GENERATING MACHINE CODE FOR HIGH-LEVEL PROGRAMMING LANGUAGES

THESIS

Presented to the Graduate Council of the
North Texas State University in Partial
Fullfillment of the Requirements

For the Degree of

MASTER OF SCIENCE

By

Chia-Huei Chao, B. S., M. A.

Denton, Texas
December, 1976

Chao, Chia-Huei, Generating Machine Code for High Level Programming Languages. Master of Science (Computer Science), December, 1976, 97 pp., 9 illustrations, 15 appendixes, bibliography, 17 titles.

The purpose of this research was to investigate the generation of machine code from high-level programming language. The following steps were undertaken:

1) Choose a high-level programming language as the source language and a computer as the target computer.

2) Examine all stages during the compiling of a high-level programming language and all data sets involved in the compilation.

3) Discover the mechanism for generating machine code and the mechanism to generate more efficient machine code from the language.

3) Construct an algorithm for generating machine code for the target computer.

The results suggest that compiler is best implemented in a high-level programming language, and that SCANNER and PARSER should be independent of target representations, if possible.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF APPENDIXES

CHAPTER I

INTRODUCTION

1.1. The Problem Definition

The purpose of this research is to investigate the generation of machine code from high-level programming languages; in particular, having proposed a model of such processing, to answer the questions:

Given a high-level programming language, how can the target computer code be generated?

How can a more efficient machine code for the target computer be generated?

1.2. Procedure

In order to accomplish the purpose of the research, the following steps were undertaken;

1) Choose a high-level programming language to be the source language and a computer as the target computer.

2) Examine all stages during the compiling of a high-level programming language and all data sets involved in the compilation.

3) Discover the mechanism for generating machine

code and the mechanism to generate more efficient machine code from the language.

4) Construct an algorithm for generating machine code for the target computer.

1.3. Source Language and Target Computer

The TPL (THIS PROGRAMMING LANGUAGE) programming language was chosen as the source language. Although it is a relative simple language, it is complex enough to display many of the quality and implementation difficulties of more advanced high-level language.

The FAIRCHILD F24 mini-computer was chosen as the target computer for the model, since it is a memory-oriented, high-speed, general-purpose digital computer with flexible addressing abilities.

Figure 1.1 is the system flow for generating machine code from the TPL programming language.

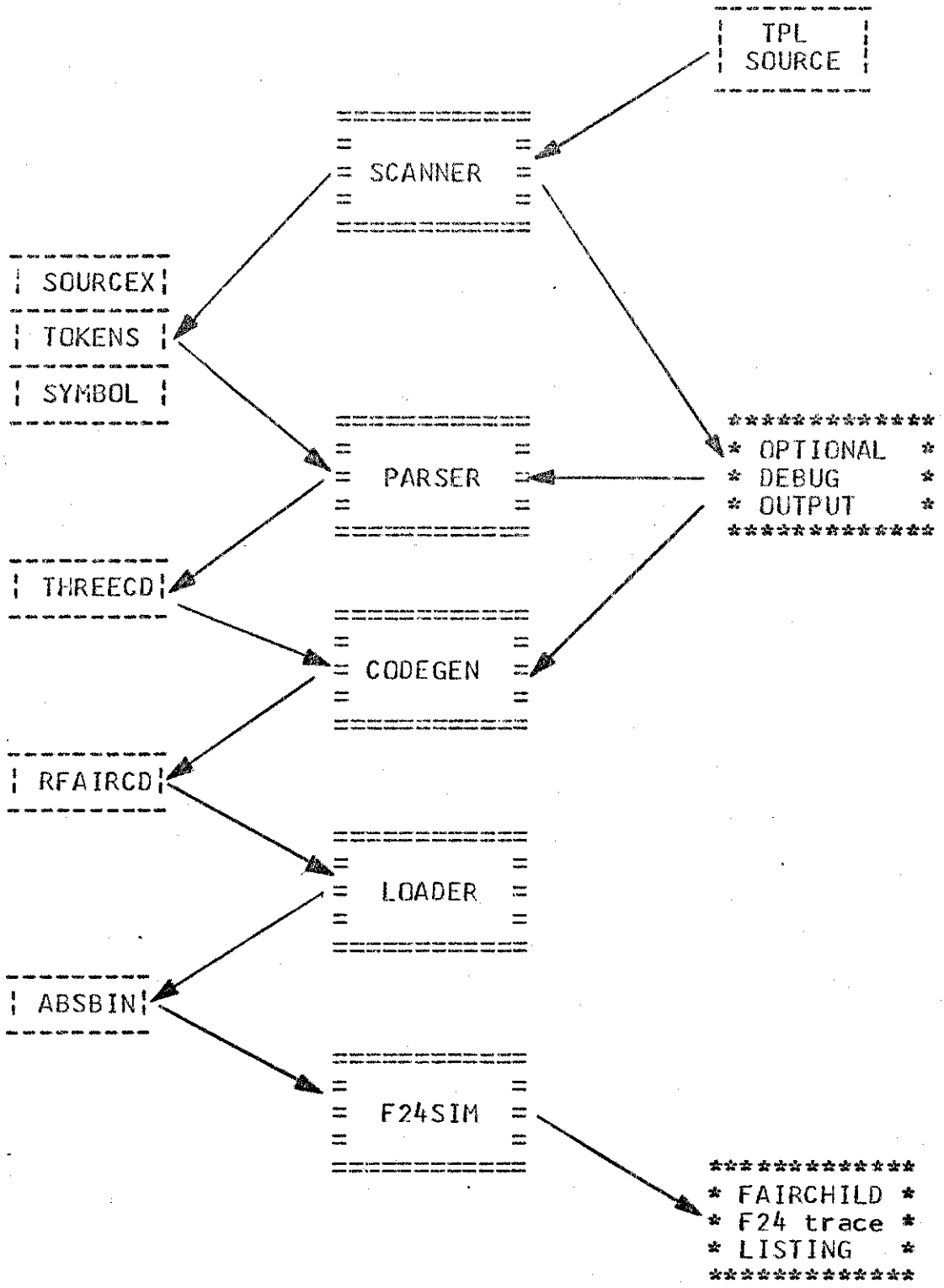


FIGURE 1.1. System Flow for Generating Machine Code

1.4. Organization

This research is organized into eight chapters; the first chapter provides a general description of the research. The second chapter describes the analysis stages during a language compilation. The third chapter describes the storage allocation algorithms used by high-level programming language. The fourth chapter discusses the function and algorithm for code generation. The fifth chapter examines several techniques for generating optimized target machine code from high-level programming languages. The sixth chapter contains an description of This Programming Language, which is the source language of the model. The seventh chapter discusses the data bases which are involved in the generation of machine code from a high-level programming language. The last chapter contains the summary and conclusion.

CHAPTER II

ANALYSIS

2.1. Introduction

The purpose of the analysis during a compilation of a programming language is to translate the input source language into a intermediate form (usually a structure tree)(3); from this intermediate form the code generator creates the target machine code for the language.

In this chapter different phases of analysis for compiling a language are depicted.

2.2. Lexical analysis

The action of parsing the source program into proper syntactic classes is known as lexical analysis. The aim of the lexical analysis of the compiler is to take the input source language, which is presented in some form, and translate this into a string of tokens. We usually call this translator the "scanner". The token stream which comes out of the scanner is the input to the parser, which is the processor in the syntax analysis phase during a language compilation.

2.2.1. Scanning

Scanning is the major processing during the lexical analysis phase. Scanning involves finding the substrings of characters that constitute units called textual elements. These elements are the words, punctuation, single- and multi-character operators, comments, sequence of spaces, and numbers of the source.

For example, consider the following line from a PL/1 program represented as a character stream.

```
IF XX < 10 THEN YY = YY + 1 ;
```

After scanning the program may be regarded as being in following form :

```
IF  XX   <  10  THEN  YY  =  YY  +  1  ;
--..--...-..--..----..--..-..--..-..-..-
ID..ID...OP..N.. ID ..ID.OP..ID..OP.N..OP
```

Where 'ID' means 'identifier', 'N' means 'integer', '.' means 'space', and 'OP' means 'operator'. The 'space', the 'identifier', the 'integer', and the 'operator' are textual elements.

2.2.2. Lexical Grammars

One can usually specify the textual elements of a programming language, its lexical level, with a regular

grammar or a regular expression, or most conveniently with a mixture of the two in the form of a transduction grammar. For example, consider the following grammar:

<TEXT>=<IDENTIFER><SPACE>¦<INTEGER><SPACE>

<IDENTIFIER>::=<LETTER>¦<LETTER><LETTER-DIGIT>

<LETTER-DIGIT>::=<LETTER>¦<DIGIT>

<INTEGER>::=<DIGIT>¦<INTEGER><DIGIT>

<LETTER>::=A¦B¦.....¦Z

<DIGIT>::=0¦1¦2¦.....¦9

<SPACE>::= ' ' ¦' '<SPACE>

This grammar describes a very simple lexicon containing identifiers, integers, and spaces. Identifiers and integers must be separated from each other by at least one space.

2.2.3. Finite-state Machine

To display more clearly the structure of scanner for the lexical grammar in section 2.2.2., I present its finite-state machine(1) diagram on Figure 2.1. In fact, the scanner procedure is a simulation of a finite-state machine which breaks the source program into tokens(1). In Figure 2.1. Two states noted by '?' need look-ahead sets(2) to determine read-reduce(2) decisions. The finite-state machine simulation is done in the usual manner with two tables(4):

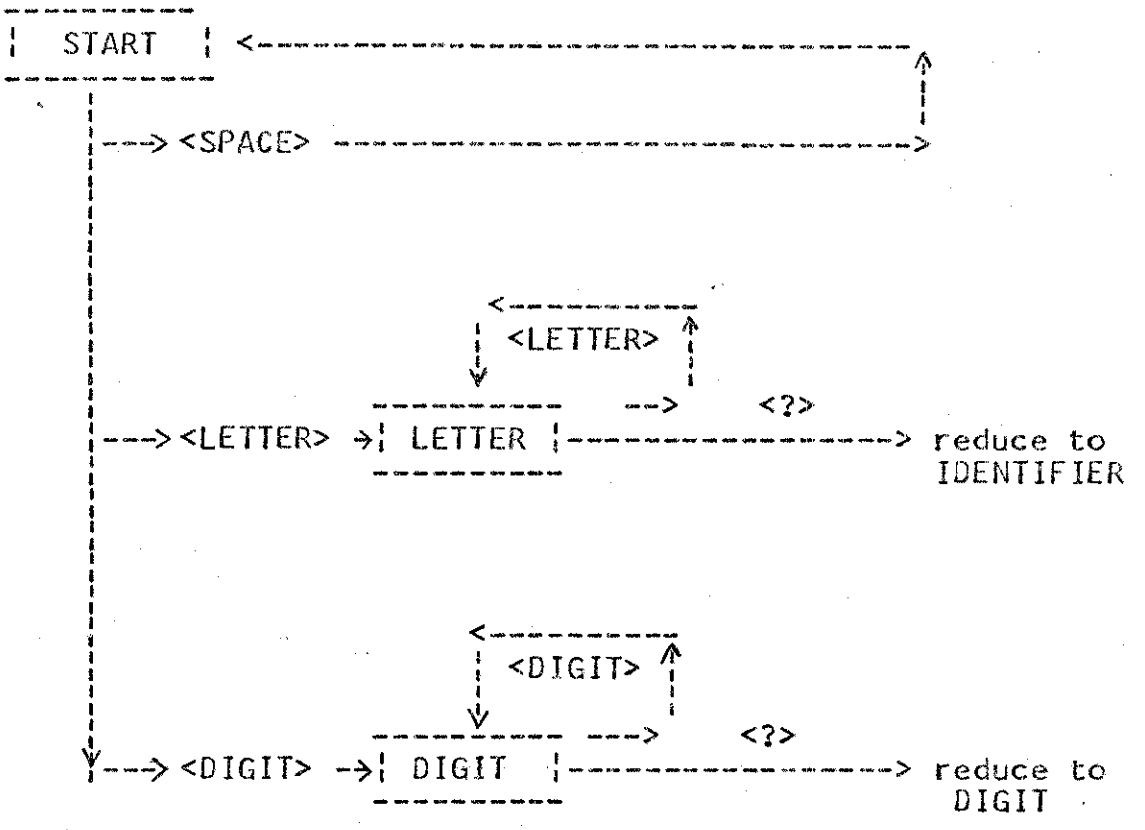


Figure 2.1. Finite-State Machine for Lexical Analysis

one table which defines the next state function, and another table which defines the action associated with each state transition.

