GENERATING MACHINE CODE FOR HIGH-LEVEL
PROGRAMMING LANGUAGES

TR 71 - 103

A Thesis
Presented to the Faculty of the Graduate School
of Cornell University for the Degree of
Doctor of Philosophy

by
Thomas Richard Wilcox

September, 1971
BIOGRAPHICAL SKETCH

Thomas Richard Wilcox was born on March 3, 1945 in Rochester, New York and passed his childhood nearby in Greece, New York. After four years of study at The University of Michigan, he received a Bachelor of Science with High Distinction and High Honors in Mathematics in April, 1967. In September of that year, he began graduate study in the Department of Computer Science at Cornell University. He is a member of Phi Kappa Phi and Phi Beta Kappa honorary societies and the Association for Computing Machinery.
# TABLE OF CONTENTS

1. Introduction -- The Compilation Process  
   1.1 The Recognition Process  
       The Abstract Program Tree  
   1.2 The Code Generation Process  
       Translation  
       Global Optimization and Coding  
   1.3 The Intermediate Text  
   1.4 Summary of Content  
       Translation Model  
       Coding Model  
   1.5 Contributions  

2. Translation  
   2.1 The Translator Environment  
       Simplifying Assumptions  
   2.2 Translator States  
       State Components  
   2.3 Semantics and Structure  
       Postfix-Translatable Operators  
       Prefix-Translatable Operators  
       Variadic Operators  
       Hybrid Text  
       Tree-Translatable Operators  
       M-pass-Translatable Operators
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>User-Defined Operators</td>
<td>55</td>
</tr>
<tr>
<td>2.4 Intermediate Text Structure and BWF-Manship</td>
<td>58</td>
</tr>
<tr>
<td>2.5 Statement Translation</td>
<td>63</td>
</tr>
<tr>
<td>2.6 Data References</td>
<td>68</td>
</tr>
<tr>
<td>An Example from PL/C</td>
<td>71</td>
</tr>
<tr>
<td>Explicit Address Functions</td>
<td>72</td>
</tr>
<tr>
<td>2.7 Summary of Translator Model</td>
<td>74</td>
</tr>
<tr>
<td>3. Coding</td>
<td>78</td>
</tr>
<tr>
<td>3.1 Communication with the Translator</td>
<td>79</td>
</tr>
<tr>
<td>3.2 Interpretive Coding Language Templates</td>
<td>82</td>
</tr>
<tr>
<td>Code Templates in the Compiler Milieu</td>
<td>84</td>
</tr>
<tr>
<td>ICL Statements</td>
<td>89</td>
</tr>
<tr>
<td>3.3 Value Descriptors</td>
<td>91</td>
</tr>
<tr>
<td>Address Representation</td>
<td>94</td>
</tr>
<tr>
<td>Values in Registers</td>
<td>99</td>
</tr>
<tr>
<td>Values in Storage</td>
<td>101</td>
</tr>
<tr>
<td>Uses of the Descriptor Mechanism</td>
<td>102</td>
</tr>
<tr>
<td>Address Chaining in PL/C -- Another Example</td>
<td>111</td>
</tr>
<tr>
<td>3.4 Coder Structure</td>
<td>116</td>
</tr>
<tr>
<td>Register Management</td>
<td>120</td>
</tr>
<tr>
<td>Storage Management</td>
<td>123</td>
</tr>
<tr>
<td>Object Machine Instruction Generation</td>
<td>126</td>
</tr>
<tr>
<td>Forward References</td>
<td>129</td>
</tr>
<tr>
<td>3.5 Summary</td>
<td>131</td>
</tr>
<tr>
<td>4. Optimization</td>
<td>132</td>
</tr>
<tr>
<td>4.1 Optimal SLM Sequences</td>
<td>132</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Chapter 1.

1.1 Logical Compilation Phases and Their Names
1.2 Internal Representations in the Recognition Process
1.3 A PL/I Procedure
1.4 Abstract Program Tree for Program in Figure 1.3
1.5 Internal Representation Flow in the Code Generator
1.6 A Typical Source Language Machine Program

Chapter 2.

2.1 Diagrams for Formal PDT and Model Translator
   a) Organization of a Formal Pushdown Transducer
   b) Basic Organization of Model Translator
2.2 Semantic Template Interpretation
2.3 Postfix Translation Algorithm
2.4 Prefix Translation Algorithm
   a) Prefix Translation Algorithm
   b) Flowchart of Operations Performed by a Driver
2.5 Variadic Postfix Translation Algorithm
2.6 Operations Performed by Variadic Driver
2.7 Hybrid Text Translation Algorithm
4.2 Piecewise Coding Constraint 136
4.3 Selection of the Proper SLM 137

An Example of the Effect of SLM Definition 138

4.4 Local Optimization in the Coder 141
4.5 Implementing Global Optimization Strategies 144

Multiple-Pass Code Generation 146

SLM Instruction Operands - Template Parameters 148

4.6 Descriptor Management -- A Second Look 152
4.7 Summary of Multi-Pass Modifications 160

5. Transportability 163

5.1 The Interface Module 166

5.2 The Proper SLM Instructions for 170

Machine Independent Translation

5.3 Summary 172

6. Conclusion 173

6.1 Further Work 173

6.2 Summary of Code Generation Model 174

Appendix I 180

References 196
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.8</td>
<td>A Complete Translator</td>
<td>50</td>
</tr>
<tr>
<td>a)</td>
<td>Translator Algorithm</td>
<td>51</td>
</tr>
<tr>
<td>b)</td>
<td>Operations Performed by Tree Driver</td>
<td>52</td>
</tr>
<tr>
<td>2.9</td>
<td>A Semantic Template for the PL/I IF</td>
<td>59</td>
</tr>
<tr>
<td>2.10</td>
<td>Driver for IF-Statement</td>
<td>60</td>
</tr>
<tr>
<td>2.11</td>
<td>Semantic Routines</td>
<td>61</td>
</tr>
<tr>
<td>2.12</td>
<td>Basic Process Flow in STATEMENT TRANSLATOR</td>
<td>65</td>
</tr>
<tr>
<td>2.13</td>
<td>DO-Statement Driver</td>
<td>66</td>
</tr>
<tr>
<td>2.14</td>
<td>Model Translator Structure</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td>Chapter 3.</td>
<td></td>
</tr>
<tr>
<td>3.1</td>
<td>One-Pass Code Generator</td>
<td>81</td>
</tr>
<tr>
<td>3.2</td>
<td>MAD Defining Sequence for A .MOD. B</td>
<td>85</td>
</tr>
<tr>
<td>3.3</td>
<td>Code Template for MOD(A,B)</td>
<td>92</td>
</tr>
<tr>
<td>3.4</td>
<td>Value Descriptor</td>
<td>95</td>
</tr>
<tr>
<td>a)</td>
<td>PL/I Structure Definition of Value Descriptor for IBM /360</td>
<td>95</td>
</tr>
<tr>
<td>b)</td>
<td>Key to Diagramatic Representation of Value Descriptor</td>
<td>95</td>
</tr>
<tr>
<td>3.5</td>
<td>Register Status Descriptor</td>
<td>100</td>
</tr>
<tr>
<td>a)</td>
<td>PL/I Structure Definition of RSD for IBM /360</td>
<td>100</td>
</tr>
<tr>
<td>b)</td>
<td>Key to Diagramatic Representation of RSD</td>
<td>100</td>
</tr>
<tr>
<td>3.6</td>
<td>Address Computer for 360 Coder</td>
<td>103</td>
</tr>
<tr>
<td>3.7</td>
<td>LGPR (Load General Purpose Register) Statement Processor</td>
<td>105</td>
</tr>
<tr>
<td>3.8</td>
<td>Management of Result Descriptors</td>
<td>107</td>
</tr>
</tbody>
</table>
a) Descriptors After Computing Result in R1 107

b) Descriptors Just After Reallocating Result
   Register (R1) 107

c) Descriptors After Loading Register R1
   with New Value 108

d) Key to Situations Described by a), b),
   and c) 108

3.9 Optimizing Base Register Loads 109

   a) Run-Time Environment 109

   b) Initial Descriptor Arrangement 109

   c) After Loading Base Address and B 110

   d) Key to Situations Described by b) and c) 110

3.10 An Example from PL/C 113

   a) Sample PL/I Program Segment 113

   b) Code Generated for Statement 12 in a)
      by PL/C 113

   c) Run-Time Environment while Executing
      Statement 12 114

   d) Descriptor Structure while Compiling
      Statement 12 115

3.11 Coder Calling Structure 117

3.12 A Control Module for Interpreting ICL 118

Chapter 4.

4.1 Optimal 360 Multiply and Divide 134

4.2 Optimal SLM Sequence Using Simple SLM 139

4.3 360 Code for SLM in Figure 4.2 139
4.4 360 Code for DO Using More Complex SLM  140
4.5 One-Pass Code Generator  147
4.6 Code Template for Binary Addition (One-Pass)  154
4.7 Flowchart for PRIV ICL Statement  (General Purpose Registers)  157
4.8 Code Template for Binary Addition (Multi-Pass)  158
4.9 Information Flow in Multi-Pass Code Generator  161

Chapter 5

5.1 One-Pass Portable Code Generator Organization  168
5.2 Multi-Pass Portable Code Generator  169

Chapter 6.

6.1 Logical Phases of Compilation and  Defining Maps  176

LIST OF TABLES

1.1 The Partitioning of Phases into Passes  15
2.1 Summary of Translator Algorithms and texts  57
3.1 Encoding of Access Function in Value Descriptor  99
To My Parents
ACKNOWLEDGEMENTS

The author wishes to express his gratitude to Professor Alan C. Shaw for his invaluable guidance and encouragement throughout the evolution of this thesis. He would also like to thank Professors Howard L. Morgan and John Hopcroft for serving on his special committee.

The author also wishes to thank Professor Richard W. Conway for providing the opportunity to participate in the PL/C compiler project and his continuing interest and support of the author's work in that effort. The experience gained in that endeavor contributed significantly to the formulation of this thesis.

The patient and accurate key punching of Miss Marilyn Perry was a considerable help and is much appreciated.

(This research has been supported in part by National Science Foundation grants GJ-108 and GJ-28176 and the PL/C Project at Cornell University.)
CHAPTER 1.  INTRODUCTION -- THE COMPILATION PROCESS

The purpose of the research reported here is to investigate the process through which machine code should be generated in compilers for modern high-level programming languages such as FORTRAN, ALGOL, and PL/I. In particular, this thesis examines a model for code generation which has been abstracted from the code generator designed by the author for PL/C -- the Cornell compiler for PL/I [Wilcox 1970]. For this examination, the code generation process is divided into four subprocesses or phases: storage allocation, translation, global optimization, and coding. In this thesis the translation and coding phases are developed in detail. The model provides for the others, but they are not discussed to any length. The code generator modeled is not only compact and flexible to the point of being both extendible and transportable, but is also capable of producing reasonably efficient, if not optimal, machine code sequences.

The process of code generation is embedded in the larger process of compilation. Consequently, it will be helpful to first review the general nature of the compilation process. The logical organization of compilation into phases is diagrammed in Figure 1.1. The function of each of these phases will be explained in the following sections. Although most of the phases of compilation are universally recognized, their names are not. The names for the compilation phases as used in this thesis are given in Figure 1.1a. The names used
Figure 1.1 Logical Compilation Phases And Their Names:

a) This Thesis
b) McKeeman [1970]
c) Gries [1971]
by McKeeman [1970] and Gries [1971] to identify the same phases are indicated in Figures 1.1b and 1.1c respectively.

1.1 The Recognition Process

The first half of compilation is the recognition process (see Figure 1.2). Its function is to determine from the stream of input characters forming the source language program what the program is attempting to do. In particular, it must determine two aspects of the program:

1) The representational aspect: the data items which are manipulated by the program and the attributes these data items are to have.

2) The functional aspect: the operations to be performed on these data items and the order in which they are to be applied.

Recognition is accomplished in three distinguishable phases: lexical analysis, syntax analysis, and semantic analysis.

The lexical analyser transforms the program from its concrete syntax representation to a tokenized form. The program is still a linear string of symbols, but the input characters have been grouped into tokens of the language -- integers, identifiers, keywords, punctuation, operators, etc. The definitions for the new tokens -- e.g., identifiers -- are recorded in the symbol table. The tokens are rearranged by the syntax analyser into a tree structure which reflects the
Figure 1.2 Internal Representations In The Recognition Process
partial ordering of operations in the program as demanded by the rules for forming and using the programming language. The semantic analyser replaces identifier names in the tree with unique variable identification. Any ambiguity resulting from naming conflicts is resolved. The symbol table is modified to reflect this resolution. This final tree representation of the program is called the abstract program tree (APT).

```
PROG: PROCEDURE OPTIONS(MAIN);
   DECLARE (C(10), A) FLOAT, B FIXED;
   
   GET DATA (C,N);
   DO I=1 TO N;
      PUT LIST (FUNC (.5));
      END;

   FUNC: PROCEDURE(X) RETURNS (FLOAT);
   DECLARE (S,X) FLOAT, I FIXED;
   IF A=B THEN RETURN(SUM(C*X));
   ELSE DO;
      S=0;
      DO I=1 TO 10;
         S=S+(C(I)*X)*(A/B);
      END;
      RETURN(S);
   END;
   END PROG;
```

Figure 1.3 A PL/I Procedure

As an example of the processing done by the recognizer, consider the PL/I procedure shown in Figure 1.3. (Throughout this thesis programming language examples will be drawn from the PL/I language as defined in the PL/I__ (F) Language Reference Manual [IBM 1969] and implemented on the IBM System/360 computer by the PL/C compiler [Conway 1970].) The recognition process would construct for this an APT such as
the one shown in Figure 1.4

The Abstract Program Tree

Note that the APT used in this thesis to formalize the output of the recognizer is different from the parse and syntax trees used by other authors [Gries 1971; McKeeman 1970] when describing compilation. For example, no higher-level constructs such as <block> or <DO-group> are present. The APT can, of course, be derived from these other trees. The APT formalism is preferred, however, because it captures the essential properties of the source program which are of interest to the code generator better than the others. That is, the structuring of the tree is used to convey the order in which the operations of the program are to be performed, and not the phrase structure of the source statements which induced this ordering.

At this stage, the operations labeling the nodes of the APT are just the operations of the source language. That is, there is a one-to-one correspondence between the operator tokens in the input and the nodes of this abstract program tree. The operation of sequencing from one PL/1 statement to the one following which is implicit in the concrete syntax, is made explicit in the APT by making the subtree which represents a statement a son of the node which is associated with the statement which logically precedes it in the program. For example, the first DO-loop in Figure 1.3 follows a GET
Figure 1.4 Abstract Program Tree For Program In Figure 1.3

Note: unlabeled leaves indicate options not present in source program

C X A B I
statement. So in the APT in Figure 1.4 the subtree for the DO-loop is the right-most son of the GET node.

In general the nodes of the abstract tree representation will be statement names, operators, built-in and user-defined function names, and subscripted variable names. The sons of a node are trees for

1) qualifying expressions in, or statements associated with, the node statement,
2) operands of the node operator,
3) arguments of the node function, or
4) subscripts of the node array variable reference.

Each leaf corresponds to an unsubscripted identifier -- variable or constant -- in the program. Associated with each leaf is the collection of attributes which have been applied in explicit declarations or by default. These are stored in the symbol table.

1.2 The Code Generation Process

The second half of compilation is the code generation process (see Figure 1.5). Its function is to transform the APT and symbol table internal representations of the source program into an equivalent object machine program. In particular, it transforms the data items of the program into suitable object machine storage locations and transforms the APT into a suitable sequence of object machine instructions. This transformation is accomplished in four distinguishable
Figure 1.5 Internal Representation Flow In The Code Generator
phases: storage allocation, translation, global optimization, and coding.

The storage allocator assigns storage space to each data item referenced by the program. The symbol table is modified to indicate the address of each data item under this allocation. This is the only code generation phase which modifies the symbol table or representational aspect of the source program's internal representation. The remaining phases massage the APT into the proper sequence of object machine instructions.

Translation

All modern computers have in common the following essential characteristics:

1) Execution proceeds from instruction to instruction unless explicitly directed otherwise by a branch instruction.

2) Each instruction operand is a single value (called a scalar in the sequel) as opposed to a function or expression or an aggregation of values.

In contrast, the APT representation of the program produced by the recognition phase is characterized in the following way:

1) The flow of control is either implicit in the branches of the APT or embedded in the definition of a statement (or other type) node.
2) In general, all operands are trees representing unevaluated expressions or functions and not necessarily single values.

3) Leaves of the APT are not necessarily scalars.

Therefore, the first steps in generating code are to

1) order the operations of the program (nodes of the tree) so that only evaluated operands are used in any operation,

2) make explicit the sequencing of expression evaluation by inserting the appropriate branch instructions, and

3) expand operations involving aggregates (arrays and structures) to repeated operations on scalars.

The phase of the code generator which performs these tasks is the **translator**.

The set of operations generated by the translator constitute instructions of an artificial machine which is tailored to the source language being compiled. Together with the data structures used implicitly in the execution of these operations, they define a **source language machine** (SLM). The instruction set of the SLM includes instructions to perform the scalar operations of the source language, the branches and tests introduced for sequencing and the operations added to select single values from aggregates.

An SLM program for the PL/I program in Figure 1.3 is presented in Figure 1.6.
SECTION 1.2

PROG LOCAL A , B, C(10) , I, N , T(9)
OPENL SISIN , GET , NOSKIP
ASSIGN T1 , =F'1'

L1 SCRIPT T2 , C , T1
GET *T2 , DATA
ADD T1 , T1 , =F'1'
GREATER T3 , T1 , =F'10'
B_FALSE T3 , L1
GET N , DATA
CLOSEL SISIN
ASSIGN I , =F'1'
ASSIGN T4 , N
BRANCH L3

L4 ADD I , I , =F'1'
L3 GREATER T5 , I , T4
BR_TRUE T5 , L2
OPENL SISPRINT , PUT , NOSKIP
ASSIGN T6 , =F'5'
ASSIGN T7(0) , =A(T8)
ASSIGN T7(1) , =A(T6)
ENTER FUNC , T7
PUT T8 , LIST
CLOSEL SISPRINT
BRANCH L4

L2 STOP

L6 LOCAL S , I , T(13)
EQUAL A , B
B_FALSE T10 , L5
ASSIGN T11 , =F'0'
ASSIGN T12 , =F'1'

L6 SCRIPT T13 , C , T12
MULT T14 , *T13 , I
ADD T11 , T11 , T14
ADD T12 , T12 , =F'1'
GREATER T15 , T12 , =F'10'
B_FALSE T15 , L6
RETURN T11

L5 ASSIGN S , =F'0'
ASSIGN I , =F'1'
BRANCH L7

L9 ADD I , I , =F'1'
L7 GREATER T21 , I , =F'10'
BR_TRUE T21 , L8
SCRIPT T16 , C , I
MULT T17 , *T16 , I
FIXPLT T18 , B
DIV T19 , A , T18
MULT T20 , T17 , T19
ADD S , S , T20
BRANCH L9

L8 RETURN S

GET SPACE FOR VARIABLES
PREPARE FILE FOR INPUT
INTERNAL LOOP INDEX
COMPUTE ADDRESS OF C(T1)
READ INTO ADDRESS IN T2
INCREMENT LOOP INDEX
T3 = T1 > 10; (PL/I EQUIV.)
IF ~T3 THEN GOTO L1;
DATA DIRECTED INPUT
END OF GET LIST
START DO I=1 TO N;
SAVE UPPER LIMIT IN T4
SKIP INCREMENT FIRST TIME
INCREMENT LOOP INDEX
T5 = I > T4;
IF T5 THEN GOTO L2;
PREPARE FILE FOR OUTPUT
FILL DUMMY ARGUMENT
PASS ADDRESS FOR RESULT
PASS ADDRESS OF ARGUMENT
CALL PROCEDURE 'FUNC'
PRINT T8, 'LIST' FORMAT
END OF PUT LIST
-> TO TOP OF DO
LINK PARN WITH ARG.
GET SPACE FOR VARIABLES
FIXED TO FLOAT CONVERSION
T10 = A = B;
IF ~T10 THEN GOTO L5
INTERNAL ACCUMULATOR
INTERNAL LOOP INDEX
COMPUTE ADDRESS OF C(T12)
T11 = T11 + A + C(T12)
INCREMENT LOOP INDEX
IF LOOP INDEX ≤ 10
THEN GO TO TOP OF LOOP
BACK TO TOP OF DO

START DO I=1 TO 10;
SKIP INCREMENT FIRST TIME
BY 1
TO 10
-> DO SPEC SATISFIED
S=S+(C(I)*X/(A/B))
FIXED TO FLOAT CONVERSION
BACK TO TOP OF DO

Figure 1.6 A Typical Source Language Machine Program
Global Optimization And Coding

The global optimization phase is optional. If present, it reorganizes the SLM instructions so that the algorithm is computed more efficiently.

The final transformation from SLM instructions to object machine code is performed by the coder, the fourth and last phase of code generation. This phase requires the existence of a well defined implementation -- a mapping between the SLM and the object machine. The mapping is defined on the data types of the source language, the operations of the SLM and the address and memory structures which define the architecture of the SLM.

For each data type in the source language, there corresponds a machine data structure which has the properties required by the source data type. Similarly for each combination of SLM operation and operand type there corresponds a sequence of object machine code which manipulates the implemented operands to produce the result defined for the operator of the source language. These correspondences constitute the implementation.

1.3 The Intermediate Text

The preceding sections considered only the logical phases of the compilation process. Eventually all of the program being compiled must pass through these phases and for some
basic unit of the source program -- a statement, say -- these phases must be applied in sequence. So far, no restrictions have been placed on the number of phases which may be applied to a statement before beginning to compile the next statement. At one extreme a statement could be processed by all phases before proceeding with the next one. At the other extreme all statements would have to be processed by a phase before it would be possible to go on to the next phase.

The term pass is usually used to mean the collection of logical phases which are carried out concurrently in any given compiler implementation. Therefore, in a one-pass compiler all phases are performed concurrently, while in a two-pass compiler the first pass might perform the tasks of the recognition process (three phases) and the second pass could perform the remaining three or four phases of code generation.

It is the partitioning of phases into passes which gives the greatest variation in compiler organization. Table 1.1 indicates the partitioning used in eleven different compilers -- PL/C [Conway 1970], PL/I (F) [IBM 1968b], FORTRAN IV (G) [IBM 1968c], FORTRAN IV (H) [IBM 1966], WATFOR [Cress 1966], MAD [Galler 1961], MAD/I [Sroda 1968], ALGOL W [Bauer 1968], BCPL/360 [Kelly 1970], and the models used by Gries [1971] and McKeeman [1970]. In the table each pass is indicated by a solid box. If a phase is not included in a pass, but the neighboring phases are, then the box is divided into sections by broken lines. For example, the last pass of the PL/C compiler includes storage allocation, translation, and coding,
<table>
<thead>
<tr>
<th>PHASES</th>
<th>PL/C</th>
<th>PL/I-(F)</th>
<th>FORTRAN G</th>
<th>FORTRAN H</th>
<th>WATFOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEXICAL ANALYSIS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SYNTAX ANALYSIS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEMANTIC ANALYSIS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STORAGE ALLOCATION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+ -  4</td>
</tr>
<tr>
<td>TRANSLATION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+ -  1</td>
</tr>
<tr>
<td>GLOBAL OPTIMIZATION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-  1</td>
</tr>
<tr>
<td>STORAGE ALLOCATION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CODING</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PHASES</th>
<th>MAD</th>
<th>MAD/I</th>
<th>ALGOL W</th>
<th>BCPL</th>
<th>Gries</th>
<th>McKeeman</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEXICAL ANALYSIS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SYNTAX ANALYSIS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEMANTIC ANALYSIS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STORAGE ALLOCATION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRANSLATION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GLOBAL OPTIMIZATION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STORAGE ALLOCATION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CODING</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1.1 The Partitioning of Phases Into Passes
but contains neither global optimization nor the storage allocation which follows it.

After being processed by a phase, the source program is no longer represented by the string of characters submitted to the compiler nor is it in the same form as read by that phase, yet it still encodes the same information as the original under the proper interpretation. These altered representations of the program are called internal representations. A different data structure is used in the internal representation for each aspect of the program. The representational aspect is encoded in a randomly accessible symbol table. The functional aspect is encoded in a sequentially accessible internal text.

Each phase concentrates on one feature of the program structure and transforms instances of that feature to a form which is more easily managed by the phases which follow it. A phase usually handles only one aspect of the program, modifying either the symbol table or internal text. Consequently each phase represents a transformation from one internal representation to another.

Whenever more than one pass is required (or used) the internal representation of the program's algorithm which has been generated by the most advanced phase of the pass must be physically constructed in its entirety within the data space of the compiler. This representation of the program's algorithm is called an intermediate text (IT). Of course, if there are more than two passes there will be more than one IT
-- one between each pass.

Some internal representation of the program’s algorithm will always be passed between phases. This has been called an internal text. Only when all of the symbols of an internal text of a program must be saved in storage before being processed by a following phase will the internal text be called an intermediate text. An internal text exists between each phase. An intermediate text exists only between passes.

The form of the IT depends on many factors. Principally it depends on the internal text which is being retained between passes. The tokenized program text, the APT, and SLM code all have different influences on the form of the IT. In general, the IT will be a linear encoding of the internal text which is easily read and manipulated using the instructions of the computer on which the compiler is implemented.

1.4 Summary Of Content

The methods used in the recognition phase to construct the APT have been studied extensively and are well documented in the literature. (See Gries [1971] for references.) In contrast, the nature of the code generation process is poorly understood, inadequately documented and, except for studies of very isolated sub-tasks, has not been treated theoretically. It is the aim of this thesis to clarify the organization of code generation by presenting a model code generator with a complete discussion of the reasons for including each
component of the model. The discussion concentrates on the translation and coding phases -- the phases which have received the least attention in the literature.

