INFORMATION TO USERS

This material was produced from a microfilm copy of the original document. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the original submitted.

The following explanation of techniques is provided to help you understand markings or patterns which may appear on this reproduction.

1. The sign or “target” for pages apparently lacking from the document photographed is “Missing Page(s)” If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting thru an image and duplicating adjacent pages to insure you complete continuity.

2. When an image on the film is obliterated with a large round black mark, it is an indication that the photographer suspected that the copy may have moved during exposure and thus cause a blurred image. You will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., was part of the material being photographed the photographer followed a definite method in “sectioning” the material. It is customary to begin photoing at the upper left hand corner of a large sheet and to continue photoing from left to right in equal sections with a small overlap. If necessary, sectioning is continued again — beginning below the first row and continuing on until complete.

4. The majority of users indicate that the textual content is of greatest value, however, a somewhat higher quality reproduction could be made from “photographs” if essential to the understanding of the dissertation. Silver prints of “photographs” may be ordered at additional charge by writing the Order Department, giving the catalog number, title, author and specific pages you wish reproduced.

5. PLEASE NOTE: Some pages may have indistinct print. Filmed as received.

Xerox University Microfilms
300 North Zeeb Road
Ann Arbor, Michigan 48106
DONEGAN, Michael Kelly, 1947-
AN APPROACH TO THE AUTOMATIC GENERATION
OF CODE GENERATORS.

Rice University, Ph.D., 1973
Computer Science

University Microfilms, A XEROX Company, Ann Arbor, Michigan
RICE UNIVERSITY

AN APPROACH TO THE AUTOMATIC
GENERATION OF CODE GENERATORS

by

Michaël Kelly Donegan

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Thesis Director's signature:

Edward Alvin Fenstel

Houston, Texas

May, 1973
ACKNOWLEDGEMENTS

The author would like to thank the members of his committee, Edward Feustel, Ken Kennedy, and Bob Minnick, not only for the encouragement and aid they provided in the preparation of this thesis, but for being the kind of educators which make attending graduate school a worthwhile pursuit. Special acknowledgement should go to Dr. Minnick who managed to give me an appreciation for hardware whether I wanted it or not.

To the many others who have provided friendship, encouragement, and enlightenment, I can only say thanks.

Finally, my love and appreciation for my wife, Jackie, cannot be expressed. Perhaps she will find out what it is like to have her husband at home more than two hours a day.
## TABLE OF CONTENTS

**CHAPTER I**  
Introduction  
1.1 Overview of the Compilation Process  
1.1.1 Lexical Analysis  
1.1.2 Parsing  
1.1.3 Code Generation  
1.2 The Plan of This Thesis  

**CHAPTER II**  
A Survey of Compiler Construction Schemes  
2.1 Machine Transfer  
2.2 Using Macros for Bootstrapping  
2.3 Macros for Machine Description  
2.4 An Interpretive Language for Code Generation  
2.5 The Cornell PL/C Compiler  
2.6 Other Schemes  

**CHAPTER III**  
A Code Generation Scheme  
3.1 Elaboration Modes and Translation  
3.2 Bookkeeping Routines  
3.2.1 DECLARE and PURGE  
3.2.2 Labels and Jumps  
3.2.3 MAKEADDRESSABLE  
3.2.4 GENADDR  
3.2.5 CODE  
3.2.6 EVALCONST  
3.2.7 Summary
CHAPTER IV

CGP -- A Preprocessor for Describing
Translations for Code Generation

4.1 A Simple Example

4.2 The Code Generation Preprocessor Language

4.3 Register Allocation

4.4 The PL/I Floating Assignment

4.5 Subscription in NEW

4.6 Other Features

4.7 Possible Expansions to the Language and Conclusion

CHAPTER V

Implementation

5.1 The Parser

5.2 The STS Builder

5.3 The Transition Analyzer

5.4 The Decision Table Processor

5.5 Optimization

5.6 Auxiliary Output

5.7 Summary

CHAPTER VI

Conclusions and Further Work

6.1 Possible Areas for Further Research
CHAPTER I
INTRODUCTION

The purpose of this thesis is to examine the process by which object code is generated for a high-level programming language. Historically code generation has been the most poorly understood task in the compilation process. This is in part due to the simplicity of early languages, since code generation is not really very difficult for them. As programming languages have become more sophisticated—allowing many different data types, more complicated control structures, and an abundance of possible operations—the task of code generation has become enormous. Modern central processing units have also become more complex, allowing many different methods of accomplishing the same task, the best choice of methods depending on the characteristics of the data being operated on.

In an attempt to provide a unified and well-organized method for attacking this complex problem, a number of approaches to code generation have appeared in the recent literature. In each case, the approach taken is to describe a translation method and then define a language which is particularly suited to the needs of the compiler writer in carrying out this translation. Such languages usually include primitives which permit accessing of the
input (which is in the form of a syntax tree or some linearized form) and the symbol table. In addition, instructions for the target machine are easily generated, and registers and temporaries are managed automatically. However, the complex logic involved in choosing the proper object code for a myriad of special cases must still be explicitly specified by the programmer. It is felt that this is a failure of these schemes and that some means must be provided for raising the conceptual level of these languages. If this can be done, then the programmer will be able to perform his task more efficiently and with less possibility for error.

The goal of this thesis is to provide a method for alleviating this problem. Rather than define a new language for code generation per se, a preprocessing language which may be used in conjunction with any suitable programming language will be described. This preprocessor will have the ability to transform incomplete descriptions of the code generation translation desired into a complete translation using certain transitions defined by the programmer. After this is done, the program for code generation will be generated automatically in the desired language. As will be seen, several advantages accrue from this method:

(1) The programmer is relieved from specifying the detailed logical specification of his translation. Rather he is only required to provide translations for those cases which are primitive.
(2) The preprocessor will be capable of checking for logical errors and incompleteness in the input.

(3) Since the specification for the code generator will be quite brief, the task of transferring the compiler to another machine will be much simpler than has been previously possible.

(4) Often it is necessary to change a compiler due to operating system changes or to take advantage of an upward compatible processor. These changes will be greatly facilitated.

Before proceeding further, a short description of the entire compilation process will be given.

1.1 OVERVIEW OF THE COMPILATION PROCESS

Compilers for high-level languages are usually thought of as containing three major entities: the lexical analyzer, the parser, and the code generator. Each of these processes is a translation from one representation of the source program to another—the final result usually being an executable object program. A brief description of each phase will be given.

1.1.1 LEXICAL ANALYSIS

When the source program is presented to a compiler, it consists of a string of individual characters. In a programming language, certain strings are regarded as single entities. For example, it is useful to distinguish among
lexical classes, such as, key words, identifiers, numeric constants, string constants, punctuation, etc.

The job of the lexical analyzer is to perform this recognition task. The output of the analysis is usually a set of ordered pairs of integers which represent the token found in the source. Each pair consists of a lexical class number and a unique identifier of the member of that class.

Figure 1.1 shows how a typical lexical analyzer might work. In the case of the string BEGIN, the output identifies the string as a member of the lexical class key word and as being the second member of that class. In the case of identifiers, such as XYZ, the proper lexical class is identified and the second co-ordinate is the location of that identifier in a symbol table.

In practice, the lexical analyzer is often implemented as a subroutine called by the syntax analyzer. This subroutine returns lexical value of the next token when called. The recognition of lexical tokens may be formally described as a recognizer of extended regular expressions [AH072]. The process used in this recognition is not significantly different among compilers so that it is possible to define one lexical analysis process which can be used almost universally. One such system which takes advantage of this is the AED RWORD system [JOH68]. Using the RWORD system one may specify each lexical class by a regular expression. The system then creates tables to be used by a general recognizer. Bohlmann [BOH73] describes the implementation of a
Input Stream

Reserved Words

1 REAL
2 BEGIN
3 END
4 INTEGER

Hash Function

Symbol Table

Constant Table

Output Stream

Lexical Classes
1 Key Word
2 Identifier
3 Constant
4 Operator
5 Separator

Figure 1.1 Lexical Analysis
lexical analyzer in hardware which is used as a front-end processor for the APL language.

1.1.2 PARSING

In order for an input string to be a proper program of the language, it must satisfy certain structural rules of the language. These rules comprise the grammar of the language. The parser or syntax analyzer must determine if the source program satisfies these rules.

The output of the parser is a record of how the program satisfies the restrictions imposed by the grammar. This record is usually in the form of a syntax tree, a data structure which is somewhat reminiscent of the diagrams used by grammar students to represent the parse of English sentences. The syntax tree is a compact representation of the derivation tree, in which non-essential elements have been elided.

Parsing is a more complex process than lexical analysis and many generalized parsing schemes are not very efficient. However, it is possible to create very efficient automatic parsers for grammars whose productions satisfy certain properties. Much research has gone into defining classes of grammars which can be automatically parsed. Some examples of such grammars are simple precedence [WIR66], operator precedence [FL063], mixed-strategy precedence
and restricted LR(k) grammars [DER69, LAL71]. The latter class of grammars is particularly suited to compiler construction since its members include the grammars of most modern languages.

1.1.3 CODE GENERATION

The code generation process is the final phase of the compiler. Its purpose is to use the syntax tree and symbol table built by the previous phases and produce the final output. This phase is often divided into three parts. Rather than produce the object program directly, it is often desirable to generate a program in some intermediate language such as three-address code. After this is done, an optimization phase is entered which eliminates redundant computations and performs certain transformations on this program to make it more efficient. Finally the optimized intermediate code is transformed into instructions for the target machine.

The major concern of this thesis is the translation of a syntax tree into a linear sequence of instructions. These instructions may either be in the form of intermediate code or an object program for the target machine. Hence, the preprocessor may be used in an optimizing compiler if desired.
1.2 THE PLAN OF THIS THESIS

In the next chapter, several compiler implementation techniques will be described briefly, in order to examine the strengths and weaknesses of each. Then, a NEW code generator [DON72] will then be described with emphasis on the organizational structure of that compiler. This organization was in part inspirational in the development of the preprocessor. In turn, the methods used in the preprocessor have aided in uncovering flaws in the design of that code generator. Finally, a description of the preprocessor, its use, and implementation techniques for its realization will be given.
CHAPTER II

A SURVEY OF COMPILER CONSTRUCTION SCHEMES

In this chapter, several methods for compiler construction and code generation in particular will be surveyed. The emphasis will be on attempts to simplify the code generation process and the merits and limitations of each technique will be discussed.

2.1 MACHINE TRANSFER

Possibly the earliest known method for compiler implementation which made use of any advanced software is the machine transfer method, usually known as bootstrapping. This technique makes use of the availability of a compiler on one machine to produce another compiler which runs on a different machine. This method may be used to implement new languages or simply to move an existing compiler from one computer to another.

Figure 2.1 represents how one might implement a compiler for language X to run on machine B using the bootstrapping technique. In the example, an X compiler running on machine A is used. For convenience, the notation

\[
P \times Q \times R
\]

is used to represent a compiler for language P, written in language Q, and generating code in language R. The aster-
Figure 2.1 Bootstrapping
isks identify programs which must be written in order to effect the transfer. In this case, two complete compilers must be written, one in language \( X \) and another in language \( Y \). If the languages are significantly different, this may be an arduous task.

If it is only required to create an \( X \) compiler for machine \( B \), then the process may be shortened; that is, of course, if there is an \( X \) compiler written in the \( X \) language. In this case, it is only necessary to rewrite the code generation phase to produce \( B \) object code. At worst, this phase may have to be completely redesigned if \( A \) and \( B \) are radically different computers.