The scanner can be implemented directly as an executable program. However, it is noteworthy that scanners frequently have states with direct loops, such as states blank, letters, and digits. Such states should be implemented as fast as possible since they typically do the bulk of the scanning.

2.3. Error Recovery in Lexical Analysis

While scanning a textual element, the scanner is always either in a context in which it has had some left context that must be matched by the some right context (e.g., the right parenthesis must matched the left parenthesis) or it is in a context that may legally end at any point. In the latter case, characters in error show up as the beginning of the next textual element and can usually be skipped or replaced with a blank to permit continuation of the processing. In the former case a scan to the end of the current line is usually done in order to try to find the desired right context. If found, the intervening text can be considered part of the current textual element; otherwise,

the rest of the line is usually best skipped and the scanner is best restored to its initial state.

After detecting and reporting an error, a module may either attempt to repair it (so it is not seen by subquent modules) or pass it along. Each approach has its problems; if a module is to be truly an error sink, it must ensure that none of the effects of the error it has repaired can propagate. Conversely, if it does not filter out all errors, then all subquent modules must be prepared to deal reasonably with them (without generating too many further messages).

In many compilers, a single error can trigger a whole avalanche of messages on the unsuspecting error; this is very nearly as unacceptable as quitting the scan after the first error(5).

2.4. Syntax Analysis

The main aim of the syntax analysis phase during a language compilation is to take the token string produced by the scanner in lexical analysis phase, and to use some parsing algorithm to verify that the token string consists of a legal string. In addition, it is required to collect information about the language, and produce as output a structure tree which could be code which is ready to be

executed or interpreted, but is more likely to be a structural representation of the token string which will be used to generate code.

2.5. Error Recovery in Syntax Analysis

Syntactic analysis also notes syntactic errors and assures some sort of recovery so that the compiler can continue to look for other compilation errors.

A syntactic error is discovered when the parser can take no further valid parsing actions, given the current state of the parser (the stack) and the current input symbol. Recovery thus requires changing the stack, the input, or both. The change may take the form of deletions or insertions (a substitution is a deletion and a insertion).

Gries(2) points out that changes to the stack are particularly dangerous, since semantic routines will have been invoked for the parsing actions leading to the current stack, and the parser can not safely undo or modify the effects of these actions.

Leinius(5) was consider augmenting the syntactic description of a language by a number of error productions, describing common errors, as so that recovery can be subsumed under normal parsing. For this strategy to be

effective, several problems must be deal with: the compiler-writer must ensure that he has really included enough error productions to cover the common errors; since so many different errors are possible, the error productions may substantially enlarge the grammar (and hence the parser); it is difficult to include error productions without making the grammar ambiguous.

2.6. Semantic Analysis

The purpose of semantic analysis is to derive an evaluation procedure from the structure tree of an expression and the attributes of its components.

An evaluation procedure is a sequence of primitive operations on primitive operands, and is completely specified by the definition of the source language. The semantic analyzer must deduce the attributes of the various components of a structure tree, ensure that they are compatible, and then select the proper evaluation procedure from those available. The input to the semantic analyzer consists of the structure tree which specifies the algorithm, and the dictionary which provides attribute information.

Two transformations, attribute propagation and

flattening, are performed in semantic analysis(6):

1) Attribute propagation:

Attribute propagation is the process of deriving the attributes of a structure tree from those of its components.

2) Flattening:

Flattening is the process of transforming a structure tree into a sequence by making explict the order in which the operators are executed (in order to produce optimized code from the code generator).

CHAPTER BIBLIOGRAPHY

1. Donovan, John J., System Programming, New York, Mcgraw-Hill, 1972

2. Gries, David, Compiler Construction for Digital Computers, New York, John Wiley, 1971

3. Hopgood, F. R. A., Compiling Techniques, London Macdonald, 1972

4. Isaacson, Portia, A Compiler For This Programming Language, Department of Computer Science, North Texas State University, Denton, Texas 1972

5. Leinius, R., Error Detection and Recovery for Syntax Directed Compiler Systems, PH.D Dissertation, University of Wisconsin, 1970

6. Newly, M., "Abstract Machine Modeling to Produce Portable Software," Software, Practice and Experience, 1972, pp. 107-136

CHAPTER III

STORAGE ALLOCATION

3.1. Introduction

High-level programming languages with different features require different types of storage management, in which a hierarchy can be distinguished; at the bottom end is the static allocation scheme for a language like FORTRAN(5), in which it is possible to know the address that each object will occupy at the run time. At the next level comes the stack techniques for languages like ALGOL60(1), where space is allocated on a stack at block entry and released at block exit. Languages like PL/1(4) permit both types of storage management.

3.2. Static Allocation

In a static allocation scheme it must be possible to decide at compile time the address that each object will occupy at run time. This requires that the number and size of the possible objects be known at compile time. This is the reason why programming languages that use static allocation have constant bounds for arrays and procedures can not be recursive.

The process which the compiler goes through in doing storage allocation for a static language is very simple; during the first pass of the text the compiler creates a symbol table in which are kept the name, type, size and address of each object encountered. During the later code generation phase, the address of each object is thus available for insertion into the object code.

3.3. Dynamic Allocation

Modern high-level programming languages allow recursive procedure calls, and this precludes any attempt at a static storage allocation scheme, since a variable which is declared and used with a recursive procedure may correspond to more than one value at a given moment during the execution of the program.

The usual storage model for a dynamic allocation is a stack, on which entry to a block or a procedure causes a new allocation, the space being freed at exit from the block or procedure. The use of a stack to model nested structures is a standard device(2).

3.4. Storage Allocation For Arrays

In a programming language in which the size of an array is known at compile time, its space can be allocated

statically ;

For example :

A(5,10)

A is a array of size of 5x10 and 50 consecutive storage locations will be reserved:

A(1,1),A(1,2)..A(5,1),A(1,2)..A(5,2)..A(5,10)

When making a reference, element A(I,J) is to be found in location

(J-1)*5+I-1 from the start of the array.

In general, given an array A with bounds Bi :

Element A(I1,I2.. In) is to be found at location

(..((In-1) * B(n-1)-1) * B(n-2)+..+ I2-1) * B1 + I1-1

In a programming language in which the limits of boundries of a array is not known at compile time, storage allocation for a array must dynamic.

3.5. Storage Allocation for Temporary Variables

A major problem of storage allocation arises when considering how to allocate storage for temporary variables required for partial results during a compilation. These

variables are not defined by the language but by the compiler. How many are required and how storage is allocated is completely dependent upon the compiler(3).

The code generator algorithms did not concern themselves with how storage for the temporary variables should be allocated. They can be thought of as taking a new variable name from an infinite set of unused names each time a temporary variable is required by the algorithm. The storage allocation algorithm has to allocate storage for these variables in such a way that the minimum number of storage locations is required.

Variables having completely disjoint ranges at execution time can be allocated the same location. The usage range of a variable is defined as the sequence of code interval between the initial definition of the vaule of a variable and its last use. There are two algorithms for finding the usage range of a variable(3):

3.5.1. Backward Scan Algorithm

Consider a set of variables : V1,V2,V3..... Vn . For each variable Vi, an statement is defined :

ST Vi

which initially sets a value to the variable and

indicates the start of the range for Vi.

A statement :

U Vi

is defined which indicates a subsequent use of this variable. The last instruction of this type using Vi indicates the end of the range for Vi.

A sequence of code produced by code generator then consists of a set of instructions independent of the variable Vi, together with the orders of the two statements defined above which use the variables Vi , for i=1 to n. The problem is to allocate storage to the Vi so that the minimum number of location is required. The assumption is that a variable is no longer required after the last appearance of it in the sequence. In this sense it assumed that the sequence is completed. Once a variable is no longer required, its storage location may be re-allocated to another variable not yet defined. It is assumed that this sequence of instructions does not contain any entry points other than at the top and that control passes straight through the sequence, leaving at the bottom (no branches).

Consider the arithmetic expression :

(A+B*C)/(F*G-(D+E)/(H+K))

The code generated for the expression might be as follows for the IBM 360:

```
L     H
ADD   K
ST    V1
L     D
ADD   E
DIV   V1
ST    V2
L     F
MPY   G
SUB   V2
ST    V3
L     B
MPY   C
ADD   A
DIV   V3
```

The names V1,V2 and V3 would be taken from the set of unused names. As far as storage allocation is concern the squence of code can be written (ST = Store, U = Use):

```
ST V1
U  V1
ST V2
U  V2
ST V3
U  V3
```

As the last of each variable appears before the next is defined, it is obviously that one storage is sufficient.

The general algorithm will be :

Scan the sequence of instructions from the end backward. For each instruction of the type U Vi (I=1 to 3), if no storage location has been allocated to Vi, take the top free storage location from the stack and assign it to Vi and replace Vi in the instruction by the address

of the storage location. For each instruction ST Vi, if no storage location has been allocated to Vi, then either there is an error or this order is redundant, as this implies that there are no subsequant uses of the variable. If storage has been allocated to Vi, then replace Vi in the instruction by the address of the storage location and, as this is the first time use of the variable Vi, the location may now be returned to the free store stack, as it is no longer required.

3.5.2. Forward Scan Algorithm

In the forward scan, the ST instruction defines the start of the range. A count of the number of uses of each variable has been kept and is used to find the last use of a variable so that the end of the usage range also can be found.

CHAPTER BIBLIOGRAPHY

1. Dijkstra, E., "ALGOL60 Translation," ALGOL Bulletin 10, 1960

2. Gries, David, Compiler Construction for Digital Computers, New York, John Wiley, 1971

3. Hopgood, F. R. A., Compiling Techniques, London, Macdonald, 1969

4. International Business Machines, PL/1 Language Specifications, IBM Form C28-6571, 1972

5. International Business Machines, FORTRAN Programmer's Guide, IBM Form C28-6835, 1974

CHAPTER IV

CODE GENERATION

4.1 Introduction

A source language definition specifies the evaluation procedure for the constructs of the language in terms of a set of primitive operators and operands provided for this purpose. Code generation is the process of an evaluation procedure in terms of the primitive of a particular target computer(5).