Translation Model

The way the APT is to be translated is dictated by a semantic mapping which takes each node operator to a semantic template consisting of

1) rules for the evaluation of each operand subtree, intermixed with

2) SLM instructions which manipulate the evaluated operands.

Chapter two presents, for the first time in the literature, a discussion of the "best" physical representation for the APT in the IT passed to the translator. The criteria used to measure the goodness of an IT representation are

1) the amount of space required to represent the operator and its operands, and

2) the complexity of the translator which is required to transform the IT representation to the proper sequence of SLM instructions.

The translator is characterized as a formal pushdown transducer (PDT). The number of states in the finite control of the PDT, its input/output tape mechanisms, and the nature of the information saved in its stack are used to measure the complexity of the translation.
Out of this discussion comes a classification of operators and their translators into one of seven categories. The classification is complete and is determined solely by the semantic map defined on the operator. Operations of different classifications can be represented in the same IT with only a few symbols added to punctuate the structure. A translator for this IT format is presented.

Coding Model

The coder works on the SLM representation of the program -- possibly modified by the global optimizer -- to produce the object machine code. The way the SLM representation is to be coded is dictated by an implementation map which takes each SLM instruction to a sequence of object machine code. It is assumed that the implementation has been determined before applying the results of this research. The automatic generation of the implementation is not within the scope of the present work, although it is an interesting topic for future research. Only the realization of the implementation in the coder will be considered here.

In chapter three a coder is described which will realize any implementation map which satisfies the following constraint: if the SLM representation of the APT, S(APT), has the form

\[
\begin{array}{c}
0(a, \ldots, a) \ldots 0(a, \ldots, a) \\
1 \quad 1 \\
1n \quad m \quad m1 \quad mn \\
1 \quad m
\end{array}
\]
where $O$ is an SLM operator and the $a_i$'s are its operands, $ij$
then the implementation, $I(E,S(\text{APT}))$, must be definable in
the form

$$I'(E,0,a_{11},...,a_{nn}) = I'(E,0,a_{1n},...,a_{nn})$$

where $I'$ is a restriction of the implementation map and $E$ is
a compile-time description of the run-time environment after
executing operator $O$ in the environment described by $E_i$. The presentation in chapter three concentrates on the
structure of the environment description and the encoding of
the restricted implementation map. The techniques discussed
are similar to those used in many different compilers, but are
presented in an expanded and more organized fashion.

The environment is described by a set of descriptors. Each value manipulated by the SLM representation is defined to
the coder in an information structure called a value
descriptor. Each hardware component of the object machine's
CPU is represented by a component descriptor.

Unlike the descriptors used in most existing compilers
and translator writing systems like FSL [Feldman 1966] and CIL
[Gries 1969], the descriptors of this model are uniformly
associated with values rather than differentiating between
those for identifiers and those for intermediate results. A
new descriptor is constructed to describe a value when the
first SLM instruction referencing that value is coded. This
descriptor is managed by the coder until the value it
describes is either destroyed or no longer referenced in the
SLM code. The descriptor is destroyed at this time.

The value descriptor contains 1) the attributes of the value in terms of the object machine, 2) its access function, and 3) the value itself if known at compile time. The value and attribute information remain constant for the life of the descriptor, but the access function is constantly changed so that it always describes the most easily accessible copy of the value. This is a unique feature of these descriptors and is available only because the descriptors are not associated permanently with any symbol table entry or other information structure used by the rest of the compiler.

The encoding of the access function is also new. Two descriptor formats are possible. The register format indicates the value is in a register and designates the component descriptor for the register. In the storage format the DISP, BASE and INDEX fields of the descriptor define the location of the value as $C(C(BASE)+C(INDEX)+DISP)$ where $C(*)$ means "the contents of." DISP is a constant, but BASE and INDEX are pointers to other descriptors in either the register or storage format. This almost universal representation can describe indirect addressing to arbitrary depth with indexing (constant or variable) available at each level. So this simple representation is capable of describing any location addressable in computers of modern architecture. Its many other virtues are discussed in chapter three.

The algorithm for coding $I'(E_{i-1}^i, a_{i-1}^i)$ for all $i=0, 1, \ldots, m$.
SECTION 1.5

E and all combinations of a 's is represented in the coder by a code template. The template is interpreted by the nucleus of the coder at compile time. The template is created using a coding language (ICL) specially designed for the object machine. The statements of the ICL isolate the functions necessary in coding: 1) instruction generation and address computation, 2) register management, 3) storage area management, 4) descriptor management, and 5) primitives for conditional coding. The sample ICL in Appendix I compactly presents the operations which are typically performed within these functional areas.

1.5 Contributions

Unlike the few other attempts to abstract the code generation process, the research reported here originated from experience with a specific, "production" compiler -- PL/C -- and not from an interest in translator writing systems. The code generator in PL/C could have ended up just another ad hoc mess had it not been for the fact that careful analysis of the process of generating code for PL/I was necessary in order to meet the performance and size constraints placed on the compiler by its implementers. This analysis led to the model presented in this thesis. For historical reasons, it was not possible to mold the PL/C code generator into exactly this model as it evolved over a period of years, but the basic principles are indeed present. Therefore, the model is
workable.

Aside from the obvious contribution of a well organized presentation of a very complex process, the following should be mentioned:

1) By separating translation from the recognition phases and putting it where it belongs in code generation, this research presents a fresh approach to translation and, as a result, is able to succinctly classify all operators according to their internal representation and the translator structure needed to process them.

2) The concept of a compile-time descriptor is not new, but here they have been detached from the symbol table and identifiers, which gives them expanded flexibility and generality, particularly in the representation of the access function.

3) The use of coding templates is also present in many compilers, but this work is the first complete presentation of them in an expanded form in which all functions of coding are specifiable.

4) Finally, this research has shown that the templates and descriptors can be combined to form a very capable coder which is economical enough to use in any compiler for PL/I-like languages -- not just in translator writing systems or compilers for extensible languages.
CHAPTER 2. TRANSLATION

The proponents of translator writing systems and syntax-directed compilers (most recently McKeeman [1970]) advocate either one-pass compilers or a two- or three-pass scheme in which the translation phase is included with the recognition phases in the first pass, leaving global optimization and coding for the final passes. (See Figures 1.1b and 1.1c.) However, a glance at Table 1.1 reveals that very few successful compilers have this structure. In addition, it should be noted, the three compilers that do have this structure (WATFOR [Cress 1966], MAD [Galler 1961], and MAD/I [Srodawa 1968]) are for very simple programming languages -- e.g. no block structuring, no aggregate operations, etc.

The model presented here assumes that the translation phase is in a pass separate from the passes containing the recognition phases as in Figure 1.1a. This is the strategy of many production compilers, most notably PL/I (F) [IBM 1968b], FORTRAN IV (G) [IBM 1968c], FORTRAN IV (H) [IBM 1966], and ALGOL W [Bauer 1968].

Most of the work on syntax-directed compiling was done in the late 1960's when ALGOL was the most complex programming language in use. Because all names in ALGOL (except labels) must be declared before they are used, it is a simple matter to add routines to the syntax analyser to do the semantic analysis. In this scheme a semantic routine is called each time a construct of the language is recognized in the source
program. The routine verifies the construct is semantically correct. In addition, the semantic routines are responsible for constructing an intermediate text.

Because of the declare-before-use rule all of the information about the program's identifiers is available during syntax analysis so it is only natural to try to complete as much of the compilation process as possible at this time. In most cases this means producing the object code directly (a one-pass scheme). In others, the source language machine instructions are produced so that optimization can be performed before coding begins (three passes). In either case translation is concurrent with recognition.

Even without the strict declare-before-use rule of ALGOL -- labels rarely need to be declared -- concurrent translation is feasible if the output text can be formatted so that when the missing semantic information -- e.g. the location of a label -- becomes available, it can be backed up and patched to complete the translation.

Concurrent translation is not feasible when compiling PL/I [IBM 1969]. PL/I does not require declaration before use and because very complex data structures can be created by such a declaration it is not possible to do even semantic analysis concurrently with syntax analysis even with very elaborate back-up-and-patch schemes, let alone attempting concurrent translation. Consequently, when compiling a PL/I-like source language a representation of the APT must be written into an IT to be processed by the translator in a
separate pass. In this chapter the structure of such an IT, and the algorithms which transform it into a proper SLM instruction sequence will be investigated.

2.1 The Translator Environment

In this model the translation phase of compilation is responsible for converting the abstract program tree (APT) into a sequence of source language machine (SLM) instructions which have as operands scalar values and which preserve the meaning of the program. The source program is presented to the translator in two distinct data structures -- the symbol table and an intermediate text representation of the APT.

The symbol table contains complete information about each variable defined by the source program. Nodes and leaves of the APT which represented occurrences of these variables in the program are replaced in the intermediate text by reference pointers to the symbol table. The information the symbol table must retain for the translator includes the attributes of the variables and a description of their run-time location. The use of this information is discussed in section 2.6.

The intermediate text representation of the APT -- referred to in the sequel as simply the intermediate text (IT) -- is a physical representation of the APT. The structuring of the APT may be conveyed implicitly through the ordering of the symbols in the text -- as in postfix notation -- or made explicit by the addition of pointers and similar punctuation.
The model allows both forms to be used in the IT of a single language, the format being determined by the unique symbols which make up the IT.

Simplifying Assumptions

Except for the additional pointers and symbols needed to punctuate the structure of the tree, each syllable of the IT corresponds to a unique node or leaf of the APT, which in turn corresponds to a token of the original program. So that the following discussions may concentrate on the total process of translation rather than starting at some half-way point, it is assumed that source operations -- corresponding to operator tokens, statements or keywords -- have neither been synthesized into more complex operations, nor have they been decomposed into simpler ones. Although this restriction is imposed here solely to facilitate the following exposition, it is not unreasonable to expect such a situation to actually exist in practice. The recognizer has enough of its own problems to contend with in analysing a programming language as complex as PL/I. There is little advantage in burdening it with tasks the translator can handle just as easily and more properly.

The following discussion of the model assumes that any errors which existed in the source have been eliminated -- either corrected or repaired -- so that the program represented in the IT is both syntactically and semantically
correct. This assumption will simplify the model of the translator and certainly is not an unreasonable request to put on the recognizer phases.

Finally, it is assumed that scanning and parsing the IT is a trivial operation. Each symbol in the IT is locally and completely defined. The notation for identifiers -- pointers to unique, scoped elements in the symbol table -- is readily distinguishable from that used for keywords, operators, punctuation and the like.

2.2 Translator States

In automata theoretic terms, the model used here for the translator is a deterministic push-down transducer (PDT) [Ginsburg 1966, pp. 103]. This device and the model translator are diagrammed in Figure 2.1. The translator's input tape is the intermediate text. Its output is the stream of SLM instructions. Aside from local finite storage, the only work space used is a push-down stack.

For the translator model to fit into the PDT formalism, the information in the symbol table must be considered part of the input tape. That is, the symbols used for identifiers in the PDT must encode all of the information which, in any implementation, would normally be found in the symbol table. Since an identifier's symbol table entry remains constant during translation, the symbol table pointers used in the model to represent identifiers can serve as this encoding
**SECTION 2.2**

---

**INPUT TAPE**

**FINITE STATE CONTROL**

**OUTPUT TAPE**

\[ \text{PDT} = \{S, Z, I, O, N, Q, s^0, z^0\} \]

- **l**: Input symbols
- **S**: Machine states
- **Z**: Stack symbols
- **R**: Read head adjustments
- **W**: Write head adjustments
- **N**: Next-state function: \[ Z \times (I \cup \text{nil}) \times S \rightarrow Z^* \times S \times R \]
- **Q**: Output function: \[ Z \times (I \cup \text{nil}) \times S \rightarrow O^* \times W \]

- **(a)** Organization of A Formal Pushdown Transducer

---

**INTERMEDIATE TEXT (APT)**

---

**SYMBOL TABLE**

**TRANSLATOR**

**WORKAREA** (FINITE STORAGE)

- **SLM**

- **INSTRUCTIONS TO NEXT PHASE**

- **STACK**

---

- **(b)** Basic Organization Of Model Translator

**Figure 2.1 Diagrams For Formal PDT And Model Translator**
without violating the PDT model. Consequently, the randomly accessible symbol table used to implement the translator model can be ignored when formalizing the translator's input in terms of the input tape of the PDT.

Although this seems to be a reasonably specific characterization, it is not detailed enough to serve as a model from which to implement an actual translator. Before that is possible several questions must be answered regarding the nature of the input tape, the finite state control, symbols on the stack and the output mechanism. Whether the input tape must be two-way or one-way is an important consideration in the management of the intermediate text. Similarly, whether the output tape may be write-only or of necessity be read-write and two-way is crucial to the design of the entire generation phase. For example, a write-only tape is required for a one-pass generation scheme. These questions and others will be discussed in the following sections and in chapter three.

The notion of translator state is similar to the concept of a state in the finite-state control of the push-down transducer. Ultimately the state the translator is in depends on three conditions -- the state the translator was in after processing the previous input symbol, the current input symbol, and the symbol(s) on the top of the stack. In turn, the translator state and the next input symbol determine the output of the translator, the symbols pushed onto or popped from the top of the stack, the movements of the read/write
heads and the translator's reaction to the next input symbol.

Without further qualification, equating translator states with the states of the PDT's finite-state control would introduce too many uninteresting translator states. Specifically, in the formal definition of the PDT the output and next-state functions depend only on the top symbol of the stack. To overcome this restriction, artificial states may be introduced into the finite-state control of the PDT which are entered on null input symbols in order to "program" an involved manipulation of the stack or complex output function. States of this type are not included in the notion of translator state as used here. Rephrased slightly, this means that the PDT used in this thesis to model translation allows the next-state and output functions to depend on an arbitrary number of symbols on the stack. Furthermore, the translator state cannot change without reading a new symbol from the input tape.

Using this analogy with the PDT as a guide, a hierarchy of ever more complex translators can reasonably be constructed based on the number of translator states and the degree to which the next-state and output functions of the translator depend on their arguments. Minor levels of the hierarchy can be differentiated by the complexity of the stack symbols -- e.g. are they used just to remember input symbols, previous translator states, or possibly more complex information structures?
State Components

For a programming language like PL/I, the translator state can be decomposed into many independent components. It is convenient to classify these according to their duration or frequency of change. **Global state components** remain constant throughout the generation process. Typically these components are specified by compiler options or one of the recognition phases. In the PL/I (F) and FORTRAN IV (H) compilers, for example, the OPT-n parameter influences global state components having to do with the efficiency of the object code generated. And in the PL/C compiler the M91 (compiler is running on a model 91) and UDEF (test variables for initialization before use) options form two global components of its translator state.

**Local state components** change value within the process of translating expressions. They may remain in one state for the translation of an entire expression or only during the translation of one operand of one operator (or sub-expression). Examples of such state components will be presented in the next section.

Between these two extremes is a third class of state components which are in effect for entire statements, blocks, or procedures. In PL/I these components have to do with exceptional condition handling and are controlled by condition prefixes on statements.
2.3 **Semantics And Structure**

In this section the PDT model for the translation from the APT to an SLM program is refined through the introduction of explicit algorithms for the finite state control and detailed analyses of the information maintained in the stack. The discussion will be in terms of operators and their operands and will view the entire tree as an expression. As discussed in chapter one, these terms should be interpreted liberally to include as operators built-in and user-defined procedures and, in general, all statements.

This investigation of the micro-structure of the translator cannot be separated from an analysis of the structure of the IT representation of the APT. The complexity of the translation algorithm depends on the relationship between the format of the IT input and the sequence of SLM instructions which is to be associated with that input. The closer the format of the information in the IT is to its desired output form, the simpler the task of the translator. If the input is a postfix representation of the APT, for example, and the SLM is a stack machine, then the translator has practically nothing to do. On the other hand, if the IT is an infix representation and the SLM is as defined in chapter one, then the translator must do considerably more in the way of reordering the input information, including some of the work already done by the recognizer.
The output of the translator is fixed by the semantics of the operators being translated. Here "semantics" can be defined exactly as the mapping from the APT operator into a semantic template. A semantic template indicates

1) when each operand is to be evaluated relative to the evaluation of the other operands and to the execution of SLM instructions involving these evaluated operands,

2) how each operand is to be evaluated, and

3) what SLM operations should be performed on the evaluated operands.

---

**Step 1)** Evaluate left operand for its arithmetic value and save it in location A.

**Step 2)** Evaluate right operand for its arithmetic value and save it in location B.

**Step 3)** Perform an ADD SLM operation on the values in locations A and B. This is the result.

---

**a) Semantic Template for Addition**

**Step 1)** Translate left operand in "arithmetic-value" state. Put description of value in A.

**Step 2)** Translate right operand in "arithmetic-value" state. Put description of value in B.

**Step 3)** Generate an ADD SLM instruction which references values as described in A and B.

---

**b) Interpretation of Semantic Template for Translating Addition**

---

**Figure 2.2 Semantic Template Interpretation**

---

For example, the semantic template for addition might be as in Figure 2.2a. A semantic template for the PL/C IF-statement is shown in Figure 2.9.
By implication, the SLM code generated from this semantic template, when executed, will evaluate the operator's operands and manipulate them to produce the result defined by the language for the particular operator-operand combination. The semantic mapping is assumed to be given. Its validity is not an issue.

Because of the duality which exists between evaluation and translation, the semantic template can be interpreted as a set of rules for translating the operator. Under such an interpretation the semantic template presents for each operator

1) the order in which all of its operands are to be translated relative to each other and to the generation of SLM instructions,

2) the translator state in which each operand is to be translated, and

3) the SLM instructions which are to be generated.

The semantic template for addition in Figure 2.2a may therefore be interpreted as shown in Figure 2.2b.

The IT will be constructed by the recognizer and so, in the final analysis, the IT structure will be limited by the capabilities of those phases. For now, however, the only restriction that will be placed on the IT is that it be a linear representation of the APT. This will permit the free selection of an IT format which is best suited to the operators and their translation.
Postfix-translatable Operators

Perhaps the simplest linear representation of the APT is postfix notation. Moreover postfix has several properties which make it an ideal intermediate text for a large class of languages.

1) Since only the names of leaves and nodes appear, it is one of the most compact representations available.

2) The leaves of the tree appear in the same order as they did in the tree -- and consequently in the source program. Therefore, assuming a left-to-right scan of this IT, the operands are translated and therefore evaluated from left to right -- a semantic rule often imposed by programming languages.

3) The operands of an operator -- i.e. their postfix form -- immediately precede the operator symbol. Thus, with a simple push-down stack the run-time location of operands can easily and efficiently be recorded during translation.

4) The operators (node names) appear in the order in which they are to be evaluated. So when the translator scans an operator, it can immediately go about the business of generating the necessary SLN instructions. Because of property 3), this simply entails using the operator and the types of its operands as an argument to a table look-up which will yield in tabular form the sequence necessary for the combination. The entire process of translating from postfix is summarized by the flowchart in Figure 2.3. Those operators
which can be translated using this algorithm will be called postfix-translatable.

The postfix translator is a very simple machine. In terms of the PDT model, it can have only one state. The output function depends only on the input symbol and the symbols on the top of the stack. The stack symbols are relatively uncomplicated in that they need only remember the name of an input symbol or serve as an indicator for an
intermediate result.

For an operator to be postfix-translatable three conditions must be satisfied.

1) The operator may not be variadic. That is all valid occurrences of the operator must require the same number of operands.

2) The semantic template which defines a postfix-translatable operator can use neither prologue code -- an initialization sequence of SLM instructions which is to precede the evaluation of all operands -- nor intervening code -- SLM instruction sequences which are to intercede between the evaluation of operands.

3) All operands of the postfix-translatable operator must be translated in the same translator state as the one which surrounds the translation of the operator-operand combination.

Addition as defined in Figure 2.2 satisfies all of these conditions and is, therefore, postfix-translatable. The PL/C IF statement defined in Figure 2.9 has intervening code sequences and is, therefore, not postfix-translatable.

Without condition 1), the postfix expression would be ambiguous. It would be impossible to determine, at the time the operator is encountered in the left-to-right scan, how many of the preceding operands belonged to it and how many were to be used in some later operations. For example, the postfix expression A B C MIN D MAX involving the variadic PL/I operators MIN and MAX could be the representation of either
MAX(A, MIN(B, C), D) or MAX(MIN(A, B, C), D). Many operators in PL/I fail to satisfy this condition. In addition to the built-in functions MAX and MIN, many of the string built-in functions -- SUBSTR, CHAR, BIT, etc. -- have optional parameters and are therefore variadic.

Conditions 2) and 3) are required because as the translator reads through the postfix expression it does not know to which operator the operand being translated is an argument. In fact it does not even know when it switches from the translation of one operand to that of another.

Here again some operators in PL/I fail to satisfy these conditions. The array generic functions SUM, PROD, ALL, and ANY require that an accumulator be initialized before their operand is evaluated. That is, prologue code is required. If a postfix representation were used, the operand of the array generic function would be translated before the operator was discovered. So there would be no chance to insert the prologue code before the code which evaluates the operand.

As a second example, the operand to the PL/I built-in function LENGTH should really be evaluated for the length of the result string rather than constructing the result and then using only its length. That is, the operand of LENGTH should be translated in a different state (length) from the normal state (value). Again using the algorithm in Figure 2.3 to translate the postfix expression, the string expression which is the operand of the LENGTH built-in function would be translated before the translator could recognize the fact that
it should be translated in the length state.

Prefix-translatable Operators

To be able to translate the operators which fail to meet the second and third conditions of postfix-translatability, the translator must discover to which operator the operand being translated belongs. To do this and at the same time to avoid a complicated scan of the IT it is necessary to move the operator from its position following its operands to a position which immediately precedes them -- i.e. prefix notation.

An operator which can be translated in this IT format will be called **prefix-translatable**. The flowchart or an algorithm for translating the prefixed text is given in Figure 2.4. Each prefix-translatable operator has associated with it a routine which directs the translation of that operator. This routine will be called a **driver**. The operations performed by a typical driver are flowcharted in Figure 2.4b. Note that an actual driver would not be structured exactly as in Figure 2.4b. The loop which directs the translation of each operand would probably be expanded into straight line code. Figure 2.10 illustrates a typical driver for a prefix-translatable operator. In this case it is the IF-statement operator in PL/I.

The complexity of the prefix driver, and consequently the entire translator, depends on which of the two conditions of
a) Prefix Translation Algorithm

b) Flowchart Of Operations Performed By A Driver
postfix-translatability are violated by the prefix-translatable operator. If the only violation is that the semantic template has prologue or intervening code, then the translator still has only one state. (In that case boxes A and B in Figure 2.4a are not required.) In the other case, the translator has one additional state for each new way of translating an operator's operand. In either case, to make each driver recursive, the stack must be used to save the location in the driver to which the translator is to return control after translating an operand. This is in addition to the use made of the stack in the postfix translator.

With the addition of more translator states, the contents of the stack become more complex. In particular, the stack must be used to save the old state component values which are changed to accommodate the incoming operator. Theoretically this really presents no extra complexity, but in any implementation, the state of the translator will usually be scattered through many bit flags and counters. To save these may be costly in both time and space.

Variadic Operators

Variadic operators can be translated in either prefix or postfix format with the addition of a single symbol to mark the end (or beginning) of the variable length operand list. For example, \texttt{MAX(A,MIN(B,C),D)} would be written as \texttt{(A (B C MIN D MAX} in variadic postfix notation and as
Figure 2.5 Variadic Postfix Translation Algorithm
MAX A MIN B C) D) in variadic prefix notation. (Here and in later examples the symbol "(" is used as the left list delimiter and the symbol ")" is used as the right list delimiter.)

![Flowchart](image_url)

Figure 2.6 Operations Performed By Variadic Driver

In either format, the translator to handle variadic operators is no more complex than the translators defined earlier. Figure 2.5 displays the algorithm for translating variadic postfix operators. A flowchart of the operations performed by the driver for a variadic prefix-translatable operator is drawn in Figure 2.6. The rest of the variadic prefix translator is the same as Figure 2.4a.
Where there is a choice between using postfix or prefix for a variadic operator, the prefix format would be preferrable because of its ability to produce better code. For example, prefix is especially advantageous when the variadic operator simply extends, as do MAX and MIN, a binary operation over its operands. As each operand after the first is translated the binary operation can be generated in the intervening code. Since no further use of the operands is needed, their descriptors may be discarded and the resources they hold released.

Hybrid Text

Clearly postfix-translatable operators can be translated from the prefix format. Since the majority of operators are postfix-translatable and the prefix format requires a more complex translator, it would be advantageous if the two formats could be combined without increasing significantly the complexity of either the IT or the translator. This can be done by combining the two forms in the natural way. Two symbols are needed. One, a comma, say, must be added to separate the operands of prefix operators and the other, a right bracket say, to mark the end of the operand list. These symbols must be distinct only if variadic operators are involved. These symbols are portion of the translator when a prefix is translated and it is time for the driver to regain control.
As long as the prefix-translatable operators are distinct from the postfix-translatable operators there can be no confusion in using the postfix format for most of the operators and changing to prefix only when that format is required by the operator. This combination of prefix and postfix notation will be called hybrid text. Thus the PL/I infix expression

\[ A = \text{SUM}(B+C) \times \text{MIN}(D,E-F,G); \]
ecomes

\[ A \text{ SUM } B \text{ C } \times \text{ MIN } D, \text{ E } F - , \text{ G} \times = \]
in hybrid text format assuming the operators \( +, -, \times, = \) are postfix-translatable, and \( \text{SUM} \) and \( \text{MIN} \) are prefix-translatable. Note that a "," could be used unambiguously instead of the ")" after \( \text{SUM} \) but the ")" following \( \text{MIN} \) must be different from the "," used as the argument delimiter.