An even better bootstrapping scheme was proposed in 1958 by the UNCOL group [STR58]. Their idea was to develop a universal intermediate language for machine communication (UNCOL is an acronym for UNiversal Computer Oriented Language). Given the existence of such a language, each compiler for a new language would be written so as to produce this universal language as output. Corresponding to each machine would be a translator which accepted the UNCOL language and produced object code for that machine. If this were done, any program written in UNCOL, (and therefore the output of any compiler) could be executed on any computer. If a compiler were written in UNCOL, the implementation of that compiler on any machine would be a trivial task.

Unfortunately, the definition of such a universal
Figure 2.2  Bootstrapping using UNCOL
language proved to be a formidable task because of the trem-
mendous differences in machine architectures (see [BEL71]).
Varying word lengths, differences in arithmetic processing,
and inefficiencies inherent in the translation process were
the downfall of the UNCOL concept.

Although UNCOL proved to be infeasible, there are com-
pilers which have been bootstrapped by this method. One
notable example is the BCPL compiler [RIC68]. However, the
difficulties in transferring BCPL to a machine with uncon-
vventional architecture can be staggering [FEU73].

2.2 USING MACROS FOR BOOTSTRAPPING

In general, the use of macro expanders is quite useful
in writing any large program. This is especially true if the
program is in assembly language. Most systems for code gene-
ration make use of macros in some form. This has the advan-
tage that often used code such as the generation of instruc-
tions for the object machine, setting up parameters, saving
machine registers on entry to procedures, and so on, can be
accomplished by the use of one or two macro commands.

It has become fairly standard for high-level languages
to have some form of macro definitional facility available
in the compiler. This may be as simple as the literally
attribute in XPL [MCK70] which provides for naming constants,
or the DEFINE command in B-5500 ALGOL [BUR70]. PL/I makes
use of a rather elaborate preprocessor which is really a programming language in its own right [IBMn].

At least one macro system has been used to generate a compiler by bootstrapping. Orgass and Waite [ORG69] describe a macro system which is very easy to implement, in which they have written a compiler for a simple list processing language. In order to perform the machine transfer for this language, one need only implement their macro language (which requires a FORTRAN program of about 50 lines) and write macros which generate assembly language for the target machine. The compiler itself produces macro calls from the source language input.

Cocke and Schwartz [GOG70] describe a language called LITTLE which can be used for bootstrapping. This language is implemented using macros in a similar fashion.

2.3 MACROS FOR MACHINE DESCRIPTION

Miller [MIL71] describes a system called DMACS which is intended to generate object code automatically from a description of the target machine. His method is diagrammed in Figure 2.3.

To use DMACS the programmer prepares a set of macros which describe the translation from an intermediate language to machine code. Next, macros which describe the target machine are written. These macros describe possible data paths in the machine, such as from storage to registers, and the object code required to move data on that path. In addition, any register conventions, such
Figure 2.3 Miller's DMACS System
as the use of even-odd pairs for multiplications on the IBM System 360 [IBMD], or restrictions on addressing boundaries may be described. These two descriptions are used to provide for automatic register management and the generation of accessing functions for data elements. Miller describes how to use his system for arithmetic expressions and data accession. Unfortunately, the description of the DMACS system is not complete, and too many simplifying assumptions and restrictions on the architecture of the target machine make it a questionable system. However, this area of research might be promising and certainly warrants further attention.

2.4 An Interpretive Language for Code Generation

Elson and Rake [ELS70] describe what they call a macro system which was developed for writing code generators. Their method has been used to write a partial implementation of PL/I for the System 360. Their language, called GCL or Generate Coding Language, is interpreted at compile-time. The input to the code generator is assumed to be a syntax tree produced by the parser. A representation of the syntax tree for the assignment statement \( A=I \) is given in Figure 2.4. Oddly enough, all semantic information is included in the tree rather than being represented in a more compact form, such as in a symbol table.

The GCL system consists of a set of user-written
Figure 2.4 Tree for Assignment A=I
procedures called OPGEN Macro Definitions or OMD's. One
OMD is associated with each node type in the tree. Elson
and Bake estimate that about three hundred OMD's would be
required for a complete PL/I implementation. As the syn-
tax tree is traversed, OMD's are invoked according to the
current node type. To keep track of the current location
in the tree a pointer or cursor is provided. Each OMD has
conventions as to where the cursor will be upon entry to
the procedure.

Aside from the tree, values may be stored in variables
declared by the user. Certain predefined variables are
global to all OMD's. These variables contain information
such as compiler options which are active. Local vari-
bles may be declared in an OMD. Variables in OMD's may
be declared to be of type CELL, REG or STRING. A CELL is
a 32-bit quantity which is used for general-purpose sto-
rage. STRING's are used to store string constants, mul-
tiple-precision arithmetic constants, edit masks, and the
like. REG values represent hardware registers in the tar-
get machine. The declaration of a REG in an OMD implies
that the register is needed to perform the calculation
performed by it. REG's are allocated to real machine
registers by a final optimization phase in the compiler.
Of course, REG's may be declared to be either fixed or
floating. The assignment of values to variables is ac-
complished by a SET statement, such as SET X=5.
Calling an OMD is accomplished by the LINK statement. An OMD may be called "blindly" by specifying a tree reference

\[ \text{LINK \@ARG(1)} \]

in which case the node type at that point in the tree determines which OMD is to be called. Alternatively, a specific OMD may be called

\[ \text{LINK \@FLOATLENGTH(LLEN).} \]

Parameters to OMD's are always passed by reference. In the example above, the FLOATLENGTH OMD returns the length of the item at the current cursor position in the CELL LLEN.

Besides the LINK statement, GCL provides control by the GOTO, IF, and LOOK commands. GOTO may be used to transfer control to a labelled statement within an OMD. The IF command is of the form

\[ \text{IF (test) true label, false label} \]

and allows simple testing. Either label may be elided if transfer is desired to the next statement on the failure of the test. Of course, if the true label is omitted, the comma is required. Further flexibility is provided by the LOOK command. In GCL one may build tables of up to 256 dimensions. A LOOK command of the form

\[ \text{LOOK errorlabel, resultcell, tablename(arguments)} \]

is used to find elements in the table, where the arguments are table indices. The indicated value is returned in resultcell; any error such as out of range indexing results
in a branch to errorlabel. Elson and Bake allude to other uses of the LOOK command, but no examples are given. The use of LOOK in conjunction with a GOTO provides for computed jumps.

In order to access the syntax tree, a powerful set of referencing functions is included in GCL. The elements of a node are referred to as ARG(1), ..., ARG(n); the current node is assumed. The syntax for a tree reference is as follows:

\[
\begin{align*}
<tref> ::= & <argref> <connector> <nameval> | \\
& <argref> <connector> <tref> \\
<argref> ::= & \text{ARG} ( <integer> ) \\
<connector> ::= & . | : | _ \\
<nameval> ::= & \text{VALUE} <id>
\end{align*}
\]

Argrefs serve to push the cursor to an arbitrary location in the subtree of the current node. For example

\[
\text{ARG}(1).\text{ARG}(3)
\]

refers to the third argument in the subtree referred to by the first argument of the current node. The pair

\[
\text{connector nameval}
\]

provides a testing facility for extracting information from the tree as follows:

\[
\text{.VALUE} \quad \text{used to extract the value at the cursor.}
\]

This is used to extract semantic information from the tree.
.<id> initiates a search of all arguments of the current node to see if any has a name matching <id>.

:<id> tests the current node to see if it matches the name <id>.

:_<id> is similar to .<id> except the search is over all descendants of the current node (in some unspecified order).

By using these tree reference primitives, the programmer may test the context of a particular node to determine any optimizations which might be possible. Optimizations of this sort are often quite dramatic and very difficult to detect in linear representations of programs.

GCL allows the use of most of the PL/I operators in forming expressions; an additional operator for exclusive-or (\oplus) is also provided. Bit testing is facilitated by the BIT function. For example,

```
BIT (ARG(2).VALUE,3)
```

obtains the value of the third bit in ARG(2).

As has been seen, GCL is a fairly powerful language with many interesting features which are useful to the compiler writer. It has several drawbacks however:

(1) It is inherently slow due to the interpretation of OMD's (Elson and Bake do not provide compile times). This could presumably be remedied by compiling the language, although Elson and Bake
suggest that this would be difficult.

(2) It is bulky. Elson and Bave estimate that 600
OMD's requiring 300,000 bytes of storage would
be needed for a full PL/I compiler. This is
really an indictment of the PL/I language and
not GCL; however it does contribute to the ob-
jection in (1).

(3) The system used is cumbersome. Since one of the
design goals of GCL was for the compiler to
execute in 100K bytes of storage, an overlay
scheme is required to page OMD's from a direct
access device. Elson and Bave recognize this
problem and propose statistical studies in order
to establish a priority scheme for the paging
algorithm.

(4) Finally, it is not clear that a compiler written
in a good high-level language which could be
compiled and which had an adequate definitional
facility (such as PL/I) would not be more effec-
tive.

2.5 THE CORNELL PL/C COMPILER

In his dissertation, Wilcox [WIL71] describes the code
generation scheme used in implementing the Cornell PL/C
compiler (see also [CON73]). His code generation scheme con-
sists of a set of drivers used to translate expressions and
statements. The expression drivers are used to evaluate all
PL/I expressions. Statements are broken up into a succession of expressions evaluations by their drivers. These expressions are then fed to the expression drivers. A statement driver may insert code for defining labels and branching such as in a DO statement.

The input to the code generator is a syntax tree which is first linearized into intermediate text (or IT). The drivers convert IT into SLM (Source Language Machine) instructions. Finally SLM programs are converted to object code by the use of coding templates written in ICL (Interpretive Coding Language). Coding templates are quite elaborate, permitting conditional branching, flag setting and testing, and calls to machine language and ICL subroutines. A simple example of a coding template is that for FLOOR(X), where X is declared to be REAL FIXED BINARY (p,q).

\[
\begin{align*}
\text{GIFR} & \quad \text{L1,REAL1} & \text{skip to L1 if X is in a register} \\
\text{GLOADG} & \quad \text{REAL1} & \text{generate a load of X into a register} \\
\text{L1} & \quad \text{GRS} & \text{SRA,REAL1,RO, (RO,A1)} \quad \text{generate a right shift to delete fraction bits} \\
\text{GFIN} & & \text{finish code sequence}
\end{align*}
\]

In this example, it is assumed that q has been placed in cell A1 by a previous instruction. GRS is an ICL command to generate an RS type instruction [IBM].
Although Wilcox's method does little to reduce the amount of coding required for the implementation of a language, he has made an obvious contribution by presenting the complete design of a real code generator; perhaps for the first time in the literature.