The basic approach is to simulate the evaluation procedure in the environment provided by the target computer (with its register organization and addressing structure). A symbolic description of the run time contents of the environment is maintained by the code generator. When the evaluation procedure indicates that the contents should be altered, then code to perform the alteration is emitted and the description is updated. The data for the code generator are structure trees. The evaluation procedure specifies the sequence in which the nodes of a structure tree are to be considered when performing the evaluation, and this sequence is largely independent of the particular target computer. The structure tree is traversed by the semantic analyzer,

which considers the entire subtree before deciding upon the best sequence of operations to perform. Thus the code generator input is a sequence of tokens specified by the nodes of the structure tree.

4.2. A Model for Code Generator

The code generator does not have arbitrary access to the structure tree, and must therefore operate on the basis of limited information(5). The model which I have chosen consists of two parts:

1. A push store transducer, which maintains the contextual information that can be derived from the sequence of input tokens.

2. A target computer machine code generator, which maintains the run-time contents of the environment and produces sequence of target computer instructions to implement the abstract primitives.

Hopgood(2) terms these components the translator and the coder respectly.

The transducer passes a sequence of tree structures to the code generator, each consisting of an operator and its associated operands. Each command is interpreted by the code generator in the light of the object environment which

exists at that point in the execution of the program. It generates appropriate code and the updates the environment to reflect effect of that code.

4.2.1. The Transducer

A pushdown store transducer has four components(5) : an input, an output, a finite-state control and a pushdown store. The input models the stream of tokens which encodes the structure tree, and the output models the abstract instructions which will be delivered to the code generator. The finite-state control and the pushdown store encode the limited contextual information derived from the sequence of input tokens.

Information pertaining to the ancestors of the current nodes, and the status of the current node itself, is encoded by the finite state control. The pushdown store contains information derived from subtrees which have been completely traversed. After all subtrees whose roots are descendants of a particular node have been traversed, their entries are deleted from the pushdown store and replace by a single entry for the entire tree rooted at that node. Information from the pushdown store is used to identify the operands of an operator.

4.2.2. Code Generator

In order to interpret the primitives of the source language in terms of the target machine, the code generator maintains descriptions of the values being manipulated and of the target machine environment. A particular value may be represented in many different ways in the target computer, and the purpose of the value image is to specify the current representation of each value. Similarly, the registers of the target computer may contain many different values during execution, and the purpose of the machine environment is to specify the current contents of each register.

A value comes under the control of the code generator when the transducer requests simulation of an operand token, giving the current transducer state as an argument. At that point the code generator creates an entry for the operand and links it to the machine environment. Values pass out of the control of code generator when they are used as operands. This is signalled when the transducer requests simulation an operator token giving the current state and one or more value as arguments. At that point the code generator deletes the operand entries from the value image, breaking any linkage to the machine environment. If a result is specified, a description of the result value is created

and linked to the appropriate entry in the target machine environment.

In fact, the code generator producing a relocatable target machine code packet for each structure tree(3).

4.3. Code Generation For Arithmetic Expressions

To produce efficient machine codes for a arithemetic assignment statement is one of the major problems in the compilation of a programming language(2). The efficiency of two compilers in the execution of a program will depend almost entirely on the code produced by the code generator.

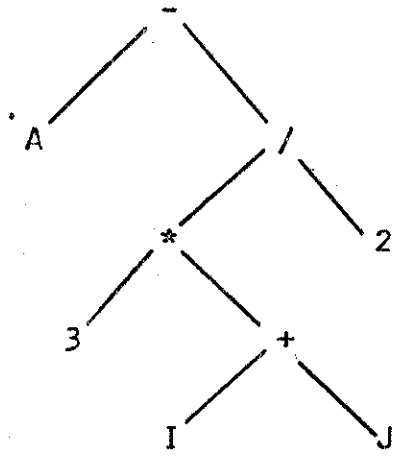
4.3.1. An Algorithm For Code Generation From A Tree Strusture

A structure tree is a graph that consists of a collection of nodes and branches, which each branch connecting two nodes(2).

Waite(5) defines a arithemetic expression as a structure tree written in linear form (figure 4.1.), with each node representing an elementary computation. A leaf of the tree represents a computation which can be carried out independently of all other nodes in the tree, while an interior node represents a computation which requires as operands the results of the computations represented by its decendants.

```
A - 3 * (I + J) / 2
```

A). A Typical Expression



B). The Equivalant Tree for (A)

Figure 4.1. The Meaning of an Expression

One possible evaluation algorithm for this structue tree is the following:

1. Select any leaf and perform the computation which it represents.

2. If the selected leaf is the root, then exit. The result of the computation is the value of the tree.

3. Otherwise, transmit the result to the parent of the leaf and delete the leaf from the tree.

4. Repeat from 1.

This procedure is strictly sequential, but nothing is mentioned about the order in which the leaves are selected.

4.3.2. Anonymous Operands

Waite(5) points out that the reason for using an expression is to avoid naming each of the intermediate results created in the course of a computation. When a leaf of an arithemetic expression is evaluated, the result is anonymous. The code generator is free to do what it well with these anonymous results because it has explicit control over the times at which they are created and the times at which they are freed, it does not need to worry about whether the programmer may access them unpredictably.

The following are three broad categories to process anonymous operands:

1. Use no register

   All instructions take their operands from momory and return their results to memory.

2. Use a single register

   Operators take their operand from the register and return their result to the register.

3. Use mutiple registers:

   Binary operators take one operand either from a register or from memory and all operators return their result to a register. Some registers may be paired to provide an analog of the extension of a single-register machine, but all have essentially the some capabilities.

CHAPTER BIBLIOGRAPHY

1. Donovan, John J., System Programming, New York, Mcgraw-Hill, 1972

2. Hopgood, F. R. A., Compiling Techniques, London, Macdonald, 1967

3. Isaacson, Portia, A Compiler for This Programming Language, Department of Computer Science, 1972 North Texas State University, Denton, Texas

4. Mckeeman, W., A Compiler Generator, New Jersey, Prentice-Hall, 1970

5. Waite, W. M., Compiler Construction, New York, Springer-Verlag, 1974

CHAPTER V

OPTIMIZATION

5.1. Introduction

Optimization is the term which is used to denote the attempt by a translator to improve upon the description of the algorithm which was given by the programming language user. Optimization is most appropriate when the source language does not provide access to all of the facilities of the target computer(6).

Any general approach to code optimization is severely limited by undecidability results(1) and by the lack of definitive optimality criteria. The compiler's optimizer therefore provides improvement (relative to some cost function), rather than true optimization(6). In order to avoid undecidable equivalence questions, the improvement is carried out by applying a series of equivalence-preserving transformations to the original algorithm(1). Each transformation is based upon information gathered from some region of the source program.

In this chapter different techniques for optimization are depicted in detail.

5.2. Folding and Propagation

When the value of the operands on an expression are known to the compiler, that expression can be folded (replaced by a single value). When a variable is set to a value at compiler time, that value can be propagated (substitued for the variable) by the compiler, if the algorithm used by the compiler results in the same value that would result for evaluation at object time.

5.3. Rearrangement

The purpose of rearrangement is to reduce the amount of temporary storage required during the evaluation of an expression. This has the effect of speeding up the evaluation, because it may be possible to compute a value using only registers for temporary storage.

Some number of anonymous operands must exist simultaneously during the evaluation of a given expression. This number depends upon the order in which the components of the expression are evaluated.

The temporary storage requirements for a given expression are fixed. In order to lower the requirement, we must transform the tree structure of this expression into another which yields the same value but has a different tree

structure.

For example, with the expression of Figure 5.1, we can transform the tree structure of the expression A*B+(C+D) from Figure 5.1(a) to obtain the tree sturcture of Figure 5.1(b), which requires only a single temporary anonymous operand rather than two. This is valid only if the data type of (A*B) is the same as (C+D); otherwise, transformation operation will have to be inserted to convert data types, as from single-precision to multiple precision, or from a fixed decimal number to a floating binary number.

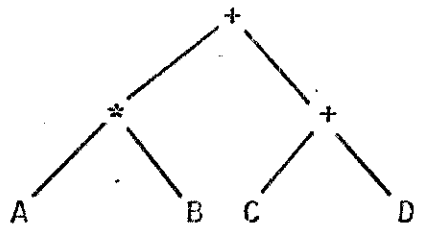
5.4. Redundant Elimination

Code is redundant if the value which it computes is already available at the point where the code occurs. The method that is used to eliminate redundancy of a expression is that of finding common sub-expressions of a expression and eliminating the excessive use of temporary storages.

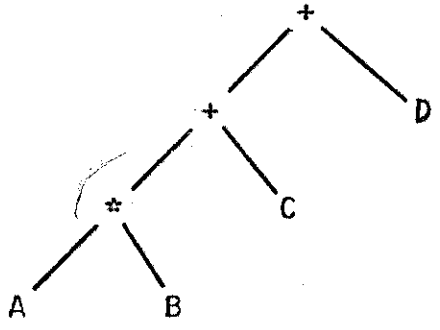
Common sub-expressions elimination is an optimization which creates a directed acyclic tree(3) rather than a structure tree to describe the program.

For example, given the expression of Figure 5.2, we can transform the tree structure of Figure 5.2(b) and create a directed acyclic tree of Figure 5.2(c) which retains only

A * B + (C + D)



(a). The Directed Tree Structure for the Expression

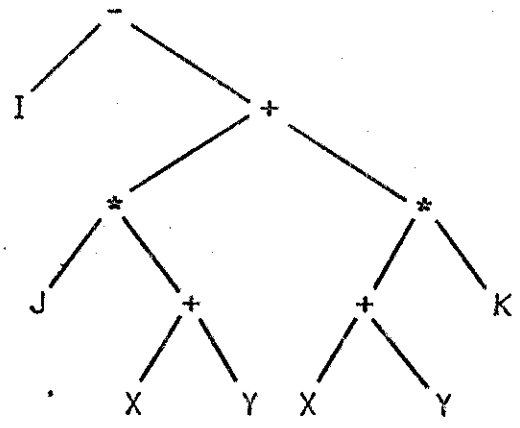


(b). The Tree Strucrure Transformated from (a).

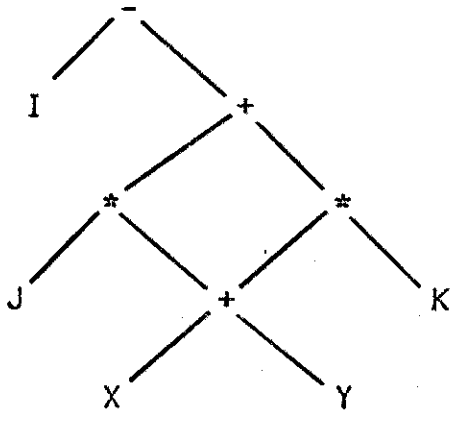
Figure 5.1. Temporary Storage Requirements

```
I - J * (X + Y) + (X + Y) * K
```

(a). An Expression With Common Sub-expression



(b). The Tree Structure for (a).



(c). The Directed Acyclic Structure for (a).