The translator for this IT is flowcharted in Figure 2.7. This translator combines the postfix and prefix translators defined in Figures 2.5 and 2.6 respectively. Note, however, that some of the operations of the postfix translator have become dependent on the translator state. As before, the translator state can be changed only by a prefix driver. Either form of prefix driver (Figure 2.4a or Figure 2.6) may be used with this translator.
Figure 2.7 Hybrid Text Translation Algorithm
Tree-translatable Operators

In each of the translation schemes presented so far the order in which symbols are translated depends on the ordering of these symbols in the IT. Fortunately the IT presents the operations in an order which is acceptable and sometimes desirable for execution. That means that in producing a linear form of the APT, the pass preceding translation has performed the bulk of one of the major tasks of translation -- that of ordering operations for sequential execution.

That this is not an unreasonable request to make of the recognizer must be emphasized. The hybrid text is, in fact, an almost natural presentation of the original source. The operands appear in the same order as in the source and, considering that most prefix-translatable operators -- all in PL/I -- appear in the concrete syntax as function application, only a few operator symbols (primarily infixed postfix-translatable) are reordered.

The recognizer should not be expected to do more than this, however. In particular it should not be asked to reorder operands for the sole convenience of the translator. The recognizer has enough work to do without catering to the translator's every whim. So, if the operands of an operator must be translated in an order other than that used in the language's syntax, the translator must be able to skip around within the intermediate text.
Using the hybrid format of the IT, this could be accomplished in the prefix driver by for commas which are not protected by other prefix symbols. The efficiency of this scan is improved if the prefix operators are followed by a (redundant) symbol to mark the start of their operand list -- a left bracket say. This avoids having to look up the operator to see if it is prefix- or postfix-translatable. For example,

\[ A = \text{SUM}(B+C) * \text{MIN}(D,E-F,G) \]

becomes

\[ A \text{ SUM}(B+C) \text{ MIN}(D,E-F,G) * =. \]

This scheme will work only if the operands always appear in the same order. If they do not, the recognition phase will have to insert pointers after the operator symbol to indicate where each operand starts in the text which follows the operator. A comma or right bracket can be used to mark the end of the operand expression. Including these pointers is, of course, equivalent to a complete explicit specification of the tree. Consequently operators requiring this treatment will be called tree-translatable.

The DO-statement operator of PL/I is a good example of a tree-translatable operator. In the IT the statement

\[ \text{DO } I(1) = \text{SUM}(B+C) \text{ BY } 1 \text{ TO } \text{MIN}(D,E-F,G); \]

becomes

```
DO \[\text{I (1), SUM (B+C), 1, MIN (D, E-F, G)}\]
```
KEY TO SYMBOLS USED IN FIGURE 2.8

A(*): Attribute of 
AA: Array application operator
C(*): Class of 
D(*): Driver selection function
d: Result descriptor
E(*): Epilogue SLM sequence selection function
FA: Function application operator
I(*): Intervening SLM sequence selection function
i: IT read head position
IS: Current input symbol
IT: i-th symbol in IT

j: Index of top symbol in stack
N(*): Next translator state function
n: Number of operands processed, driver state
O(*): Identifier descriptor construction function
O: Operator symbol
P(*): Prologue SLM sequence selection function
p: Pointer to first operand pointer in IT
Q: Output function
q: State surrounding translation of operator
R(*): Result descriptor construction function
S: State of translator
s0: Initial translator state
T(*): SLM sequence selection function (postfix)
t: Selected template
Z(j): j-th symbol in stack vector

#(*): Number of operands to *
Figure 2.8a A Complete Translator
Figure 2.8b Operations Performed by Tree Driver
Note that the delimiters of the source language —i.e., =, to, by, and ; -- could be used instead of the ",," and ")" in such a text representation.

As with the prefix operators, a driver is associated with each tree-translatable operator. The driver uses the pointers in the IT to guide the rest of the translator through the translation of the operator's operands. The operations performed by such a driver are flowcharted in Figure 2.8b. A driver for the simplified DO-statement operator used in the above example is presented in Figure 2.13. This is typical of the drivers for tree-translatable operators. The rest of the translator is defined in Figure 2.8a. This translator is basically the same as the hybrid translator. It has been modified slightly to accept changes in the order of scanning as directed by the tree driver. (Additional modifications to Figure 2.7 have been made to create Figure 2.8a. The significance of these changes will be discussed shortly.)

The need to translate operands out of IT order does not increase the complexity of the translation algorithm to any great extent, but it does necessitate a more complicated mechanism for managing the intermediate text. In terms of the PBT model, the one-way tape is no longer adequate; a two-way read-only tape is required.
N-pass-translatable Operators

A two-way tape is also required if the operands of an operator must be translated more than once. The two scanings of the text will, of course, be made in different translator states. There would be no use for the second scanning otherwise. Operators of this type will be called *n-pass-translatable*.

N-pass-translatable operators arise when some attribute of the operand must be determined at execution time, prior to computing its value. In the PL/C implementation of PL/I, for example, if the argument to a user-defined function is a string array expression, temporary space for the dummy argument must be allocated on a run-time stack. Then the array expression is evaluated element by element and the resulting values are stored into the temporary area. The size of the temporary area, however, depends on the lengths of the resulting strings. Since, in PL/I, string lengths may be computed at run time, it is not possible for the compiler to know in every case how long the result will be. Consequently, object code must be generated which will compute the length of the result prior to allocating the space for the dummy argument.

To do this the translator first scans the argument (operand) in a state in which 1) subscripting is ignored, 2) arithmetic operators are ignored, 3) the relational operators generate a length result of one and 4) the string operators
generate SLM code sequences to compute the length of their result as a function of the length of their string operands. After this is completed, intervening code is generated which will allocate the temporary array.

Before scanning the operand for the second time, code is generated which sets up and initializes compiler-generated variables which will serve as the omitted subscripts in the array expression. The second translation of the operand text generates code to evaluate the string expression for one element of the array. The element which is selected is determined at run time by the value of the compiler-generated subscripts. After the second translation additional code is generated -- in another intervening code sequence -- which stores the result into the temporary, increments the compiler-generated subscripts, and then branches back to re-evaluate the expression with the new subscript values.

\textit{N}-pass-translatable operators are translated exactly the same way as tree-translatable operators. The flowchart in Figure 2.8a defines the translator for such an operator. An \textit{n}-pass driver is not flowcharted since it is only a minor variation of the tree driver in Figure 2.8b.

\textbf{User-defined Operators}

The class of operators introduced by the program in the form of user-defined procedures deserves some discussion. Because arguments are traditionally associated with parameters
by matching the order of presentation with the order of definition, in any reasonable implementation user-defined procedures can be processed as either prefix- or postfix-translatable operators.

Within the tables of the translator all user-defined procedures are represented by one operator — function application (FA in Figure 2.8a). When the identifier for a user-defined procedure is scanned, it is converted by the translator into this operator. The information about the user-defined procedure is an implied first operand of the function application operator. (See Figure 2.8a.)

To be postfix-translatable user-defined procedures must be implemented with call-by-value arguments and with no automatic conversion to match parameter types. This is the situation in BCPL/360 [Kelly 1970]. Other cases must be handled as prefix-translatable operators. The prefix notation allows the translator to find the description of each parameter before the corresponding argument is translated. From this description the proper state for the translation and whether any conversion or other intervening code is required can be determined.

This completes the discussion of translator structure and intermediate text representations of the APT. The hierarchy of operators developed here and the corresponding translators and intermediate texts are summarized in Table 2.1. A flowchart for a translator which is capable of handling any class of operator is presented in Figure 2.8. The translator
### Table 2.1 Summary of Translator Algorithms and Texts

<table>
<thead>
<tr>
<th>TRANSLATION CLASS</th>
<th>ALGOR. FIGURE</th>
<th>INPUT TAPE MECHANISM</th>
<th>NO. OF STATES</th>
<th>STACK SYMBOLS</th>
<th>EXTRA TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>POSTFIX</td>
<td>2.3</td>
<td>ONE-WAY</td>
<td>1</td>
<td>1</td>
<td>---</td>
</tr>
<tr>
<td>VARIADIC POSTFIX</td>
<td>2.5</td>
<td>ONE-WAY</td>
<td>1</td>
<td>I, LB</td>
<td>(</td>
</tr>
<tr>
<td>PREFIX</td>
<td>2.4</td>
<td>ONE-WAY</td>
<td>1</td>
<td>I, D</td>
<td>---</td>
</tr>
<tr>
<td>VARIADIC PREFIX</td>
<td>2.4</td>
<td>ONE-WAY</td>
<td>&gt;1</td>
<td>I, D, S</td>
<td>---</td>
</tr>
<tr>
<td>VARIADIC PREFIX</td>
<td>2.6</td>
<td>ONE-WAY</td>
<td>&gt;0</td>
<td>I, D, S</td>
<td>)</td>
</tr>
<tr>
<td>HYBRID</td>
<td>2.7</td>
<td>ONE-WAY</td>
<td>&gt;0</td>
<td>I, D, S</td>
<td>( , )</td>
</tr>
<tr>
<td>TREE</td>
<td>2.8B</td>
<td>TWO-WAY</td>
<td>&gt;0</td>
<td>I, D, S, R</td>
<td>( , ) P</td>
</tr>
<tr>
<td>N-PASS</td>
<td>---</td>
<td>TWO-WAY</td>
<td>&gt;1</td>
<td>I, D, S, R</td>
<td>( , ) P</td>
</tr>
</tbody>
</table>

1=operand and result value descriptors; LB=left list bracket; D=driver state; S=translator state; R=input tape head position

---

<table>
<thead>
<tr>
<th>TRANSLATION CLASS</th>
<th>INTERMEDIATE TEXT&lt;EXPR&gt; ::=</th>
</tr>
</thead>
<tbody>
<tr>
<td>POSTFIX</td>
<td>&lt;ID&gt;</td>
</tr>
<tr>
<td>VARIADIC POSTFIX</td>
<td>&lt;ID&gt;</td>
</tr>
<tr>
<td>PREFIX</td>
<td>&lt;ID&gt;</td>
</tr>
<tr>
<td>VARIADIC PREFIX</td>
<td>&lt;ID&gt;</td>
</tr>
<tr>
<td>HYBRID</td>
<td>&lt;ID&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>TREE</td>
<td>&lt;ID&gt;</td>
</tr>
<tr>
<td>N-PASS</td>
<td>&lt;ID&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2<ID>=identifier symbol; symbol table pointer; |=exclusive or; [ ]=arbitrary number of (≥0); [*]=optional symbol; P=pointer; <POST-OP>, <VAR-POST-OP>, <PRE-OP>, <VAR-PRE-OP>, <TREE-OP>=operator symbol.
proper (Figure 2.8a) is the hybrid translator (Figure 2.7) modified to handle a two-way input tape, user-defined procedures, and subscripted variables (discussed in Section 2.6). The driver selected to process a non-postfix-translatable operator may be any of the types discussed in this section -- i.e., prefix (Figure 2.4b), variadic prefix (Figure 2.6), tree (Figure 2.8b), or n-pass (not shown).

2.4 Intermediate Text Structure And BNF-manship

An alternative view of the translator is that of a tree walker. In these terms the structuring of the IT determines the order in which nodes and leaves of the tree are visited by the translator. Postfix corresponds to left-to-right, bottom-up. Prefix corresponds to left-to-right, top-down. Tree-translatable corresponds to a top-down walk which is probably not left-to-right.

Based on this observation, it would appear that syntax-directed techniques can be used only to translate postfix-translatable operators. This is because regardless of the parsing algorithm used by the syntax analyser the semantic routines are always called in a bottom-up fashion. The situation, fortunately for its proponents, is better than this. Translation of prefix- and tree-translatable operators is possible with some additional work. Translation of n-pass-translatable operators, however, is not possible because the syntax analysers allow only one look at the language
constructs as they are parsed. The translation of prefix- and tree-translatable operators is made possible by altering the BNF description of the language so that the parser will present the information to the semantic routines in the proper order.

The general technique is to replace selected terminal and non-terminal symbols in a production with new non-terminals and then add productions for these new non-terminals which will generate the symbols replaced. The new non-terminal forces the syntax-directed parser to make a reduction and in so doing call a semantic routine sooner than it would have in the old grammar.

By way of example, consider the PL/I compound IF-statement:

<Statement¹> ::= IF <Bit string expression>

THEN <Statement²>

[ ELSE <Statement³> ]

<table>
<thead>
<tr>
<th>(Evaluate &lt;Bit string expression&gt; (=0 ))</th>
<th>(Evaluate &lt;Bit string expression&gt; (=0 ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFALSE 0,L1 1</td>
<td>BFALSE 0,L1 1</td>
</tr>
<tr>
<td>(Execute &lt;Statement²&gt;)</td>
<td>(Execute &lt;Statement²&gt;)</td>
</tr>
<tr>
<td>LABEL L1</td>
<td>BRANCH L2</td>
</tr>
<tr>
<td></td>
<td>LABEL L1</td>
</tr>
<tr>
<td></td>
<td>(Execute &lt;Statement³&gt;)</td>
</tr>
<tr>
<td></td>
<td>LABEL L2</td>
</tr>
</tbody>
</table>

a) ELSE Clause Omitted  
b) ELSE Clause Present

Figure 2.9 A Semantic Template for the PL/I IF

the semantic template which defines this operator is given in
Figure 2.9. Because of the optional ELSE clause, and the intervening code between the operands this operator is one-state variadic prefix-translatable. The prefix driver for this is given in Figure 2.10.

```
IF: DRIVER;
   DECLARE (L1, L2) LABEL DESCRIPTOR;
   DECLARE (BFALSE, BRANCH) EXTERNAL SLM INSTRUCTION;

   CALL TRANS;  /* Translate <Bit string expression> */
   CALL GEN(BFALSE, STACK(J), L1);  /* Generate SLM */
   J=J-1;  /* Pop Stack. Finished with descrip. */
   CALL TRANS;  /* Translate <Statement> */
   IF IT(I)=ELSE THEN /* ELSE clause present? */
      DO;
         CALL GEN(BRANCH, L2);  /* Gen. branch over ELSE */
         CALL DEFINE_LABEL(L1);
         CALL TRANS;  /* Translate <Statement> */
         CALL DEFINE_LABEL(L2);
      END;
   ELSE CALL DEFINE_LABEL(L1);
   RETURN;  /* to Translator */
END IF;
```

Figure 2.10 Driver for IF-statement

The BNF statements with sufficient structure to handle the translation of this operator are given by Gries [1971, pp. 277-279] for ALGOL. In the PL/I syntax these become:

```
<Statement> ::= <True part> <Statement> | <If clause> <Statement>  
              <If clause> <Statement> ELSE

<True part> ::= <If clause> <statement> ELSE
```

The semantic routines which accomplish the same tasks as the driver in Figure 2.10 are given in Figure 2.11.
A multiple state translator can be implemented in a syntax-directed compiler only if the parse algorithm is top-down. In that case the BNF programmer has two strategies. In the first strategy a global switch is set by the semantic routine associated with the reduction of the production

\[ \text{<Op-name>} ::= \text{Operator-symbol} \]

which will be called as the operator symbol is scanned — assuming functional notation. With this switch set, as the operand is parsed, its operations will be given special translation.

\[ a) \quad \text{<If clause>} ::= \text{IF <Bit string expression>} \text{ THEN} \]
\[ \quad \text{GEN(BFALSE,<Bit string expression>.ENTRY,L1)}; \]
\[ b) \quad \text{<Statement1>} ::= \text{<If clause> <Statement2>} \]
\[ \quad \text{DEFINE_LABEL(L1)}; \]
\[ c) \quad \text{<True part>} ::= \text{<If clause> <Statement2> ELSE} \]
\[ \quad \text{GEN(BRANCH,L2)}; \]
\[ \quad \text{DEFINE_LABEL(L1)}; \]
\[ d) \quad \text{<Statement1>} ::= \text{<True part> <Statement3>} \]
\[ \quad \text{DEFINE_LABEL(L2)}; \]

Figure 2.11 Semantic Routines

Alternatively, the grammar could be rewritten so that a whole new set of productions would apply to the operand — one set of productions for each different translator state. The semantic routines associated with these new productions would
translate according to the new state rules.

Tree-translatable operators can be handled in this same way. Non-terminals are inserted to direct the order of translation. N-pass-translatable operators, however, cannot be handled by syntax-directed techniques because the parser gives only one parse of the source text. Reductions for a particular construct are produced only once.

So, although it is possible to translate the semantically more intricate operators using syntax-directed techniques, it is done at the cost of large and more obscure defining grammars. In addition, the actual translation process is clouded by the bit manipulations employed by the semantic routines to "get around" the weakness of the basically bottom-up parse/translation algorithm.

Consequently, even if it were possible to translate concurrently with recognition -- the declare-before-use problem having been solved -- it would still be advisable to separate the translation into another pass. The obvious advantages of clearer organization and concise documentation outweigh any disadvantages caused by an increase in the space or time required for the multi-pass scheme.

All this is not to say that syntax-directed techniques are antiquated. On the contrary, they have very useful applications in the recognition phases. In particular, they are ideal for generating the intermediate text representation of the abstract program tree and can perform the error correction assumed by this model.
2.5 Statement Translation

Up to this point the discussion has been under the premise that each statement in a language could be considered an operation and thereby easily included in the framework of an expression tree. As operators, statements come in all operator complexities. In PL/I, for example, scalar assignments, SIGNAL, and REVERT are postfix-translatable statements. IF- and ON-statements are prefix-translatable. Most of the others -- DO, GET, PUT, etc. -- are tree-translatable because of their many options and the insignificance of the ordering of these options in the concrete syntax.

The author's experience and their treatment in most compilers would dictate special treatment for statements. Statements are set apart from the other operators, because as a class they have properties which make their inclusion in a general expression translator clumsy and detrimental to the performance of the translator.

Basically, statements have a linear, non-recursive structure. Compare the following grammar for statements with that for hybrid text in Table 2.1:

\[
\text{<Program>} ::= \{\text{<Statement>}\}
\]

\[
\text{<Statement>} ::= \{\text{<Prefix>}\} [\text{<Keyword>}]
\]

\[
[\{\text{<Keyword>}[\text{<expr}>\{,\text{<expr}>\}]\}]
\]

The recursion which is apparent in the nesting of statements within the scope of DO-, BEGIN-, PROCEDURE-, and
IF-statements can virtually be ignored because only a simple stacking mechanism is needed to do the translation. The structuring in the IT is correct so no parse information need be remembered. The IT will have the necessary scope delimiters either as constructs in the language like the END-statement and THEN or ELSE prefixes or supplied by the recognizer -- e.g., a flag on the statement following an IF-THEN-ELSE construct. Finally, the mode of translation is not altered for the translation of the embedded statements.

The linear structure of statements allows the statement translator to be non-recursive -- a much simpler structure than that required in the expression translators described in section 2.3. In addition, by translating statements separately from expressions the expression translator can be simpler. All of the tree-translatable operators used in present-day programming languages are, in-fact, statements. Even PL/I cannot claim a built-in function which cannot be translated from left-to-right -- although POLY comes close.

The flowchart for a statement translator is given in Figure 2.12. Its structure is quite similar to that of tree-translatable operators. Conspicuously absent is the saving and restoring of translator states. The statement drivers are analogous in function to the drivers of section 2.3. They tend to be more elaborate, however.

The principle activity of the statement driver is scanning the IT to discover what options are present and then guiding the expression translator through the expressions
Figure 2.12 Basic Process Flow in STATEMENT TRANSLATOR
DO:
  DRIVER;
  DECLARE TO_CLAUSE BIT(1) INITIAL('0'B), MAXI;
  DECLARE TO_TEST LABEL, P, (I, J, IT(*)) EXTERNAL;
  DECLARE GEN ENTRY(TEMPLATE, DESCRIPTOR, DESCRIPTOR);
  DECLARE (INDEX_ASSIGN, BRANCH, ADD_AND_ASSIGN, COMPARF,
             BGREATER) EXTERNAL TEMPLATE;

  J=J+1; STACK(J).TYPE=%DO_ENTRY; /* GET WORKSPACE */

  P=I+1;  /* SAVE IT CURSOR IN P */
  I=I+4;
    CALL TRANS;  /* TRANSLATE <ID> */
  I=IT(P);
    CALL TRANS;  /* TRANSLATE <EXPR> */
  MAXI=I;
    /* REMEMBER POSSIBLE END OF DO-STATEMENT */

  IF IT(P+1)=-0 THEN DO; /* TO <EXPR> PRESENT? */
    I=IT(P+1);
    CALL TRANS;  /* TRANSLATE <EXPR> */
    MAXI=MAX(MAXI,I);
    /* MAXI=POSSIBLE END OF DO */
    TO_CLAUSE='1'B;
    IF STACK(J).TYPE=%EXPR THEN CALL /* PUT COMPUTED*/
        STORE INTO TEMP(STACK(J));  /* UPPER LIMIT */
        /* INTO A TEMP.*/
  ELSE J=J+1; /* DUMMY STACK ENTRY */

  IF IT(P+2)=-0 THEN DO; /* BY <EXPR> PRESENT? */
    I=IT(P+2);
    CALL TRANS;  /* TRANSLATE BY <EXPR> */
    MAXI=MAX(MAXI,I);
    /* MAXI=END OF DO-STATEMENT */
    IF STACK(J).TYPE=%EXPR THEN CALL /* PUT COMPUTED*/
        STORE INTO TEMP(STACK(J));  /* INCREMENT */
        /* INTO A TEMP.*/
  END;
  ELSE DO; /* SUPPLY 1 IF NO BY <EXPR>, BUT TO <EXPR> */
    J=J+1;
    IF TO_CLAUSE THEN STACK(J)=CONSTANT_ONE;
  END;

  CALL GEN(INDEX_ASSIGN, STACK(J-3), STACK(J-2));

  IF TO_CLAUSE | IT(P+2)=-0 THEN DO; /* INCREMENT <ID> */
    CALL GEN(BRANCH, 0, TO_TEST);  /* SKIP FIRST TIME */
    CALL DEFINE_LABEL(STACK(J-4), RETURN_LABEL);
    CALL GEN(ADD_AND_ASSIGN, STACK(J-3), STACK(J));
    CALL DEFINE_LABEL(TO_TEST);
  END;
  ELSE CALL DEFINE_LABEL(STACK(J-4), RETURN_LABEL);

  IF TO_CLAUSE THEN DO; /* STOP IF <ID> = <EXPR> */
    CALL GEN(COMPARF, STACK(J-3), STACK(J-1));
    CALL GEN(BGREATER, 0, STACK(J-4), END_OF_LOOP);
  END;

  J=J-4;  I=MAXI+1;  /* POP STACK; ADVANCE IT CURSOR */
END DO;

Figure 2.13 DO-statement Driver
which make up the statement in the proper order. Occasionally the driver will generate SLM code to connect expressions or to order their evaluation at run time, but the bulk of the code for statements is generated by the expression translator.

As an example, a driver for a simplified form of the PL/I DO statement is given in Figure 2.13. In the IT this statement has the form

\[
\text{DO}<\text{EXPR-PTR}><\text{TO-PTR}><\text{BY-PTR}><\text{ID}>=<\text{EXPR}>(\text{TO}<\text{EXPR}^2>) [(\text{BY}<\text{EXPR}^3>)]
\]

The three syllables following the operator syllable are pointers to the initial expression (<EXPR>), the TO-expression (<EXPR^2>), and the BY-expression (<EXPR^3>) respectively. The pointer will be zero if the expression has been omitted. The TO and BY expressions, if present, may appear in any order. Examples of the SLM code generated by this driver are illustrated in Figure 1.6.

Unlike expression operators, statements do not have results which are to be used in later statements. The notable exceptions being the compound statements discussed earlier in this section. Thus the overall statement translator structure does not include a stack.

In automata theoretic terms the statement translator is a finite automaton which calls a PDT (the expression translator) from selected states. The formal language describing such a device is called a regular context free language — i.e., a regular expression some of whose terminals generate context free languages. These languages and the parsing structures they induce have been studied by Tixier [1967] and were found
ideally suited for handling modern programming languages.

A feature peculiar to statements is the prefix -- labels, THEN, ELSE, and, in PL/I, condition prefixes. Since these may appear on all statements, their translation is factored out of the statement drivers and into the main translator algorithm. Each prefix causes the translator to generate some SLM code and, as in the case of PL/I condition prefixes, possibly change the state of the translator.

2.6 Data References

Associated with each variable in a program is a procedure for computing the run-time address of the location containing the value of the variable. The function which maps the variable into this address will be called an access function. The access function is defined only within the scope of the variable and depends on both the variable and the environment. A variable is accessible at a point in execution, if the access function is defined for the variable at that point.

To reference a variable in the object code its access function must be realized. A variable whose access function can be completely realized within the addressing structure of the SLM will be called addressable. That is, a variable is addressable if and only if its address can be encoded in the address field of a single SLM instruction. If a variable is not addressable, then SLM code must be used to manipulate portions of the environment so that it becomes addressable.
This code will be called addressing code and the process of
generating it address translation. A subscripted variable,
for example, is not addressable because at least the value of
the subscript must be loaded into an index register before the
indexed value can be referenced in an SLM instruction.

In general, when a variable is scanned by the translator
a working copy of the descriptive material in the symbol table
entry for the variable is made in the translator stack. This
descriptive material includes the attributes of the variable
and its access function. The attributes of a variable are
determined by the recognition phase and are used in the
processing of operators to determine the appropriate sequence
of SLM code to generate. The storage allocation phase
determines the run-time location to be used for the variable
and thereby defines the access function. The access function
is encoded in the symbol table by this phase for use in the
translator and coder.

If the SLM is designed properly most variables will be
addressable. So, as the variable is used in SLM instructions,
only the description of its access path in the architecture of
the SLM need be passed in the internal text. In the rare
event that a non-addressable variable is referenced, its
addressing code must be generated some time before the first
SLM to reference the variable is generated.