2.6 OTHER SCHEMES

Wilcox is not the first to use coding templates in code generation. One of the most elegant uses is in the IBM FORTRAN IV 
H compiler [LOW69]. The templates used are known as bit strips, an example of which is given in Figure 2.5. In this scheme, each intermediate code instruction has associated with it an eight bit status field. The meaning of these bits is as follows:

<table>
<thead>
<tr>
<th>Bit Number</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The first operand must be fetched from storage if zero and is in a register if one.</td>
</tr>
<tr>
<td>2</td>
<td>If one, the first operand must be left in a register for later use.</td>
</tr>
<tr>
<td>3 &amp; 4</td>
<td>Same as 1 &amp; 2 but refer to the second operand.</td>
</tr>
<tr>
<td>5</td>
<td>The base register for the first operand must be loaded if one.</td>
</tr>
<tr>
<td>6</td>
<td>Same as bit 5 for the second operand.</td>
</tr>
<tr>
<td>7</td>
<td>Same as bit 5 for the third operand.</td>
</tr>
<tr>
<td>8</td>
<td>The result must be stored if one.</td>
</tr>
<tr>
<td>INDEX</td>
<td>SKELETON INSTRUCTIONS</td>
</tr>
<tr>
<td>-------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>1</td>
<td>L B2, D(0, BD)</td>
</tr>
<tr>
<td>2</td>
<td>LH R2, D(0, B2)</td>
</tr>
<tr>
<td>3</td>
<td>LH R1, D(X, B2)</td>
</tr>
<tr>
<td>4</td>
<td>L B3, D(0, BD)</td>
</tr>
<tr>
<td>5</td>
<td>LCR R3, R3</td>
</tr>
<tr>
<td>6</td>
<td>LR R1, R2</td>
</tr>
<tr>
<td>7</td>
<td>LH R3, D(0, B3)</td>
</tr>
<tr>
<td>8</td>
<td>LCR R1, R3</td>
</tr>
<tr>
<td>9</td>
<td>SH R1, D(X, B3)</td>
</tr>
<tr>
<td>10</td>
<td>SR R1, R3</td>
</tr>
<tr>
<td>11</td>
<td>AH R3, D(X, B2)</td>
</tr>
<tr>
<td>12</td>
<td>AH R1, D(X, B2)</td>
</tr>
<tr>
<td>13</td>
<td>AR R3, R2</td>
</tr>
<tr>
<td>14</td>
<td>L B1, D(0, BD)</td>
</tr>
<tr>
<td>15</td>
<td>STH R1, D(0, B1)</td>
</tr>
</tbody>
</table>

Figure 2.5 A Bit Strip Table
These status bits are set by a previous register optimization pass through the intermediate code. The appropriate code is selected by letting the first four status bits choose one of the columns in the table. The last four bits are used to replace the X's in the selected column in order from top to bottom. Finally, any one in that column selects an instruction to be generated.

This system is mentioned since it is the only use of decision tables in code generation known. The use of decision tables is central to the system described in this thesis. The problem with this particular scheme is that the bit strips are apparently prepared by hand, a somewhat laborious task. In addition, the compiler phase which sets the appropriate status bits is quite complicated. On the other hand, bit strips are very compact and the resulting object code is quite good.

It is sometimes possible to merge the code generation phase of a compiler with the parsing phase by associating with each production of the grammar a sequence of code to be generated when that production is applied. This process is known as syntax-directed translation (see [AH072]). This method has been used in compilers for ALGOL 60 [IRO61] and other simple languages. Wilcox points out that this technique is not feasible for more complicated languages since variables may be used before they are declared. This presents little problem in ALGOL 60, for example, where labels
need not be declared before their use. However patching a forward branch is much simpler than patching a forward reference to a complicated data item. In addition to this objection, one may mention languages such as ALGOL 68 [WIJ69] and NEW [WAR72] where the interpretation of a construct depends on its context, which may not be known when the construct is recognized.
CHAPTER III
A CODE GENERATION SCHEME

In this chapter, the organization of the code generator used in the NEW [WAR72] compiler will be outlined. The reason for this is to illustrate some typical routines and data structures which occur in a code generator. The details of the NEW implementation are not given here since they are available elsewhere [DON72] and are dependent on the structure of the Rice Research Computer [RIO68a].

The input to the NEW code generator consists of a syntax tree, a symbol table, and a table of constants. The code generator produces object code directly from this input. This object code is organized into modules which may be independently linked with other modules from previous compilations. Each routine in the source program produces an object module.

The syntax tree consists of variable length nodes; each node represents some production of the grammar which has been applied by the parser. In order to distinguish the different nodes, a two-level identification field is provided. The first level divides the nodes into certain classes which may be handled in common by the code generator. An example is the class of commutative operators. The second level identifies an individual member of the group. Each node may have a number of operands associated with it. These operands are classified according to the data they repre-
sent. The operand types are as follows:

(1) Tree pointer -- A reference to a subtree representing a program piece or an expression to be evaluated.

(2) Literal -- A constant which may be used as an immediate operand in an instruction.

(3) Constant -- A reference to the constant table.

(4) String -- A pointer to a string in a string constant table. String constants share the same area used by the symbol table to represent variable names.

(5) Intrinsic -- Used to represent a function which is to be evaluated in-line.

(6) Variable -- A reference to the symbol table.

In addition to the operands which appear in the tree, certain special operands may be generated internally by the code generator. These are used to refer to hardware registers or temporary areas.

The symbol table consists of a current definition table and a push-down area. The push-down area is used to hold symbol table references when a variable is re-
declared in an inner block. Each entry contains a block and procedure nesting count to preserve the block-structuring of the language. As declarations are encountered during translation, space is taken for the variable and the compile-time address is placed in the symbol table. This address consists of an ordered pair. The first coordinate identifies the run-time area in which the variable will reside. Such areas are registers, the stack, parameter vectors, etc. The second coordinate of the address is the offset in the run-time area.

3.1 ELABORATION MODES AND TRANSLATION

In the NEW language, identical constructs may be translated differently according to their context. A global variable is used to keep track of the current elaboration mode (as it is called). The elaboration modes are as follows:

(1) VALUE -- The normal mode in which the value of a construct is desired.

(2) REF -- The address of the value is required. Only certain constructs may appear in REF context. For example, arithmetic expressions have no address assigned and hence cannot be elaborated in this mode.

(3) PARAM -- Used for parameters to procedures and indicates that a reference to the value should be constructed if possible.
(4) CC  -- In certain contexts, the value of an expression is not needed but only an indication of its value in the condition code. This mode is used to set up tests for conditional jumps.

(5) ASSIGN  -- In the strict definition of NEW, all constructs on the left-hand side of assignments are to be elaborated in REF mode. However, some savings in object code can be achieved by treating assignments as a special case. Thus, the code generator usually loads the right-hand side of an assignment into a register and uses ASSIGN mode to cause a store of that value to the left-hand side.

(6) VOID  -- In certain cases, the value of a construct is not required but must be elaborated for any side effects which may be present. VOID mode is used to prevent the translation of any expression which does not produce side effects.

(7) GOTO  -- The elaboration of an expression such as GOTO IF A THEN L1 ELSE L2 FI is costly if the conditional is elaborated
in VALUE mode since the creation of a label value is implied. Using GOTO mode elaboration achieves the intended effect by treating the expression as if it had been written

IF A THEN GOTO L1 ELSE GOTO L2 FI.

The main routine in the code generator is the TRANS procedure. This procedure takes one parameter which is assumed to be an operand. It is the duty of TRANS to evaluate the operand and leave the result in a register. If the operand is a tree pointer, the node referred to determines how elaboration is to proceed. The register to be loaded is appointed by whoever calls TRANS, and TRANS must restore any other registers it uses to their original values. An alternative method might be for TRANS to return a pointer to the value created. However, this is unsatisfactory since TRANS may not know which registers are currently being used by the caller. If TRANS chose to put the value in one of these registers, confusion could result.

Certain elaboration modes (GOTO, VOID, CC, and ASSIGN) do not require a value to be loaded. ASSIGN mode uses the target register instead to indicate where the value to be stored is located.

The TRANS routine makes use of several other procedures during its operation. These procedures perform the bookkeeping operations required during translation. They will
be outlined in the next section.

3.2 BOOKKEEPING Routines

Many details must be attended to during code generation. In the NEW code generator, a number of routines are used to take care of these actions. The various routines will be discussed and in some cases related to similar routines which might be required in a code generator for other computers.

3.2.1 DECLARE and PURGE

When a new block is entered during the translation of a program, the symbol table must be adjusted to reflect the new nomenclature introduced by declarations in that block. Similarly, the old nomenclature must be restored again on exit from the block. These actions are performed by the DECLARE and PURGE routines.

The current nesting level in a NEW program is maintained by two counters, RN and PN, the range and procedure nesting levels. These level numbers are adjusted as needed during execution of the code generator. When a symbol is entered in the symbol table, the current values of RN and PN are stored with the entry. In this manner, up-level addressing may be detected as well as branches to labels outside the current range. All symbols which are declared in a particular range are chained together in the symbol table in a linked-list fashion. This facilitates deleting the entries on block exit. If a symbol is re-declared in
an inner range, the old definition is saved in the push-down area mentioned in an earlier section.

3.2.2 LABELS AND JUMPS

Jumps to labels which have not yet occurred present some problem to the code generator. In the NEW code generator, every label is given a unique identifying number which is an index into a label array. The label array is used to keep track of forward jumps. When a jump to a label is generated and that label has not occurred, the entry for that label is chained to the jump instruction. Fortunately, the format of R-2 instructions permit this chain field to be placed in the instruction itself. As more jumps to this label are generated, the instructions are linked to this chain. When the label finally appears, the instructions are patched to be jumps to the correct location.

In compiler generated code, it sometimes happens that a jump is generated to a label which is in turn labelling another jump instruction. One place this occurs is in nested conditionals. In the NEW code generator, this is handled by deferring the label patching process until the labelled instruction is generated. If it is a jump, all jumps to the label are rechained to proceed to the destination of the second jump.

3.2.3 MAKEADDRESSABLE

In some cases, it may be necessary that a value be
made addressable so that the address may be used in an instruction. This operation is equivalent to loading a base register in some machines. When this action is required, the routine MAKEADDRESSABLE is used. If the value happens to be an expression, then MAKEADDRESSABLE will simply call TRANS, otherwise the appropriate is taken. In some machines, several modes of addressing are available and a particular mode may be requested of a similar routine.

3.2.4 GENADDR

If an operand is addressable, the fields of the instruction which will address this operand need to be created. This is done in the NEW code generator by the GENADDR routine. As mentioned above, code generators for other machines might require the addressing fields in a number of different modes.

3.2.5 CODE

The CODE routine is used to output instructions and format them for listing if required. In the R-2, the CC or condition code may represent a number of different types of conditions depending on the last instruction executed. In order to produce a meaningful listing, the CODE procedure keeps track of the current CC mode by setting it depending on the instructions received. If a conditional branch is generated, the proper mnemonic for the condition may be inferred. The CODE routine also uses the symbol table to obtain the names of variables when they are accessed by instructions. These actions make for a more readable object listing.
3.2.6 EVALCONST

Often constant expressions occur in programs. The NEW code generator attempts to detect these expressions and evaluate them at compile-time. If such an expressions is found, a pointer to the tree is passed to EVALCONST whose job is to evaluate it. EVALCONST may return an operand of type LITERAL or CONSTANT depending on the resulting value.

3.2.7 SUMMARY

This chapter has presented a short outline of some of the routines which are used in the NEW code generator. The list is by no means complete as an exposition of the routines which might be required by a code generator, but a number of common actions have been presented. In general, it is desirable for actions which are commonly required in the code generator to be placed in subroutines. This simplifies the TRANS routine and makes it more readable.
CHAPTER IV
CGP -- A PREPROCESSOR FOR DESCRIBING
TRANSLATIONS FOR CODE GENERATION

Before a code generator can be written it is necessary to decide on an implementation of the language for the target machine. An implementation is a set of mappings which define how a particular program will be realized on the machine. To define the implementation, the compiler designer must first decide how each possible data type will be represented in the computer. Then to each action required by the language a stereotype set of instructions must be determined which simulate that action when executed by the computer. There may be several such stereotypes assigned to a particular action depending on the characteristics of the data being operated on.

As an example, Figure 4.1 shows the code sequences which might be chosen for implementing the assignment \( A=B \) in PL/I, where \( A \) and \( B \) are each floating point arithmetic variables. The choice of instructions to be used depends on the alignment and the length of the operands.

In a code generator, many such decisions must be made and each contributes to the bulkiness of the code generation phase. Given an implementation, the compiler writer is faced with the problem of realizing a complex translation in as efficient a manner as possible. The desire for effi-
ciency is normally at odds with the equally important task of making the program readable and easily debugged. Readability is important since someone other than the author of the program may be called upon to debug the program if it is found to be in error. Similarly, it may be desirable to change the program in order to provide a new feature in the language or support an instruction which is available on an upward-compatible central processor. Elson and Rake state that the stereotypes given in Figure 4.1 are not the most efficient for every model of the System 360.