FIGURE 5.2. Eliminating a Common Sub-expression

one copy of each sub-expression.

5.5. Strength Reduction

Strength reduction is the general process of replacing an expensive (use relative more CPU cycles) operation by a cheaper (use relative less CPU cycles) one and dose not alter the value of the expression. When a value is raised to a constant power it may be possible to replace the exponent by a series of multiplications:

For example, the polynomial

```
X**4 + X**3 + X**2 + X + 1
```

may be replaced by the expression

```
X*(X*(X*(X+1)+1)+1)+1
```

Another example is that the control variable of a loop is multiplied by an expression whose value remains constant over the loop; than that multiplication may be replaced by an addition.

For example, the following code

```
DO I = 3*X+Y TO Z ;
   ......
END ;
```

```
may be replaced by
    TEMP = 3*X+Y ;
    DO I = TEMP TO Z ;
     ........
    END ;
```

5.6. Frequency Reduction

Frequency reduction attempts to move operations from one part of the program which are entered frequently to the part which are entered rarely. The most important use of this transformation is to remove invariant calculations from loops, on the assumption that the code inside a loop is executed more frequently than that surrounding the loop.

```
For example :
    DO I = 1 TO 100 ;
/* ASSUME THAT A, B, C, D ARE NOT EVALUATED */
      .........
      X = A+(B*(C+D)) ;
      .........
    END ;

may be replaced by
    X = A+(B*(C+D)) ;
    DO I = 1 TO 100 ;
```

```
      ......
      ......
END ;
```

5.7. Register Allocation

Register allocation can be performed on the basis of global flow analysis which provides the code generator with information on the future use of values currently residing in registers; this type of allocation falls in the province of optimization(2).

The so-called 'peephole optimization'(5), which avoids locally redundant fetches and stores of register contents, is an instance of intelligent code generation.

CHAPTER BIBLIOGRAPHY

1. Aho, A. V., "A Formal Approach to Code Optimization," SIGPLAN Notices, July 1970, pp. 86-100

2. Beatty, J. C., "Register Assignment Algorithm for Generation of Highly Optimized Object Code," IBM J. Res. Develop. 1974, pp. 20-39

3. Cocke, J., "Globol Common Sub-Expression Elimination," SIGPLAN Notices, July 1970, pp. 20-24

4. Hopgood, F. R. A., Compiling Techniques, London, Macdonald, 1971

5. Mckeeman, W. M., "Peephole Optimization," CACM 8, 1965, pp. 443-444

6. Waite, W. M., Compiler Construction, New York, Springer-Verlag, 1974

CHAPTER VI

A DESCRIPTION OF TPL

6.1. Introduction

TPL is a goto-less structured programming(2) language with an unusually complete set of control structure of the non-procedure variety.

CASE and LEAVE statement are among the more unusual statements types found in TPL. SELECT, CASE, LEAVE, IF-THEN-ELSE, UNLESS-THEN-ELSE, and REPEAT-WHILE staements are designed for structured programming in TPL. Functions are avaiable with zero or one argument and a single result.

The syntax is PL/1 (1) -like except there are reserved words (APPENDIX 1). A usual set of arithmetic and comparison operations on integer numbers and one-character strings are included. Arrays are available. Input and output are unformatted. Storage allocation is static.

The compiler of TPL is written in PL/1 to run on the IBM 360/370 computer. The compiler produces relocatable object code which can contain external references.

6.2. Basic Language Elements

Identifier

An indentifier is a string of characters. The first character must be alphabetic. The remaining characters can be alphabetic, numeric, or the underscore character. An identifier may not contain embedded blanks. An identifier can be any length; however, the first six characters must be unique in identifiers used as external function names. Identifiers are used as variable names, as control structure names, and as reserved words.

Constants

TPL has two types of constants. A numeric constant consists of up to five decimal digits optionally preceded by a sign. A character constant consists of one character preceded by a single quote.

Spaces

Spaces may not be embedded in identifiers or constants, but may be embedded anywhere and/or any number of spaces in the source program.

Comment

A comment is any character string started and ended with a bar character ('|'). A comment may extend over any

number of input records and may appear anywhere a space may appear.

Examples:

1. ¦ THIS IS A COMMENT ¦
2. SET A TO ¦ THIS IS A COMMENT ¦ B;

Vectors

A vector is a array which defined by naming the vector and its length in a vector declaration statement. An element of a vector is chosen by following the vector name with an expression enclosed in parenthesis.

Examples:

1. VECTOR V(10), X(50) ;
2. SET V(A+5) TO V(FUNCX(X-1)+B) ;

6.3. Statements

Statements are free form. Each statement is terminated by a semicolon. One statement can be spread over as many input records (cards) as necessary and one input record can contain several statements.

Examples:

1. SET A TO 0; SET B TO 0; SET C TO 0;
2. SET A TO A+B;

```
3. SET A TO A + B
              * C
              D / E
              ;
```

There are five types of statements in TPL. They are :

1. Declaration statements
2. Simple statements
3. Statements that start a control structure
4. Statements that end a control structure
5. Statements that cause termination of a control Structure

Type 3, 4, 5, are described later in this chapter.

6.3.1. Declaration Statement

There are two types of declaration statements:

1. EXTERNAL declaration statement
2. VECTOR declaration statement

All EXTERNAL declaration statements must preceed all other statements in the program. An EXTERNAL declaration statement specifies a list of function names which are external to the program being compiled.

Examples:

```
EXTERNAL EX1, EX2;
```

All VECTOR declaration statements must proceed all

other statements in the program except EXTERNAL declaration statements. A VECTOR statement specifies a list of arrays and their lengths.

Examples:

```
VECTOR A(10), V(20);
```

6.3.2. Simple Statement

There are three types of simple statements:

1. SET
2. INPUT
3. OUTPUT

The SET statement is the usual assignment statement.

Examples:

1. SET A TO 5;
2. SET V(X*Y+Z) TO FUNC(J)+7;

The INPUT statement causes input of the items in the list. An input list item can be a subcripted array or a simple variable.

Examples:

```
INPUT A, B(I), J, C(J);
```

The OUTPUT statement causes output of the vaules of the items in the list. An output list item can be any expression.

Examples:

```
OUTPUT A+B*C, 17, D, '=,  FUNC(X);
```

There is no control over format for either INPUT or OUTPUT statements.

6.4. Operations

Arithmetic Operations

A set of usual arithmetic operations are available in TPL. They are addition (+), subtraction (-), multiplication (*), and division (/) from left to right, except the following precedence of operations.

```
Priority 1
   - prefix minus
   + prefix plus

Priority 2
   / divide
   * multiply

Priority 3
   - infix minus
   + infix plus
```

Relational Operations

Relational operations are provided for use in IF and UNLESS statements. They are equal (=), not equal (~=), less than (<), less than or equal (<=), greater than (>), and greater than or equal (>=).

Examples:

```
1. IF A = B THEN ....
```

2. UNLESS X = Y THEN ....
3. IF CHARACTER = 'X THEN ....

6.5. Control Structures and Flow Of Control

The physical beginning of each control structure, except the source program structure, is identified by a reserved word specified to the control structure.

Examples:

1. FUNCTION
2. IF
3. UNLESS
4. SELECT
5. REPEAT

The physical end of each control structure is identified by a reserved word specific to the particular control structure.

Examples:

1. END (END OF PROGRAM)
2. END_FUNCTION
3. END_IF
4. END_SELECT
5. END_REPEAT

Any control structure except a program can be named by placing a label on the statement starting the control structure.

Example:

A: SELECT .....

```
X: IF A=B THEN ......
```

Control structures can be nested to any level. On the compiler listing there is a level count which starts at one with the first statement in the source program. The count is incremented each time a control structure is started and decremented when the corresponding end is encountered. Any control structure may appear at any level except the program and the function. The program is at level zero. All function definition must be at level one. Normal exit from any control structure is made by executing the statement physically ending the control structure. Premature exit can be made from a control structure by executing a LEAVE statement.

6.5.1. Program Control Structure

There is no statement which specifically identifies the beginning of the program control structure. The END statement identifies the physical end of the source program. A object program can be terminated by executing a STOP statement anywhere in the program or by executing the END statement.

Example:

```
SET X TO 5;
......
IF X=Y THEN STOP;
```

```
      .....
      END_IF;
      .....
      END;
```

6.5.2. FUNCTION Control Structure

A FUNCTION control structure must be at level one and prior to any reference to the function. The FUNCTION statement identifies the start of a function. The FUNCTION statement should have a label which becomes the function name.

All variables defined in the program can be referenced from within the function and all variables defined in the function can be referenced in the main program (storage allocation is static).

Example:

```
FUNC: FUNCTION;
      .......
SET I TO I+1;
      .....
IF A=B THEN LEAVE FUNC;
      ......
END_FUNCTION;
```

A function is invoked by using the name in an expression. The function will return a value which will be used in the place of function name in the evaluation of the expression. The value returned by the function will be the

value assigned to the function name within the function, within the function the function name is treated as a simple variable in all contexts.

```
FACTOR : FUNCTION(N);
    SET FACTOR TO 1 ;
    REPEAT SET I TO FROM 1 TO N;
           SET FACTOR TO FACTOR *.I ;
    END_REPEAT ;
END_FUNCTION;
    .
    .
SET X TO FACTOR(Q+R*S);
    .
    .
 END;
```

The END_FUNCTION statement physically ends a function. If the END_FUNCTION is executed, control transfers to the point at which the function is invoked. A function can also be exited by a LEAVE statement.

6.5.3. SELECT Control Structure

The SELECT statement causes a choice to made among several possible cases. Each case consists a unique sequence of simple statement and/or control structure.

The choice is made in the following way :

1. The expression on the SELECT statement is evaluated.

2. Each CASE statement expression is evaluated in

turn until all cases are exhausted or until the select expression equals a case expression in value. The matched case is chosen as the one to be executed. If no case is matched, none is executed.

3. After the chosen case is executed, if any, execution continues at the statement following the END_SELECT.

The domain (range of statements) for a given CASE is from the statement follows the CASE statement to the statement which precedes the next CASE statement or the END_SELECT statement.

Example :

```
1. SELECT I;
       CASE 1;
         ...
       CASE 2 ;
         ...
   END_SELECT;

2. SELECT A+B ;
       CASE X+Y ;
           ......
        CASE 5 ;
           ......
        CASE X+Y+FUN(I) ;
           .....
   END_SELECT ;

3. SELECT CHARACTER ;
     CASE 'A ;
     ........
     CASE '3 ;
```

```
        .........
        CASE '# ;
        .........
    END_SELECT ;
```

6.5.4. IF Control Structure

The IF control structure is started by the IF statement by the END_IF statement.

Examples:

1. IF A=B THEN .....
2. IF A+5 = B/C THEN .....

Any number of simple statement and/or control structures may follow the reserved word THEN.

The IF control structure is terminated by the reserved word END_IF or the reserved word ELSE. If the ELSE is used, it may be followed by a sequence of simple statement and/or control structure terminated by the reserved word END_IF.

Examples:

```
IF A=B THEN
SET X TO Z ;
.....
END_IF ;
ELSE
SET Y TO Z
.....
END_IF ;
```

6.5.5. UNLESS Control Structure

Contrary to the IF statement, in the UNLESS statement the sequence of simple statement and/or control structures following the reserved word THEN is executed if the conditional expression is false. If the conditional expression is true, the statements following the ELSE are executed.