Addressing code may be generated conveniently in one of
two places: when the variable is first scanned or when
processed by the operator in which it is used. The principle
of latest binding argues for the latter location. This placement, however, would tend to complicate the already involved process of selecting the proper code sequence. If the language allows, as most do, operators to manipulate all accessible variables, address translation can be quite complex. To reduce program size, the translation must be implemented as a subroutine to be called by every operator processor.

The major argument against generating addressing code when the variable is scanned is that the practice ties up critical resources -- such as registers -- prematurely. Despite this, the author feels that address translation should be conducted at scan time. Placing it there seems cleaner because it separates addressing from evaluation. This is also more consistent with the treatment of operators. They are translated as they are scanned. In a certain sense variables requiring addressing code are actually operators with no (explicit) operands. In fact, subscripted variables and user-defined functions have been treated as operators from the start. Finally, many languages require left-to-right evaluation of operands -- especially subscripts. In the translation schemes presented here this is equivalent to address translation on scanning.
An Example From PL/C

In the PL/C implementation, the SLM supports indirect addressing to an arbitrary depth. Addresses are given in terms of a base address and an offset. In this case the notion of addressability has the following definition:

A location is addressable (in PL/C) if it is either a register or is a constant positive offset less than 128K from an absolute machine base address contained in an addressable location.

Variables are assigned space in a data area, called an activation record [Wegner 1968], which is associated with the block in which the variable is declared. In the run-time environment, the base addresses of selected activation records are maintained in registers. The base addresses of other accessible activation records are stored in these selected records.

In this way almost all PL/C variables are addressable. One exception is BASED variables. The storage address of the data referenced through a BASED variable is contained in a POINTER variable associated with the BASED variable. To guarantee the integrity of the PL/C compiler, the address in the POINTER variable cannot be used until it is determined at run time that it is a valid machine address and in fact addresses data of the same type as the BASED variable.

If it were not for this test, the value of the BASED variable would be addressable -- i.e., accessible indirectly
through the POINTER variable. The addressing code generated when a BASED variable is scanned passes the POINTER and BASED variables to a run-time routine which verifies the address and returns it in a register. The fact that, at run time, the value of the BASED variable will be located zero bytes from the address contained in that register is indicated, at compile time, in the stack entry for the variable along with its attributes. This addressable value may be used with impunity in any operation which follows.

Explicit Address Functions

Whereas the translation of every operator may generate code to compute a value, the information of interest to the rest of the translator is the description of the type and location of the resulting value. With this in mind, it is easy to see how array references and other left-side functions are handled by this model.

Unlike other operator drivers, left-side function drivers usually generate code to compute an address rather than a value. Occassionaly they only modify the description of the address of their operand(s) and use that as the result. Nonetheless, just like all operators, the result at compile time is a description of a value and its (computed) location. The driver for the PL/I built-in function and pseudo-variable IMAG, for example, generates no code at all. Instead it produces at compile time a description of the imaginary part
of the COMPLEX variable which was its operand in such a way that it will be treated by the rest of the translator as a REAL variable.

Array references are the most common left-side functions. As in the case of user-defined procedures all array names are mapped to an array application (AA in Figure 2.8a) operator when they are encountered in the scan. In one-state languages such as FORTRAN and BCPL, the postfix notation can be used with this operator, although prefix format may make possible the generation of better code. In multi-state languages such as PL/I, the prefix form is necessary to insure that subscripts are always evaluated in the proper state (compute scalar value).

For example, in translating LENGTH(A(I)) the prefix operator LENGTH will set up the translator state to compute the length of A(I). If the subscript I is not prefixed by the array name A the translator will stumble onto the subscript in this length translation state. In fact, the subscript should be evaluated for its value. Whether prefix or postfix is used, it must be emphasized that array application requires no more processing structure than the other prefix operators of the language.

Other left-side functions appear in PL/I at all levels of translation complexity. For example, REAL and IMAG are postfix-translatable. SUBSTR is one-state prefix-translatable, and STRING is multi-state prefix-translatable.
2.7 **Summary of Translator Model**

The overall structure of the model translator which has been developed in this chapter is shown in Figure 2.14. The model differs from other formal presentations in that it assumes the APT, in its entirety, is passed as an IT from the recognition phase. Starting from this new viewpoint it has been possible to concentrate on the process of translation and analyze the form of the translator and IT which is best for the translation of an operator rather than the parsing of its source language representation.

As indicated in Figure 2.14 the model translator is basically a pushdown transducer -- a finite-state machine with a tape for input, a tape for output, and a pushdown stack structure for working storage. Section 2.1 discusses the nature of the input tape, which in this model is the IT representation of the APT. Section 2.2 establishes the concept of a translator state and its analogy with the state in the finite-state automaton of the PDT.

Section 2.3 adds more detail to the model by giving structure to the finite-state machine of the PDT. The internal structure of the translator depends on the semantics of the operators which are being translated. Based on a functional definition of operator semantics -- called semantic templates -- seven distinct classes of operators can be distinguished. Associated with each class is an IT format, which minimizes the efforts of the translator, and an
Figure 2.14 Model Translator Structure
algorithm for translating the operators of the class from that IT format into the SLM sequences defined by the semantic template. These are summarized in Table 2.1.

Operators of all classes can be represented in one IT format and can be translated using the algorithm in Figure 2.8. The overall structure of this translator is indicated in Figure 2.14. The expression translator proper, handles the translation of postfix operators and deals with the special punctuation of the IT to sort out the operands of the more semantically complex operators. When a non-postfix-translatable operator is scanned by the translator proper, it relinquishes control to a driver subroutine.

Each driver controls the translation of all the operands associated with the operator symbol which caused the driver to be called. In particular, each driver determines --

1) which operands are to be translated and how many times,

2) the order in which the operands are to be translated,

3) the state in which they are to be translated, and

4) the proper sequences of SLM instructions which should be generated to manipulate the evaluated operands or order their evaluation.

Needless to say both the translator and each driver are recursive algorithms.

In Section 2.4, the model is compared with the semantic routines used for translation in a syntax-directed compiler.
This is the only other context in which translation is discussed formally.

Although statement keywords may be considered operators and therefore handled by the expression translator developed in Section 2.3, there is good reason to treat them specially. In general, statements are non-recursive structures and therefore do not need the stack mechanism of the expression translator. In addition, considered as operators, they are the most complex and are the sole members of the tree-translatable class. As a result, separating statement translation from expression translation simplifies both translators. The organization of the statement translator and the nature of the statement drivers is discussed in Section 2.5.

Finally, Section 2.6 focuses on those special operators which generate new addresses rather than new values. Nonetheless, they can be handled naturally within the framework of the translator presented in Section 2.3.

So, this chapter has analyzed in detail, the process of translation and has developed a model translator which is defined succinctly by the flowcharts in Figure 2.8. Furthermore, the translator used in the PL/C compiler is essentially this model and is therefore evidence of the practicality of the model.
CHAPTER 3. CODING

The last major component of the code generator model to be discussed is the coder.

Stated briefly, the function of the coder is to produce for each SLM instruction a sequence of object code -- i.e., machine instructions for the computer on which the program is to execute -- which when executed will manipulate the operands of the SLM instruction in a way which will produce the result(s) specified for the SLM instruction. This is analogous to the coding process a machine language programmer goes through as he transforms the flowchart and program specifications into machine language code. The coding process involves choosing an appropriate sequence of code for each SLM operation, deciding what data items are needed and where they can be placed, and assigning the critical resources of the machine -- principally CPU registers -- so as to make the best use of them.

Like the translator, the coder must realize a pre-defined mapping between internal representations. In the case of the coder this mapping is the implementation map from the SLM internal representation to the object machine representation. The implementation is well defined before the coder is constructed and is always assumed correct. The semantic map was dismissed this same way in chapter one.

The implementation map has two distinct dimensions. The representational dimension carries the SLM data structures
into the selected object machine data structures. The
**functional dimension** maps each SLM instruction into an
equivalent sequence of object machine instructions. These
dimensions are almost independent. Although different machine
instructions may be needed to manipulate different operand
data types, the sequence of machine operations performed in
the implementation of an SLM instruction will be essentially
the same for all operands. That is, ADD will always compute
the sum of its operands, but the operands may be in quite
different number representations and have arbitrarily complex
access functions. In general the coder must be able to handle
all combinations of SLM instruction and operand data
structures.

### 3.1 Communication with the Translator

The principal input to the coder is the stream of SLM
instructions generated by the translator -- the internal text
between these phases. A distinguishing characteristic of this
input is that there is little, if any, interaction between SLM
instructions -- certainly not that present in the intermediate
text representation of the APT.

The model assumes that the SLM operator and designations
for its operands are passed to the coder as a single unit.
Furthermore, the coder must see the SLM instructions of a
straight line sequence in the order in which they are to be
executed. Consequently, as each SLM is processed by the
coder, or \textit{coded}, all of the object code to be associated with it -- except possibly for forward references to code -- can be generated immediately. And after the code has been generated, the input can be forgotten.

The only information which must be retained from one SLM instruction to the next is the status of the object machine or \textit{run-time environment}. Each SLM instruction, when executed, changes the state of the SLM. This SLM state has many components. The primary components are the contents of SLM CPU registers, the content of core storage, the instruction location counter, and the machine interrupt status. For example, an ALGOL machine will have a stack storage, program area (blocks), and some registers to hold the display of environment pointers [Wegner 1968].

Each of these SLM state components has a counterpart in the run-time environment. Some subset of the object machine's architecture will be used to implement each component of the SLM machine as dictated by the representational dimension of the implementation. The object code generated for each SLM instruction, when executed, will manipulate the run-time environment to reflect the change in SLM state which would be brought about by the SLM instruction when it is executed.

In the coder a description of the run-time environment is maintained. In addition to generating object code for an SLM instruction, the coder also updates the description of the run-time environment so that it describes the new environment resulting from the execution of the object machine.
instructions generated for the SLM instruction.

This description of the run-time environment is the only information which must be retained from one SLM instruction to the next.

![Diagram](image)

**Figure 3.1 One-pass Code Generator**

Because of the incremental nature of the coder, two different partitionings of the code generation phase are possible. Coding may be done as the SLM instructions are generated -- a one pass code generator -- or the SLM
Instructions can be written into an intermediate text and coded in a later pass. In this chapter a coder is described which exists in the same pass as the translator. (See Figure 3.1.) In this one-pass model whenever the translator decides an SLM instruction should be generated, the internal text for the instruction is passed to the coder, which immediately generates the corresponding object code. The two pass organization is discussed in chapter four.

3.2 Interpretive Coding Language Templates

Constructing the bit patterns which are to be the object machine instructions is a tedious process and one which potentially requires many instructions in the coder. For example, approximately five instructions are needed to put together from its parts one instruction of the object code on the IBM 360/370 computers [IBM 1968a]. If these instructions were encoded in-line, the size of the coder would be prohibitive.

Obviously the solution is a subroutine which does the coding. The argument to the coder must indicate which type of instruction is to be produced and must designate the location of the various parts of the instruction -- the operation code, registers and addresses. These arguments must be a constant part of the code generator. They must be assembled into it and not constructed at compile time. If this is not done, it would take as many instructions to set up the coder arguments
as it would to generate the object instructions in the first place. Because the arguments are constants, they must be interpreted by the coder during code generation to produce the desired object code. This is the basis for the interpretive coder used in this model.

There are many small tasks which are always associated with the generation of object code. These tasks include the acquisition and release of registers and temporary locations, management of the translator stack, and the determination of the types and location of SLM operands. Consequently, provision should be made to perform these tasks within the context of the interpretive coder. That is, add argument codes which when interpreted by the coder, request it to carry out these tasks. The result is the kernel for an interpretive coding language (ICL). To complete the ICL, instructions should be added to provide for unconditional and conditional branching, subroutine calls and the elementary manipulation of bit flags.

The proposed ICL is powerful enough so that any SLM instruction can be coded using only ICL statements. The set of all ICL statements which, when interpreted by the coder, will code an SLM instruction is called a code template. A code template is associated with each SLM instruction generated by the translator. In the one-pass model under investigation here, therefore, the task of the translator reduces to selecting the appropriate templates in the proper order, setting up arguments referenced in the template and
with each template selected passing control to the coder. (The dotted arrow to the boxed "/" in Figure 3.1 and other figures in this thesis is used to indicate this type of control the translator and other modules have over the coder's access to data structures.)

Code Templates in the Compiler Milieu

The concepts of code templates and interpretive coding are not new to compiler construction. ILTRAN [Dewar 1969], FORTRAN IV (H) [IBM 1966], MAD [Arden 1965], MAD/I [Siodawa 1968], PL/I (Boulder) [Elson 1970], and PL/C [Wilcox 1970] are some of the more prominent compilers using some form of interpretive coding.

A very successful compiler for MAD developed at the University of Michigan uses such templates to define on the IBM 7090/7094 computer [IBM 1963], every operator in the MAD language. In fact, it allows the programmer to write his own templates to define new (unary or binary postfix-translatable) operators. Using the templates of the MAD compiler -- called defining sequences -- it is possible to generate any 7090/94 instruction, indexed if desired. Permitted in the address and decrement fields of these instructions are non-negative integers less than 256 or references to 1) either operand of the operator being defined, 2) a temporary, 3) code locations relative to the position of the instruction being generated, 4) any constant permitted in MAD or 5) address constants for
variables and functions named in the program.

If an operand is subscripted the coder automatically supplies the proper index register. Referencing a constant or temporary causes the coder to allocate space for the item. Conditional coding, based on the content of the AC and MQ registers -- the accumulator -- and whether the operands are in the AC or MQ is also possible. The defining sequence also indicates the state of the AC and MQ on exit.

```
DEFINE BINARY OPERATOR .MOD., PRECEDENCE SAME AS /
MODE STRUCTURE INTEGER = INTEGER .MOD. INTEGER

1 JMP "+7,AT,"++1 -> LINE 8 IF LEFT OPND IS IN AC/MQ
2 JMP "+1,AC,"++2 -> LINE 4 IF AC IS FREE
3 STO DT CODE TO SAVE CONTENT OF AC
4 JMP "+1,MQ,"++2 -> LINE 6 IF MQ IS FREE
5 STQ DT+1 CODE TO SAVE CONTENT OF MQ
6 CLA A CODE TO LOAD LEFT OPND INTO AC
7 JMP "+3 -> LINE 10
8 JMP "+2,AC,"++1 -> LINE 10 IF LEFT OPND IS IN AC
9 XCA CODE TO MOVE LEFT OPND INTO AC
10 LRS 35 CODE TO SHIFT LEFT OPND TO MQ
11 DVP B CODE TO DIVIDE LEFT OPND BY RIGHT
12 TZE LOC+6 CODE TO BRANCH ON A .MOD. B = 0
13 TPL LOC+5 CODE TO BRANCH IF A .MOD. B > 0
14 STO =TEMP CODE TO SAVE NEGATIVE REMAINDER
15 CLA B CODE TO LOAD LEFT OPERAND
16 SSP CODE TO TAKE ABSOLUTE VALUE OF OPND
17 ADD =TEMP CODE TO SUBTRACT REMAINDER FROM B
18 OUT AC END OF SEQUENCE. RESULT IN AC
19 END
```

Figure 3.2 MAD Defining Sequence for A .MOD. B

As an example, the defining sequence for A .MOD. B is presented in Figure 3.2. The first two lines of the sequence (unnumbered) define the name of the operator (.MOD.), its precedence (same as the division operator), and the types of
its operands and its result (integer). The remaining lines (numbered) are a mixture of 7090 instructions (STO, STQ, CLA, XCA, etc.) and instructions to the MAD coder (JMP, OUT, and END) which define the code which is to be generated for the operator.

As far as it goes, the MAD definition sequence is a very powerful coding technique. For some reason it has not received the recognition it deserves. The technique is usually extolled only for its virtue as a means for extending an existing language [Arden 1969]. The author feels that the template scheme is invaluable for coding any language. The advantages increase with the size and complexity of the language being compiled.

The definitional facility in MAD, unfortunately, does not demonstrate the feasibility of this technique for modern programming languages and third generation computers. The implementation of a more ambitious language, MAD/I [Bolas 1970a], was attempted by the same group for the IBM System/360 computer, but seems to have had only moderate success [Bolas 1970b]. The MAD/I coder, which expands a macro to code each SLM instruction, seems not to be at fault, however.

The feature of both MAD and MAD/I which makes them less interesting is that they are not block-structured languages -- at least not in the ALGOL or PL/I sense. As a result, all access functions are very simple -- all variables are allocated in a linear array as in FORTRAN. Consequently their coders need not worry about indirect addressing or changes in
the run-time environment which may alter access functions.

Also, the lack of block-structuring and the architecture of the IBM 7090/94 combine to sizably reduce the forward reference problem in MAD. On the IBM 7090/94 any location can be addressed in a single instruction. This means that, in the absence of block structuring which may require several levels of indirect addressing per variable reference, a one-to-one correspondence can be established between the instructions in the template and each instruction of the final object code. With this guarantee the limiting of branches to relative branches in the MAD definitonal facility is acceptable for coding operator definitions.

Relative code addressing is not sufficient for coding statements where an arbitrarily long sequence of code may intervene between the referencing instruction and the location referenced. Presumably MAD had some other mechanism within its coder to handle this problem for its THROUGH and WHENEVER statements.

Without some restrictions, relative addressing is not adequate when coding for a machine such as the IBM System/360 whose instructions have a severely limited range of addressability. The most innocent looking operand reference may require extra instructions to compute its address. So in this case, the coder may not be able to determine at the point of reference the exact address of the operand even though it occurs within the same template.
For these reasons the template scheme used in the MAD and MAD/I compilers is an unsatisfactory one for use in a model of a coder for modern languages.

An IBM research group at Boulder, Colorado, worked on a PL/I compiler which used a template scheme for both translation and coding. The aim of the group was to write a compiler which would produce very efficient PL/I object code -- competitive with that produced by FORTRAN IV (H). This the compiler was able to do, but at the cost of considerable compilation speed. Yet as complex as it was, it did not address itself to the problems stated above. The resolution of forward references and the addition of addressing code was performed by an undocumented coding pass following the interpretive translation/coding pass.

The model presented here is based on the coder for PL/C. While PL/C is technically a subset of PL/I, it is a very rich subset which includes array and structure expressions, fully recursive procedures, and all data types except PICTURE. Code generation is not trivial. It must handle all of the major problems including forward references, a variety of access functions, a constantly changing run-time environment, and the management of both general purpose and floating point registers on the IBM System/360 computer. These facts are adduced to support the contention that the model presented here is fully capable of coding a language as rich as PL/C and can do it with reasonable efficiency.
ICL Statements

A statement in the ICL looks much like an assembly language statement. That is, each statement will have one operation code and list of operands. This format is used for two reasons. First, and foremost, machine-like instructions are easily scanned and induce a simple structuring in the coder. Second, until a useable high-level compiler implementation language becomes widely available, compilers will continue to be written in assembly language. The macro facilities of most existing assemblers are an efficient (and readable) means for assembling the code templates.

The operands of ICL statements may be constants or via some constant code designate the location of the operand value. The operand code could indicate, for example, that the operand is the sixth element in the translator stack, or the first element in a global vector known to both the coder and translator. The addresses of operands may also be passed to the coder when it is called by the translator. A code in the template instructs the coder when to use this address for an operand value.

The templates are parameterized in this way so that a single template may be used for all possible operands of the SLM instruction it defines. This feature also allows SLM instructions with similar implementations to share the same template and so further reduce the space needed for coding.
In most ICL statements the operands will reference a description of some element in the run-time environment or specific portions of that description. The form of the description is detailed in the following sections. When the operand represents the address portion of a machine instruction, as it often does, the ICL statement provides a means for generating in the object code an address field which is a slight modification of the address described by the operand. Possible modifications include additional offsets or indexes, or indirect addressing. These modifications may themselves be constants or parameterized references. In fact, a complete address field -- base address, index, and displacement, for example -- may be specified.

The ability to generate modified addresses using ICL statements is a very powerful tool for the template programmer and yet does not complicate the coder structure unnecessarily. With this facility, for example, it is possible to generate code to reference any field of an array's dope vector given only a description of the address of the start of the dope vector.

A complete ICL for coding the IBM System/360 is defined in Appendix I. This ICL is similar to the one used in the PL/C coder. They differ primarily in areas where, for "historical" reasons, the PL/C ICL is not clearly defined or too specialized. Except for the instruction generation statements, which are machine dependent, this ICL can be adapted easily to other machines. The coder structure
underlying this ICL will be described in the next section.

As an example of the use of this ICL, Figure 3.3 shows a code template which implements on the IBM System/360 the PL/I function MOD(A, B) where A and B are FIXED BINARY (31,0) operands. The template assumes that when it is expanded, descriptions of the operands A and B will be on the top of the translator's stack, followed by an empty stack position which can be used to describe the result.

3.3 Value Descriptors

The principal data structure in the code system is the value descriptor. Every value referenced by the program -- variable, constant, label, program base, data area base, pseudo-register vector, etc. -- must be represented by a descriptor before it may be used in the coder. A value's descriptor contains all of the information pertaining to the implementation of that value.

In the one-pass version of this code generator model, the translator is responsible for ordering the creation and destruction of descriptors. To keep the interface clean, however, the actual creation and destruction is done by the coder in response to certain ICL statements. For variables, labels, and constants the information in the symbol table entry for the element is used by the translator to determine the appropriate description. In the one-pass scheme, their descriptors are created and placed on the compile-time stack.
SECTIO N 3.3 92

MODB BNR REAL2,MODB1
BP REAL2,MODB2
PAIR REAL2

MODB2 BOR REAL2,MODB3

GRS SRDA,RF L,R0,32(R0)

B MODB4

MODB1 LGPR REAL2,ODD,PAIR

MODB3 RDESC REAL2,EVEN
GRX M,REAL2,FONE

MODB4 BR MODB5
LGPR REAL1

MODB5 GRR DR,REAL2,REAL1
GRX LTR,REAL2,REAL2
GRX BC,MM,LABEL
GRR LPR,REAL1,REAL1
GRR AR,REAL2,REAL1
DLAB LABEL
UNPR REAL2
FREE REAL1
SET REAL3,REAL2
POP 2
EXIT

->MODB1 if A is not in register.
->MODB2 if A is in register pair.
Allocate register pair to A
->MODB3 if A is in odd register of pair.
Generate shift to propagate sign.

->Merge.

Generate load of A into odd register of pair.
Indicate A is in even register.
Generate multiply to propagate sign of A to even register.

->MODB5 if B is in a register.
Generate load of B into any general purpose register.

Generate A/B.
Generate test of remainder.
Generate branch not minus.
Generate ABS(B).
Generate ABS(B)-remainder.
Complete forward branch.
Free odd register of pair.
Free B register.
Fill in result descriptor.
Remove entries for A and B.
Return to translator.

1Designates descriptor for real part of second operand on stack (A).
2Designates descriptor (in global area) for constant F'1".
3Designates descriptor for real part of operand on top of stack (B).
4Designates null descriptor in global area which may be used for local forward branches.
5Designates descriptor for real part of third operand on stack. In this case, the stack entry for the result.

Figure 3.3 Code Template For MOD(A,B)
when they are scanned in the IT and destroyed after their first reference.

Other elements will have predetermined descriptors and are created and destroyed in a manner appropriate to the changing environment. The descriptor for the storage area associated with a block, for example, will be created when translation of the block begins and will be destroyed after the last statement of the block has been translated.

Most of the information in the descriptor refers to the object machine. For example, the attributes of a variable or constant are transformed into the attributes of the object machine component which the variable or constant is to occupy. Furthermore, all data items represented by descriptors must be addressable. The access function is encoded in the descriptor in a form which is closely related to the addressing structure of the object machine.

Therefore, once descriptors have been constructed, all program elements have a consistent representation which may be manipulated by the coder without regard for the significance of the element in the program.

The format of a value's descriptor depends on the hardware component of the machine in which the value is located. In general, a descriptor is formatted differently for each different component of the object machine, one format for each component which is made available to the machine language programmer. For the IBM System 360/370 line of computers, for example, there would be a format for general
purpose registers, one for floating point registers, one for each type of addressable storage structure -- bytes, halfwords, fullwords, short and long floating point numbers, byte strings, and possibly even special formats for channel command words, channel status words and program status words. To facilitate the management and manipulation of descriptors, however, these formats should also be closely related. All descriptors should be the same size and from component to component depart only slightly from a general format.

As an example, a value descriptor is presented in Figure 3.4 which is suitable for use with the IBM System/360/370 computers. The hardware component which the value occupies is indicated by a few flags and type indicators. The function of the various fields of this VALUE_DESCRIPTOR will be explained in the following sections.

An important part of the value descriptor is the encoded access function. The enormous utility of the descriptor is due, in large part, to the method of encoding described in the next section.