In order to debug or modify a code generator, one must be capable of easily discerning the logic of the program. The author submits that this is not a simple task when currently available techniques are employed.

As a case in point, the OMD for floating point assignment given by Elson and Rake which realizes the translation specified in Figure 4.1 is given in Figure 4.2. Parts of the program have been deleted to shorten the example. This example is typical of most code generation routines. Equally cryptic examples could have been extracted from the NEW code generator.

The author has carefully studied this example in order to obtain a feeling for the GCL language (to the extent of removing several errors which existed in the original). The program appears rather formidable at first due to unfamiliarity with GCL, but it represents a fairly straight-
<table>
<thead>
<tr>
<th>Precision</th>
<th>Both single</th>
<th>A single</th>
<th>A double</th>
<th>Both double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alignment</td>
<td>Both aligned</td>
<td>B double</td>
<td>B single</td>
<td>Both double</td>
</tr>
<tr>
<td>both aligned</td>
<td>L R,B</td>
<td>L R,B</td>
<td>SDR R,R</td>
<td>LD R,B</td>
</tr>
<tr>
<td></td>
<td>ST R,A</td>
<td>ST R,A</td>
<td>LE R,B</td>
<td>STD R,A</td>
</tr>
<tr>
<td>both unaligned</td>
<td></td>
<td></td>
<td>MVC A(4),B</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>XC A+4(4),A+4</td>
<td></td>
</tr>
<tr>
<td>A unaligned</td>
<td></td>
<td></td>
<td>MVC A(4),B</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>XC A+4(4),A+4</td>
<td></td>
</tr>
<tr>
<td>B unaligned</td>
<td></td>
<td></td>
<td>MVC A(4),B</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>XC A+4(4),A+4</td>
<td></td>
</tr>
<tr>
<td>either unaligned</td>
<td>MVC A(4),B</td>
<td>MVC A(4),B</td>
<td></td>
<td>MVC A(8),B</td>
</tr>
</tbody>
</table>

**Figure 4.1** Stereotype code for the assignment A=B in PL/I
START @FLOATASSIGN
DCL (LLEN,RLEN,WKCELL, RATR, LATR, LO, LB, LI, LL, LR, RO, RB, RI, RR, RL) CELL, GPR REG(FIXED)
*FIND BYTE LENGTHS OF SOURCE AND TARGET
*@FLOATLENGTH UTILITY EXPECTS CURSOR AT PARENT OF
*ARITH NODE
PUSH ARG(1)
LINK @FLOATLENGTH(LLEN)
POP
PUSH ARG(2)
LINK @FLOATLENGTH(RLEN)
POP
SET LALN=2-ARG(1).UNALIGNED
SET RALN=2-ARG(2).UNALIGNED
*NOW DO TABLE LOOKUP AND GO TO RESULT LABEL TO
*SET UP REQUIREMENTS FOR SOURCE RESULT, DEPENDING
*ON LENGTHS AND ALIGNMENTS
LOOK ERR1,WKCELL,TBL1(LALN,LLEN/4,RALN,RLEN/4)
GO TO WKCELL
ERR1 MSG 'ERROR IN TBL1 LOOKUP IN @FLOATASSIGN'
RTN
*FOLLOWING ARE THE RESULT LABELS OF LOOKUP
*TARGET 4 BYTES ALIGNED, SOURCE ALIGNED, ASK FOR
*RX REFERENCE OR FLOATING REGISTER
RXFRI SET RATR = M'F0001000'
GO TO LRX
*8 - BYTE RESULT NEEDED IN FLOATING REGISTER, SO
*SOURCE WILL DO SDR, LE OR LD OR MVC(4),SDR,LE
FRFUL SET RATR=M'30000000'
*GET ADDRESSABILITY OF TARGET AS RX OR RS REFERENCE
LRX SET LATR = M'C0000000'
GOTO LINK
*REQUEST BOTH SOURCE AND TARGET AS RS REFERENCES
*SINCE MVC WILL BE DONE
RS1 SET RATR = M'40000000'
SET LATR =M'40000000'
*NOW LINK TO EACH ARGUMENT
*STANDARD CALLING SEQUENCE HAS BIT ATTRIBUTE CELL,
*OFFSET, BASE, INDEX, LENGTH, AND ONE EXTRA CELL
*FOR SPECIAL USE IN SOME CONTEXTS
LINK LINK ARG(1)(LATR,LO,LB,LI,LL,LR)
LINK ARG(2) (RATR,RO,RB,RI,RR)
*NOW DO LOOKUP AS BEFORE, BUT THIS TIME TO DECIDE
*WHERE TO GO TO FINISH WORK
LOOK ERR2,WKCELL,TBL2,(LALN,LLEN/4,RALN,RLEN/4)
GOTO WKCELL
ERR2 MSG 'ERROR IN TBL2 LOOKUP IN @FLOATASSIGN'
RTN

Figure 4.2 Elson and Fiske's Floatassign OMD
*FOLLOWING ARE THE VARIOUS LABELS RESULTING FROM
*THE LOOKUP
*SOURCE IS EITHER RX (IF DATA REFERENCE) OR FLOATING
*REGISTER (IF EXPRESSION). IT HAS SET RATR TO INDICATE
*WHICH
RXFR2 IF(BIT(RATR,RXREF) | BIT(RATR,RSREF)) LST
*IN FLOAT REGISTER GIVEN IN RB FIELD
    STE RB,LO(LI, LB)
    RTN
*IN CORE
    LST L GPR,RO(RI, RB)
    ST GPR,LO(LI, LB)
    RTN
*DOUBLE LENGTH RESULT IN REGISTER, TARGET ALIGNED
FRFUL2 STD RB,LO(LI, LB)
    RTN
*TARGET IS SHORT FLOAT, EITHER IS UNALIGNED.
*SOURCE WAS RETURNED AS RS REFERENCE
RS24 MVC LO(4, LB), RO(RB)
*TARGET IS LONG FLOAT UNALIGNED, SOURCE LONG
*FLOAT RS REFERENCE
RS28 MVC LO(8, LB), RO(RB)
    RTN
*TARGET LONG FLOAT UNALIGNED, SOURCE SHORT FLOAT
*RS REFERENCE
RS24XC MVC LO(4, LB), RO(RB)
    XC LO+4(4, LB), LO+4(LB)
    RTN
*FOLLOWING ARE THE TWO TABLES, GIVEN IN ROW
*MAJOR ORDER
TBL1 TBL (2,2,2,2) REF
    ARRAY RXFR1,RXFR1, RS1, RS1, FRFUL1, FRFUL1, FRFUL1, RS1,
    RS1, RS1, RS1, RS1, RS1, RS1, RS1, RS1
TBL2 TBL (2,2,2,2) REF
    ARRAY RXFR2, RXFR2, RS24, RS24, FRFUL2, FRFUL2, FRFUL2, RS28,
    RS24, RS24, RS24, RS24, RS24xc, RS28, RS24sc, RS28
END

Figure 4.2 (Continued)
forward procedure. Initially, the length of each operand is determined. Then, using a table lookup and a subsequent GOTO, it is determined whether an MVC or a Load and Store sequence is necessary to perform the assignment. In the former case, an RS reference to the right-hand side is requested; otherwise an RX reference or a register reference is required. Finally another table lookup is performed to transfer to the code which will complete the operation. Although the example is liberally commented, it is difficult to determine if a particular situation is indeed handled correctly. This is because the entire logical flow of the OMD is implicit in the tables used, and the correspondence between operand conditions and the label jumped to is not obvious. Thus the translation from the table previously given has been completely obscured by the language. This difficulty might have been somewhat alleviated by laying the table initialization out better (for example, similar to the way APL would print such a table) and identifying the meaning of table dimensions by comments at the point of declaration. Unfortunately, one cannot usually depend on programmers to be so perspicuous.

The situation is not much better when other languages are used. For example, most of the generator routines in the NEW compiler make use of intricately nested IF-THEN-ELSE blocks. To complicate things further, the desire for efficiency required that numerous jumps and labels be in-
asserted to merge common actions.

The point of this example was to emphasize that the table given in Figure 4.1 is superior in readability to any translation of its meaning into currently available programming languages. It would be most desirable if the table itself could be used as the description of the desired translation. The CGP system described in this thesis is an attempt to apply this idea; that is, the programmer will be allowed to specify his implementation in simple terms. This specification will in turn be transformed into a complete program realizing the desired translation. Rather than being a "code generation" language, CGP is a preprocessor which may be imbedded in a host language. CGP translates statements in its language (CGPL) into statements in the host language.

In order for CGP to be feasible, it is necessary to make certain requirements on the structure of the code generation scheme. However, no restriction is made in how the translation is described since the programmer may define his own terms to fit his needs. This is an improvement over Miller's [MIL71] scheme where descriptions are highly restricted, allowing only a certain class of target machines.

It will be assumed that the input to the code generator will be a syntax tree. This is a reasonable restriction since many code generators exist which are organized
this way. Elson and Bake have discussed the advantages of a syntax tree over linear representations in PL/I. These arguments are valid for most high-level block-structured programming languages. As has been seen, a code generator which uses a tree as input normally consists of a number of recursive routines, one such routine for each node type in the tree. Within each routine, the action performed is dependent on the operands or arguments of the current node and on global information available to all routines. Eventually, the routine will generate the object code to evaluate that node and perhaps return a reference to the result. This process may be thought of as a series of transitions from an initial state (the characteristics of the node arguments) through some set of intermediate states and finally to a state in which the final code may be generated. CGP is designed to allow the programmer to specify these actions in a simple notation from which a program for performing the actions may be generated. Guided by information concerning possible input states, transitions, and allowable terminal states, CGP determines the best sequence of actions for achieving a terminal state from each initial state. Finally the program for making the necessary decisions is generated in the host language. In addition, CGP will check for any logical inconsistencies or unresolvable input states.

In this thesis, the host language is assumed to be
PL/I. The features which make PL/I suitable for this purpose are as follows:

(1) PL/I allows nest IF-THEN-ELSE statements making conditional tests such as will be generated by CGP simple to translate.

(2) PL/I permits declarations to appear at any location in a block. This is convenient since the preprocessor may generate declarations without regard to any language structure.

(3) PL/I has a very convenient macro facility. The use of this facility makes the input to the preprocessor more readable.

(4) It is possible for the PL/I compiler to generate a formatted listing. Hence CGP need not be overly concerned with creating readable output, although some effort in this direction is certainly desirable.

The remainder of this thesis deals with the use, philosophy and implementation of CGP. An attempt will be made to show how the design of the code generation scheme can be molded to realize the best utilization of the system. Since using CGP is an ad hoc process (in the same sense that programming itself is ad hoc), several examples will be given to illustrate its usefulness in various situations.
4.1 A SIMPLE EXAMPLE

In this section, code generation for a simple machine with an equally simple language will be considered. The example will illustrate the code generation scheme to be used with CGP. This computer, the T-machine (T for trivial), consists of an accumulator, a random-access memory, and a built-in hardware stack. The instruction set is as follows:

- **CLA M**: load the contents of memory location M into the accumulator
- **ADD M**: add the contents of memory location M to the accumulator
- **SUB M**: subtract the contents of memory location M from the accumulator
- **PUSH**: push the accumulator onto the stack
- **ADD**: add the top of the stack to the accumulator and pop the stack
- **SUB**: subtract the top of the stack from the accumulator and pop the stack

The language for which object code will be generated can be defined by the following grammar:

```
Program    S op S
S          (S op S)
S          variable .
op         + -
```

A program in this language might be

\[(A+B)-(C-D)\]
which would have a corresponding syntax tree such as

```
    +
   / \    / \    / \
  A   B  C   D
```

A code generator for this language would consist of a translation routine, TRANS, which accepts a tree pointer as input. Upon investigation of the node referred to by this pointer, it will call on one of two generator routines to produce object code. To start the process, TRANS is called with a pointer to the root of the syntax tree.