Examples:

```
1. UNLESS X=Y THEN SET X TO A+B ;
                   ELSE SET X TO C+D ;
   END_UNLESS ;
```

The UNLESS control structure is terminated by the reserved word END_UNLESS statement.

6.5.6. REPEAT Control Structure

The REPEAT control structure is used for forming a loop around a sequence of simple statements and/or control structures. The REPEAT control structure is terminated by the END_REPEAT statements. When the END_REPEAT statement is encountered during execution, control returns to the beginning of the REPEAT control structure.

The WHILE and SET are two optional phrases for the REPEAT statement. The WHILE phrase supplies a conditional

expression. The conditional expression is evaluated each time the top of the REPEAT is encountered. The sequence of simple statement and/or control structures of the REPEAT is executed only if the conditional expression evaluated to true. If the conditional expression evaluate to false, the REPEAT control structure is terminated and the next statement to be executed will be the one following the END_REPEAT statement.

Example :

```
REPEAT WHILE X<100 ;
  .....
END_REPEAT ;
```

The SET phrase specifies a variable referred to as the repeat-variable. Each time the top of the REPEAT statement in encountered, the REPEAT variable is assigned the next value specified by the SET phrase.

Example:

```
1. REPEAT SET I TO 1, 5, -8, 'X ;
     .....
   END_REPEAT ;

2. REPEAT SET J TO X+Y, X*Y ,
          FROM X+5 TO Y*3 BY A/3 ,
          FROM 10 TO -100 BY -3 ;
     .....
   END_REPEAT ;
```

```
3. REPEAT WHILE A>B SET X TO FROM 1 TO 500
   BY 5;
   ............
   END_REPEAT;
```

The list of values follows the reserved word TO is the SET phrase. A list item can be expression or a FROM phrase. The FROM, TO, and BY, values can be expressions which evaluate to either positive or negative values, if the BY value is not specified, the default value is one.

6.5.7. Termination Statements

The STOP and the LEAVE are the termination statements other than the END statements. The STOP statement causes the program to be halted and no further statements will be executed. The LEAVE statement names a control structure. When the LEAVE statement is executed, the named control structure will be terminated. A LEAVE statement at level N can cause termination of any control structure from level 1 through N-1.

Example:

```
P: REPEAT WHILE X<1000 ;
      ....
   IF X=50 THEN LEAVE P ;
   IF X = 60 THEN STOP;
      ....
   END_REPEAT ;
```

CHAPTER BIBLIOGRAPHY

1. International Business Machines, PL/1 Language Specifications, IBM Form C28-6571, 1972

2. Yourdon, I., "A Brief Look at Structured Progrmming And Top-Down Design", Modern Data, June 1974, pp. 30-34

CHAPTER VII

DATA BASES

7.1. Introduction

The data bases which are related to the generation of machine code are depicted in detail in this chapter. The manner in which the various data bases are used is shown by Figure 1.1 in Chapter I, which should now be refered to.

7.2. SOURCEX

SOURCEX is an external file built by the SCANNER and input by the PARSER. It contains a copy of the card input which is the source program of TPL.

7.3. TOKENS

TOKENS is an external file built by the SCANNER and input by the PARSER. The entries in the file TOKENS are numbers which identify a certain token-type followed by a variable which may be a pointer to a symbol list entry or a number or a null value for those tokens not requiring operands. Figure 7.1 shows the three types of entries.

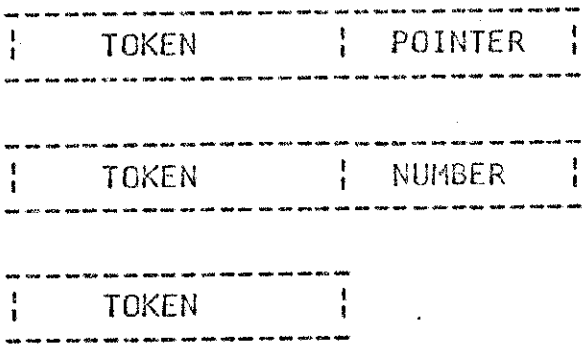


Figure 7.1. ----- TOKEN ENTRIES

7.4. Symbol List

The symbol list is a one-way linked list. The nodes of the list are dynamically allocated as they are needed. Each node contains six fields. Nodes are variable in size since the NAME field is variable in size. Figure 7.2. Shows the layout of a node.

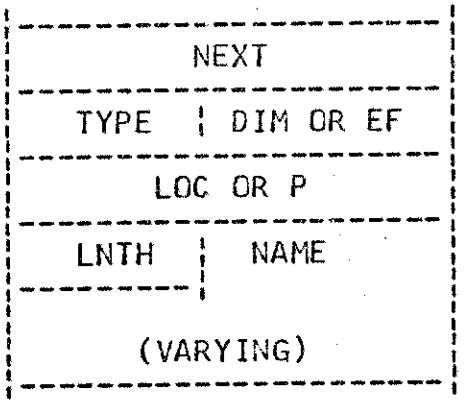


FIGURE 7.2. -- A SYMBOL NODE

The field called NEXT is always a pointer to the next entry in the one-way chain, except that the last entry in

the symbol list has a value of NULL, a special PL/1(1) value.

The field called TYPE is an integer value which indicates the type of node. TYPE has values for each phase of the compilation. The types of nodes built by the SCANNER are:

1. Label
2. Indentifier
3. Constant

The types of nodes recognized by the PARSER are:

1. Label
2. Vector
3. Simple variable
4. Constant
5. Function
6. Active function
7. Dummy argument

The types of nodes recognized by the code generator are:

1. Label
2. Define label
3. Vector
4. Simple variable
5. Constant
6. Function
7. Defined function

The field called DIM or EF is filled by the PARSER. For a vector DIM contains the length of the vector. For a function EF contains a flag indicating whether or not the function is external. The field called LOC or P is used by

the PARSER and CODEGEN. PARSER uses P when compiling a function. If the node is an active function, P is a pointer which points to the simple variable where the function value is stored. If the node is a dummy argument, P is a pointer points to the simple variable where the dummy argument is stored. During the storage allocation stage CODEGEN sets LOC for vectors, simple variables, or constant. During the machine code generation CODEGEN sets LOC for labels or functions.

The field called LNTH is entered when the code is allocated. It contains a number which is the length of the NAME field in characters.

The field called NAME is entered when the word is allocated. It contains an identifier which names a label, a vector, a simple variable, or a function; or it contains the character string which defins a constant.

Nodes are allocated by every phase of the compiler. The SCANNER builds nodes for every identifier, label, or constant found in the source. The PARSER builds nodes for temporary storage and system-generated labels.

The PARSER also builds nodes for function values and dummy argument values. Code generator builds a node for a system generated label which is defined at the beginning of the executable code.

7.5. THREECD

THREECD is an external file built by the PARSER and input to the code generator (CODEGEN). The entries in the file THREECD are numbers which identify the type of code optionally followed by one to three pointers or a number. All entries in the file have fields for three pointers. If a field is not used for a particular code, it is set to NULL (a special PL/1 value). Figure 7.3 shows the five types of entries in THREECD.

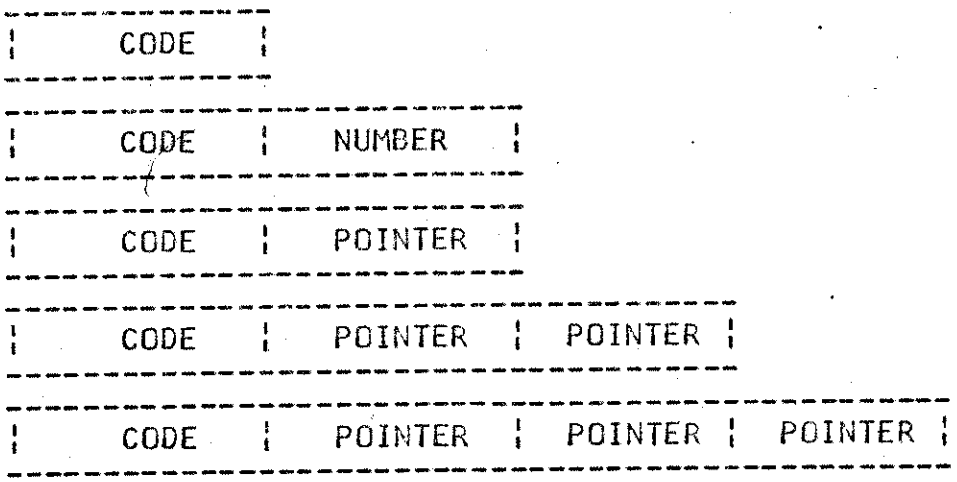


Figure 7.3. --- THREECD ENTRIES

Appendix 2 defines the various three address and lists their respective operands, if any.

7.6. RFAIRCD

RFAIRCD is an external file built by the code generator and input to the LOADER, which transforms the relocatable object code module into an executable module.

RFAIRCD contains the relocatable object code along with the loader control information necessary to load the program by the LOADER. Each entry in RFAIRCD consists of a loader control character and optionally an operand. The operand can be a number or a name.

Appendix 4 defines the various loader control codes and lists their respective operands, if any.

7.7. ABSBIN

ABSBIN is an external file output from the relocatable loader and input to the Fairchild F24 computer, which is the target computer. ABSBIN contains the absolute Fairchild F24 machine codes which are executed by the Fairchild F24 computer hardware or by the simulator.

7.7. ERFORSS and ERRORSP

ERRORSS and ERRORSP are external files built by the SCANNER and the PARSER respectively. Both files contain error messages generated during the compilation of a TPL

program. Both files are merged into the output listing. Each entry in the files consists of a line number on which the error occurred and up to 100 characters of text describing the error.

CHAPTER BIBLIOGRAPHY

1. Gries, David, Compiler Construction for Digital Computers, New York, John-Wiley, 1971

2. International Business Machines, PL/1 Language Specifications, IBM Form C28-6571, 1974

CHAPTER VIII

SUMMARY AND CONCLUSION

8.1. Testing

SCANNER, PARSER, and CODEGEN will generate a debug listing when requested. The debug output options are controlled from the source input stream. The options can be turned on or off and changed at any point by one source card with '??' on column 1 and 2. There are several levels of debug output that can be selected. Figure 8.1 shows the output for each level. Samples of debug output are shown in Appendix 7-13.

This compiler was tested in several stages:

Stage 1: SCANNER output was hand-checked using the debug output.

Stage 2: PARSER output was hand-checked using the debug output. The PARSER and SCANNER were integrated for this stage.

Stage 3: CODEGEN output was hand-checked using the debug output. CODEGEN was integrated with PARSER and SCANNER for this stage.

Stage 4: LISTER output was hand-checked using

Figure 8.1 Debug Output Options

PHASE	LEVEL	OUTPUT
SCANNER	0	None.
	1	Source, Errors and tokens.
	2	Source, errors, tokens, and detail trace of finite-state machine.
PARSER	0	None
	1	Source, errors, and three-address codes.
	2	Source, errors, three-address codes, tokens, and subroutine trace.
CODEGEN	0	Source and Fairchild F24 assembly language listing.
	1	Source, F24 assembly language listing, and object code.
	2	Source, F24 assembly language listing, object code, and three-address codes.

debug output. LISTER was integrated with PARSER, SCANNER, and CODEGEN for this stage.