Address Representation

A major task of the coder is the completion of the representational dimension of the implementation. By the time the coder is asked to interpret a template, all references to source program values will be to addressable locations. The storage allocator will have assigned most of the data to
1 VALUE_DESCRIPTOR UNALIGNED BASED(P),
2 DESCRIPTOR_MANAGEMENT,
3 STATUS,
4 USES_LEFT FIXED BIN(7), /* USED BY OPTIMIZER */
4 VALUE BIT(i) INIT(1). /* 0=RSD, 1=VALUE_DESCRIPTOR */
4 STORAGE BIT(1), /* 0=RPD; BASE -> RSD */
4 TEMPORARY BIT(1), /* 1=TEMPORARY_ALLOCATED */
4 IMAGE BIT(1), /* 1=IMAGE OF A REGISTER */
4 OTHER_FLAGS /* (20),
3 FORWARD_LINK POINTER, /* LINKS IN DESCRIPTOR */
3 BACKWARD_LINK POINTER, /* MANAGEMENT CHAINS */
2 ACCESS_FUNCTION,
3 BASE POINTER, /* =ADDR(VALUE_DESCRIPTOR) */
3 DISP FIXED BINARY(31), /* OR ADDR(RSD) */
3 INDEX POINTER, /* =ADDR(VALUE_DESCRIPTOR) */
2 ATTRIBUTES, /* OR NULL */
2 MACHINE_TYPE,
4 S360_STORAGE BIT(8), /* SEE BELOW FOR MEANING */
4 SCALE FIXED BINARY(7), /* RADIX POINT POSITION */
4 PRECISION FIXED BINARY(7), /* SIGNIFICANT DIGITS */
3 SOURCE_TYPE BIT(8), /* SOURCE LANGUAGE DEPENDENT */
2 VALUE BIT(64); /* IF KNOWN A. COMPILE TIME */

/* S360_STORAGE VALUE AND MEANING

<table>
<thead>
<tr>
<th>VALUE MEANING</th>
<th>VALUE MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 CONDITION CODE</td>
<td>8 DOUBLEWORD INTEGER</td>
</tr>
<tr>
<td>1 BYTE</td>
<td>9 CHANNEL COMMAND WORD</td>
</tr>
<tr>
<td>2 HALFWORD INTEGER</td>
<td>10 CHANNEL STATUS WORD</td>
</tr>
<tr>
<td>3 24 BIT ADDRESS</td>
<td>11 PROGRAM STATUS WORD</td>
</tr>
<tr>
<td>4 FULLWORD INTEGER</td>
<td>12 B I T S</td>
</tr>
<tr>
<td>5 SHORT FLOATING POINT</td>
<td>13 P A CK E D D E C I M A L</td>
</tr>
<tr>
<td>6 31 BIT ADDRESS</td>
<td>14 Z O N E D D E C I M A L</td>
</tr>
<tr>
<td>7 LONG FLOATING POINT</td>
<td>15 BYTE STRING</td>
</tr>
</tbody>
</table>

a) PL/I Structure Definition Of Value Descriptor For IBM /360

VALUE_DESCRIPTOR

| STATUS | HELDATIONAL |
| DESCRIPTOR_MANAGEMENT | FORWARD_LINK |
| | BACKWARD_LINK |
| BASE | (BASE) |
| | DISP | (DISP) |
| INDEX | (INDEX) |
| ATTRIBUTES | VALUE |

b) Key To Diagrammatic Representation Of Value Descriptor

Figure 3.4 Value Descriptor
storage locations. The translator will generate the addressing code for non-addressable values before invoking the coder.

For the following discussions the concept of addressability defined in chapter two must be refined by defining the following forms of addressability:

1) **Absolute**: Bit pattern for address of value is known at compile time.

2) **Direct**: Address can be expressed as a constant offset (known at compile time) from an absolutely addressable base.

3) **Indirect**: Address can be expressed as a constant offset from a non- absolutely addressable base.

4) **Index**: Address can be expressed as a constant offset plus an addressable variable offset (computed at run time) from an addressable base.

Note that under these definitions, a value in a register is absolutely addressable. Its address is the bit pattern needed to reference that register. Therefore, in these terms, the base/displacement form of address specification is a direct address.

If the address (access function) of a value can be encoded directly into the address field of an object machine instruction, the value will be called immediately addressable. Whether a data element is immediately addressable depends on the object machine and not necessarily on the complexity of the access function. Immediate addresses may be indirect or indexed. On the other hand, directly addressable data may not
be immediately addressable. On the IBM/360 computer, for example, any data element which is located less than 4096 bytes beyond an address contained in a general purpose register is immediately addressable. It is also directly addressable. A data element which lies, say, 5000 bytes from a base register is still directly addressable, but it is not immediately addressable because the instruction format on the 360 allows only positive offsets of up to 4095 bytes. On the other hand, on the IBM 7090, all directly addressable values are immediately addressable. Note, however, that direct addresses on the 7090 require the "indirect addressing" (their term) feature of this machine. Also, all absolutely addressable locations are immediately addressable on the 7090, whereas only those absolute addresses which are registers or in the first 4096 bytes of storage are immediately addressable on the 360.

The immediate addressability of a data element may also depend on the object machine instruction. Again, on the 360, index addressable data cannot be immediately addressed if the instruction is anything but an RX instruction, and then only if the index is in a general purpose register. Similar restrictions exist for the 7090.

The access function (address) of an addressable value is encoded in its descriptor. The encoding depends primarily on the complexity of the addressing structure of the SLM. In any event, it should be possible to encode in the descriptor the address of any immediately addressable value. The full
capabilities of the object machine could not be utilized otherwise. On the other hand, for every access function which can be encoded in a descriptor, it must be possible to generate object code which will make the value described immediately addressable. So, considering the architecture of modern computers, a general descriptor mechanism must handle all forms of addressability defined above.

Since almost every object machine instruction involves a data reference, and each data reference means the address in the data element's descriptor must be decoded, the access function must be encoded in descriptors so that it can be decoded efficiently. As a general rule this means that all access functions should be encoded according to a general, consistent scheme. In this way the address computer need not test for special cases. In particular, the "address" of a value contained in a CPU register should have a natural encoding. In addition, the encoding should be flexible so that as the environment changes, the access function can be modified to take advantage of the changes.

These objectives can be realized using only the BASE, DISP, and INDE X fields of the VALUE_DESCRIPTOR defined in Figure 3.4. In this scheme, the address portion of each descriptor may have one of three formats -- register, address, or storage. The use of each field in each format is summarized in Table 3.1.
Values In Registers

A register formatted descriptor (RFD) specifies that the value described is in a CPU register. In the model, the register format is indicated by a zero bit in the STORAGE field of the VALUE_DESCRIPTOR. The BASE field points to a REGISTER_STATUS_DESCRIPTOR (RSD) for the CPU register which the value occupies. A model RSD for the 360 computers is defined in Figure 3.5.

<table>
<thead>
<tr>
<th>Addressability</th>
<th>BASE Field Value</th>
<th>DISP Field Value</th>
<th>INDEX Field Value</th>
<th>Format Abbrev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute</td>
<td>Null</td>
<td>Bit pattern for address</td>
<td>Null</td>
<td>AFD</td>
</tr>
<tr>
<td>Pointer to RSD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Direct</td>
<td>Pointer to AFD for base.</td>
<td>Offset (Integer)</td>
<td>Null</td>
<td>RFD</td>
</tr>
<tr>
<td>Indirect</td>
<td>Pointer to Non-AFD for base.</td>
<td>Offset (Integer)</td>
<td>Null</td>
<td>SFD</td>
</tr>
<tr>
<td>Indexed</td>
<td>Pointer to descriptor for base.</td>
<td>Offset (Integer)</td>
<td>Pointer to descriptor for base.</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1 Encoding of Access Function in Value Descriptor

An RSD is associated with each CPU register which can be referenced in a machine instruction. It must be emphasized that RFD's and RSD's are distinct structures. The RFD describes a value, which happens to be in a register. The RSD
1 REGISTER_STATUS_DESCRIPTOR UNALIGNED BASED(P),
2 DESCRIPTOR_MANAGEMENT,
3 STATUS,
4 USES_LEFT FIXED BIN(7), /* SEE CHAPTER FOUR */
4 VALUE BIT(1) INIT(0'), /* 1=VALUE_DESCRIPTOR, 0=RD*log */
4 DEDICATED BIT(1), /* 0=REGISTER IS MANAGED */
4 GENERAL_P BIT(1), /* 0=FLOATING POINT REGISTER */
4 PAIRED BIT(1), /* 1=EVEN/ODD PAIR */
4 SAVED BIT(1'), /* VALUE STORED IN TEMPORARY */
4 OTHER_FLAT BIT(3),
4 REGISTER_INFORMATION,
5 REG# BIT(4), /* ADDRESS OF REGISTER */
5 ALLOCATION_CLASS BIT(4), /* SEE BELOW */
5 STORE_OP BIT(8),
3 (FORWARD_LINK, BACKWARD_LINK) POINTER,
2 CONTENT_ADDRESS,
3 OLD_BASE POINTER,
3 OLD_DISP FIXED BINARY(31),
3 OLD_INDEX POINTER,
2 CONTENT_ATTRIBUTES,
3 MACHINE_TYPE,
4 S360_TYPE, /* SAME AS S360_STORAGE */
4 SCALE, PRECISION) FIXED BIN(7),
3 SOURCE_TYPE BIT(8), /* SOURCE LANGUAGE DEPENDENT */
2 CONTENT_DESCRIPTOR POINTER; /* POINTS TO RFD FOR CONTENT */
/* VALUE MEANING OF ALLOCATION_CLASS */
0 REGISTER IS EMPTY -- CONTENTS UNKNOWN.
1 NO USES LEFT -- ALTERNATE ADDRESS KNOWN
2 USES LEFT AND IMAGE IS IN STORAGE.
3 USES LEFT AND IMAGE IS NOT IN STORAGE.
4 INDEX REGISTER.
5 UNREFERENCED MEMBER OF EVEN/ODD PAIR. */

a) PL/I Structure Definition Of RSD For IBM /360

```plaintext
REGISTER_STATUS_DESCRIPTOR
  DESCRIPTOR_MANAGEMENT
    STATUS
    FORWARD_LINK
    BACKWARD_LINK
    OLD_BASE (BASE)
  ACCESS_FUNCTION
    OLD_DISP (DISP)
    OLD_INDEX (INDEX)
  CONTENT_ATTRIBUTES
    CONTENT
```

b) Key To Diagramatic Representation Of RSD

Figure 3.5 Register Status Descriptor
describes the state of a hardware component. These descriptors work together to provide a flexible description of the run-time environment. The RFD specifies the register involved by pointing to the appropriate RSD. In turn, the CONTENT_DESCRIPTOR field of the RSD points to the RFD which describes its current content. Since only one value may be in a register at a time, only one content pointer is needed in the RSD. This cross-referencing allows the register manager easy access to the contents of any register at any time during compilation.

Values In Storage

The address formatted descriptor (AFD) specifies that the value described is in an absolutely addressable storage location. In this case both the BASE and INDEX fields are null; the address is stored in the DISP field.

The power of this scheme lies in the storage formatted descriptor (SFD). In this format the BASE field specifies another VALUE_DESCRIPTOR and the DISP field contains an integer constant. A descriptor in this format specifies that the value is located in storage the number of address units specified in the DISP field from the address contained in the location defined by the VALUE_DESCRIPTOR designated by the BASE field. If the BASE->VALUE_DESCRIPTOR is an RFD or AFD, the SFD defines a directly addressable value, otherwise it describes an indirectly addressable value.
Note the universality of this address representation. Since the BASE descriptor can itself be in any of the three formats, any level of indirection can be described. If the BASE descriptor specifies a general purpose register, then the descriptor has encoded the base-displacement address field of the IBM 360 or any other base register machine. Alternatively, if the DISP field is zero and the BASE descriptor is in address format, then the descriptor has encoded indirect addressing on a second generation computer such as the IBM 7094.

Despite this flexibility and generality in describing access functions, all access functions described in this fashion can be implemented on either type of computer. An algorithm for generating IBM 360 addressing code from this access function encoding is given in Figures 3.6 and 3.7 as an example.

In the storage format, the INDEX field designates a VALUE_DESCRIPTOR just like the BASE field. With this addition -- a variable displacement actually -- practically any access function imaginable can be described. Still, any descriptor so encoded can be implemented on both machines, although the algorithms are somewhat more complex.

Uses of the Descriptor Mechanism

RSD's and value descriptors work together to keep track of intermediate results. An intermediate result is usually
MRXA AND MRSA ICL STATEMENT PROCESSORS:

INPUT -- ARG = POINTER TO A VALUE DESCRIPTOR

ACTION -- IF NECESSARY, CODE IS GENERATED TO MAKE THE VALUE DEFINED BY ARG->SFD IMMEDIATELY ADDRESSABLE IN EITHER AN RS OR RX INSTRUCTION DEPENDING ON THE ENTRY POINT USED.

OUTPUT -- ARG->SFD IS MODIFIED TO DEFINE THE NEW IMMEDIATE ADDRESS.

MRXA: PROCEDURE (ARG); RECURSIVE; DECLARE ARG POINTER;
DECLARE RS BIT(1) INITIAL('1'B); /* 1 FOR NULL INDEX
DECLARE R POINTER; /* TO SPFD FOR SCRATCH REGISTER
DECLARE PART1 BIT(12);

RS '=' '0'B; /* AN INDEX REGISTER MAY BE USED

MRSA: ENTRY (ARG); /* BASE/DISPLACEMENT ADDRESS ONLY

IF NOT ARG->STORAGE THEN DO;

/* VALUE IS IN A REGISTER. STORE IT INTO TEMPORARY
/* STORAGE TO MAKE IT RS-ADDRESSABLE
/* COLLECT INFORMATION FROM RSD
/* (CONSTRUCT FIRST PART OF STORE INSTRUCTION)

PART1=ARG->BASE->STORE_OP || ARG->BASE->REG0;
CALL FREE(ARG); /* FREE REGISTER VALUE OCCUPIES

/* ALLOCATE TEMPORARY STORAGE AND
/* RE-FORMAT VALUE RFD TO AN SFD

CALL ASE(TEMP_DATA_AREA, ARG);

/* MAKE TEMPORARY RX-ADDRESSABLE

CALL MRXA(ARG); /* SHOULD BE RS-ADDRESSABLE ALREADY

/* FINISH GENERATION OF STORE INSTRUCTION

PUT AREA(CODE_AREA) EDIT /* EXPLAINED ON NEXT PAGE */
(PART1, ARG->INDEX->BASE->REG0) (H(3), H(1))
(ARG->BASE->BASE->REG0, ARG->DISP) (H(1), H(3));
RETURN;
END;

1ICL statement processor for FREE <SD>. See Appendix I for definition.
2ICL statement processor for ASE <SAD>, <D>. See Appendix I for definition.

Figure 3.6 Address Computer For 360 Coder
/* IF BASE IS NOT IN A REGISTER THEN GENERATE LOAD */

IF ARG->BASE->STORAGE¹ THEN CALL LGPR(ARG->BASE,%ANY)²;

IF ARG->DISP<0 OR ARG->DISP>4095 THEN DO;

/* DISP WON'T FIT INTO AN INSTRUCTION */
/* GET A REGISTER TO HOLD THE DISPLACEMENT */

R=GGPR(... l,3,NUL)³; /* R -> RPFD FOR NEW REGISTER */
R->USES_LEFT = ARG->USES_LEFT; /* RETAIN WITH VALUE*/

/* GENERATE INSTRUCTIONS TO COMPUTE IN REGISTER */
/* |FLOOR(DISP/4096)*4096| */

PUT AREA(CODE_AREA)* EDIT /* PUT CODE INTO AREA */
(MLA,R->BASE->REG#,ABS(FLOOR(ARG->DISP/4096)),
 MSLA,R->BASE->REG#,12) (H(2),H(1),Z(2),H(3))⁵;

/* IF DISP<0 THEN GENERATE CODE TO NAGATE REGISTER */

IF ARG->DISP<0 THEN PUT AREA(CODE_AREA) EDIT
(MLCR,R->BASE->REG#,R->BASE->REG#) (H(2),2 H(1)):

/* IF NO INDEX ALLOWED, THEN GENERATE CODE TO ADD */
/* BASE ADDRESS TO DISP AND FORM NEW BASE ADDRESS */

IF RS THEN DO; /* MSA ENTRY POINT USED */

PUT AREA(CODE_AREA) EDIT
(MLB,R->BASE->REG#,ARG->BASE->BASE->REG#) (H(2),H(1),H(1));
CALL FREE(ARG->BASE); /* ONE LESS USE LEFT */
ARG->BASE = R; /* NEW REGISTER IS NEW BASE */
END;
ELSE ARG->INDEX = R; /* NEW REGISTER IS NEW INDEX */

/* REST OF DISP GOES IN REFERENCING INSTRUCTION */

ARG->DISP=MOD(ARG->DISP,4096);
END;
END MRAA;

¹See Figure 3.4 for definition of terms.
²See Figure 3.7 for definition of LGPR.
³<RD>=GGPR(<TYPE>,<CLASS>,<SD>) is GGPR statement processor.
(See Appendix I.)
⁴This is not PL/I, but it has the flavor of the language.
⁵Two new format items: H(n) right justifies data in an n-digit hexadecimal field (4n bits), pads with zero or truncates on the left; Z(n) generates n hexadecimal zeroes.

Figure 3.6 (con'd) Address Computer For 360 Coder
LGPR: PROCEDURE(ARG,TYPE) RECURSIVE;
DECLARE ARG POINTER, TYPE FIXED BINARY(7);
/*
  INPUT -- ARG = POINTER TO A VALUE DESCRIPTOR
  TYPE = CONST FOR TYPE OF REGISTER DESIRED.
  (*EVEN, %OOL, %EVEN_PAIR, %ODD_PAIR, %ANY, ...
  %ANY_PAIR*)
  ACTION -- CODE IS GENERATED TO LOAD VALUE DEFINED BY
  ARG INTO A (NEW) REGISTER.
  OUTPUT -- ...
  VALUE_DESCRIPTOR IS MODIFIED TO DEFINE
  THE NEW LOCATION.
  DECLARATION R POINTER; /* TO NEW REGISTER */
*/
IF ARG->STORAGE THEN DO; /* VALUE IS IN STORAGE */
  CALL MRX(A)(ARG); /* MAKE VALUE RX-ADDRESSABLE */
  /* AFTER LOAD, ONE LESS USE OF ARG WILL BE LEFT */
  CALL FREE(ARG); /* ACCOUNT FOR THIS FACT NOW */
  /* ALLOCATE A NEW REGISTER FOR THE VALUE */
  /* GPPR MAY REALLOCATE ARG->BASE OR ARG->INDEX */
  R=GPPR(TYPE,2,ARG);
  /* GENERATE CODE TO LOAD VALUE INTO NEW REGISTER */
  PUT AREA(CODE_AREA) EDIT
  (%L,R->BASE->REG#,ARG->INDEX->BASE->REG#, 
   ARG->BASE->BASE->REG#,ARG->DISP)
  (H(2), H(1), H(1), H(1), H(3));
END;
ELSE DO; /* VALUE IS IN A REGISTER */
  /* AFTER LOAD, ONE LESS USE OF VALUE WILL BE LEFT */
  CALL FREE(ARG); /* ACCOUNT FOR THAT NOW */
  R=GPPR(TYPE,ARG->ALLOCATION_CLASS,ARG);
IF ARG->BASE==R->BASE THEN DO;
  /* NEW REGISTER IS NOT THE SAME AS THE OLD ONE */
  /* SO GENERATE CODE TO LOAD VALUE INTO NEW ONE */
  PUT AREA(CODE_AREA) EDIT
  (%L,R->BASE->REG#,ARG->BASE->REG#)
  (H(2), H(1), H(1));
END;
END;
/* ASSOCIATE ORIGINAL VALUE WITH REGISTER AND FORMAT */
/* VALUE_DESCRIPTOR AS AN RFD (SEE APPENDIX I) */
CALL LINK(ARG,R);
END LGPR;

Figure 3.7 LGPR (load General Purpose Register) Statement
Processor
left in the register in which it was computed. The descriptor
for the result sits in the stack and points to the appropriate
RSD (see Figure 3.8a). Later, however, that register may be
needed for other computations, and its contents must be saved.
When this happens, the register manager obtains temporary
space (from the storage area manager) and generates the
appropriate store instruction. Then the RFD for the
intermediate result is changed to an SPD which defines the new
temporary location. The RSD is finally cleared for its new
assignment (see Figure 3.8b and 3.8c).

Figure 3.9 illustrates how the descriptor mechanism can
be used to locally optimize base register loads required to
implement indirect addressing. The method is effective only
as long as descriptors of elements with bases which are
logically equivalent reference the same BASE descriptor. For
example, A and B in Figure 3.9a have been allocated space in
the same storage area, SA1. Therefore, logically they have
the same base address. So, to take full advantage of the
descriptor mechanism, the BASE fields of the descriptors for
both A and B should point to the descriptor for SA1 as in
Figure 3.9b and 3.9c. When, in the address computation
algorithms in Figures 3.6 and 3.7, the base address, SA1, is
loaded into a register, its SPD is changed to an RFD for that
register. The description of the base address's original
storage address -- D3(SA2) in this case -- is saved in the
CONTENT_ADDRESS portion of the RSD (see Figure 3.9c). Any
later references to SA1 will automatically reference that base
a) Descriptors After Computing Result in R1

b) Descriptors Just After Reallocating Result Register (R1)

Figure 3.8 Management of Result Descriptors
c) Descriptors After Loading Register R1 with New Value

```
A       R1,D1(B)
```

b) `ST R1,D2(R6)`

c) `L R1,D3(X,B)`

d) Key to Situations Described by a), b), and c)

Figure 3.8 (con'd) Management of Result Descriptor
a) Run-time Environment

b) Initial Descriptor Arrangement

Figure 3.9 Optimizing Base Register Loads
c) After Loading Base Address and B

  b) \[ \rightarrow \quad \text{LOAD B} \]

  c) \[ \rightarrow \quad \text{LOAD } A \]

  d) Key to Situations Described by b) and c)

Figure 3.9 (Con'd) Optimizing Base Register Loads
register. When the register is needed for another purpose, the ACCESS_FUNCTION field of SA1’s descriptor is reset to its original value from the CONTENT_ADDRESS field of the RSD.

Address Chaining in PL/C -- Another Example

The code generator for PL/C uses this access function encoding — without the index field — for all data references generated by the PL/C coder. The definition of addressable in PL/C (Section 2.6) assures that all base address chains must eventually terminate in some register descriptor. This eventuality reflects the fact that to reference anything on the IBM 360 computer an address must be specified in a register so that the base/displacement address can be used in an instruction.

The address decoder used in PL/C is analogous to that used in Figure 3.6, but because the maximum length of an address chain is known to be five descriptors long, the algorithm is not implemented as a recursive routine. By minimizing the time spent in this critical routine, the efficiency of the code generator is greatly improved.

The indirectness of the descriptor chains is used to advantage in PL/C to solve the tricky problem of changing environments in the block structured language. When space is allocated to a variable a descriptor which describes that location is constructed and placed in the variable’s symbol table entry. When the variable is referenced in the IL, this
information will be moved to the stack and form the ACCESS_FUNCTION field of its value descriptor. The DTSP field is the offset of the variable in some activation record (AR). If the storage class of the variable is STATIC, the BASE field will point to the F.J for register 11 -- the run-time location of the base address for the STATIC AR. If the storage class is AUTOMATIC, the BASE field will designate the descriptor for the base address of the AR associated with the block in which the variable is declared. All of the variables in Figure 3.10 are AUTOMATIC. Note how the descriptors shown in Figure 3.10d are constructed to describe the access functions of these variables in the run-time environment diagramed in Figure 3.10c.

This single AR descriptor is manipulated by the translator to reflect changes in the environment in which the code will be executed. Thus, when the block is first entered -- during code generation -- the AR descriptor will be an RDF for register 13 -- a dedicated register pointing to the current AR at run-time. As internal blocks are entered, the descriptor will be changed to point to the RSD for register 9 and then register 8 -- dedicated registers which hold the base addresses of the AR's for the blocks at "current depth minus one" and "current depth minus two." Finally the AR descriptor becomes an SPD defining the location in the current AR -- based on register 13 -- which contains the base address of the AR for the variable's block. Figure 3.10d shows the state of the descriptor area after the code generator has worked its
STMT LEVEL W3ST BLOCK PL/I SOURCE STATEMENT

1    M:    PROCEDURE OPTIONS(MAIN);
2    1     DECLARE 1 S, 2 A, 2 B;
3    1     CALL P(S);
4    1     PROCEDURE(PS);
5    2     DECLARE 1 PS, 2 A, 2 B;
6    2     B1:     BEGIN;
7    3     DECLARE C;
8    3     B2:     BEGIN;
9    4     DECLARE D;
10   4     B3:     BEGIN;
11   5     DECLARE E;
12   5     PS.B=(E-D)*C/S.B+PS.A

a) Sample PL/I Program Segment

LD F2,D9(0,R13) (E
SD F2,D6(0,R9)
MD F2,D4(0,R8)  -D)
L R15,D7(0,R13)  *C
L R15,D2(0,R15)  (R15 is a PL/C scratch register.)
DD F2,0(0,R15) /S.B
L R10,D8(0,R13)  (R10 is the other scratch register.)
L R10,D3(0,R10)
L R10,0(0,R10)
AD F2,0(0,R10)  +PS.A
L R15,D8(0,R13)
L R15,D3(0,R15)
L R15,4(0,R15)
STD F2,0(0,R15)  =PS.B

b) Code Generated for Statement 12 in a) by PL/C

Figure 3.10 An Example From PL/C
Figure 3.10c Run-time Environment While Executing Statement 12
Figure 3.10d Descriptor Structure While Compiling Statement 12
way to a block nesting level of five. As a result, some variables are shown at each stage of this progression.

The only explicit IC statement associated with address computation is "make immediately addressable." (In the 360 ICL of Appendix I there is such a statement for each address format -- i.e., "make HS-addressable," and "make RX-addressable." ) However, address computation is an automatic and necessary part of instruction generation statements. Many other ICL statements alter the ACCESS_FUNCTION portion of the descriptor.

3.4 Coder Structure

The functions performed by the coder can be divided into five categories:

1) descriptor construction and management,
2) address computation,
3) register management,
4) storage area management,
5) instruction generation.

Some of these functions will be requested by each ICL statement.