Since generator routines will cause the value created to be left in the accumulator, a convention will be made that before TRANS may be called, the accumulator must be freed if it is currently in use. This is done with a PUSH instruction. To keep track of the accumulator status, a global variable ACCFREE will be used. When a PUSH is performed, ACCFREE will be set to true; a CLA will set ACCFREE to false. Other operations will have no effect on ACCFREE.

In designing the code generation routines, it is noticed that on entry to the routine, an operand may either be a tree pointer to a subtree or a variable. These conditions will be called TREEPTR and VAR respectively. Conditions which might hold true on entry to a generation routine will be called input conditions.
Obviously it is not possible to generate an ADD or SUB command immediately from the input states. It is necessary therefore to introduce new states which will represent conditions which may exist while the generator is being executed. These states will be called INACC, signifying that the operand has been brought to the accumulator, and ONSTACK, signifying that the operand has been pushed onto the stack. After these conditions are defined, it is possible to list configurations (sets of states) from which the final code may be generated. For example, if the first operand is in the accumulator, and the second is in memory, then the instruction ADD OP2 will compute the desired result. Configurations from which the final result may be computed will be called terminal configurations.

The entire code generation process may now be summarized in tabular form. Figure 4.3a shows such a table for the ADD generator. This form of table will be called a state transition schema for the ADD generator. Each row in a state transition schema (STS) corresponds to some condition which might hold during the translation process. The columns represent states of the code generator which are numbered at the top. The X's represent conditions that are true in that state. Together they comprise the configuration associated with that state. Underneath each column is the text representing actions to be performed when the code generator is in that state. In some cases,
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAR</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TREEPTR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INACC</td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>7</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ONSTACK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>VAR</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TREEPTR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>INACC</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>5</td>
<td>X</td>
<td></td>
<td>9</td>
<td>X</td>
</tr>
<tr>
<td>ONSTACK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACCFREE</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.3a The STS for ADD kodes
<table>
<thead>
<tr>
<th>VAR</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>TREEPTR</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INACC</td>
<td>5</td>
<td>7</td>
<td>X</td>
<td>X</td>
<td>11</td>
<td>11</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ONSTACK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VAR</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TREEPTR</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INACC</td>
<td>6</td>
<td>8</td>
<td>X</td>
<td>X</td>
<td>9</td>
<td>10</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ONSTACK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACCFREE</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 4.3b** The STS for SUB nodes
this action may cause a transition to another state. This is indicated by an arrow from some condition to a new condition row. The head of the arrow points to the state number of the new state to be entered. For terminal states, the state number is circled and no transition is indicated.

The first four columns of the STS for ADD represent initial states (or input states). By following the transitions and listing the actions performed until a terminal state is reached, it is possible to see exactly what object code is generated from each input state. Figure 4.3b shows the STS for SUB and Figure 4.4 shows the object code generated by each of these schemata.

Examining the STS for ADD, one can see that it is not the only possible STS which could be given. For example, State 1 could have been changed so that **CLA OP1** was the action specified. This would result in a next state of 6, since the VAR condition on OP1 would be changed to INACC. Therefore, the choice of actions performed was quite arbitrary. This is not the case for SUB, since subtraction is a non-commutative operation. For example, choosing the transition from VAR to INACC on OP2 in state 1 for subtraction would result in the following:

```
CLA OP2
PUSH
CLA OP1
SUB
```
<table>
<thead>
<tr>
<th>State</th>
<th>Object Code</th>
<th>State</th>
<th>Object Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CLA OP2</td>
<td>1</td>
<td>CLA OP1</td>
</tr>
<tr>
<td></td>
<td>ADD OP1</td>
<td></td>
<td>SUB OP2</td>
</tr>
<tr>
<td>2</td>
<td>TRANS OP2</td>
<td>2</td>
<td>TRANS OP2</td>
</tr>
<tr>
<td></td>
<td>ADD OP1</td>
<td></td>
<td>PUSH</td>
</tr>
<tr>
<td>3</td>
<td>TRANS OP1</td>
<td></td>
<td>CLA OP1</td>
</tr>
<tr>
<td></td>
<td>ADD OP2</td>
<td></td>
<td>SUB</td>
</tr>
<tr>
<td>4</td>
<td>TRANS OP1</td>
<td>3</td>
<td>TRANS OP1</td>
</tr>
<tr>
<td></td>
<td>PUSH</td>
<td></td>
<td>SUB OP2</td>
</tr>
<tr>
<td></td>
<td>TRANS OP2</td>
<td>4</td>
<td>TRANS OP2</td>
</tr>
<tr>
<td></td>
<td>ADD</td>
<td></td>
<td>PUSH</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>TRANS OP1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SUB</td>
</tr>
</tbody>
</table>

Figure 4.4 Object Code Generated by the ADD and SUB Routines
Obviously, the choice of loading OP2 first is superior. This is often the case in implementing any real language on an actual computer, therefore it will be necessary for any system which automatically generates coding routines to determine the best possible transition path based on some cost function. A method for performing this task will be discussed in a later chapter.

Having developed a method for depicting the transformations involved in a code generation routine, it is necessary to develop a notation to be used in representing these transformations for input to the computer. This notation will be called CGPL or Code Generation Preprocessor Language and is the subject of the next section.

4.2 THE CODE GENERATION PREPROCESSOR LANGUAGE

It has been seen that code generation transformations may be represented as a sequence of state transitions: from a set of input states to a set of terminal states. CGPL is the language used to represent the various elements of this scheme. Figure 4.5 shows the CGPL for the ADD and SUB routines developed previously.

A line by line explanation of the SUB routine follows:

(1) $\$GENERATE$ indicates the beginning of a set of preprocessor statements. SUB is an identifier for this routine and is used in any labels generated by the CGP.
(2-4) The conditions which will be used in the routine are declared. VAR and TREEPTR are the input conditions and are further identified as being mutually exclusive. The mutually exclusive attribute tells CGP that it is not necessary to allow for cases in which both conditions are true. INACC and ONSTACK are conditions which result from transitions and as such are declared INTERNAL. INTERNAL conditions are automatically understood to be mutually exclusive.

(5-7) Here the terminal states are identified. The list in parentheses is considered to be ordered and applied to operands in that order. For example, (INACC,VAR) is taken to mean that INACC is true for OP1 and VAR is true for OP2. To the right of the arrow is the code to be generated for this state.

(8-11) The set of possible transitions is listed. Transitions are ordered pairs of conditions and the code specified is that which will cause that transition.

(12) END specifies the end of preprocessor statements.

When the CGP translates this routine certain conventions are used in producing the PL/I source code. Input
$\text{GENERATE ADD;}$

DECLARE CONDITIONS (VAR, TREEPTR) MUTUALLY EXCLUSIVE,
  (INACC, ONSTACK) INTERNAL;

TERMINAL
  (VAR, INACC) -> 'GEN(ADD, OP1)',
  (INACC, VAR) -> 'GEN(ADD, OP2)',
  (ONSTACK, INACC) -> 'GEN(ADD, NIL)';

TRANSITION
  (VAR, INACC) -> LOADUP,
  (TREEPTR, INACC) -> TRANS,
  (INACC, ONSTACK) -> 'GEN(PUSH, NIL)';

END;

$\text{GENERATE SUB;}$

DECLARE CONDITIONS
  (VAR, TREEPTR) MUTUALLY EXCLUSIVE,
  (INACC, ONSTACK) INTERNAL;

TERMINAL
  (INACC, VAR) -> 'GEN(SUB, OP2)',
  (INACC, ONSTACK) -> 'GEN(SUB, NIL)';

TRANSITION
  (VAR, INACC) -> LOADUP,
  (TREEPTR, INACC) -> TRANS,
  (INACC, ONSTACK) -> 'GEN(PUSH, NIL)';

END;

Figure 4.5 CGPL routines for ADD and SUB
IF VAR(OP1) THEN
  IF VAR(OP2) THEN DO;
    LOADUP(OP1);
    GO TO $SUB1;
  END; ELSE DO;
    TRANS(OP2);
    GEN(PUSH,NIL);
    LOADUP(OP1);
    GO TO $SUB2;
  END;
ELSE IF VAR(OP2) THEN DO;
    TRANS(OP1);
    GO TO $SUB1;
  END; ELSE DO;
    TRANS(OP2);
    GEN(PUSH,NIL);
    TRANS(OP1);
    GO TO $SUB2;
END;
$SUB1: GEN(SUB,OP2); GO TO $SUB_EXIT;
$SUB2: GEN(SUB,NIL); GO TO $SUB_EXIT;
$SUB_EXIT:

Figure 4.6 Possible PL/I program generated from CGPL
conditions are assumed to be boolean functions which take a tree operands as an argument. For example, VAR will be translated to VAR(OP1) when OP1 is to be tested for that condition. The programmer may choose to use either the PL/I preprocessor or a PL/I function procedure to realize this test. Notice that in the terminal definitions, the text to be substituted is in quotes. In some instances, it may be desirable (and is permissible) to use an identifier name to represent the text. The identifier will probably be a preprocessor variable to be expanded by the PL/I compiler. In either case, simple substitution is used by CGP.

In the case of transitions, the simple variable name (if it appears) is assumed to be a preprocessor routine with one argument. For example, LOADUP(OP1) is used to cause the transition from VAR to INACC on OP1. After the PL/I preprocessor has completed its task, this might read as

GEN(CLA,OP1)

Notice that the transition (INACC,ONSTACK) requires no parameters and is enclosed in quotes. Figure 4.6 shows the PL/I text (before preprocessing) which might be produced by CGP.

4.3 REGISTER ALLOCATION

In the above examples, the variable ACCFREE never appears, even though it was represented in the STS given.
Indeed, it was never really needed since the transitions always pushed the accumulator onto the stack when necessary. However, the case is not so simple if more than one computational register is available in a machine. Suppose that the T-machine were to be expanded so as to have a number of registers. Any attempt to assign states to each possible register conditions would be rather hopeless. However some method must be used to keep track of the use of registers so that the inadvertent use of a register will not destroy its contents. In some cases it may not be necessary to do this in the code generator however. For example, the GCL language requires that registers be declared whenever a new register is used. This results in assigning a new unique number to that register. When object code is finally generated, an optimizing routine will assign a real register number to these pseudo-registers and generate any necessary code for storing register contents. However, one often desires to generate object code directly, in which case a different scheme is needed.

Gries [GRI71] describes one method which is often used to keep track of registers at code generation time. A similar scheme is used in the NEW code generator. In this scheme, an array is used to store information about each register. Such information might include whether the register is currently in use, the contents of the register, some indication of how important it is to leave the regis-
ter contents undisturbed, etc. When a register is needed for computation in a generator, a request is made for a free register via a routine which will be called RESERVE. RESERVE takes one parameter which if non-negative, represents a register number. If that register is currently active, code is generated to save its contents; otherwise, the register is marked as being in use. On the other hand, a negative parameter indicates that any available register will suffice. If a register is free, that register is marked as being in use and the register number returned via the parameter; If no register is free, some priority scheme may be used to determine which register to store. When the generator no longer requires the register, the FREE routine is called and the old contents, if any, will be restored.

Unfortunately, it is quite difficult to keep track of when to FREE a register (or even if it is necessary). This is because if an unneeded value is in a register at some point, it may not be known whether it was put there by the generator or if it is there by virtue of having been there before the generator was entered.

Another difficulty is that RESERVE may choose to store a register which is currently being used by a generator. If this happens (or has a possibility of happening) constant checks may be necessary in order to determine whether a particular value is still in a register as is thought. Miller calls such a condition an accidental transition and
provides for it by performing the checks.