Stage 5: LOADER output was hand-checked using debug output. LOADER was integrated with PARSER, SCANNER, LISTER and CODEGEN for this stage.

Stage 6: the TPL compiler was integrated with the LOADER and the Fairchild F24 simulator for this stage. TPL programs were compiled and executed. The output from the Fairchild F24 simulator was hand-checked.

Test programs for stage 1 though 4 were designed to test the compiler rather than go into execution. The programs were designed to test each major feature of the compiler on a statement by statement basis. Programs for stage 5 and 6 were designed to produce output that could be intepreted as correct or incorrect depending on whether or not the interpretation had been correct.

Samples of testing programs are shown in Appendix 14-15.

8.2. Choice of Implementation Language

It is believed that the compiler presented in this thesis is best implemented in a high-level programming language. This language needs not support dynamic arrays or

data-directed input-output; in fact, any feature requiring a complex run-time environment is a liability.

The language should make it easy to produce modular software systems, that is, a system containing a large percentage of components. In support of this notion, the language should posses a reasonable, efficient subroutine call facility. Full call-by-name(2) is certainly not needed; a simple call-by-reference(2) would be satisfactory. Also a programmer should be able to get at out-of-module variables, say in a shared data base, without having to go to any great trouble.

A language (especially an implementation language) can not afford to hide very many attributes of the target-machine hardware and monitor, as it is difficult and dangerous to predict which features will never be useful. This implies that perhaps the best implementation language is the one customer-tailored for the target machine environment that it must routinely deal with. So we have two alternatives:

1) Define a reasonably machine-independent language and hope for the best with regard to monitor interfacing and complete instruction-set utilization, or

2) Define a language suitable for a fairly large family of machine (like UNIVAC 1100s or the IBM 360/370 line), allow the language to become rather machine-dependent, then count on defining a new one when it is time to cross family lines.

Of the two, the first is more attractive from a mobility standpoint. The second is more attractive to those wishing to exercise very close control over hardware and monitor.

8.3. Evaluation

Probably most compilers have been written with "conversation routines" embedded in and/or called by the lexical analyzer(1). A call to such routines usually occures immediately after each token is discovered. Such routines usually convert digit strings to some "internal" integer representation, for example, or if a decimal point is encountered, to some representation of real numbers; or they may interpret special characters inside string constants; etc.

All too often the "internal" representation chosen is that of the machine on which the language is initially being implemented, with little or no thought that the compiler

might later be moved to another machine or be modified to generate code for a different machine. Such decisions are usually made because of "efficiency".

It is desirable to keep the entire front-end (scanner and parser) of the compiler indpendent of target representations, if possible. If constants are translated to target representations by the lexical analyzer, tables of several different types usually must be maintained and some processors that do not need to know those representations, nonetheless must be programmed in term of, or around them. For example, if constants are converted and an error message should relate to one, it must be converted back to source representations for printing.

In summary, the author suggests that the scanner and the parser should be independent of target representations, if possible.

8.4. Future Research

In the author's opinion, future research activities lie in the following areas:

1) In the optimization for the machine code (CHAPTER V), the author described various methods for

generating optimized machine code. Further extensions of the present research might include investigation of other mechanism on which optimized machine code can be generated.

2) In the design of the TPL compiler, the author chose the static storage management for storage allocation. Run-time storage allocation may be added in the future.

3) In the description of error recovery in syntax analysis (CHAPTER II), the author has presented decriptions of error recovery, further research might include development of repair techniques to transform programs containing syntactic errors into programs that are both valid and similar.

CHAPTER BIBLIOGRAPHY

1. Early, J. C., "An Efficient Context-Free Parsing Algorithm," Comm. ACM 13, 1970, pp. 94-102

2. Waite, W. M., Compiler Construction, New York, Springer-Verlag, 1974

APPENDIX 1

LIST OF RESERVED WORDS IN TPL

BY	IF
CASE	INPUT
ELSE	LEAVE
END	OUTPUT
END_FUNCTION	REPEAT
END_IF	SELECT
END_REPEAT	SET
END_SELECT	STOP
END_UNLESS	THEN
EXTERNAL	TO
FOR	UNLESS
FROM	VECTOR
FUNCTION	WHILE

APPENDIX 2 THREE-ADDRESS CODES

CODE	OPERAND AND CODE DEFINITION
ADD	P1,P2,P3. add P1 to P2 and store the result in P3.
CALL0	P1,P2. the function, P1, is invoked and the returned result is store in P2.
CALL1	P1,P2,P3. the function, P1, is invoked with argument, P2, and the returned result is stored in P3.
COMPARE_EQ	P1,P2,P3. if P1 is equal to P2, P3 is set to true; otherwise, P3 is set to false.
COMPARE_GT	P1,P2,P3. if P1 is greater than to P2, P3 is set to true; otherwise, P3 is set to false.
COMPARE_GTEQ	P1,P2,P3. if P1 is greater than or equal to P2, P3 is set to true; otherwise, P3 is set to false.
COMPARE_LT	P1,P2,P3. if P1 is less than P2, P3 is set to true; otherwise, P3 is set to false.
COMPARE_LTEQ	P1,P2,P3. if P1 is less than or equal to P2, P3 is set to true; otherwise, P3 is set to false.
COMPARE_NEQ	P1,P2,P3. if P1 is not equal to P2, P3 is set to true; otherwise, P3 is set to false.
DOTIMES	P1,P2,P3. multiply P1 by P2 and store the result in P3.
DODIVIDE	P1,P2,P3. divide P1 by P2 and store the result in P3.

APPENDIX 2 THREE-ADDRESS CODES (CONTINUED)

CODE	OPERAND AND CODE DEFINITION
END_TAC	No operand indicates the end of three-address codes.
ENTER0	P1. enter a function by reserving a location P1 for the return address.
ENTER1	P1,P2. enter a function by reserving a location P1 for the return address and the argument value at P2, the dummy argument.
EXIT	P1. exit a function by returning the function value, P1.
GO	P1. branch to P1.
GOIF_FALSE	P1,P2. if P1 is false, branch to P2; otherwise, execute next code.
GOIF_TRUE	P1,P2. if P1 is true, branch to P2; otherwise, execute next code.
IN	P1. input P1.
INDEX	P1,P2,P3. calculate the address of P1 subscripted by P2. Store the resulting address in P3.
LOCDEF	P1. label P1 is defined at this location.
IN	P1. input P1.

APPENDIX 2 THREE-ADDRESS CODES (CONTINUED)

CODE	OPERAND AND CODE DEFINITION
MOV	P1,P2. move P1 to P2.
MOV_I	P1,P2. move P1 to the location defined by the address in P2.
MOVI	P1,P2. move the value at the location difined by the address in P1 to P2.
NEGATE	P1,P2. negate P1 and store the result in P2.
OUT	P1. output P1.
RETJMP	P1. return indirect through P1
STOPCD	no operand stop
SUBSTRACT	P1, P2, P3 substract P2 from P1 and store the result in P3.
XDEF	P1. the entry point to function P1 is defined at P1.

APPENDIX 3

BNF SPECIFICATION OF THIS PROGRAMMING LANGUAGE

```
<PROGRAM> ::= <PROGRAM_TAIL>
            | <PROGRAM_HEAD> <PROGRAM_TAIL>
<PROGRAM_HEAD> ::= <EXTERNAL_DECLARATION_LIST>
                   <VECTOR_DECLARATION_LIST>
<PROGRAM_TAIL> ::= <SAME_LEVEL_SEQUENCE> END;
<EXTERNAL_DECLARATION_LIST> ::= <EXTERNAL_DECLARATION_STATEMEN
            | <EXTERNAL_DECLARATION_LIST>
                   <EXTERNAL_DECLARATION_STATEMENT>
<VECTOR_DECLARATION_LIST> ::= <VECTOR_DECLARATION_STATEMENT>
            | <VECTOR_DECLARATION_LIST>
                   <VECTOR_DECLARATION_STATEMENT>
<EXTERNAL_DECLARATION_STATEMENT> ::= EXTERNAL <EXTERNAL_LIST>
<EXTERNAL_LIST> ::= <FUNCTION_NAME>
            | <EXTERNAL_LIST> , <FUNCTION_NAME>
<VECTOR_DECLARATION> ::= VECTOR <VECTOR_LIST> ;
<VECTOR_LIST> ::= <VECTOR_SPECIFICATION>
            | <VECTOR_LIST> , <VECTOR_SPECIFICATION>
<VECTOR_SPECIFICATION> ::= <VECTOR> ( <LENGTH> )
<VECTOR> ::= <IDENTIFIER>
<LENGTH> ::= <CONSTANT>
<SAME_LEVEL_SEQUENCE> ::= <TERMINATION_STATEMENT>
            | <SEQUENTION_BLOCK>
            | <SEQUENTIAL_BLOCK> <TERMINATION_STATEMENT>
<TERMINATION_STATEMENT> ::= <STOP_STATEMENT>
            | <LEAVE_STATEMENT>
<SEQUENTIAL_BLOCK> ::= <CONTROLLED_BLOCK>
            | <SEQUENTIAL_BLOCK> <CONTROLLED_BLOCK>
<CONTROLLED_BLOCK> ::= <SIMPLE_STATEMENT>
            | <FUNCTION_BLOCK>
            | <SELECT_BLOCK>
            | <REPEAT_BLOCK>
            | <IF_BLOCK>
            | <UNLESS_BLOCK>
            | NULL
<SIMPLE_STATEMENT> ::= <SET_STATEMENT>
            | <INPUT/OUTPUT_STATEMENT>
<FUNCTION_BLOCK> ::= <FUNCTION_HEADER> <SAME_LEVEL_SEQUENCE>
                     END_FUNCTION;
<SELECT_BLOCK>  ::= <SELECT_HEADER> <CASE_LIST> END_SELECT;
<REPEAT_BLOCK>::=<REPEAT_HEADER><SAME_LEVEL_SEQUENCE>END_REPEA
<IF_BLOCK> ::= <IF_HEADER> <SAME_LEVEL_SEQUENCE> <IF_TAIL>
<UNLESS_BLOCK>  ::= <UNLESS_HEADER> <SAME_LEVEL_SEQUENCE>
                    <UNLESS_TAIL>
```