The coder is organized into three major components: control module, statement processors, and utility modules. The calling structure surrounding these components is shown in Figure 3.11. The control module scans the template, decodes each statement and its operands, and then calls the processor
Figure 3.11 Coder Calling Structure
CONTROL_MODULE:
PROCEDURE(T,PARGS);
DECLARE (T /*TO TEMPLATE*/, D /*TO DESCRIPTOR*/) POINTER;
DECLARE PARGS(*) POINTER; /* PARAMETER ADDRESSES */
DECLARE SFD BASED(D) :IKE VALUE_DESCRIPTOR;
DECLARE 1 ICL_STATEMENT_FORMAT BASED(T) UNALIGNED,
2 OP BIT(8),
2 OPNM(*) ,
  INDIRECT BIT(1),
3 DISP,
  4 TYPE BIT(2),
  4 VALUE BIT(13),
3 BASE,
  4 TYPE BIT(2),
  4 VALUE BIT(8),
3 INDEX,
  4 TYPE BIT(2),
  4 VALUE BIT(8);
DECLARE OPERAND(4) POINTER; /* DECODED OPERANDS */
DECLARE STATEMENT_PROCESSOR(256) ENTRY VARIABLE
1 INITIAL(HRSA,HRXA,GRI,GAR,...,SET,PRIV);

/* FETCH DECODES THE OPERAND FIELD OF AN ICL STATEMENT */
FETCH: PROCEDURE(FIELD) RETURNS(PINTER);
DECLARE 1 FIELD UNALIGNED, /* =<PARN>,APPENDIX I */
2 TYPE BIT(2), /* DATA AREA INDEX */
2 VALUE BIT(*), /* OFFSET INTO AREA */
GO TO L(TYPE); /* DECODE TYPE FIELD */

L(0): /* NULL OPERAND -- RETURN RDF FOR 0 */
RETURN(ADDR(RQ_RFD));

L(1): /* OPERAND IS IN TRANSLATOR'S STACK */
RETURN(ADDR(STACK(J-VALUE))); /* INDEX FROM TOP */

L(2): /* OPERAND IS IN GLOBAL WORKAREA */
RETURN(ADDR(GLOBAL_WORKAREA(VALUE)));

L(3): /* OPERAND REFERENCED THROUGH CODE PARAMETER */
RETURN(PARGS(VALUE));
END FETCH;

/* DECODE DISP RETURNS VALUE ENCODED IN ICL DISP FIELD */
DECODE_DISP: PROCEDURE RETURNS(BIT(32));

IF OPND(I).DISP.TYPE=0 THEN /* CONSTANT DISPLACEMENT FIELD */
RETURN(OPND(I).DISP.VALUE);
ELSE /* DISPLACEMENT IS IN A VALUE_DESCRIPTOR */
RETURN(FETCH(OPND(I).DISP)->SFD.VALUE);
END DECODE_DISP;

Figure 3.12 A Control Module For Interpreting ICL
SECTION 3.4 119

FETCH_LOOP: /* LOOK AT INSTRUCTIONS IN SEQUENCE UNTIL EXIT */
    DO WHILE(OP=EXIT);
        SAVE_OP=OP;

DECODE_LOOP: /* DECODE F'CH OPERAND IN STATEMENT TO THE */
    /* ADDRESS OF A DESCRIPTOR */
    DO I=1 TO NUMBER_OF_OPERANDS(OP);
        /* IF ADDRESS MODIFICATION HAS BEEN SPECIFIED */
        /* THE OPERAND DESCRIPTOR IS JUST THE BASE FIELD */
        OPERAND(I)=FETCH(OPND(I).BASE);
        IF OPND(I).INDIRECT THEN DO;
            /* INDIRECT ADDRESSING HAS BEEN SPECIFIED */
            /* CONSTRUCT DESCRIPTOR FOR THIS ADDRESS */
            ALLOCATE SFD IN (DESCRRIPTOR_AREA) SET(D);
            D->ATTRIBUTES=OPERAND(I)->ATTRIBUTES;
            D->SFD.BASE=OPERAND(I);
            D->SFD.INDEX=FETCH(OPND(I).INDX);
            D->SFD.DISP=DECODE_DISP;
            OPERAND(I)=D; /* NEW DESCRIPTOR WILL BE USED */
        END;
    ELSE IF STRING(OPND(I).DISP) ≡ OPND(I).INDX.TYPE THEN DO;
        /* DIRECT MODIFICATION(S) SPECIFIED */
        /* CONSTRUCT DESCRIPTOR FOR MODIFIED ADDRESS */
        ALLOCATE SFD IN (DESCRRIPTOR_AREA) SET(D);
        D->ATTRIBUTES=OPERAND(I)->ATTRIBUTES;
        D->SFD.BASE=OPERAND(I)->SFD.BASE;
        D->SFD.DISP=OPERAND(I)->SFD.DISP+DECODE_DISP;
        D->SFD.INDEX=FETCH(OPND(I).INDX);
        OPERAND(I)=D; /* NEW DESCRIPTOR WILL BE USED */
        END;
    END DECODE_LOOP;
    T=ADDR(OPND(I)); /* POINT TO NEXT ICL STATEMENT */
    /* T WILL BE CHANGED IF AN ICL BRANCH IS TAKEN */

    /* SELECT STATEMENT PROCESSOR AND ENTER IT */
    CALL STATEMENT_PROCESSOR(SAVE_OF) (OPERAND);

    END FETCH_LOOP;
    END CONTROL_MODULE;

Figure 3.12 (con'd) A Control Module For Interpreting ICL
associated with the statement. A PL/I procedure for a control module for the coder for the ICL statements of Appendix I is given in Figure 3.12. Figures 3.6 and 3.7 are typical of ICL statement processors. Each statement processor calls on selected utility modules to help it perform its task. Except for instruction generation, which is distributed over many statement processors, there is a utility module for each major function listed above.

The areas of descriptor management and address computation have already been discussed in detail. The remaining areas will be discussed briefly.

Register Management

In any implementation, some subset of the available CPU registers will be reserved or dedicated to a specific function -- e.g., data area base, transfer vector (pseudo-register vector) base or subroutine communication. The rest of the registers are used where possible for arithmetic or logical accumulators, index registers, or scratch registers for address computation. In the model, only this later set of registers is managed by the register manager. The dedicated registers are controlled by the translator.

The managed registers are partitioned into groups of registers with the same physical characteristics -- e.g., general purpose and floating point registers on the IBM 360 computer. Associated with each group is a group status
descriptor (GSD). The RSD's of registers in the group are linked to the GSD on one of several class chains (using the FORWARD_LINK and BACKWARD_LINK fields of the RSD (Figure 3.5)). Which class chain an RSD is on depends on how the register it describes is being used. Figure 3.5 gives a set of use classes adequate for most applications on the 360. Gries gives two other classifications [1969; 1971].

The basic function of the register manager is the allocation of registers. The attitude toward register allocation taken here is to give the templates complete freedom in requesting and releasing registers. The only restriction on this is that the number of registers needed at any point of any template must not be greater than the number of registers being managed by the coder at the time. The number of registers needed includes those used explicitly by the template plus any others required for address computation. The register allocator must be able to satisfy all requests within this restriction.

Assuming that all registers in a group are in fact equivalent in every way, the crux of the allocation algorithm is deciding, when no unused registers are left, which one of the registers in use is to have its contents moved into storage and reallocated to satisfy the new request.

In a two-pass coder a scheme which uses the optimal page replacement strategy of Belady [1966] can be implemented to produce register allocations in a given straight-line sequence of code. The allocation will be optimal for the given
sequence of references, but there may be a better allocation if the sequence of references can be rearranged. This may also be implemented in a one-pass scheme with elaborate "back-and-patch" techniques. Day [1970] discusses another multi-pass register allocation technique. Short of this, a one-pass register reallocation algorithm must rely on heuristics -- and normal behavior -- to select a register. Gries gives a such an algorithm based on his register classification scheme [1971].

The register classification and reallocation algorithms are not critical components of this coder model. In PL/C, for example, there are only two register classes -- empty and in-use. The register allocator uses perhaps the simplest reallocation algorithm. It always reallocates the register which has been on the in-use chain for the longest time. With two or three accumulators available at all times, the frequency of register reallocation is very low in the average mix of programs which it compiles.

More elegant schemes for register management "on-the-fly" have been and will continue to be devised. These can be easily installed in the coder as defined here. However, the improved schemes will probably not produce a significant improvement in the performance of the object code.
Storage Management

The storage manager is another self-contained utility module of the coder. The algorithms used within it to allocate space within an area are not important to this model and so will be discussed only briefly.

Associated with each storage area is a descriptor. The address portion of the storage area descriptor (SAD) defines in one of the three possible formats the first location in the area. Like any other descriptor, the SAD may be manipulated by the translator or coder to reflect the addressability of the storage area. The SAD is specified in the BASE field of any descriptor which represents an element allocated within the storage area.

Storage areas can be classified according to the allocation scheme used and whether or not the elements are initialized by the compiler. The types of storage areas present in most compilers can be readily identified: code area, constant data area, and variable data area.

The code area is allocated sequentially, space is never reallocated and is substantially initialized with the object code instructions. To maintain such a storage area a simple location counter is required to keep track of the next available storage location. This counter could easily be kept in the VALUE field of the SAD. Only the instructions in the code area which are labeled need a descriptor. The descriptor, of course, completely defines the label and may be
manipulated like any other program element.

The translator may use more than one code area, but only one may be filled by the coder at any one time. An ICL statement is used to identify to the coder which SAD is to be used as the code area.

The constant data area is much like the code area in that space within it is allocated basically sequentially, never deallocated and completely initialized. In fact, constants are allocated in the same storage area as code in some compilers. The constant storage area differs from code areas in two ways:

1) almost all elements are represented via descriptors;

2) each request to enter a new element into the area may not actually cause new space to be allocated. If the constant is already in the area another copy does not have to be generated.

To keep track of the contents of the constant data area an additional data structure will have to be associated with the area's SAD. Alignment problems may also complicate the allocation scheme. Data areas are created and entries made in them through explicit ICL statements. The address computer in Figure 3.6 could have generated a constant for oversized displacements. In that case it would have to use a constant data area which would have to be designated to the coder via the appropriate ICL statement. If no such area were available, the necessary constant could be generated in-line on most machines as in Figure 3.6.
Variable data areas differ from code areas and constant data areas in many ways, but primarily in that the elements are not initialized. The variable data area is used to define the layout of an area which is to be created and initialized at run time. The activation record in most block-structured language implementations is a good example of a variable data area.

A very important variable data area is the temporary storage area. The temporary storage area is used primarily by the register manager as a save area for the contents of registers which must be reallocated but may also be used by the templates themselves for similar purposes — save areas, argument lists, etc. Elements of the temporary storage area are short-lived and usually used only once after their creation. Only if use information from a global optimizer is available, can a temporary be used more than once. To make optimum use of the temporary storage area, the allocation scheme should be able to reclaim elements which are no longer needed so they may be reallocated.

A feature of the temporary storage area which is required by the register manager and convenient for other users is that the temporary storage area be immediately addressable. If the temporary storage area is not immediately addressable, then attempts to save the contents of a register which is to be reallocated may require a register to compute the address — out, of course there is no register available.
The coder must be concerned with the physical representation of elements in a storage area at compile time only if the storage area element is initialized. If the compiler is load-and-go then the initialized storage areas will be assigned space within the compiler and filled with the proper bit patterns as needed. In other cases the initialized portion of the program will be written onto an auxiliary file in the form of an object module.

The two schemes are compatible up to the point of initialization as long as the object module has sufficient structure to accept information in any order relative to its physical order in storage at execution time. This feature is necessary to implement the "back-and-patch" schemes used to handle forward references in the object code.

Object Machine Instruction Generation

ICL statements which generate code come in four levels of complexity called basic, macro, generic and conversion.

**Basic level generation statements:** At this level any machine instruction may be generated by supplying the operation code and value descriptors for the operands of the instruction. The addressability of the operands must comply with the requirements of the instruction. That is, if the instruction format requires that an operand be in a register then the corresponding descriptor must be in register format. Descriptors which correspond to storage references must be in
either storage or address format. They need not be immediately addressable. Additional code will be generated to make them addressable if they are not already. Also, the attributes of the operands must be or are assumed to be compatible with the requirements of the instruction and with each other. That is, no conversion of data types is attempted at this level. At the basic level the code does little more than piece together the instruction from the parts specified in the ICL statement.

At this level heavy use will be made of the ICL statements which test the compile-time and run-time environments and control the sequencing of the interpretation of the template. The tests are used to insure that the operands are compatible before using a basic level instruction generation statement. If they are not compatible, additional ICL statements will be used to generate code to convert them. Also at this level, registers -- except those used for addresses -- and temporary storage -- except that used by the register manager -- must be explicitly allocated and freed using the appropriate ICL statements.

Macro level instruction generation statements: At this level of ICL instruction coding the more common sequences of basic level statements are reduced to primitive ICL statements. In particular, the coder will generate code to make the addressability of the operand conform to the requirements of the instruction. That is, if an instruction requires an operand to be in a register and a storage
formatted descriptor is supplied, the coder will automatically allocate a register of the proper type and then generate code to load the operand value into the register. If the operands are in temporary storage or registers these resources will automatically be released after the last reference to the operand. Instruction and operand data types are presumed compatible as in the basic level statements.

Generic level instruction generation statements: The need to specify an explicit operation code in the ICL statement is dropped at this level of coding. Instead, a generic operation such as ADD, SUB, SHIFT-LEFT, or DIVIDE is specified. The coder will determine which machine operation code to use based on the type of the operands supplied and their addressability. Both operands must still be compatible. Conversion of data types is not generated automatically. A generic instruction generation statement would be used where the machine has many data types on which the same basic operation may be performed. On the IBM System/360 computer, for example, addition may be performed on 16-bit or 32-bit integers, 32-bit or 64-bit floating point numbers and variable length decimal values. The integer and floating point operations may be performed on data in two registers or in a register and in storage.

Conversion level instruction generation statements: This is the highest level of ICL coding complexity. In addition to performing the functions of the generic level, the coder will also generate code to convert operands to compatible types before determining which operation code to use.
The statements at the basic level are all that are needed for a complete coder. The other levels could be written as macros which expand to basic level statements or could be implemented as ICL subroutines to be called from templates which need them. The subroutine would be a template written in the basic level ICL statements. The macro, generic and conversion level ICL statements are used for the convenience of the ICL programmer and the efficiency of the coding operation.

Forward References

A problem that must be faced by the coder is what to do about undefined addresses. This situation can not be avoided in a one-pass coder. The situation often arises in which it is necessary to generate a forward branch to a location further along in the code sequence. The addressing code for an undefined reference cannot be generated until the location is defined. So, in the one-pass code generator, space must be left in the object code area at the time the referencing instruction is generated. When the location of the undefined address becomes known, the code area must be patched with the proper addressing code. This is a real problem that has no firm solution.

Having to back up in the output stream to fill in the unfinished instruction when the reference is finally defined is not a problem. Whether code is being placed where it will
be executed or being written into an object file, techniques are available for backing up and filling in the missing information. [Gries 1971]

The problem is that the coder does not know how much room, if any, will be needed for the addressing code. In the majority of applications, when the reference is finally defined it will be immediately addressable, so one "solution" is to allow only those undefined references which will resolve to immediately addressable locations. In this case, just the incomplete address field has to be filled in. If this remedy is used and the reference turns out not to be immediately addressable, the coder has two options. The compilation may be discarded with a plea to rearrange statements to bring the reference within reach. RCPL/360 [Kelly 1970] is a compiler which uses this approach. Alternatively, if the forward reference occurs in a branch instruction, then a branch to an immediately addressable area may be substituted and in that area the addressing code can be generated to complete the branch. This is the technique used by Gries in CII [1969]. These solutions have the advantage that for the main stream of programs, the most efficient code will be generated.

If the undefined reference must be implemented at all costs and an immediately addressable area cannot be guaranteed, then two additional options are available. If the longest possible descriptor chain is known, then sufficient space to implement that reference can be allocated. What is not actually used can be filled with no-operation
instructions. Alternatively, a special, fixed length scheme, guaranteed to compute the address of any location in the machine could be adopted. The such scheme would entail making the address of the desired location a constant in the code area -- with code generated to branch around it -- and generating an indirect reference through that code location.

3.5 Summary

This completes the definition and discussion of this model for a code generator. The technique of coding by interpreting templates, used to some extent in several existing compilers, has been expanded to include all of the processes associated with the functional dimension of coding. The representational dimension is incorporated in the descriptors used to define program elements to the coder. The use of machine data types in these descriptors and the introduction of a powerful and uniform encoding of access functions make the descriptor a unifying force in the coder.

These descriptors provide a common interface between the functional coding processes -- instruction generation, register management, and storage management. As a result any number of algorithms may be used interchangeably in these processes to suit the aims of the compiler implementation. Only the interfacing of these algorithms has been discussed. Sample algorithms have been included for completeness, but there is no claim as to their universality or optimality.
CHAPTER 4. OPTIMIZATION

Optimization is the process of rearranging and changing operations in the program being compiled in order to produce a more efficient object program. Of any of the areas of code generation the area of code optimization has received the most attention in the literature. Algorithms for optimizing various aspects are discussed by Lowry and Medlock [Lowry 1969], J. Cocke and J. Schwartz [Cocke 1970], Day [1970], and others. Cheatham and Standish [Cheatham 1970] summarize a wide variety of optimization techniques used and Gries [1971] references other sources.

The intent of this chapter is not to expand on this vast store of information. Rather, the methods by which these techniques can be incorporated into the present model are discussed. First, techniques which are applicable within the one-pass structure of the code generator are discussed. In the last sections, a model for a multi-pass code generator is developed which is capable of handling the remaining optimization strategies.

4.1 Optimal SLM Sequences

In this thesis, code generation has been defined as the process of realizing a mapping, C, from the abstract program tree (APT) to a functionally equivalent program in the object machine (OM). The realization of C presented here decomposes
into the realization of the semantic map, \( S \), from the APT to an equivalent program in a source language machine (SLM) and the realization of the implementation map, \( I \), from the SLM to the OM.

Because of its full tree-walking abilities, the translator defined in chapter two is capable of realizing any practical semantic mapping. Since SLM optimization is really nothing more than defining a clever semantic mapping, the translator model is capable of generating an optimal sequence of SLM instructions.

The cost of optimizing the SLM instruction sequences is a generally more complex translator. Operators which are postfix-translatable if just any code will do may require prefix or tree translation to produce optimum code. One-state prefix-translatable operators may become multi-state translatable. Even simple postfix-translatable expression operators may become tree-translatable or n-yass translatable if the SLM code is to be optimized.

Here are a few examples --

1) **Postfix-translatable operators become prefix-translatable**: Integer multiplication and division can usually be thought of as postfix-translatable operators, but on the IBM System/360 computers they must be translated in prefix format if the best code sequence is to be generated in all cases. On this machine the integer dividend must occupy the odd numbered register of an even-odd pair and the two's complement sign must be propagated into the even register. If
the dividend is the result of a multiplication, it will already be in the proper format. In any other case the dividend should be computed in the even register so that a single shift will propagate the sign as the value is moved into the odd register. As an example, two expressions and the optimal coding for them are given in Figure 4.1. In the postfix format the coder will not know which register is best for computing A+B because the translator has not seen the division or multiplication operator yet. In prefix, the division operator could set a translator state that would request the dividend be placed in an even register. If present, the multiplication operator would change the state to request that the operand be computed in an odd register.

```
L R1,A
A R1,B
M R0,C
D R0,D
```

```
L R0,A
A R0,B
A R0,C
SDBA R0,J2
D R0,D
```

<table>
<thead>
<tr>
<th>a) Optimal 360 code sequence</th>
<th>b) Optimal 360 code sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>for ((A+B)*C/D)</td>
<td>for (A+B+C)/D</td>
</tr>
</tbody>
</table>

Figure 4.1 Optimal 360 Multiply and Divide

2) One-state prefix-translatable becomes multi-state translatable: In chapter two, the PL/I IF-statement was introduced as a one-state prefix-translatable operator. It may be possible to determine at compile time whether the relation in the IF-statement is always going to be true or always false. If that is the case, then the THEN- or ELSE-
clause should be ignored entirely. To do this in the prefix format a new state is required -- "skip to next operand."

3) **Postfix-translatable** operators become **tree-translatable**: The standard optimizing technique of folding, when performed on APT, requires a walk over the entire expression tree to determine what constants are present and which operations can be performed at compile time. For example, because of the associativity of addition, before any code is generated for the expression $6 + 8 + 2$, all three leaves of the expression tree must be visited to find the 6 and 2. Then code to compute $8 + 2$ can be generated.

So, although any semantic map may be realized within the framework of the present translator model, the price of increased translator structure may be too great. Alternative schemes are discussed in Section 4.5.

Having argued that optimal SLM instruction sequences are producible by the translator model, can it be concluded that optimal object machine instruction sequences are possible? Technically the answer is "yes." There is nothing preventing the SLM from being identical to the object machine, except for the overwhelming complexity of the translator in that case. So, it only for reasons of clarity and manageability there is justification for keeping some "distance" between the SLM and the object machine. Therefore before any practical answer can be given, the capability of the code to realize arbitrary implementations must be analysed.
4.2 Piecewise Coding Constraint

Unfortunately, the code defined in chapter three cannot realize all implementations. Let \( S(\text{APT}) = \)

\[
\begin{array}{c}
0 (a, \ldots, a) \\
1 \\
1^n \end{array}
\]

where 0 is in SLM operator and the a's are its operands, then the implementation, \( I(E, S(\text{APT})) \), must be definable in the form

\[
I'(E, 0, a, \ldots, a), \ldots, I'(E, 0, a, \ldots, a)
\]

where \( I' \) is a restriction of the implementation map and \( i \) in a compile-time description of the run-time environment after executing operator 0 in the environment described by \( E \).

This piecewise coding constraint is imposed by the one-pass structure of the present code generator model.

In general, better code requires more contextual information. After all, optimizing an operation is just the process of recognizing the existence of a special situation and then generating code to perform the operation which is just adequate for the situation and more efficient than that produced for the most general case. For example, shifting left on a binary computer is more efficient for multiplying by a power of two than the machine's normal multiply instruction. So multiplication can be optimized by generating a "shift left \( N \)" when it can be determined that one of the multipliers is \( N \). This substitution is available only if the code has
information about the value of the operands of an operator in addition to the required type and location information.

Within the framework of piecewise coding the coder obtains contextual information from three sources: 1) the operand descriptor, 2) global information such as the translator state or run-time environment description, and 3) the name of the SLM operator in the form of a template. The information contained in these structures can be increased as needed to aid the coder in producing more efficient code.

For example, the descriptors can be expanded to include the value of constants and, if known, variables. (This is the purpose of the VALUE fields in Figures 3.5 and 3.6.) Associative unary operations such as negation or complementation can be deferred to the last moment by noting in the descriptor that the operation is to be performed. Unnecessary branches and procedure exit coding can be eliminated by keeping track of the reachability of code segments through a flag in the global workarea. Code following an unconditional branch or procedure exit can not be executed (reached) unless the statement following it is labeled.

4.3 Selection of the Proper SLM

The efficiency of the code generator model and the object code it produces depends largely on the selection of the proper SLM instruction set. A complex SLM instruction conveys
generally more contextual information to the coder than if that SLM instruction were broken into a sequence of simpler SLM instructions. That is, if $O$ is an SLM instruction which is equivalent to the sequence $O^1 O^2 O^3$ then $I(E,O)$ can potentially be better code than $I'(E,O^1)I'(E^1,O^2)I'(E^2,O^3)$ because the coding of $O$ may be able to use the fact that $O^1$ will be followed by $O^2$ and then $O^3$ -- an assumption that can not be made when coding $O^1$ separately.

An Example of the Effect of SLM Definition

In chapter one, the DO-statements of Figure 1.3 were defined by the sequence of SLM instructions given in Figure 1.6. With the aim of improving the performance of the DO-statement implementation and at the cost of increasing the size of the stack element used by the DO-statement driver, the translator could produce the SLM sequence in Figure 4.2. In order to generate this code the translator will have to save the descriptors for $T1$, $I$, $L2$, and $L1$ on the compile time stack until it is time to generate the epilogue code. From this the coder could generate for the IBM System/360 the code in Figure 4.3.

If, however, the SLM includes the instruction

DO <Index>,<Start-expr>,<TO-value>,<BY-value>

to replace the initial sequence and ENDDO to replace the epilogue, then the coder can produce the code in Figure 4.4.
There are, of course, trade-offs in the selection of the proper SLM instructions. The more complex instructions can produce better code, yet require more template space for their definition. Since the templates are interpreted, the time taken in the coder to do the special casing that could be done in the translator will be greater than if programmed into the translator. On the other hand, the special casing done in the translator would require more code in the compiler because the
algorithms are not compacted into templates or tables.

The freedom in selecting the SLM instructions is limited both theoretically and practically speaking. On the high end of the complexity scale the only restrictions on the SLM are that it be sequentially organized and that all instruction operands be scalar values. At the low end of the complexity scale, for practical reasons, the SLM instructions should be at least no simpler than the object machine instructions. The piecewise coding constraint makes it difficult for the coder to synthesize machine instructions from several SLM instructions. The translator could have generated, for example, instruction by instruction, the SLM (OM) sequences in Figure 4.4. In this case, of course, the coder has very little to do but piece together the various parts of the OM instructions. Only the basic level instruction generators are needed in the ICL.

![Figure 4.4 360 Code For DO Using More Complex SLM](image)
As a general rule of thumb, optimization involving unevaluated operands and the reordering of operators — such as folding or common subexpression elimination — should be done in the translator and optimization involving the manipulation of single-valued operands should be performed by the coder. In chapter five additional reasons will be given to support an SLM that is some distance from the object machine.