A better solution to the problem may be arrived at if it is assumed that there are more registers available in the machine than any generator ever requires. This is a fairly safe assumption since sixteen registers is typical of most computers which have them. In the method to be proposed, it is assumed that a generator is called via one routine, TRANS, which determines the generator to be called by investigating the current node in the syntax tree. In addition, a stack is kept which will keep track of register transactions.

Upon entry to the TRANS routine, a marker is placed on the stack to indicate entry to a new generator. Now when the generator calls RESERVE, any register stores will be recorded on the stack. If a choice must be made as to which register to free, none of the registers in the current stack region may be considered. If the generator sees that a register may be FREE'd then this is done normally. However after the generator completes, the TRANS routine will check the stack to see if any registers need to be restored and generate appropriate object code. Finally the stack marker is popped.

This rather simple scheme insures that all registers are freed after their current use is no longer needed. The overhead involved is not really much more than is usually required. Most importantly, the confusing logic which is often required for register allocation schemes
is eliminated.

4.4 THE PL/I FLOATING ASSIGNMENT

In this section, the floating assignment routine for PL/I given in Figure 4.2 will be rewritten in CGP. The purpose of this example is to show how the state machine concept of code generation and decomposition of actions can lead to a more compact representation of this routine. In addition, some new features of CGP will be introduced.

Looking at Figure 4.1 it is easy to determine that there are only two cases which need to be considered. In one case, the assignment is done by a Load followed by a Store; otherwise an MVC is used. These two cases are characterized by the fact that the Load-Store sequence is only used when both operands are aligned. Using this fact, and by generalizing the Load, Store, and Move operations, assignment can be made a trivial task.

When a Load is required, it is sometimes necessary to initially clear the register in order to accommodate a short data item. Similarly, a Move from a short data item to a long one may require an additional XC instruction to clear out low-order bits. These actions can all be consolidated into three general purpose routines, LOADARITH, MOVE, and STOREARITH, having the following characteristics:
LOADARITH(R,D,L1,L2) -- R specifies a register.
   D is a data reference to the item
   being loaded.
   L1 is the "implied" length of the
   data; that is, how the item will
   be treated after it is loaded.
   L2 is the actual length of D.
   The quantities L1 and L2 are used to
   determine the exact code sequence.

MOVE(T,D,L1)   -- T is a tree reference to the target.
   D is an RS reference to the source.
   L1 is the length of the source.
   The target length may be obtained
   from the tree.

STOREARITH(T,R)   -- T and R are as above. The length
   of the data item and the register
   type determine the instruction
   used.

Each of these routines could be written with the aid of
CGP to decide which case is applicable. In addition, these
routines are general enough that they would be useful in
almost all computational generators. Note that the speci-
fication of MOVE implies that there is some way to get an
RS reference for a data item.

Figure 4.7 shows a possible CGP routine for accomplish-
ing floating assignments when the above routines are assumed.
$DECIDEIF ASSIGNFLOAT;

DECLARE CONDITION ALLIGNED;

(ALIGNED, ALIGNED) ->

'RESERVE(R); RX=GETRXADDR@OP2);
LOADARITH(R, RX, LENGTH(OP1), LENGTH(OP2));
STOREARITH(OP1, R)';

ELSE ->

'RES=GETRS(OP2);
MOVE(OP1, RS, LENGTH(OP2))';

END;

Figure 4.7  A DECIDEIF block for floating assignment
This routine illustrates several additional features of CGPL. First of all, a $DECIDEIF block is used. DECIDEIF may be used in simple situations where no transitions are necessary, but decisions based on input conditions must be made. A DECIDEIF block might be useful in writing the routines outlined above. The ELSE condition is used to indicate that an action is to applied if all other conditions fail. It is a reserved condition name in CGP.

Admittedly, this example has been somewhat contrived since certain error conditions which might arise are not handled. However, it is a simple matter to provide for these. The example does illustrate a basic principle that careful analysis of commonly used actions in a code generator can lead to easily understood generators -- generators which are inherently easy to debug.

4.5 SUBSCRIPTION IN NEW

A more complicated example of a generator is illustrated by the subscription operator in NEW. This example will serve to introduce additional features of CGPL. It is interesting because it shows how one may write a generator for a complicated situation taken from a real compiler.

Recall that NEW defines several modes of elaboration which enforce various interpretations on certain constructs. In the case of subscription, the elaboration modes may be grouped as follows:
ISAVAL group
VAL CC
GOTO VOID

ISAREF group
REF PARAM
ASSIGN

In each group, the elaboration is the same until the subscription has been evaluated after which final actions may be required. For example, PARAM mode is treated as REF mode except that the result is tagged with a chained address word tag.

The interpretation of subscription for each group is as follows:

ISAVAL group: -- The subscription indicates an indexed load to the target register. The evaluation of the first operand must yield an address word and the second operand evaluates to an integer.

ISAREF group -- The value generated is equivalent to the expression

A MOD (B-I0 A)

where A and B are the values of the first and second operand respectively. This creates an address word which refers to the word at A[B].

ASSIGN -- The value in the target register is to be stored to the word at A[B].
The R-2 has seven instructions which are candidates for use in subscription operations. They are:

CLA  -- Used to obtain the CC value of a subscription in the special case that the index is a literal and the primary (or first operand) is in a register.

LD   -- Similar to CLA but used for VAL mode. LD can direct the value to any register, whereas CLA always loads to register 1 (or U).

STO  -- Used in ASSIGN mode if the primary is in a register and the index is a literal.

GET  -- Used in VAL mode to obtain the value in register 1. The primary must be in a register and the index addressable.

PUT  -- Used in ASSIGN mode when the value to be stored is in U. The primary must be in a register and the index addressable.

DOT  -- Used in VAL mode if the primary is in the target register and the index is addressable.

MOD  -- Used in REF mode. The primary is obtained in the target register. The quantity (B-I0 A) must be addressable.

Naturally, each of these instructions will be assigned to a terminal state in the generator.

Studying the description of these instructions leads to the following specifications for input states:
OP1 -- INREG or NOTINREG
OP2 -- LITERAL, ADDRESSABLE, or TREEPTR.

Since the code required is dependent on the current elaboration mode, global conditions for the three mode classes ISAVAL, ISAREF, and ASSIGN must be specified.

In the instruction descriptions, the U register was required to be used due to the way the R-2 operates. This causes some difficulty in VAL and REF mode since the U register may be the target register. In the NEW code generator, the assignment generator was written so that U was never picked as the target register. Such a restriction is not feasible for other classes. Since U might be the target register and this condition will affect the way code is generated, an additional global condition, TISU, is necessary.

Finally, some internal conditions must be specified to indicate states which arise in the generator. These states are:

GOTIT -- Indicates that \texttt{B-IO A} has been calculated.
GOTIT1 -- Same as GOTIT, except that the target register is U. In this case, it must be moved so that A may be brought to U prior to a MOD being generated.
ASSIGN1 -- Indicates that a PUT may be done since the source value has been moved to U.
The CGPL for this routine is given in Figure 4.8. This routine shows how CGPL statements might be merged with PL/I statements. Initially the variables A, B, and R are declared. A and B are used in the routine rather than OP1 and OP2 since accessing the tree is not as efficient a process as accessing local variables. To apprise CGP of this action, A and B are declared to be synonyms for OP1 and OP2, respectively. R will be used to hold a register reference if necessary. It is initialized to -1 in anticipation of being used as an argument to RESERVE.

In declaring the conditions for this routine, it becomes necessary to declare the domain of definition of certain conditions. This is because subscription is not symmetric as were the ADD and SUB operations in the T-machine. Here also MUTEX is introduced as an abbreviation for the longer MUTUALLY EXCLUSIVE.

Specifications for terminal states in the routine are similar to the previous examples, except for the necessity for specifying global conditions. Global conditions follow a semi-colon.

The transition section introduces several new notations. In certain instances, it is necessary to restrict the application of a transition or to choose one transition over another in the presence of certain conditions. For example,
SUBSCRIPTIONS: BEGIN;
DCL (A,B,R) FIXED BIN (31);
A=OP1; A=OP2; R=-1
$GENERATE SUBS;

DECLARE CONDITIONS
(INREG,NOTINREG) MUTEX DOMAIN OP1,
(LIT,ADDRTREEPTR) MUTEX DOMAIN OP2,
(ISAVAL,ISAREF,ASSIGN) MUTEX GLOBAL,
TISU GLOBAL, ASSIGN1 INTERNAL,
(GOTIT,GOTIT1) INTERNAL DOMAIN OP2;
DECLARE SYNONYM (A,OP1), (B,OP2);

TERMINAL
(INREG,LIT;ISAVAL) ->
'CODE(T,R2LD,0,A,VALUE(B));
GO TO HANDLEVALS',
(INREG,ADDR;ISAVAL) ->
'CPY(T,A); CODEA(T,R2DOT,B);
' GO TO HANDLEVALS',
(INREG,LIT;ASSIGN) ->
'CODE(T,R2STO,0,A,VALUE(B))',
(INREG,ADDR;ASSIGN1) ->
'CODEA(A,R2PUT,B)',
(INREG,GOTIT) ->
'CPY(T,A); A=T; CODER(A,R2MOD,B);
GO TO HANDLEREF';

Figure 4.8 CGPL for NEW subscription
TRANSITION

(ASSIGN, ASSIGN1) -> 'RESERVE(1); CPY(1,T)',
(NOTINREG; ASSIGN, INREG) -> TRANS_TO_T,
(((INREG, TREETR); ISAREF, GOTIT1) ->
 'RESERVE(1);
 CORDER(4,FLD,R2XFA,A); CODEA(3,R2ADD,B)',
(GOTIT1; TISU, GOTIT) ->
 'IF NOFREE=0 THEN DO; PUSH; B=0; END;
 ELSE DO; RESERVE(R); CPY(R,1); B=R; END',
(GOTIT1, GOTIT) -> '','
(NOTINREG; ASSIGN, INREG) ->
 'RESERVE(R); TRANS(A,R,VALMODE); A=R',
(TREETR, ADDR) -> 'MAKEADDRESSABLE(B, VALMODE)';

END;

GO TO EXIT;

HANDLEVALS: IF CCMODE THEN CORDER(8, R2USRGB, B);
ELSE IF GOTO_MODE THEN DO;
RESERVE(1); CPY(1,T); CALL_GOTOSOLVER;
END; GO TO EXIT;

HANDLEREFS: IF PARAMMODE THEN TAG(T, R2CHAD);
EXIT: END SUBSCRIPTIONS;

Figure 4.8 (Continued)
the second transition may be applied only if the mode is not ASSIGN mode. The third transition requires two side conditions; first, OP1 must be in a register, and the mode must be PARAM or REF. Actually two transitions are specified, one from LIT to GOTIT1 and one from ADDR to GOTIT1. It is clear that the transition is applied to a condition on the second operand since GOTIT1 is in the domain OP2. If some ambiguity could arise, the notation (.GOTIT1) is used for the transition target.

Placing extra conditions on a transition could possibly lead to ambiguities in deciding which transition to apply. This ambiguity is resolved canonically by CGP in the following manner:

(1) If two transitions specify the same mapping, then the one which has more side conditions has precedence.

(2) If similar transitions specify the same number of conditions, then they may not be logically ambiguous. For example, the transitions

(X;Y,T1)
(X;Z,T2)

are ambiguous unless Y and Z are in the same mutually exclusive class.

This example makes use of the fact that the NEW compiler includes provisions for peephole optimization [MCK65]. Thus, some of the redundant operations specified by the generator will be eliminated. Such operations are CPY's
in which the argument registers are the same. Another case is CC mode, where the LD, CLA sequence is replaced by a single CLA. One special case of subscriptions is not handled by this example. This case arises in the fourth transition, where it is necessary to move $B-I0 A$ out of U in order to make room for the value $A$. If the sixth transition had occurred previously, the $A$ value will be no longer needed after the generator completes its task. Hence a simple exchange of register $A$ and $U$ will suffice. This may be handled easily by introducing a new internal state at the sixth transition, say GOTINREG.

