rules in LTAB—an acceptable pattern, LTAB indicates that either a search of a certain information table in the table processor is required or by the nature of the string it is the BCD representation corresponding to a certain entry reference number and no search is required. An example of this second case is when LEXICAL is checking for specific terminal symbols such as “+”, “-”, or “*”. It is possible that both the table processor and LTAB are initialized in such a way that the correct entry reference number for each terminal symbol is encoded into LTAB, and thus when the specific terminal symbol is discovered no table processor search is required.

LTAB is divided into many blocks each one of which contains the analysis information for one syntactic type. In general LEXICAL is called by a sequence of several ATAB lines each one pointing to a different LTAB block; LEXICAL performs the analysis described in each block in order of occurrence at LTAB. If a block of analysis fails, LEXICAL automatically—without returning to ACTION—performs the lexical analysis pointed to by the next ATAB line. When the analysis succeeds, the pointer is formed and placed in STAB, the system truth variable is set to TRUE; the ATAB index is cycled down to beyond the last LEXICAL operation; and control is returned to ACTION. If the block of analysis pointed to by the last ATAB line in the LEXICAL sequence fails, the system truth variable is set to FALSE; and control is returned to ACTION.

The table processor

When doing the analysis, the syntactic analyzer picks up information that should be organized, processed, and made available later on to the assembler. For example, all the variable names appearing in a program should be collected and storage registers be allocated to them before binary machine codes can be assembled. The allocation of storage registers requires information such as whether a variable is an array, the size of the array, and the dimensions of the array, and so on. Unlike the generation of macros which can be carried out when the syntactic analyzer has recognized a syntactic type that is “large” enough (for example, a statement) to warrant a generation, these information can be processed only at the end of the syntactic analysis phase. The table
The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System

The Design And Implementation Of A Table Driven Compiler System
binary machine codes. The assembler is controlled by the macro interpretation table which contains information on the meanings of the macros. The assembler uses, but is not controlled by, another table which contains the binary as well as the BCD representation of the machine instructions of the target machine.

The macro list
A macro generated by the syntactic analyzer contains the following information:

1. The macro name, which is actually a number by which the macro is referred to in the macro interpretation table.
2. A count, which is the number of times the result of this macro will be referred to by succeeding macros.
3. The list of arguments of the macro with the type of each argument appropriately identified. There are three types of arguments:
   A type 0 argument is an entry in an information table. It is the pointer of the information table entry.
   A type 1 argument is the result of a preceding macro in the list. It is a pointer to that particular macro.
   A type 2 argument is a number.

A macro is stored in a block of registers with each argument occupying one register. There is also a blank register in the block, called the association register, the usage of which will be explained in the following:

The macro interpretation table
For each of the macros in the macro list, the assembler will generate the corresponding binary machine codes using information in the macro interpretation table. In general, a macro may be interpreted along different paths to yield different sets of binary codes. For example, depending on the modes of the arguments, whether the results of the preceding macros were left in some of the active registers, whether the result of the macro being interpreted will be referred to by succeeding macros and so on, the binary codes generated by the assembler for the same macro will be different. In the macro interpretation table, the way in which the macros are to be interpreted are stored. Also, in the macro interpretation table, information on the usage of temporary storage registers, the usage of active registers and error comments are also available for each of the macros.

Although we have chosen a fixed format for the macros in the system, flexibility is retained by the use of the macro interpretation table. It is interesting to observe that for the same source language and the same set of macros, different macro interpretation tables can be used. One might want to choose an interpretation table that generates highly efficient machine codes at the expense of compilation time, or one might want to choose an interpretation table that will give fast compilation but generates less efficient machine codes. Similarly, by changing the table containing the binary representation of the machine instructions, we can, from the same macro list, generate object programs for different target machines.

Temporary storage pool
When an algebraic expression or Boolean expression is being evaluated, it is necessary, in most cases, to save the intermediate results of computation in temporary storage registers. When the intermediate results are no longer needed, the registers used to save these intermediates may be freed and used in some other computations later on. The maximal number of temporary storage registers that can be used in the compiler is declared by the user. These temporary storage registers may be regarded as a pool from which registers can be requested and to which registers are returned.

Because there is no way of knowing the length of a program or the total number of registers used as temporary storage within the program until all the binary machine codes are generated, the block of temporary storage will have to be attached to the end of an object program. This means that the addresses of the registers used as temporary storage cannot be assigned until the machine instructions are all generated. Therefore, the assembler shall leave the addresses of temporary storage registers as floating addresses starting from zero in the course of compilation, and return to add to these floating addresses the "program break" at the end of the generation of all the machine codes. In order to identify these floating addresses in the machine instructions identification bits are attached to each instruction.

The temporary storage pool is organized as a chained list whose size is initialized by the user. When a temporary storage register is called for, the first register in the list of available registers will be used and thus be deleted from the chain. Whenever a temporary storage register is released, it will be put back in the pool and is placed at the beginning of the list of available temporary storage.

Active registers usage
The execution of a machine instruction would invariably involve the usage of one or more active registers. The register used may be the accumulator, or the multiplier-quotient register, or an index register. Keeping track of the contents of various active registers, the assembler can generate more efficient binary codes by the removal of redundant machine
instructions. For this purpose, an association list is set up in the assembler. For each active register, an entry is reserved in the list. If the execution of the instructions. For this purpose, an association list is in a certain active register, a two-way pointer will be set up between the corresponding entry in the association list and association register in the macro block. When an active register is used later on by some other computations, the two-way pointer will be erased. It is, therefore, necessary to include information on the active registers a macro will use and the active register in which the result of the macro is left in the macro interpretation table. With this information, the two-way pointer will be set up and erased correctly.

CONCLUSION
Besides providing the users with an environment in which they can write their own compilers, it is also hoped that the experience of designing such a system will lead to the understanding of the general theory of compiler structure and the general technique of compiler writing. We emphasize, in our design, the segmentation of the system so that the functions of each section will be clearly defined and be brought out in evidence. The communication problem between the segments is not a difficult one to handle as illustrated in our design. It should also be pointed out that for the generality and flexibility we try to attain, less consideration is placed on efficiency. A case in point is the general structure for the information tables in the table processor.

REFERENCES
1. E. T. Irons
   The structure and use of the syntax-directed compiler
   Annual Review in Automatic Programming Vol 3 1963
2. E. T. Irons
   A syntax-directed compiler for Agol-60
   Comm ACM 4 1961
3. T. E. Cheatham, K. Sattley
   Syntax-directed compiling
   Proceedings SJCC Spring 1964
4. S. Warshall, R. M. Shapiro
   A general purpose table driven compiler
   Proceedings SJCC Spring 1964
5. G. D. Chang
   A table driven compiler generator system
   S M Thesis MIT June 1966
6. R. E. Marks
   A table driven syntactic analyzer
   S M Thesis MIT August 1966
7. C. L. Liu, G. D. Chang, R. E. Marks
   The design and implementation of a table driven compiler system
   Technical Report Project MAC to be published