4.4 Local Optimization in the Coder

Within the framework of the piecewise coding constraint an important topic for discussion is the local optimization of SLM instructions. **Local optimization** is defined here as the process of generating the shortest and/or fastest sequence of machine code which will implement an SLM instruction based solely on the semantics of the SLM instruction, the types and access functions of its operands, and the run-time environment description. Consideration is not given to coding algorithms which through the inspection of several SLM instructions will, for example, maintain in registers operands and base addresses which are common to several SLM operations or operands, nor select registers or object instructions based on the requirements of later SLM instructions. However, whenever possible nothing will be done which would preclude these desireable circumstances from happening by chance.
In the model, local optimization can take place on two levels in the coding process -- in programming the ICL template or in the processing of an ICL statement. The conditional branch instructions are included in the ICL just to aid the template programmer in selecting an efficient code sequence. To use these ICL facilities most effectively, the philosophy of latest binding should be followed -- assume nothing about the location or type of operands and leave all result values where they are computed. Adhering to this philosophy will expand the templates because of all the test ICL statements which must be added to determine where the operands are located and what the best code will be, but the code generated can be the best possible based on this local information. Sometimes the different possible locations of an operand will determine entirely different code sequences for an operator. For others, the code sequences may differ only in the format of the instructions used. In either case, the optimization is accurately documented in the code template.

As an example, consider the template for the PL/I MOD built-in function in Figure 3.3. Note how it uses the conditional branch instructions to determine whether or not its operands are in registers and then generates either RR or RX instructions based on this information. (The templates in Figures 3.2, 4.6, and 4.8 also adhere to this philosophy.)

Local optimization can also be built into the ICL statement processors inside the interpretive coder. The manipulation of descriptor address chains performed by the
address computer to minimize base register loads is a good example of this form of local optimization. In a way the automatic features of the instruction generation statements at the macro, generic and conversion levels are also forms of internal local optimization, but can also be interpreted as just a more efficient implementation of conditional template coding.

In the discussion of the different levels of instruction generation statements it was implied that the machine instruction generated was the one specified -- possibly only generically -- in the instruction generation statement. When the coder is optimizing, this may not be the case. Consider the GLA (Load Address) instruction generation statement as implemented in PL/C for the IBM System/360 computer.

Suppose the ICL statement is

```
GLA R,A.
```

If A defines a simple base-displacement address, D(B), with D non-zero, then the machine instruction

```
LA R,D(B)
```

is generated. If D is zero and B is a non-zero register, then the machine instruction

```
LB R,B
```

is generated. This has approximately the same effect as

```
LA R,O(B)
```

but is shorter and faster. If B is the same register as R then no code is generated. (Note that because of this optimization, GLA cannot be used to generate code to clear the
high order byte of a register. A GRX LA,R,A must be used instead.)

Similarly, if D is zero and B is register zero, then

\[ \text{SR R,R} \]

is generated, which computes a zero in R faster than

\[ \text{LA R,0(R0)} . \]

(But unlike the LA, the SR instruction sets the condition code.)

Finally, if D is zero and B is not a register, but some core reference, the effect of

\[ \text{LA R,0(B)} \]

is implemented by a

\[ \text{GRX L,R,B} . \]

Other ICL statements could also be optimized in this manner. Multiplication or division by a power of two could be generated as shift instructions. A

\[ \text{SLA R,1} \]

could be coded as

\[ \text{AR R,R} . \]

Whether an ICL statement is optimized depends on its frequency of use.

4.5 Implementing Global Optimization Strategies

Global optimization algorithms differ from local strategies in that they require a general knowledge of an entire program -- or at least a large block of SLM code --
before deciding on the best solution. Invariably, global strategies manipulate the structures of the source language -- either in the APT representation or more popularly in the SLM representation. Optimization of object code is usually limited to local strategies as discussed in the previous sections, although global strategies for optimal register assignment is a topic of current interest [Day 1970].

As long as global optimization strategies can be applied to either the APT or object code, then the one-pass model presented so far may be used intact -- the optimization being performed in separate passes on either side of the translation-coding pass. It, however, the optimization algorithm must work on the SLM, then the translator and coder must be separated and the SLM instructions laid down in an intermediate text to be reread by the coder after it has been processed by the optimization pass. (See dashed box in Figure 1.1a.)

Unfortunately, the majority of optimization algorithms do work on the SLM. Hopgood [1968] discusses some techniques based on a tree representation of the program. The other algorithms in the literature could probably be adapted to work on the APT, but because of the ability of modern languages to represent complex expressions compactly, much of the work in the adaptation would be exploding the language constructs into simpler, more manageable ones. This has been the task of the translator exclusively so far, and in keeping with the tenor of this thesis so it should remain.
Multiple-pass Code Generation

What then are the problems, if any, connected with separating the one-pass model into several passes — translator, optimizer(s), coder, and possibly more optimizers? The diagram in Figure 4.5 illustrates the structure of the one-pass code generation model and the flow of information through it. The dashed line indicates the boundary between the logical phases of translation and coding. When the code generator is separated into several passes, any information crossing from the translator to the coder must be written into the translated intermediate test or intermediate code (IC). The IC must include at least the SLM instructions generated by the translator.

As the code generator is constructed, information passes only from the translator to the coder. There is no feedback from the coder. The translator is unaffected by anything the coder does. Furthermore, the ICL has been designed so that anything the translator may want to do to control the coding of an SLM instruction — e.g., set a bit to signal a new state or change a descriptor to update the environment description — can be done by the coder in response to some ICL statement.

In the one-pass scheme it is often convenient to circumvent this particular capability for reasons of efficiency. There is no use in invoking the overhead of the coder just to set a single bit in the global workarea. In the multi-pass scheme, however, the coder must be informed —
through the appropriate ICL statements -- of every change to the compile-time environment which might affect the coding.

If this is done, it does not matter, theoretically, when the coder is called upon to code an SLM instruction. It could be as each one is generated -- the one-pass scheme -- or after they all have been generated -- the multi-pass objective. Here the term "SLM instruction" corresponds exactly to a call to the coder in the one-pass scheme. It does not matter
whether this call is to generate code or just set up a new state description. This definition may add instructions to the SLM which might normally not be considered machine instructions.

Some practical problems must be considered before the multi-pass scheme can be dismissed. Although viewed as simple machine-like instructions, in the one-pass scheme the SLM instructions are in fact rather complex structures. The SLM operation codes correspond to code templates. The SLM instruction operands are principally descriptors, but these are scattered throughout the code generator: primarily on the translator stack, but also in the global workarea and other places designated only by addresses passed from the translator to the coder.

Each code template is constant and therefore presents no problems. The templates can be moved to the coding pass with only a few minor changes. For each SLM instruction the template to be interpreted is indicated by a constant code -- the SLM instruction's IC operation code.

SLM Instruction Operands - Template Parameters

The operands of an SLM instruction in the IC correspond to the parameters of the template associated with that instruction. The one-pass model allows three types of parameters -- stack references, global workarea references and indirect references to locations in the stack or workarea
through argument pointers passed to the coder by the translator (called pointer references in the sequel). A new mechanism will have to be introduced to handle these in the multi-pass version of the coder.

Most template parameters reference a descriptor. Unfortunately, the descriptor is too large to write into the IC in its entirety with any economy. In addition, the descriptor contains pointers which may lose their meaning if moved. Therefore, some means must be devised for encoding in the IC the operand descriptor so that it may be reconstructed by the coder.

Considering that even in the one-pass code generator, descriptors could be created, modified, and destroyed solely by the coder, this is a relatively easy task. Basically, each time the translator wishes a new descriptor created, it generates a unique name -- e.g., a serial number -- for it. When the descriptor is to be used in an SLM instruction, only the name of the descriptor goes into the IC.

The descriptors for source identifiers deserve special handling. In the one-pass code generator, when an addressable source identifier is scanned in the IT, a descriptor for the identifier is constructed and placed on the translator stack. This descriptor is not referenced further until the SLM instruction in which it is used is generated. In the multi-pass scheme, if the procedure outlined above is followed, the identifier would have to be assigned a name and an SLM instruction would have to be generated which would have the
coder create a descriptor for it. This SLM would have to indicate the symbol table entry from which to obtain the access function and the values of variable attributes such as the length of a string. Since the generation of the descriptor is based entirely on the symbol table entry and the translator state -- two items already known by the coder -- this operation can be performed in the same pass with the coder. The pointer to the identifier's symbol table entry can serve as its name.

Moving the formation of identifier descriptors to the coding pass reduces the size of the IC by eliminating the SLM instructions which would have been emitted when the identifier was scanned by the translator. It also transforms the IC into the traditional n-tuple internal representation. In its greatest generality an SLM instruction in the IC has the following canonical form:

\[
<\text{SLM-Op-code}> \quad (<\text{Output-name}>\} \quad (<\text{Input-name}>\}).
\]

Descriptors with translator assigned names -- primarily intermediate results -- will first appear as an \(<\text{Output-name}>\).

Since the bulk of the descriptor defined for the one-pass code generator (Figure 3.5) is used exclusively by the coder, much of that descriptor does not have to be part of the descriptor used by the translator in the two-pass scheme. The translator's descriptor must record only the source attributes, the access function of non-addressable operands and the name to be used in the IC.
The coder must be modified to accept these descriptor names. (See Figure 4.9.) The handling of the names is done in the scanner module, which must be added anyway to scan the IC and from the SLM operation code select the proper template for the rest of the coder to work with. When a descriptor name is read by the IC scanner, it is associated with a descriptor in a descriptor area which has been added to the coder pass for this purpose. Some hashing technique would probably be used to make this association. The address of this descriptor is passed to the coder for processing. It is referenced through a pointer reference in the template.

If a descriptor has not been associated with the IC name, then a new descriptor must be created. If the name is a symbol table pointer, the scanner allocates the space for the descriptor in the descriptor area. The newly created descriptor is formatted and initialized from information in the symbol table.

Other descriptors are created in the descriptor area as the result of interpreting a CRED ("create descriptor") ICL statement. The argument to the CRED statement is the name of a descriptor. The scanner passes this name when it cannot find a descriptor associated with the name in the descriptor area. At the time it is created, the descriptor is formatted and initialized from information supplied in the CRED statement. Descriptors remain in the descriptor area until they are no longer needed or destroyed by a DESD ("destroy descriptor") ICL statement.
The mechanism dictates some changes in the code templates:

1) All stack references must be eliminated and replaced by pointer references to the appropriate SLM operand in the IC format.

2) All stack management ICL statements must be removed and replaced with ICL statements to explicitly create descriptors for results and destroy descriptors for operands which are no longer needed.

References to the global workarea do not have to be changed. A copy of this area is maintained in the coding pass.

4.6 Descriptor Management -- A Second Look

Note that the more general "random access" descriptor area has completely replaced the simple stack structure of the one-pass code generator. This is to be expected when global optimization is used. The crucial piece of information that the optimization pass can supply to the coder is the number of times a particular value is referenced by SLM instructions.

The number of uses is passed to the coder in the IC with the SLM instruction which computes the new value. This is the SLM instruction which causes the descriptor for the value to be created. The number is stored in the USES_LEFT field of the VALUE_DESCRIPTOR. Each time the descriptor is set up as an input argument to the coder, this field is decremented by
one. After coding the SLM in which the USES_LEFT field becomes zero, the descriptor for the value is destroyed. Note, identifier descriptors and intermediate result descriptors are handled the same in this respect. There is no reason to clutter up the descriptor area with identifier descriptors which are not being referenced.

Since descriptors are created and destroyed more or less at random the more general descriptor management algorithm is needed. This is in contrast to the unoptimized one-pass model where a new descriptor is created for each identifier reference in the IT and each result descriptor is destroyed immediately after its first use.

Since a new descriptor is not constructed for each reference to either an identifier or intermediate result care must be taken not to change the value defined by the descriptor. This care need not be exercised in the one-pass scheme. Since the descriptor was used in at most one template no harm arose from using the same descriptor for several different values.

Technically a descriptor defines a value and not a location that happens to have that value. This is evident in the fact that the description of the access function follows the value as it is moved from core to a register for example. If code is generated which would change the value of a location defined by a descriptor, that descriptor should be destroyed -- the value it represents no longer exists -- and a new one constructed to define the new value. Since the two
values have the same location and attributes, it would be
convenient if the old and new descriptor were in fact the same
one. In the one-pass code generator, this was the convention
adopted and was in most cases harmless. Where the semantics
of an operator required an operand to be used in more than one
SLM instruction -- such as the DO-statement operator -- the
translator would have to take special precautions to see to it
that the descriptor and corresponding value were protected
from modification.

<table>
<thead>
<tr>
<th>ADDB</th>
<th>BR</th>
<th>A,ADDDB1</th>
<th>-&gt;ADDDB1 If A is in a register</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BR</td>
<td>B,ADDDB2</td>
<td>-&gt;ADDDB2 If B is in a register</td>
</tr>
<tr>
<td></td>
<td>LGPR</td>
<td>A</td>
<td>Generate code to load A into</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>register</td>
</tr>
<tr>
<td>ADDB1</td>
<td>BR</td>
<td>B,ADDDB3</td>
<td>-&gt;ADDDB3 If B is in a register</td>
</tr>
<tr>
<td></td>
<td>GRI</td>
<td>A,A,B</td>
<td>Generate A+B</td>
</tr>
<tr>
<td>B</td>
<td>ADDB4</td>
<td></td>
<td>-&gt;Merge</td>
</tr>
<tr>
<td>ADDB3</td>
<td>GRR</td>
<td>A,A,B</td>
<td>Generate A+B</td>
</tr>
<tr>
<td>ADDB4</td>
<td>FREE</td>
<td>B</td>
<td>Release resources assigned to B</td>
</tr>
<tr>
<td>ADDB5</td>
<td>POP</td>
<td>1</td>
<td>Remove B descriptor from stack</td>
</tr>
<tr>
<td></td>
<td>EXIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDB2</td>
<td>GRI</td>
<td>A,B,A</td>
<td>Generate A+B</td>
</tr>
<tr>
<td></td>
<td>FREE</td>
<td>A</td>
<td>Release resources assigned to A</td>
</tr>
<tr>
<td></td>
<td>SET</td>
<td>A,B</td>
<td>A now designates result location</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>ADDB5</td>
<td>-&gt;Merge</td>
</tr>
</tbody>
</table>

Figure 4.6 Code Template for Binary Addition (one-pass)

In the multi-pass model, where any value may be
referenced in more than one SLM instruction using the same
descriptor, care must be taken in the template not to disturb
either the value or its descriptor. As an example, consider
the template in Figure 4.6 for the coding of binary addition
of the IBM System/360. This template is suitable for the one-
pass code generator, but not the multi-pass version.

Suppose the two SLM instructions

\[ \text{ADD}^T \ T, ID, ID \]
\[ \text{ADDB} \ T', ID, T \]

are to be coded. When the expansion of the templates begins ID is in storage at location D(B).

In the one-pass scheme, A and B would reference two different descriptors on the stack with identical contents. When the template is expanded for the first ADDB, the first two branch statements fail and so the LGPR is executed. This allocates a register, R say, and generates

\[ L \ R, D(, B) \]

and then links the descriptor for A to the RSD for R. The branch statement in line 4 (Figure 4.6) fails because the descriptor for B is still pointing to the core image of the identifier's value. So, line 5 generates

\[ A \ R, D(, B) \]

the attempt to free B fails (ID is permanently allocated), the top of the stack -- B's descriptor -- is erased, leaving on the top of the stack as the result descriptor, the former descriptor for A. Everything is correct. The second ADDB will generate

\[ A \ R, D(, B) \]

using this descriptor and a new descriptor for ID.

In the same situation in the multi-pass scheme, however, the outcome is disastrously different. This time in expanding the first ADDB, A and B are the same descriptor. As before
the branches in lines 1 and 2 fail and line 3 generates
\[ L \ R, D(B) \]
and links the descriptor for ID to the RSD for R. This time, since B references the same descriptor, the branch in line 4 is successful causing the GRR in line 7 to generate
\[ AR \ R, R. \]
This is a better sequence than the one-pass mechanism was able to produce. However, there is trouble. The first ADDDB leaves the descriptor for the identifier ID pointing to register R, but R does not contain the value of ID. When this descriptor is used in expanding the second ADDDB instruction, another
\[ AR \ R, R \]
will be generated producing an incorrect answer.

A new ICL statement, PRIV, is needed to correct this situation. PRIV is flowcharted in Figure 4.7. It does two things. It makes a private copy of the descriptor when necessary and also guarantees that the original value will not be destroyed. The effect is to make a local copy of the descriptor and so effect a call-by-value to replace the usual call-by-reference used in the template system. The private descriptor is destroyed after interpreting the template in which it is created.

In Figure 4.8, the template for ADDDB in Figure 4.6 has been modified to make use of the PRIV statement. It also has been updated for use in the multi-pass version of the coder.

Assume
\[ ADDDB T1, ID, ID \]
ADDB T², T¹, ID

is to be coded under the same conditions as earlier. Line 1 of the ADDB template in Figure 4.8 creates an empty descriptor named T¹. Lines 2 and 3 fall through since ID is not in a register. Lines 4 through 7 allocate a register (GGPR), generate

\[ L \ R, D(), B \]

(GRX) and then point the descriptor for ID to the RSD for R (LINK). (This is the equivalent of LGPR A.)

---

**Figure 4.7 Flowchart for PRIV ICL Statement**

(General Purpose Registers)

---

The PRIV ICL statement is now executed (see Figure 4.7). There is one use of ID left in the current sequence -- the two
references in the current SLH instruction having been subtracted from the total -- and it is currently in a register. So a new descriptor is created and assigned the attributes of ID (box a). The register manager is requested to allocate a register (box b). The register allocated will be the "best" choice based on the current usage and any future usage indicated by the optimizer.

<table>
<thead>
<tr>
<th>ADDB</th>
<th>T, A, B</th>
<th>(IC instruction format)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRED</td>
<td>T</td>
<td>Create result descriptor 1</td>
</tr>
<tr>
<td>BR</td>
<td>A, ADDB1</td>
<td>--&gt;ADD81 If A is in a register 2</td>
</tr>
<tr>
<td>BR</td>
<td>B, ADDB2</td>
<td>--&gt;ADD82 If B is in a register 3</td>
</tr>
<tr>
<td>GGPR</td>
<td>T</td>
<td>Get general purpose register for result 4</td>
</tr>
<tr>
<td>FREE</td>
<td>A</td>
<td>Release resources assigned to A 5</td>
</tr>
<tr>
<td>GRX</td>
<td>L, T, A</td>
<td>Generate code to load A into T 6</td>
</tr>
<tr>
<td>LINK</td>
<td>A, T</td>
<td>Point A to register T 7</td>
</tr>
<tr>
<td>ADDB1</td>
<td>PRIV A</td>
<td>Get private copy of A if needed 8</td>
</tr>
<tr>
<td>BR</td>
<td>B, ADDB3</td>
<td>--&gt;ADD83 If B is in a register 9</td>
</tr>
<tr>
<td>GRX</td>
<td>A, A, B</td>
<td>Generate A+B 10</td>
</tr>
<tr>
<td>B</td>
<td>ADDB4</td>
<td>--&gt;merge. 11</td>
</tr>
<tr>
<td>ADDB3</td>
<td>GRR A, A, B</td>
<td>Generate A+B 12</td>
</tr>
<tr>
<td>ADDB4</td>
<td>SRC A, 3</td>
<td>After addition, register value is not known 13</td>
</tr>
<tr>
<td>FREE</td>
<td>B</td>
<td>Release resources assigned to B 14</td>
</tr>
<tr>
<td>SET</td>
<td>T, A</td>
<td>Set result descriptor 15</td>
</tr>
<tr>
<td>EXIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDB2</td>
<td>PRIV B</td>
<td>Get private copy of B if needed 16</td>
</tr>
<tr>
<td>GRX</td>
<td>A, B, A</td>
<td>Generate B+A 17</td>
</tr>
<tr>
<td>FREE</td>
<td>A</td>
<td>Release resources assigned to A 18</td>
</tr>
<tr>
<td>SET</td>
<td>T, B</td>
<td>Set result descriptor 19</td>
</tr>
<tr>
<td>SRC</td>
<td>T, 3</td>
<td>Indicate register contents unknown 20</td>
</tr>
<tr>
<td>EXIT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.8 Code Template for Binary Addition (Multi-pass)

If R is not the best choice, the register allocated will be different, say S. So the test in box c fails and as a
result the PRIV statement generates

LR S,R

and an RPD for S is returned (boxes d and e). Future references to A in the template will reference this new descriptor. This is the local copy of ID's descriptor. The descriptor for ID will remain an RPD for R. The branch in line 9 of Figure 4.8 will be successful since B -- still pointing to the descriptor for ID -- indicates the value of ID is in register R. So,

AR S,R

is generated and after finishing the first ADDB, T' will indicate the result value is in S.

The expansion of the second ADDB will proceed as before. The PRIV statement will be inoperative since there are no references to ID left. Consequently another

AR S,R

is generated. The sequence is optimal.

Returning to the PRIV statement in the first ADDB expansion, if R must be reallocated, then normally the access function field of the descriptor for R's content -- in this case the descriptor for ID -- would be reset to its original value -- D(B). Since the value of ID is actually still in the register, it is better to delay this change until an instruction is generated which will destroy the value in the register. Consequently, flags are set in the RSD for R to indicate that the change should be made. The SNC in line 13 of the ADDB template will trigger the change.
With this arrangement, the test in box c of Figure 4.7 succeeds and so no code is generated by PBIIV. The branch in line 9 will again be successful and

AR R,R

will be generated. The SRC in line 13 will cause the descriptor for ID to be reset so that the core location of the value -- the only valid copy now -- is once again referenced. In this case the second ADDB will generate

A R,D(R,B).

The sequence is optimal assuming that the reallocation of R was the best choice.

4.7 Summary of Multi-Pass Modifications

Figure 4.9 diagrams the structure of the code generator when an optimization pass is added between translation and coding.

In summary, to convert the translator for use in the multi-pass code generator:

1) Remove coder's information from descriptor's and global workarea.

2) Add emitter to format SLM instructions in the IC and manage stack -- previously functions of the coder.

These modifications are needed whether optimization is to be performed or not.

To convert the coder:
Figure 4.9 Information Flow in Multi-pass Code Generator.
1) Add SCANNER module to:
   a) decode IC SLM instructions,
   b) allocate and destroy descriptors, and
   c) set up addresses of descriptors for template pointer references.

2) Replace stack with descriptor area:

3) Modify templates as follows:
   a) Change all stack references to pointer references.
   b) Replace stack management ICL statements with descriptor management ICL.
   c) Add PRIV statements as necessary.

4) Maintain a copy of the old global workarea.

Most of these changes are to aid the processing of the IC. The substitution of the descriptor area for the stack and the PRIV statement are required exclusively for optimization as explained above. In fact, these should also be used in the one-pass scheme if future-use information is available from, say, an optimization of the APT. The stack is still required by the translator. The descriptors for the values on the stack, however, are not part of the stack. They are in the descriptor area. The entry on the translator's stack contains a pointer to the appropriate descriptor. This pointer would be changed by the PRIV processor when the private descriptor is created.
CHAPTER 5. TRANSPORTABILITY

A topic of continuing interest in the field of software engineering is the transportability of compiler software. This chapter discusses the feasibility of using the code generator model developed in the preceding chapters as the basis for a software system which will facilitate the implementation of a programming language on a wide range of computers and the implementation of several languages on a single type of computer.

Basically, the proposed system capitalizes on the fact that the tasks performed by the translator, the phases which precede it in compilation, and the global SLM optimizer are independent of the object machine, while most of the work the coder performs does not depend on the source language being compiled. In the model, the templates and the routine which translates from symbol table references to descriptors provide the only interface between the source language and the object machine.

The scheme proposed here for the easy portability of compiler software is to separate the compiler between the coder and the translator -- or the coder and the SLM optimizer, if it is present. Each computer would come supplied with a coder, i.e., Figure 3.11, and an ICL tailored to that computer -- with all the local, machine dependent optimization built in -- just as an assembler is supplied now.
CHAPTER 5

To transfer a compiler from machine A to machine B only the templates corresponding to each SLM instruction have to be rewritten. This would be done in the ICL designed for machine B. Similarly, to import a new language compiler for a machine just the templates for its SLM instructions would have to be rewritten.

Of course, all of this assumes the existence of a high level language which will allow the transportability of the recognizer, translator, and optimizer algorithms. Such a high level language will also be needed to program the routine which constructs descriptors from symbol table references and which, in a multi-pass organization, does the decoding of the IC.

This method has several advantages over others that have appeared in the literature.

1) The ability to easily redefine SLM instructions with each new language avoids the problem of developing a universally applicable SLM as is required by the UNCOL approach [Strong 1953; Steel 1961].

2) The ICL template approach provides a much more powerful vehicle for generating code than could any readily available general purpose macro processor -- such as that used in the MOBILE system [Orgass 1969; Waite 1969]. Address construction, environment testing, and the higher level instruction generation statements are particularly noteworthy in this regard. The power of a macro processor goes only so far as its ability to make symbolic substitutions. Efficient
coding requires more than this. For example, the macro prototype statement

\[
L \quad \&R, \&A
\]

can be used only to generate code for immediately addressable \&R's and \&A's, whereas the ICL statement

\[
GRRRX \quad L, R, A
\]

can handle any addressable values.

3) The coder can be incorporated into the same pass as the translator. This possibility is not obviously available in the other proposals.

Of course, the task of writing the coder for a particular computer is considerable, but since it must be written only once for all compilers on the machine, the effort is certainly justifiable. This aside, the only disadvantage of this scheme is that the templates must be rewritten when either the language or computer is changed. This disadvantage is shared by other macro schemes as well. However, because of the piecewise coding constraint, rewriting templates is much easier than starting from nothing. In practice, the changes from machine to machine or language to language should not be too great. In addition, the templates are about as close to a specification of the implementation map as one could possibly imagine.