Learning a new notational device always creates some comprehension problems at first; therefore, this example may seem somewhat unreadable at first. However, the simplicity of the notation compared to the usual maze of nested conditional statements in most programming languages must be stressed. This example was chosen since it describes possibly the most complex routine in the NEW code generator, one which the author spent many hours developing. On the other hand, using an STS which required about thirty minutes to prepare, an hour was required to discover four logical errors in the original routine. Most of that time was spent in discovering how each case was handled by tracing through the complicated logical flow.
4.6 OTHER FEATURES

The notation which has been developed so far is sufficient to handle a number of common situations. In this section several features will be mentioned which are intended for handling more specialized cases and some mention of "statement" generators.

When CGP is used to develop an entire code generator, it will probably happen that a number of conditions and transitions will occur with great frequency. To provide for this some notation is necessary so allow naming transitions and groups of conditions. For this purpose, the DECLARE block is introduced. A DECLARE block may be used to name sets of declarations and transitions. For example,

\$DECLARE TRANSITION TTQA

(TREEPTR, ADDR) - MAKEADDR;
END;

may be used to declare a transition from TREEPTR to ADDR which may be used later in some generator. In addition several such transitions may be grouped together by a TRANSITIONLIST declaration

\$DECLARE TRANSITIONLIST TLIST (T1, T2, T3);

Similarly, a number of condition declarations may be grouped by

\$DECLARE CONDITIONLIST CLIST
condition declarations
END;
One possibility that has not appeared in the examples is that the domain of a transition might be restricted by certain conditions. This can arise if conditions do not have restricted domains; hence a transition might be applied to either operand. One real case in which this occurs is the algorithm developed by Sethi and Ullman [SET70] for generating optimal object code for arithmetic statements. The purpose of this algorithm is to determine which operand of a binary operation should be evaluated first. The process of evaluation is the same for each operand; hence, the transitions involved are the same. The algorithm might be specified by introducing a number of internal states, but this would be cumbersome at best. Instead, a notation is provided in CGP to allow transitions to have restricted domains. The syntax is

\[(C1, C2) \text{ APPLY TO OPn ONLY IF } C3 - \ldots\]

which indicates that condition $C3$ must hold true if the transition is applied to a condition in the OPn domain.

So far, only generators for operators have been discussed. However, it is possible to use CGP for "statement" generators also. As Wilcox has pointed out, statements present some difficult since they are usually translated into a series of expression evaluations which may have intervening code. An example is the FOR statement in ALGOL or the DO in PL/I for expressing repetitions or loops. In such a statement, it is necessary to perform
several translations. First an iteration variable must be created and initialization of the variable must be performed. Then step sizes and iteration limits must be evaluated, either of which may be absent. Finally the body of the loop and tests for completion must be generated. The form of the test usually depends on conditions which existed at the time initialization was done. In CGP, such a translation may be performed by using separate DECIDEIF or GENERATE blocks for each evaluation if necessary. Normally communication between the various pieces of a statement is done by using a boolean variable to indicate if certain conditions existed in a previously executed part. The reason for this is to prevent a complicated test to determine the same thing. In CGP, the creation of such a variable may be done automatically if desired by the use of a RECALL prefix. A RECALL prefix may precede any transition or terminal and indicates that a boolean variable should be declared and set to TRUE if that transition or terminal is applied during the translation. The form is

\[
\text{RECALL(B) (T1,T2) ...}
\]

In a later block, the variable B may now be specified as a global condition.

4.7 POSSIBLE EXPANSIONS TO THE LANGUAGE AND CONCLUSION

In this chapter, the CGP language has been outlined. Since CGP is intended to be an aid to the programmer, it is
possible that some extensions to it may be desirable when it is used in environments other than those mentioned here. For instance, the host language might be changed to some language other than PL/I. In this case, it might be desirable to introduce a preprocessor similar to that in the PL/I language. Similarly, if a language other than PL/I is used, it may be necessary for the generation of conditionals to be changed and label generation conventions may be required. For instance, if FORTRAN is the host language, CGP would need a reserved block of statement numbers for its own use.

If CGP were to be used for several compilers, all to produce object code for a common machine, some facility might be provided for a common method of instruction generation.

The point of this is that the description given here is certainly flexible and is only intended as a model in developing a useful system. Actual implementations of such a system may vary. It is the "spirit" of the language which is important and not the details of syntax and implementation.

In the next chapter, an outline of how CGP might be implemented will be given. In that chapter, the major concern will be how one may organize the system so as to make its implementation efficient. In addition a discussion of methods for generating decision trees will be given.
CHAPTER V
IMPLEMENTATION

Figure 5.1 is a block diagram of the internal organization of the CGP. The first phase is a parser which searches the source for CGP statements. When these are found, they are analyzed for proper syntactic structure and a symbol table is built. Depending on the nature of the block found, the decision processor or the STS builder is called next. The STS builder uses the information gathered by the parser to build an initial STS which contains only input states. Any input states which are also terminal states are marked at this time. Terminal states which include internal conditions are then added. The STS is further expanded by the transition analyzer. Configurations which result from transitions are added. After all transitions have been added, they are analyzed in order to determine the best transition path for each input state. At this time errors may result due to incompleteness or logical inconsistencies in the input. Finally the decision processor determines the logical flow of tests needed to realize the translation. Finally, the optimizer examines this flow in order to merge redundant actions.

5.1 THE PARSER

Since the actual parsing of CGPL is fairly simple, the major work done by this phase is to build a data
Figure 5.1 The CGP System
structure containing all input information. All condition names are stored in a symbol table containing a pointer into this structure which identifies the condition. Another table is required for storing the text of transitions and terminal actions. The data structure built is fairly elaborate in order to make access to any information readily available.

Figure 5.2 shows the various elements of the data structure built by the parser. Each element will be described in turn.

CONDITION ENTRIES

Conditions are the most complicated elements of the CGP structure since they are interrelated in various ways. A condition entry contains the following information:

| TYPE             | -- Identifies the entry as a condition |
| MUTEX CHAIN      | -- A circular linked list containing all mutually elements in a class. |
| DOMAIN           | -- A bit field identifying the domain of the condition if it is a local or internal one. Each bit represents an operand in order. |
| CONDTYPE         | -- Identifies the condition as being local, global, or internal. |
| TRANSITION CHAIN | -- A circular list of transitions which originate with this condition. |
### Figure 5.2
Elements of the data structure built by parser.
NAME -- Pointer to the condition name.

TRANSITION ENTRIES

Transition entries describe transitions as follows:

TYPE -- Identifies the entry as a transition.

FROMPTR -- Indicates the initial condition.

TOPSTR -- Indicates the terminal condition.

RESTRICTION COUNT -- The number of side-conditions on this transition.

DOMAIN -- The domain of this transition. This is used if an APPLY option was present.

RESTRICTION LIST -- A pointer to a condition list containing the restricting conditions. The number of elements on this list is equal to RESTRICTION COUNT.

TEXTPTR -- A reference to the text associated with the action for this transition.

QUOTED -- Indicates if the text was originally in quotes.

NAME -- The name of this transition if it was declared.

TRANSITIONLIST ENTRIES

A transition list entry is built in response to a TRANSITIONLIST declaration. The fields of the entry are:

TYPE -- Identifies entry as a transition list.
TRANSITION LIST
-- Pointer to a chain of transition pointers.

NAME
-- The name assigned to the transition list.

CONDITIONLIST ENTRIES

This entry is similar to the one for a transition list, except the list is a list of conditions.

In addition to building this data structure and the symbol table, the parser will assign row numbers in the STS to each condition. In addition, bit strings representing the terminal configurations will be built. Of course, the parser will also be responsible for detecting syntax errors, duplicate declarations, and violations of domain restrictions.

After the parser finishes scanning all statements, the decision analyzer or the STS builder is called.

5.2. THE STS BUILDER

Before further analysis can be carried out an initial STS must be built. This STS is simply a bit matrix for marking conditions. It consists of a column for each initial configuration and each terminal configuration. A check is made to insure that duplicate columns are not created when an input configuration is also a terminal configuration. Associated with each row and column of the STS is an information field. The row information field
contains information about the domain of the condition and a pointer to the condition represented. Conditions with domains over several operands are duplicated. The column information field is really an array containing information about transitions which apply to each operand. This information is the type of configuration, the condition level, and a pointer to a chain of applicable transitions. The condition level is a count representing the number of conditions applicable in any transition from this configuration. This field is used to enforce the precedence of transitions mentioned earlier.

After the initial STS has been built, the transition analyzer is called.

5.3 THE TRANSITION ANALYZER

The job of the transition analyzer is to determine transitions which will generate the shortest path from each initial state to a terminal state. Initially CGP will consider all transitions to have equal weight. However it is a simple matter to account for transitions which have cost functions applied. This might produce more efficient code generation schemes, but has not been considered here.

Before the best path can be found, all possible transitions must be determined by the analyzer. To do this, the transitions are applied one at a time in descending order according to the restriction count. If a transition with restrictions is applied to a configuration,
its condition level is set accordingly. If a transition generates a configuration which is not present in the STS, then that configuration is added. This is preferable to building an STS for all possible configurations since some may not be realizable from the transitions given.

After all transitions have been discovered, each path from input states must be examined to determine the shortest one. This problem may be solved as a "shortest route problem" as discussed in Hillier and Lieberman [HIL67].

In performing this last algorithm, it will be necessary to discover if any circular transitions exist or if an input state has no applicable transitions (that is, if it is not also a terminal state). Any errors of this nature will be reported.

Finally, a transition list will be built for each input state. If input states have transition lists in common, they will be merged and considered as a common state. The decision analyzer will make use of this information to generate the tests necessary to complete the generator routine.

5.4 THE DECISION TABLE PROCESSOR

The last difficult task in the CGP is to determine a testing scheme for conditions in order to arrive at each transition stream with the fewest number of tests. For this purpose, the STS may be considered to be a decision table. The process of generating a program from a decision
table will be considered in this section.

A decision table is a device which has been used to represent complicated logical situations when a flowchart representation proved to be untenable. It consists of a number of rows, representing conditions, and columns representing actions to be performed when certain conditions are met. Usually an entry in a decision table may consist of only the values \( Y \), \( N \), and \( \_ \), representing true, false, and don't care, respectively.

The decision table analyzer will build a decision table from the STS by merging certain rows if they specify the same transitions. Only the input configurations are considered.

Given a decision table, an unsolved problem is to find an efficient algorithm which will generate a decision tree from that table having the minimum number of tests. A decision tree is a rooted, binary tree satisfying the conditions (1) each internal node of the tree is a condition; and (2) no condition appears more than once in any path from the root to a terminal node. Figure 5.3 shows a typical decision table and two decision tables which realize it. If the cost of a decision tree is defined to be the number of internal nodes, then the cost of the first tree is 12, while the cost of the second tree is 10.

All algorithms which have been proposed for generating decision trees from decision tables use what might be called
Figure 5.3  A decision table with two trees which realize it.
a top-down approach. That is, they proceed by picking a condition to test and then partitioning the decision table into two new decision tables. That is, suppose condition C is picked, then two new decision tables are created by deleting the C row. The new decision tables are made up of those columns in which C was specified to be true and false, respectively. Columns in which C was a don't care are duplicated in each of the new tables. This process proceeds in a recursive fashion until a tree has been generated. Figure 5.4 shows an APL function which performs one such algorithm and generates a skeleton PL/I program. A sample input and skeleton program are given in Figure 5.5. Here 1, 0, and -1 represent true, false, and don't care.

The basic fact on which all decision table algorithms are based was proved by Yasui [YAS71] and is stated below. By a loss-free condition is meant one which has no don't cares in its row of a decision table.