APPENDIX 3

BNF SPECIFICATION OF THIS PROGRAMMING LANGUAGE (CONTINUED)

```
<SET_STATEMENT> ::= SET <TARGET> TO <EXPRESSION> ;
<TARGET> ::= <VARIABLE>
<EXPRESSION> ::= <TERM>
                 | <EXPRESSION> <ADD/SUBTRACT_OPERATOR> <TERM>
<TERM> ::= <SIGNED_OPERAND>
         | <TERM> <MULTIPLY/DIVIDE_OPERATOR> <SIGNED_OPERAND>
<SIGNED_OPERAND> ::= <ADD/SUBTRACT_OPERATOR> <SIGNED_OPERAND>
                 | <OPERAND>
<OPERAND> ::= <CONSTANT>
              | <VARIABLE>
              | <FUNCTION_REFERENCE>
              | ( <EXPRESSION> )
<VARIABLE> ::= <SIMPLE_VARIABLE>
               | <SUBSCRIPTED_VARIABLE>
<SIMPLE_VARIABLE> ::= <IDENTIFIER>
<SUBSCRIPTED_VARIABLE> ::= <IDENTIFIER> ( <EXPRESSION> )
<INPUT/OUTPUT_STATEMENT> ::= <INPUT_STATEMENT>
                           | <OUTPUT_STATEMENT>
<INPUT_STATEMENT> ::= INPUT <INPUT_LIST> ;
<OUTPUT_STATEMENT ::= OUTPUT <OUTPUT_LIST> ;
<INPUT_LIST> ::= <TARGET>
               | <INPUT_LIST> , <TARGET>
<OUTPUT_LIST> ::= <EXPRESSION>
                | <OUTPUT_LIST> , <EXPRESSION>
<FUNCTION_HEADER> ::= <FUNCTION_HEADER> ;
                    | <FUNCTION_HEAD> EXTERNAL;
<FUNCTION_HEAD> ::=<FUNCTION_NAME> : FUNCTION(<DUMMY_ARGUMENT>
                  | <FUNCTION_NAME> : FUNCTION ;
<FUNCTION_NAME> ::= <IDENTIFIER>
<DUMMY_ARGUMENT> ::= <IDENTIFIER>
<FUNCTION_REFERENCE> ::= <FUNCTION_NAME>
                       | <FUNCTION_NAME> ( <EXPRESSION> )
<SELECT_HEADER> ::= SELECT <EXPRESSION> ;
                  | <SELECT_NAME> SELECT <EXPRESSION> ;
<SELECT_NAME> ::= <IDENTIFIER>
```

APPENDIX 3

BNF SPECIFICATION OF THIS PROGRAMMING LANGUAGE (CONTINUED)

```
<CASE_LIST> ::= <CASE>
              | <CASE_LIST> <CASE>
              | NULL
<CASE> ::= CASE <EXPRESSION> ; <SAME_LEVEL_SEQUENCE>
<REPEAT_HEADER> ::= REPEAT <REPEAT_SPECIFICATION> ;
              | REPEAT_NAME: REPEAT <REPEAT_SPECIFICATION>
<REPEAT_NAME> ::= <IDENTIFIER>
<REPEAT_SPECIFICATION> ::= <SET_PHRASE>
              | <WHILE_PHRASE>
              | <WHILE_PHRASE> <SET_PHRASE>
<WHILE_PHRASE> ::= WHILE <CONDITION>
<SET_PHRASE> ::= SET <SET_VARIABLE> TO <SET_LIST>
<SET_LIST> ::= <VALUE_LIST>
              | <SET_LIST> , <VALUE_LIST>
<VALUE_LIST> ::= <EXPRESSION>
              | <RANGE_SPECIFICATION>
<RANGE_SPECIFICATION> ::= FROM <START> TO <FINISH>
              | FROM <START> TO <FINISH> BY <INCREMENT>
<START> ::= <EXPRESSION>
<FINISH> ::= <EXPRESSION>
<INCREMENT> ::= <EXPRESSION>
<SET_VARIABLE> ::= <TARGET>
<IF_HEADER> ::= IF <CONDITION> THEN
              | <IF_NAME> : IF <CONDITION> THEN
<IF_NAME> ::= <IDENTIFIER>
<IF_TAIL> ::= END_IF;
              | ELSE <SAME_LEVEL_SEQUENCE> END_IF;
<UNLESS_HEADER> ::= UNLESS <CONDITION> THEN
              | <UNLESS_NAME> : UNLESS <CONDITION> THEN
<UNLESS_NAME> ::= <IDENTIFIER>
<UNLESS_TAIL> ::= END_UNLESS;
              | ELSE <SAME_LEVEL_SEQUENCE> END_UNLESS;
<CONDITION> ::= <EXPRESSION> <RELATIONAL_OPERATOR> <EXPRESSION
<LEAVE_STATEMENT> ::= LEAVE <BLOCK_NAME> ;
<BLOCK_NAME> ::= <FUNCTION_NAME>
              | SELECT_NAME>
              | <REPEAT_NAME>
              | <IF_NAME>
              | <UNLESS_NAME>
<STOP_STATEMENT> ::= STOP;
```

APPENDIX 4

LOADER CONTROL CODES

CODE	OPERAND	CODE DEFINITION
A	Number	Load the number at current load address. Increment the load address.
R	Number	Relocate the number by adding to the bias. Load the relocated number. Increment the load address.
D	Number	Define a label at this location by filling in a chain of references starting at number plus bias.
X	Number	Load the number at the current load address. This instruction is a reference to an external function. Do not increment the load address.
blank	Name	There is at this location a reference to an external function. If the function has been loaded previously, enter its address in the address of the instruction at this loaction; otherwise, link this reference to the reference chain. Increment the load address.
Z	Name	The entry point to an external function is defined at this location. Fill in all previous references and enter its name in the external symbol directory.
V		Reserve number storage locations at this location. Add number to the load address.
E	None	Set bias to load address. This is the end of this program.
*	None	This is the end of this load procedure.

APPENDIX 5
TPL COMPILER DIAGNOSTIC MESSAGES (1)

```
LINE     LEVEL

0001         1        ||
0002         1        | ERROR MASSAGES IN TPL  |
0003         1        ||
0004         1         REPEAT SET I FROM 1  TO 10;
**********MISSING TO SUPPLIED
0005         2            IF A > 0 THEN SET A TO 10;
0006         3            IF A > 20 THEN SET A TO 20 ;   END_IF;
0007         3            UNLESS A ¬ 30 THEN SET A TO 40 ;
**********NO '=' FOLLOWING '¬'
0008         4          END_REPEAT;
**********MISSING END_UNLESS SUPPLIED
**********MISSING END_IF SUPPLIED
0009         1          END_IF;
**********EXTRA OR UNRECOGNIZABLE STATEMENT IS IGNORED
0010         1        LEAVE NO_SUCH_BLOCK;
**********LEAVE OF INACTIVE BLOCK INVALID
0011         1         IF A ¬>< B THEN STOP;  END_IF;
**********MISSING END IS SUPPLIED
```

APPENDIX 6
TPL COMPILER DIAGNOSTIC MESSAGES (2)

```
LINE     LEVEL

0001        1      ||
0002        1      | ERROR MESSAGES IN TPL |
0003        1      ||
0004        1        VECTOR  Z(0);
**********INVALID VECTOR SIZE SET TO 1
0005        1        SET A12 TO 12A;
**********SEMICOLON NOT FOUND WHERE EXPECTED
0006        1        SET B TO 5*B+(C-D));
**********SEMICOLON NOT FOUND WHERE EXPECTED
0007        1         WHAT IS THIS ?
**********ILLEGAL CHARACTER
**********EXTRA OR UNRECOGNIZABLE STATEMENT IS IGNORED
0008        1         IF A = B THEN SET C TO 10;
0009        1         REPEAT SET I FROM 1 TO 10;
**********MISSING TO SUPPLIED
0010        2           SET J TO J + 1;
0011        2         END;
**********MISSING END_REPEAT SUPPLIED
```

```
OUTN TOKEN  DEBUG                    0
OUTN TOKEN  DEBUG                    0
OUTN TOKEN  LINE_NUMBER              2
   2       SET A TO 1 ;
OUT TOKEN  SET
OUTP TOKEN  IDENTIFIER      A
OUT TOKEN  TO
OUTP TOKEN  CONSTANT        1
OUT TOKEN  SEMICOLON
OUTN TOKEN  LINE_NUMBER              3
   3       SET B TO A * 3 + 2;
OUT TOKEN  SET
OUTP TOKEN  IDENTIFIER      B
OUT TOKEN  TO
OUTP TOKEN  IDENTIFIER      A
OUT TOKEN  MULTIPLY
OUTP TOKEN  CONSTANT        3
OUT TOKEN  PLUS
OUTP TOKEN  CONSTANT        2
OUT TOKEN  SEMICOLON
OUTN TOKEN  LINE_NUMBER              4
   4       IF B > A THEN SET C TO A ;   END_IF;
OUT TOKEN  IF
OUTP TOKEN  IDENTIFIER      B
OUT TOKEN  GT
OUTP TOKEN  IDENTIFIER      A
OUT TOKEN  THEN
OUT TOKEN  SET
OUTP TOKEN  IDENTIFIER      C
OUT TOKEN  TO
OUTP TOKEN  IDENTIFIER      A
OUT TOKEN  SEMICOLON
OUT TOKEN  END_IF
OUT TOKEN  SEMICOLON
OUTN TOKEN  LINE_NUMBER              5
   5       REPEAT SET C TO FROM 1 TO B ;
OUT TOKEN  REPEAT
OUT TOKEN  SET
OUTP TOKEN  IDENTIFIER      C
OUT TOKEN  TO
OUT TOKEN  FROM
OUTP TOKEN  CONSTANT        1
OUT TOKEN  TO
OUTP TOKEN  IDENTIFIER      B
OUT TOKEN  SEMICOLON
OUTN TOKEN  LINE_NUMBER              6
   6       IF C = 2 * A THEN SET B TO C ;  END_IF;
OUT TOKEN  IF
OUTP TOKEN  IDENTIFIER      C
OUT TOKEN  EQUAL
OUTP TOKEN  CONSTANT        2
OUT TOKEN  MULTIPLY
OUTP TOKEN  IDENTIFIER      A
OUT TOKEN  THEN
OUT TOKEN  SET
OUTP TOKEN  IDENTIFIER      B
OUT TOKEN  TO
OUTP TOKEN  IDENTIFIER      C
OUT TOKEN  SEMICOLON
```