The following sections discuss details of the proposal and more clearly define what must be rewritten when the transfer is made.
5.1 The Interface Module

The format of the descriptors used by the coder is totally machine dependent, but it should provide some fields for source language dependent information. The ICL statement processors and the utility modules of the coder use only the machine dependent information in the descriptors. Any source related information in the descriptor is there for the convenience of the template system and is manipulated solely through the interpretation of ICL templates.

For example, arithmetic conversions in PL/I are based on the scale and precision declared for the value involved in the conversion. In a particular machine implementation, these may not be meaningful -- e.g., the hardware may not support variable precision arithmetic. Consequently, in that implementation the scale and precision of a value would be saved in the source dependent fields of the descriptor to be used in the SLM instruction templates to program around this shortcoming of the machine.

The template format is unique to the coder and so its decoding in the coder control is not changed when the SLM is changed. So the coder control module, every module it calls, and the format of every data structure they reference remain constant for all SLM's defined. (See Figure 3.2.)

In the one-pass code generator, the template parameters are somewhat of a problem. The translator must supply the address of the global work area and also a means for obtaining
the address of the current top of the stack, so that references to these areas can be resolved by the coder when they occur in the template. An alternative to this would be to use the multi-pass version of the coder and pointer references. In this case an interface module, similar to the multi-pass scanner, would have to be supplied to establish the address of template parameters. This scheme is perhaps preferable because then only one coder is needed to accommodate all code generator organizations.

In the multi-pass scheme, the scanner must be rewritten for each new SLM to decode properly the SLM-dependent IC format and so establish the proper association between SLM operation code and coder template.

In both organizations, the routine which constructs descriptors from symbol table references must be rewritten for each new symbol table. This routine realizes the mapping from source data types to machine data types and converts the access function of each variable into the appropriate BASE, DISP and INDEX fields of the machine dependent descriptor.

In Figures 5.1 and 5.2, the portable code generators are diagrammed. In the one-pass code generator (Figure 5.1), the descriptors are maintained in the descriptor area, controlled by the coder, and referenced via pointers from the translator's stack. The interface module translates references to the stack into descriptor addresses to be used in pointer references in the template. It also constructs descriptors for identifiers when directed by the translator.
The interface module in the multi-pass code generator (Figure 5.2) decodes the IC and constructs identifier descriptors as in the multi-pass scanner. (See Figure 4.9.)

Figure 5.1 One-pass Portable Code Generator Organization
Figure 5.2 Multi-pass Portable Code Generator
5.2 The Proper SLM Instructions For Machine Independent Translation

Since the translator can generate any sequence of SLM instructions desired, the only question which must be answered concerning translation is how machine independent can the SLM be? More properly stated, how should the SLM be defined to be easily coded on a large class of machines?

To conform with the architecture used in the mainstream of computers today, the SLM instructions should be organized for sequential execution and have a format of the form \(<\text{OP} >\) \(<\text{Operand}>\). The instructions should have at least three addresses -- two input operands and a result operand. Such an SLM can be coded very efficiently on a zero-, one-, two-, or three-address machine with no complications. All of the examples in this thesis have used this format to illustrate coding on a one-address computer.

An SLM instruction with less than three addresses -- for binary operations -- could not be accommodated easily on a three-address machine. In this case the coder would have to synthesize one object machine instruction from several SLM instructions -- a difficult task in the coder defined for the model.

In general, there is no need for the SLM to specify registers. It should work on named storage locations only. The implementation will decide how physical registers are to be allocated to these data items and intermediate results. An
optimizing pass could put retention levels on each data item to help the register allocator make the best assignments. The retention level would indicate the strength of conviction with which a value should be maintained in an easily accessible register. The ability to dedicate a register allows the implementer to remove (from the control of the coder) registers with pre-assigned functions.

For the purposes of transferability the SLM instructions should be chosen so that they are as general as possible. They should be very close to the source language structures and far from any particular machine. This will give the templates maximum context for optimization and greatest freedom in deciding on the best code sequences.

In the interest of optimization, contextual information which may be of use to some coder -- but not necessarily of interest to the translator -- should be included in the SLM instruction sequence where possible. For example, the fact that an intermediate result will be used in a multiplication or division would be helpful to the coder for the IBM System/360. To provide this information, either prefix translation must be used by the translator or it could be determined by the optimizer. Because there is no feedback from the coder, anything more complex than prefix-translation could not be used to improve the coding operation.
5.3 **Summary**

As presented in the previous chapters, the code generator is organized for easy transfer from one machine to another. To perform the transfer the following steps should be executed:

1) Decide on the implementation of each SLM instruction on the new object machine. This will probably be described in the form of a flowchart covering all possible run-time situations.

2) For each SLM instruction, program the implementation flowchart as an ICL program segment. This program segment becomes the template for the SLM instruction.

3) Write the interface module to:
   a) associate SLM operation codes with the proper template;
   b) construct descriptors in the descriptor area from symbol table references in the SLM instructions;
   c) convert all SLM instruction operand references to descriptor addresses; and
   d) pass the template and these addresses to the code control module to code the SLM instruction.

4) Recompile the front end of the compiler on the new machine.

5) Link together the front end, templates, interface module and the object machine's coder to form the compiler for the new machine.
CHAPTER 6. CONCLUSION

6.1 Further Work

Having carefully defined the components of a code generator and their interaction, the next logical step is to use this model as the basis for the automatic generation of a code generator.

To this end, it seems feasible that an interpretive translation language (ITL) -- analogous to the ICL -- could be developed to reduce the size of the translator and to better isolate the mechanisms of translation. The rules for translating each source operator defined by a semantic template would be written in the ITL. This semantic template would specify the order and mode of evaluation of operands and the specific SLM instructions which are to be generated for the operator. From the template the classification of the operator can be determined. Then the appropriate flowchart in chapter two, which served there as a description of the logical flow of operations in the translation of operators in that class, could now be used as the skeleton for the interpretation of the semantic templates of that class. The prologue, intervening, and epilogue code, state changes, and result specifications would be supplied from the template at the appropriate times during translation.

Similar to the syntax-directed recognition system, the basis for a template-driven translation system would be a
semantic template defining each node operator of the APT. The translator-compiler would classify the operators and generate output which would include the proper tables for each operator and drivers for each class along with a core translator similar to that charted in Figure 2.8a which would scan the IT and set up the templates to be interpreted for each input symbol. The class of each operator would be passed to the recognizer-compiler so that the proper IT format could be constructed by the recognizer.

For a given machine, the coder is already built and so only the templates must be rewritten and combined with that stock coder to produce the rest of the code generator. A very interesting problem is the automatic generation of a coder for a machine from a description of the machine. In his thesis, P.L. Miller [Miller, 1971] has done some work in this direction. Essentially, from a description of the basic operational machine instructions and a description of the instructions for moving data between hardware components, his scheme is capable of generating the equivalent of the statement processor for a macro level instruction generation ICL statement. What further progress toward this end is possible, is difficult to predict.

6.2 Summary Of Code Generation Model

This thesis has examined a model for the code generation process in compilers for modern high level programming
languages. For this examination code generation has been divided into four major phases which are initiated in the order storage allocation, translation, global optimization and finally coding. In chapters two and three of this thesis, models for the translator and coder phases are developed in detail. In chapter four, the model is expanded to provide for a global optimization pass. Chapter five exploits the dichotomy of tasks inherent in the model and discusses the portability of the code generator.

The translator works with an abstract tree representation of the program which has been constructed by the earlier recognition phases (APT). (See Figure 6.1.) From the APT it produces a sequence of instructions for an artificial machine based on the operations of the source language (SLM). As introduced in chapter one, the form of the SLM has only two restrictions --

1) Instructions must be executed sequentially unless an explicit branch instruction is executed, and

2) Each instruction must have a fixed number of evaluated, single-valued operands.

In chapters four and five additional criteria for the proper selection of SLM instructions were introduced to facilitate optimization and transportability.

The way the APT is to be translated is dictated by a semantic mapping which ultimately takes each node operator of the APT to a sequence of SLM instructions.
Figure 6.1 Logical Phases of Compilation and Defining Maps
The coder works on the SLM representation of the program -- possibly modified by the optimizer -- to produce the object machine code. The way the SLM representation is to be coded is dictated by an implementation which takes each SLM instruction to a sequence of object machine code.

Chapter two develops a classification of source operators having seven major categories: postfix-translatable, variadic postfix-translatable, one-state prefix-translatable, multi-state prefix-translatable, variadic prefix-translatable, tree-translatable, and n-pass-translatable. The classification is complete and can be determined solely from the operator's semantic template. Associated with each operator class is an intermediate text format and a translation algorithm which together characterize the most efficient means of handling the translation of operators in the class. Operations in different classes can be represented in the same IT with only a few symbols added to punctuate the structure. A translator for this combined IT is presented.

In chapter three a coder is described which will realize any implementation map which can be restricted to a concatenation of code sequences each of which is the implementation of a single SLM instruction. The coding of an SLM instruction may depend only on the SLM instructions which precede it. The presentation in chapter three concentrates on the structure of the compile-time description of the run-time environment and the encoding of the restricted implementation maps. The techniques discussed are similar to those used in
many different compilers, but are presented in an expanded and more organized fashion.

The environment is described by a set of value and component descriptors. Unlike the descriptors used in most existing compilers and translator writing systems, the descriptors of the model are associated with values rather than locations. When code to compute a value is generated a new descriptor is constructed to describe it. This descriptor exists until the value it describes is either destroyed or no longer referenced in the SLM code.

The value and type information in each value descriptor remains constant for the life of the descriptor, but the access function is constantly changed so that it always describes the most easily accessible copy of the value. This is a unique feature of these descriptors and is available only because the descriptors are not associated permanently with any symbol table entry or other information structure used by the rest of the compiler.

The encoding of the access function is also new. The almost universal representation used can describe indirect addressing to arbitrary depth with indexing (constant or variable) available at each level. The representation is very simple, yet is capable of describing any location addressable in computers of modern architecture. The use of pointer references in the representation makes possible a very precise and flexible description of the run-time access function.
The algorithm for coding an SLM instruction is represented in the coder by a code template which is interpreted by the nucleus of the coder at compile time. This allows a very compact implementation of the coding algorithms and plays a key role in the transportability of the model. The template is created using a coding language (ICL) especially designed for the object machine. The statements of the ICL isolate the functions necessary in coding.
Here is a complete ICL for the IBM System/360 computer. The general format of an ICL statement is:

\[ \text{<Label> <ICL-op> <ICL-opnd> \{<ICL-opnd>\}} \]

There are semantic and syntactic restrictions placed on the <ICL-opnd>'s of some <ICL-op>'s. These restrictions are defined below in BNF and English:

\[ \text{<Parm> ::= a pair of numbers, (n, m), encoded in an ICL operand.} \]
\[ \text{n identifies an area known to the coder -- e.g., 1=translator stack, 2=global workarea, 3=coder argument list. The number m specifies the offset in the area where the value of the parameter starts. If n is zero, the value of the operand is m.} \]

\[ \text{<D-ref> ::= <Parm> referencing a descriptor.} \]
\[ \text{<Cn-ref> ::= <Parm> referencing an n-bit constant.} \]
\[ \text{n may not be zero.} \]

\[ \text{<RD> ::= <D-ref> to VALUE_DESCRIPTOR in register format (RFD).} \]
\[ \text{<SD> ::= <D-ref> to VALUE_DESCRIPTOR in storage or address format (SFD).} \]
\[ \text{<D> ::= <RD> | <SD>} \]
\[ \text{<SAD> ::= <D-ref> to storage area descriptor.} \]
\[ \text{<CAD> ::= <D-ref> to code area descriptor.} \]
<RS> ::= <Parm> referencing a register group save area.

<SR> ::= reference to program element. It may take any of the following forms and will have the meaning indicated:

<table>
<thead>
<tr>
<th>Form</th>
<th>Value Referenced in Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;SD&gt;</td>
<td>C(&lt;SD&gt;)</td>
</tr>
<tr>
<td>&lt;SD&gt;[i&lt;Disp&gt;][(&lt;D&gt;)]</td>
<td>C(&lt;SD&gt;[i&lt;Disp&gt;][+C(&lt;D&gt;))]</td>
</tr>
<tr>
<td>&lt;DISP&gt;(&lt;D1&gt;[,&lt;D2&gt;])</td>
<td>C(&lt;DISP&gt;+C(&lt;D1&gt;)[+C(&lt;D2&gt;))]</td>
</tr>
</tbody>
</table>

<Disp>::=<C24-ref>

<ID> ::= <C8-Ref>       Immediate data

<Len> ::= <C8-Ref>       String length

<Int> ::= <C32-Ref>      Integer constant

<DR> ::= <RD> | <SB>    Other symbols used in the following definitions of the ICL statements should be self-explanatory.

ADDRESS COMPUTATION STATEMENTS

Make_RX_Addressable: MRIA <D>

Code is generated where necessary and <D> updated accordingly so that <D>.BASE and <D>.INDEX point to general purpose RPD's and <D>.DISP is a positive integer constant less than 4096.
**Make RS Addressable: MRSR <D>**

Same as MRIA except <D>.INDEX is made NULL.

**INSTRUCTION GENERATION STATEMENTS**

**Generate RX/RR/RS/SI/SS Instruction:**

GRX <RX-op>,<RD>,<SR>

GRR <RR-op>,<RD>,<RD>

GRS <RS-op>,<RD>,<RD>,<SR>

GSI <SI-op>,<SR>,<ID>

GSS <SS-op>,<SR>[ (L=Len)],<SR>[ (L=Len) ]

An instruction in the 360 instruction format indicated by the ICL operation code is generated. If necessary, address code is generated for <SR> operands which are not immediately addressable. Provision is also made for undefined <SR>'s -- e.g., forward label references. No descriptors are changed unless they are involved in address computation.

**Define Label: DLAB <D>**

The address portion of <D> is set to define the current location in the code area being used by the coder. If <D> has been referenced, the coding of those references is completed at this time.
Generate Load Address: GLA <RD>,<SR>

Code is generated to load the address of <SR> into register <RD>. The code generated does not necessarily preserve the condition code setting.

Macro Generate RR or RX Instruction:

MGRX (<RX-op> | <RR-op>),<DB>¹,<DR>²

If <DR>¹ is not in a register, code is generated to load it into one. If <DR>² is in a register the RR form of the operation is generated, if one exists. Otherwise <DR>² is stored in temporary storage (if necessary) and an RX instruction is generated. A FREE <DR>² is issued.

Macro Load General Purpose Register:

LGPR <DB>[,(ODD | EVEN)][,PAIR]

An [ODD or EVEN numbered] general purpose register [PAIR] is allocated as requested. Code is generated to load <DR> into the [ODD or EVEN numbered] register [of the PAIR]. <DR> is formatted into an RFD identifying the register allocated, provided <DR> does not specify address modification.

Macro Load Floating Point Register:

LFPR <DB>[,(LONG | SHORT)]
A floating point register is allocated. Code is generated to load a [LONG or SHORT] floating point number from <DR> into the register. <DR> is formatted into an RPD for the allocated register as long as <DR> does not specify address modification.

**Macro Generate RS Instruction:** MGRS <RS-Op>,<DR>\textsuperscript{1},<DR>\textsuperscript{2},<DR>\textsuperscript{3}

If <DR>\textsuperscript{1} or <DR>\textsuperscript{2} is not in a register, code is generated to load it. If unmodified, <DR> is formatted into an RPD when it is loaded. Code to make <DR>\textsuperscript{3} RS-addressable is generated if needed. The RS instruction is then generated. Finally, a FREE <DR>\textsuperscript{3} is generated.

**Macro Generate SI Instruction:** MGSI <SI-Op>,<DR>,<ID>

<DR> is made RS-addressable -- stored into temporary storage if necessary. The SI instruction is generated.

**Macro Generate SS Instruction:**

MGSS <SS-Op>,<DR>\textsuperscript{1}[(L=<Len>)],<DR>\textsuperscript{2}[(L=<Len>)]

<DR>\textsuperscript{1} and <DR>\textsuperscript{2} are made RS-addressable -- stored into temporary storage if necessary. Then the SS instruction is generated. If L is specified, it overrides the length specification for <DR>. 

Generate Generic Instruction: GGI \( <\text{Gen-op}>,<\text{DR}_1>,<\text{DR}_2> \)

\( \text{Gen-op} ::= \text{ADD}|\text{SUB}|\text{MPY}|\text{DIV}|\text{SHIFT}_L|\text{SHIFT}_R|\text{COMP}|\text{MOD} \)

\( <\text{DR}_1> \) and \( <\text{DR}_2> \) must be the same type. The operation code used in the instruction is determined from this type information. \( <\text{DR}_1> \) and \( <\text{DR}_2> \) are made RS or RX addressable as needed and then the instruction is generated.

Convert: \text{COMV} \( <\text{D}>,<\text{DR}> \)

Code is generated to convert \( <\text{DR}> \) to the type indicated in \( <\text{D}>> \). \( <\text{D}>> \) is made to reference the converted value.

Generate Instruction with Conversion:

\text{GICRX} \( (<\text{RX-Op}∥<\text{RR-Op}>>, <\text{DR}> , <\text{DR}> \)

\text{GICRS} \( <\text{RS-Op}>, <\text{DR}>, <\text{DR}>, <\text{DR}> \)

\text{GICSI} \( <\text{SI-Op}>, <\text{DR}>, <\text{ID}> \)

\text{GICSS} \( <\text{SS-Op}>, <\text{DR}>, [\{L=\text{Len}\}], <\text{DR}>, [\{L=\text{Len}\}] \)

\text{GICG} \( <\text{Gen-Op}>, <\text{DR}>, <\text{DR}> \)

Operands are first converted to highest compatible type. Then the proper operation code is determined, the operands are made addressable, and the instruction generated.
REGISTER MANAGEMENT STATEMENTS

**Dedicate Register**: DEDR <RD>

The register specified by <RD> is removed from the control of the coder. If necessary, its content is saved in temporary storage.

**Manage Register**: MANR (GPR | FPR),<RD>

The register specified by <RD> is assigned to the general purpose (GPR) or floating point (FPR) register group. Its allocation will be controlled by the coder until further notice.

**Get General Purpose Register**: GGPR <RD>[<TYPE>[,<CLASS>[,<SD>]]]

<Type> ::= [,(ODD | EVEN)][,PAIR]

An [ODD or EVEN numbered] general purpose register [PAIR] is allocated and assigned to the descriptor <RD>. A register will be reallocated if necessary. If <CLASS> is specified an SRC <RD>,<CLASS>,<SD> is issued. Otherwise, the register is marked "in use, contents unknown."
**Get Floating Point Register:** GPRR <RD>[<CLASS>[,<SD>]]

A floating point register is allocated and linked to <RD>. A register will be reallocated if needed. If <CLASS> is specified an SHR <RD>,<CLASS>,<SD> is issued. Otherwise, the register is marked "in use, contents unknown."

**Free Register or Temporary:** FREE <RD>|<SD>

If the <RD> register is managed and no further references to it are indicated, the register it specifies is returned to the pool of empty registers in the register group it was originally associated with. If the SD is a temporary storage element allocated by the register manager and no further references to it are indicated, the space it defines is made available for reallocation in the storage area from which it was obtained.

**Form Register Pair:** PAIR <RD>[,(ODD | EVEN)]

An even/odd register pair is allocated and <RD> is linked to it. If <RD> specifies a general purpose register, an attempt is made to allocate the pair which includes that register. If the pair allocated does not include <RD>, code is generated to move <RD> to the [ODD or EVEN member of the] pair. In this case, the register <BD> had occupied is freed.
**Release Member of Pair**: UNPR <RD>

If <RD> specifies a register allocated as an even/odd pair, the other member of the pair is released.

**Change Register Description**: RDESC <RD>, (EVEN | ODD)

If <RD> specifies a register allocated as an even/odd pair, the <RD> is set to indicate the value is in the EVEN or ODD member of the pair. RDESC is a no-op in any other case.

**Link Descriptor to Register**: LINK <SD>, <RD>

<SD> is linked to the RSD specified by <RD> and so indicates <SD> is in the register. <RD> is cleared.

**Save Register Status**: SAVE <RS>

The status -- contents, class, uses left, etc. -- of all managed registers are stored in the register status save area <RS>.

**Restore Register Status**: REST <RS>

The current status of each managed register is made to agree with the status of that register as specified in the register status save area <RS>.
**Join Register Statuses:** JOIN <RS>

For each currently managed register, if its status differs from the status recorded in <RS> it is marked "empty." If they agree, it remains unchanged.

**Set Register Class:** SRC <RD>,<Class>[,<SD>]

The ALLOCATION_CLASS of the register specified is set to <Class>. If <Class> specifies the contents are known, <SD> supplies the CONTENT_ADDRESS fields for the RSD. It is not LINK.

**Storage Management Statements**

**Create Storage Area:**

CSA <D>,TYPE=(CA | CDA | VDA)[,START=<Int>] [,MAXSIZE=<Int>]

<D> is formatted as a storage area base descriptor. MAXSIZE sets an upper bound on the area (0=infinite). START is used to initialize the location counter for the area. TYPE indicates the allocation algorithm and whether any elements will be initialized (CA=code area, CDA=constant data area, and VDA=variable data area).
Establish Code Area: ECA <SAD>

After interpreting this instruction, all code generated by the coder will be placed in the storage area defined by <SAD>. SAD must have the "initialized" attribute.

Establish Temporary Area: ETA <SAD>

After interpreting this instruction, temporary storage allocated by the register allocator will be obtained from the storage area defined by <SAD>.

Establish Constant Data Area: ECDA <SAD>

After interpreting this statement, constants used in the address computer will be allocated in the constant storage area defined by SAD.

Allocate Storage Element: ASE <SAD>,<D>

Space is allocated in the area defined by <SAD> based on the type information in <D>. <D> is formatted to define the space allocated.

Free Storage Element: FSE <D>

The space originally allocated to <D> in some storage area (by ASE) is made available for reallocation.
**Initialize Storage Element:** ISE <SAD>,<D>

If <D> has not been allocated space, it is assigned from <SAD> as in ASE. The VALUE field of <D> is used to initialize the space allocated.

**FLOW OF CONTROL STATEMENTS**

**Branch If (not) Register Pair:** B(N)P <D>,<Label>

Interpretation continues at <Label> only if <D> is (not) a member of a register pair.

**Branch If (not) In Register:** B(N)R <D>,<Label>

If <D> does (not) define a register, then the ICL statement at <Label> will be interpreted next.

**Branch If (not) In Odd Register:** B(N)OR <D>,<Label>

If <D> does (not) define an odd numbered general purpose register, the coder will interpret the statement at <Label>. Otherwise the next statement in sequence will be executed.

**Branch If (not) In Even Register:** B(N)ER <D>,<Label>
If <D> does (not) define an even numbered general purpose register, the branch is taken.

**Branch If (Not) RS-addressable:** B(N) RSA <D>,<Label>

Interpretation continues at <Label> only if <D> is (not) RS-addressable.

**Branch If (Not) RX-addressable:** B(N) RXA <D>,<Label>

Interpretation continues at <Label> only if <D> is (not) RX-addressable.

**Branch:** B <Label>

Interpretation continues at <Label>.

**Branch and Link:** BAL <Label>

Interpretation continues at <Label>. The location of the next statement is saved in a push-down stack internal to the coder.

**Return From Subroutine:** RTN

Interpretation continues at the location specified on the top of the internal push-down stack. The label is removed.
from the top of the stack.

**Exit from Coder:** EXIT

The coder relinquishes control to the translator (one-pass) or SCANNER (multi-pass).

**BIT MANIPULATION Routines**

**Bitwise And/Or/Exclusive-Or Flags with Mask:**

APM <Flags>,<Mask>

OPM <Flags>,<Mask>

IFM <Flags>,<Mask>

The bitwise operation specified in the statement is performed between <Flags> and <Mask>. The result is left in <Flags>.

**Branch Under Mask Ones/Zero/Mixed:**

BMO <Flags>,<Mask>,<Label>

BMZ <Flags>,<Mask>,<Label>

BMM <Flags>,<Mask>,<Label>

For each bit of <Mask> which is a one, the corresponding bit in <Flags> is tested. Interpretation continues at <Label> if the tested bits are all zero (BMZ), all ones (BMO), or
mixed (BMM).

STACK MANAGEMENT STATEMENTS (One-Pass Code Generator)

Enter New Stack Element:  PUSH [<D>]

A new element is generated on the top of the translator stack. If <D> is specified, the stack entry is initialized to <D>.

Remove Element from Top of Stack  POP <Integer>

<Integer> elements are removed from the top of the translator stack.

DESCRIPTION MANAGEMENT STATEMENTS

Format Descriptor:  FORD <D-Ref>[<D>]

The location <D-Ref> is formatted as an empty descriptor. It is initialized to <D> if specified.

Create Descriptor:  CRED <D-Ref>[,<D>]

<D-Ref> must be a pointer reference (n=3). A value descriptor is allocated in the descriptor area. <D-Ref> will reference the new descriptor. If <D> is specified, the new descriptor will be formatted like <D> and initialized with a copy of <D>.

Destroy Descriptor: DESD <D>

The space occupied by <D> in the descriptor area is released.

Copy Descriptors: SET <D>1, <D>2

The STATUS, ACCESS_FUNCTION, and ATTRIBUTES fields of <D>2 are copied into the corresponding fields of <D>1.

Create Private Copy: PRIV <D-Ref>

After the coding interpreter this statement, <D-Ref> will reference a private copy of the descriptor which <D-Ref> originally referenced. The new descriptor may be modified without affecting the globally known description of the value. The value described by <D-Ref> will be protected from alteration, if other uses for the value remain.
REFERENCES


[Dewar 1969] Dewar, R. B. K., R. R. Hochspring, and


REFERENCES 198

Cornell University, 1970.


[Tixier 1967] Tixier, V.; Recursive Functions of Regular Expressions in Language Analysis, TR#58, Computer Science Department, Stanford University, March 1967.