**THEOREM:** Assume that a decision table has a loss-free condition C and consider the set of all trees which realize that table. Then for any such tree T, a tree T' can be exhibited which has C as the root and is such that cost(T') ≤ cost(T).

In practical terms, this theorem means that if a loss-free condition exists in a decision table, it should be
\[ \psi \text{DECISION[\(t\)]}\psi \\
\]

\[ \psi \text{DECISION } t; z \]

[1] \(\text{LEVEL}+\text{LEVEL}+1\)
[2] \(\rightarrow(\wedge/\text{''1 2 }_{=pT})/\text{ACTION}\)
[3] \(\rightarrow(2=1+pT)/\text{TEST}\)
[4] \(z\rightarrow=1 =1\ 1 +T\)
[5] \(z\rightarrow=1+z\rightarrow=1)/z\)
[6] \((3\times\text{LEVEL})p\ 'i''IF'c'':T[z\rightarrow=0]'; 'THEN DO';\)
[7] \(\text{DECISION } t[(z\rightarrow=1+pT)/i1+pT/(1,1+0=\text{T}[z\rightarrow=0])/i1+pT]\)
[8] \((3\times\text{LEVEL})p\ 'i''END';\)
[9] \((3\times\text{LEVEL})p\ 'i''ELSE DO';\)
[10] \(\text{DECISION } t[(z\rightarrow=1+pT)/i1+pT/(1,1+1=\text{T}[z\rightarrow=0])/i1+pT]\)
[11] \((3\times\text{LEVEL}+1)p\ 'i''END';\)
[12] \(\text{OUT}:\text{LEVEL}+\text{LEVEL}-1\)
[13] \(\rightarrow0\)
[14] \(\text{ACTION}:(3\times\text{LEVEL})p\ 'i''A':T[0;1]'; 'i''\)
[15] \(\rightarrow\text{OUT}\)
[16] \(\text{TEST}:(\wedge/\text{''1} =1+pT[1];1)/\text{ACTION}\)
[17] \('\text{LOGICAL ERROR IN DECISION TABLE}'\)
[18] \(T\)
[19] \(\rightarrow\text{OUT}\)
\]

Figure 5.4 An APL Function for creating programs from decision tables
\[
\begin{array}{cccccc}
A & 0 & 1 & 2 & 3 & 4 & 5 \\
1 & 0 & 1 & 1 & 1 & 0 \\
2 & 1 & 0 & 1 & 0 & 1 \\
3 & 0 & 1 & 1 & 0 & 1 \\
\end{array}
\]

**DECISION A**

```plaintext
IF C1 THEN DO;
  IF C2 THEN DO;
    A3;
  END;
  ELSE DO;
    IF C3 THEN DO;
      A2;
    END;
    ELSE DO;
      A4;
    END;
  END;
ELSE DO;
  IF C3 THEN DO;
    IF C2 THEN DO;
      A5;
    END;
    ELSE DO;
      A2;
    END;
  ELSE DO;
    A1;
  END;
END;
```
picked as the condition to test. It is trivial to show that all loss-free conditions are equivalent. The difficult comes in choosing among conditions which are not loss-free. Yasui discusses two algorithms for generating decision trees, each of which uses a different policy for distinguishing among conditions having some loss. The first merely chooses the first such condition; the second chooses the condition having the least loss. Neither algorithm can be favored since decision tables may be constructed for which either algorithm produces the least costly tree.

An algorithm will be presented here which is based on the above algorithms and on the following:

**THEOREM:** Given a decision tree $T$ with root $C$ having subtrees $T_1$ and $T_2$, the left and right subtrees of $C$ respectively. Suppose that $T_1$ and $T_2$ have a leaf in common, $B$. In addition, the path from the root of $T_1$ to $B$ is the same as the path from the root of $T_2$ to $B$. Then a new tree $T'$, which realizes the same decision table as $T$, may be constructed such that $\text{cost}(T') < \text{cost}(T)$.

**PROOF:** The tree $T'$ is built as follows. Simultaneously, walk $T_1$ and $T_2$ in preorder [KNU68], comparing them at each node. If at any step, $T_1$ and $T_2$ contain the same node, then include that node in $T'$. If the nodes of $T_1$ and $T_2$ are not the
same, then in $T'$ insert a subtree consisting of $C$ and having as left and right subtrees the trees whose roots are currently being compared in $T_1$ and $T_2$, respectively. The tree $T'$ so created will have at least one less leaf than $T$ since the leaf $B$ will only appear once in $T'$. This is because the paths to $B$ in $T_1$ and $T_2$ are the same. The procedure does not produce any new leaves. Therefore, $\text{cost}(T') \leq \text{cost}(T)$.

This theorem provides a procedure for reducing the cost of a decision tree when the hypothesis is satisfied. It is possible to build an initial decision tree in a manner which insures the hypothesis of this theorem will be satisfied whenever possible. The algorithm consists of two algorithms which will call each other recursively:

**ALGORITHM A**

The input to this algorithm is a decision table, $D$. It chooses a condition by picking the row in the decision table which has the least number of don't cares. If several rows have this property, the first one is picked. Then the decision table is partitioned as usual into tables $D_1$ and $D_2$. Apply Algorithm B to these decision tables.
ALGORITHM B

The input to this algorithm is two decision tables, D and D'. If D and D' have no common action column, then apply Algorithm A to D and then to D'. Otherwise, three cases need to be considered.

Case 1: D and D' each have a loss-free condition.
If some condition C is loss-free in both D and D', then choose condition C as the next condition to be tested and go to step 1.

Case 2: D or D' has a loss-free condition, but not both.
Suppose D has the loss-free condition. If more than one condition in D is loss-free, choose the condition which has minimum loss in D'. Otherwise, the unique loss-free condition is chosen. Go to step 1.

Case 3: Neither D nor D' has a loss-free condition.
Choose a condition such that the total loss in D and D' is minimized. Go to step 1.

Step 1: Partition D as usual using the chosen condition to obtain tables D1 and D2. Apply Algorithm B to D1 and D2.
Step 2: Partition D' as usual using the chosen condition to obtain tables D1' and D2'. Apply Algorithm B to D1' and D2'.
By applying Algorithm A to a decision table, a sequence of tests is generated from which a tree may be built. Algorithm B is fairly complicated. Its purpose is to choose conditions so that common paths are created in the resulting decision table whenever possible. After these algorithms have been used to build the decision tree, the algorithm in the proof of the second theorem may be used to reduce the cost of the resulting tree if possible.

The above algorithms are suggested as an alternative to those given in the literature. They are not optimum algorithms but are somewhat better than previous algorithms. Unfortunately, these algorithms are somewhat costly since the decision tree must be built. However, it does not seem possible to generate decision trees directly from decision tables in an optimal fashion. Obviously this area of research requires further attention. The algorithms given here are intended as motivation for finding relationships between a decision table and transformations on decision trees. Another approach to the problem might be to find a bottom-up procedure which combines actions which differ in only one condition (assuming that a don't care always matches a Y or N). The procedure would then build a decision tree in a bottom-up fashion.

Fortunately, the number of conditions to be considered in a code generation routine is small. Therefore, the cost induced by not generating the optimum decision tree
will be small. Although the loss due to condition testing cannot be recovered, common actions may be merged. This is done by the optimization routine to be discussed next.

5.5 OPTIMIZATION

After the logic flow of the generator has been prescribed by the decision table analyzer, a certain amount of optimization is possible. This is done by examining the transition sequences from the input states. If any of the sequences are the same or end in common sequences, then the text specified is moved out of the nested conditional statement and replaced by a GOTO statement. The deleted text is inserted at the end of the routine, appropriately labeled. An example of how this optimization works was given in Figure 4.6.

5.6 AUXILIARY OUTPUT

Aside from its intended purpose, that of generating a PL/I program from a CGPL description, the CGP can aid the programmer in other ways. The most obvious aid is that of producing error messages when logical errors are found in the input.

However, CGP may also be directed to produce optional listings which document how each input state is handled in the routine. This feature of being self-documenting is certainly unique among code generation systems. The listing produced can be an invaluable aid in debugging.
5.7 SUMMARY

The purpose of this chapter was to demonstrate the feasibility of implementing the CGP system. The major routines have been outlined and various algorithms have been suggested. None of the algorithms presented is particularly difficult to implement; however, the CGP system is likely to be somewhat inefficient. It is believed that the time spent in using CGP will however be worthwhile when balanced against the advantages of using the system.
CHAPTER VI

CONCLUSIONS AND FURTHER WORK

This thesis attacks the problem of developing a method for specifying the logical flow of a code generation in a simple form from which a code generator may be automatically created. That the problem is a real one is substantiated by Elson and Bade when they state that the logic required for an optimizing PL/I compiler is about fifty times that of a similar FORTRAN compiler. Modern programming languages are much more complex than their predecessors.

The measure of a compiler's success is almost completely dependent on whether or not it generates correct object code in every situation. Current methods of compiler construction provide almost no aid to the programmer in achieving success.

All systems or languages for code generation in use today provide little more than the ability to easily access the input data structures and generate instructions for the target machine. These features have been completely eschewed in the CGP system in favor of a more general approach. It is felt that such routines are the most easily written and debugged parts of a compiler, and hence, contribute little to the overall difficulty of writing a working compiler. CGP is intended to be used for any compiler, generating code for any machine. The one re-
striction being that the input to the code generator be in the form of a syntax tree. Even this restriction could be removed if desired.

The advantages of CGP over other systems are as follows:

1. The notation used is concise and readable.
2. The programmer is freed from specifying the detailed logic of his routines, but instead may think of the code generation process as a state-transition process.
3. The system is capable of producing diagnostics which inform the programmer of logical errors or incompleteness in his specification.
4. The system is capable of documenting the code generation routines by producing a readable listing of actions taken for every possible situation.
5. The programmer may define his own terms rather than being tied to any particular scheme; hence a code generator for virtually any machine may be produced.
6. Since the readability of the host language source is not an issue, CGP may generate programs which are much more efficient than the human programmer might produce.

The CGP system as presented here is not by any means perfect. Its syntax is somewhat clumsy and somewhat un-
natural. This failing is, however, somewhat minor. The greatest drawback is that no implementation exists. Therefore the author has tried to demonstrate (as briefly as possible) the feasibility of implementing such a system. Since no implementation has been attempted, CGPL has not been put to the ultimate test of using it in writing a production compiler. However, the author believes that subsequent implementation of the system will prove it to be a usable one. If nothing else, the author believes that this thesis adequately demonstrates the advantages of the state-transition approach to code generation. This technique is quite helpful in disentangling extremely complicated code generation routines.

The work of many others has been helpful in the production of this thesis. Perry Miller's work on code generation from a machine description is most enlightening. The idea of considering code generation as a finite state machine is his, however, the use of state transition schema is this author's. The state transition schema is obviously inspired from similar representations of sequential switching circuits. CGPL is an attempt at generalizing Miller's system to make it more usable.

6.1 POSSIBLE AREAS FOR FURTHER RESEARCH

The CGP is a system which in some sense "understands" the process of code generation. As such, it aids the programmer by being capable of discovering certain errors in a
code generator. As the study of formal semantics and the
development of descriptional languages such as those of
Mutshler [MUT73], Lucas, et. al. [LUC68], and Bell and
Newell [BEL71] continues, it becomes plausible that sys-
tems could be designed which would be capable of under-
standing the meaning of programming languages. Such a
system could conceivably "discover" implementations for
programming languages given a description of the target
machine.

One possible approach to this goal might be a program
which can find programming "tricks" for a given computer,
much as a human programmer does. Such a program might be
capable of writing simple programs similar to those as-
signed in elementary programming courses. The next step
would be for this program to learn how to organize more
complicated programming tasks. This approach might even-
tually lead to truly automatic code generation.
BIBLIOGRAPHY