APPENDIX 8
TRACE LEVEL 2 OF SCANNER

```
OUTN TOKEN  DEBUG                    0
OUTN TOKEN  DEBUG                    0
OUTN TOKEN  LINE_NUMBER              2
   2       SET A TO 1 ;
LN =    2   STATE =  1   CHAR =      CHAR_TYPE =  5   TOKEN =
LN =    2   STATE =  1   CHAR =      CHAR_TYPE =  5   TOKEN =
LN =    2   STATE =  1   CHAR =      CHAR_TYPE =  5   TOKEN =
LN =    2   STATE =  1   CHAR =      CHAR_TYPE =  5   TOKEN =
LN =    2   STATE =  1   CHAR = S    CHAR_TYPE =  1   TOKEN =
LN =    2   STATE =  2   CHAR = E    CHAR_TYPE =  1   TOKEN = S
LN =    2   STATE =  2   CHAR = T    CHAR_TYPE =  1   TOKEN = SE
LN =    2   STATE =  2   CHAR =      CHAR_TYPE =  5   TOKEN = SET
OUT TOKEN  SET
LN =    2   STATE =  1   CHAR =      CHAR_TYPE =  5   TOKEN = SET
LN =    2   STATE =  1   CHAR = A    CHAR_TYPE =  1   TOKEN = SET
LN =    2   STATE =  2   CHAR =      CHAR_TYPE =  5   TOKEN = A
OUTP TOKEN  IDENTIFIER     A
LN =    2   STATE =  1   CHAR =      CHAR_TYPE =  5   TOKEN = A
LN =    2   STATE =  1   CHAR = T    CHAR_TYPE =  1   TOKEN = A
LN =    2   STATE =  2   CHAR = O    CHAR_TYPE =  1   TOKEN = T
LN =    2   STATE =  2   CHAR =      CHAR_TYPE =  5   TOKEN = TO
OUT TOKEN  TO
LN =    2   STATE =  1   CHAR =      CHAR_TYPE =  5   TOKEN = TO
LN =    2   STATE =  1   CHAR = 1    CHAR_TYPE =  2   TOKEN = TO
LN =    2   STATE =  3   CHAR =      CHAR_TYPE =  5   TOKEN = 1
OUTP TOKEN  CONSTANT       1
LN =    2   STATE =  1   CHAR =      CHAR_TYPE =  5   TOKEN = 1
LN =    2   STATE =  1   CHAR = ;    CHAR_TYPE =  4   TOKEN = 1
LN =    2   STATE =  7   CHAR =      CHAR_TYPE =  5   TOKEN = ;
OUT TOKEN  SEMICOLON
LN =    2   STATE =  1   CHAR =      CHAR_TYPE =  5   TOKEN = ;
OUTN TOKEN  LINE_NUMBER              3
   3       SET B TO A * 3 + 2;
LN =    3   STATE =  1   CHAR =      CHAR_TYPE =  5   TOKEN = ;
LN =    3   STATE =  1   CHAR =      CHAR_TYPE =  5   TOKEN = ;
LN =    3   STATE =  1   CHAR =      CHAR_TYPE =  5   TOKEN = ;
LN =    3   STATE =  1   CHAR =      CHAR_TYPE =  5   TOKEN = ;
LN =    3   STATE =  1   CHAR = S    CHAR_TYPE =  1   TOKEN = ;
LN =    3   STATE =  2   CHAR = E    CHAR_TYPE =  1   TOKEN = S
LN =    3   STATE =  2   CHAR = T    CHAR_TYPE =  1   TOKEN = SE
LN =    3   STATE =  2   CHAR =      CHAR_TYPE =  5   TOKEN = SET
OUT TOKEN  SET
LN =    3   STATE =  1   CHAR =      CHAR_TYPE =  5   TOKEN = SET
LN =    3   STATE =  1   CHAR = B    CHAR_TYPE =  1   TOKEN = SET
LN =    3   STATE =  2   CHAR =      CHAR_TYPE =  5   TOKEN = B
OUTP TOKEN  IDENTIFIER     B
LN =    3   STATE =  1   CHAR =      CHAR_TYPE =  5   TOKEN = B
LN =    3   STATE =  1   CHAR = T    CHAR_TYPE =  1   TOKEN = B
LN =    3   STATE =  2   CHAR = O    CHAR_TYPE =  1   TOKEN = T
LN =    3   STATE =  2   CHAR =      CHAR_TYPE =  5   TOKEN = TO
OUT TOKEN  TO
LN =    3   STATE =  1   CHAR =      CHAR_TYPE =  5   TOKEN = TO
LN =    3   STATE =  1   CHAR = A    CHAR_TYPE =  1   TOKEN = TO
LN =    3   STATE =  2   CHAR =      CHAR_TYPE =  5   TOKEN = A
OUTP TOKEN  IDENTIFIER     A
LN =    3   STATE =  1   CHAR =      CHAR_TYPE =  5   TOKEN = A
```

```
          GDEBUG                    0
          GLLNO                     2
0002        1       SET A TO 1 ;
          MOV                       1                       A
          GLLNO                     3
0003        1       SET B TO A * 3 + 2;
          DOTIMES                   A                       3                       T$1
          ADD                       T$1                     2                       T$2
          MOV                       T$2                     B
          GLLNO                     4
0004        1       IF B > A THEN SET C TO A ;    END_IF;
          COMPARE_GT                B                       A                       T$1
          GOIF_FALSE                T$1                     L#2
          MOV                       A                       C
          LOCDEF                    L#2
          LOCDEF                    L#1
          GLLNO                     5
0005        1       REPEAT SET C TO FROM 1 TO B ;
          LOCDEF                    L#5
          MOV                       1                       C
          LOCDEF                    L#7
          COMPARE_LT                1                       0                       T$2
          GOIF_TRUE                 T$2                     L#8
          COMPARE_GT                C                       B                       T$3
          GO                        L#9
          LOCDEF                    L#8
          COMPARE_LT                C                       B                       T$3
          LOCDEF                    L#9
          GOIF_TRUE                 T$3                     L#6
          CALLO                     L#4
          ADD                       C                       1                       C
          GO                        L#7
          LOCDEF                    L#6
          GO                        L#3
          LOCDEF                    L#4
          ENTERO                    L#10
          GLLNO                     6
0006        2       IF C = 2 * A THEN SET B TO C ;  END_IF;
          DOTIMES                   2                       A                       T$4
          COMPARE_EQ                C                       T$4                     T$5
          GOIF_FALSE                T$5                     L#12
          MOV                       C                       B
          LOCDEF                    L#12
          LOCDEF                    L#11
          GLLNO                     7
0007        2       END_REPEAT ;
          LOCDEF                    L#10
          RETJMP                    L#4
          LOCDEF                    L#3
          GLLNO                     8
0008        1       STOP;
          STOPCD
          GLLNO                     9
0009        1       END;
          STOPCD
   9    EOF
STORAGE     T$5                         2
STORAGE     T$4                         3
STORAGE     L#12                        0
```

```
UPTOK                          DEBUG
        GDEBUG                          0
UPTOK                          DEBUG
        GLLNO                           2
0002        1       SET A TO 1 ;
UPTOK                          LINE_NUMBER
PROGRAM_TAIL
SAME_LEVEL_SEQUENCE
CONTROLLED_BLOCK
UPTOK                          SET
SET_STATEMENT
TARGET
UPTOK                          IDENTIFIER
UPTOK                          TO
EXPRESSION
TERM
SIGNED_OPERAND
UPTOK                          CONSTANT
        MOV                       1                     A
FIND_SEMICOLON
UPTOK                          SEMICOLON
CONTROLLED_BLOCK
        GLLNO                           3
0003        1       SET B TO A * 3 + 2;
UPTOK                          LINE_NUMBER
UPTOK                          SET
SET_STATEMENT
TARGET
UPTOK                          IDENTIFIER
UPTOK                          TO
EXPRESSION
TERM
SIGNED_OPERAND
UPTOK                          IDENTIFIER
UPTOK                          MULTIPLY
SIGNED_OPERAND
UPTOK                          CONSTANT
        DOTIMES                   A                     3                     T$1
UPTOK                          PLUS
TERM
SIGNED_OPERAND
UPTOK                          CONSTANT
        ADD                       T$1                   2                     T$2
        MOV                       T$2                   B
FIND_SEMICOLON
UPTOK                          SEMICOLON
CONTROLLED_BLOCK
        GLLNO                           4
0004        1       IF B > A THEN SET C TO A ;    END_IF;
UPTOK                          LINE_NUMBER
UPTOK                          IF
IF_BLOCK
CONDITION
EXPRESSION
TERM
SIGNED_OPERAND
UPTOK                          IDENTIFIER
UPTOK                          GT
EXPRESSION
```

```
0002          1       SET A TO 1 ;
R  LDA     1
R  STA     A
0003          1       SET B TO A * 3 + 2;
R  LDA     A
R  MUL     3
R  STA     T$1
R  ADD     2
R  STA     B
0004          1       IF B > A THEN SET C TO A ;   END_IF;
R  LDA     B
R  SUB     A
R  BNZ     L#2
R  LDA     A
R  STA     C
D  L#2
D  L#1
0005          1       REPEAT SET C TO FROM 1 TO B ;
D  L#5
R  LDA     1
R  STA     C
D  L#7
R  LDA     0
R  SUB     1
R  BP      L#8
R  LDA     C
R  SUB     B
R  STA     T$3
R  BRU     L#9
D  L#8
R  LDA     B
R  SUB     C
R  STA     T$3
D  L#9
R  LDA     T$3
R  BP      L#6
R  BRU     L#4
R  LDA     C
R  ADD     1
R  STA     C
R  BRU     L#7
D  L#6
R  BRU     L#3
D  L#4
0006          2       IF C = 2 * A THEN SET B TO C ;  END_IF;
R  LDA     2
R  MUL     A
R  STA     T$4
R  SUB     C
R  BNEZ    PC+3
A  LDA     200
R  BRU     PC+2
A  LS       24
R  BNZ     L#12
R  LDA     C
R  STA     B
D  L#12
D  L#11
0007          2       END_REPEAT ;
```

```
0002          1      SET A TO 1 ;
R  LDA     1
R  STA     A
0003          1      SET B TO A * 3 + 2;
R  LDA     A
A +00000002 V  00000001 D  00000001 R  24000011 R  14000010 R  24000010
R  MUL     3
R  STA     T$1
R  ADD     2
R  STA     B
0004          1      IF B > A THEN SET C TO A ;   END_IF;
R  LDA     B
R  SUB     A
R  34000013 R  14000007 R  20000014 R  14000012 R  24000012 R  22000010
R  BNZ     L#2
R  LDA     A
R  STA     C
D  L#2
D  L#1
0005          1      REPEAT SET C TO FROM 1 TO B ;
R  02300000 R  24000010 R  14000015 D  00000027 D  00000000 D  00000000
D  L#5
R  LDA     1
R  STA     C
D  L#7
R  LDA     0
R  SUB     1
R  BP      L#8
R  24000011 R  14000015 D  00000000 R  24000005 R  22000011 R  02400000
R  LDA     C
R  SUB     B
R  STA     T$3
R  BRU     L#9
D  L#8
R  LDA     B
R  24000015 R  22000012 R  14000004 R  01000000 D  00000036 R  24000012
R  SUB     C
R  STA     T$3
D  L#9
R  LDA     T$3
R  BP      L#6
R  BRU     L#4
R  22000015 R  14000004 D  00000042 R  24000004 R  02400000 R  01000000
R  LDA     C
R  ADD     1
R  STA     C
R  BRU     L#7
D  L#6
R  BRU     L#3
R  24000015 R  20000011 R  14000015 R  01000034 D  00000047 R  01000000
D  L#4
0006          2      IF C = 2 * A THEN SET B TO C ;  END_IF;
R  LDA     2
R  MUL     A
R  STA     T$4
R  SUB     C
D  00000050 R  24000014 R  34000010 R  14000003 R  22000015 R  02500065
R  BNEZ    PC+3
A  LDA     200
```

```
          GLLNO                    2
0002        1      SET A TO 1 ;
          MOV                      1                    A
R  LDA     1
R  STA     A
          GLLNO                    3
0003        1      SET B TO A * 3 + 2;
          DOTIMES                  A                    3                    T$1
R  LDA     A
A +00000002 V  00000001 D  00000001 R  24000011 R  14000010 R  24000010
R  MUL     3
R  STA     T$1
          ADD                      T$1                  2                    T$2
R  ADD     2
          MOV                      T$2                  B
R  STA     B
          GLLNO                    4
0004        1      IF B > A THEN SET C TO A ;   END_IF;
          COMPARE_GT               B                    A                    T$1
R  LDA     B
R  SUB     A
R  34000013 R  14000007 R  20000014 R  14000012 R  24000012 R  22000010
          GOIF_FALSE               T$1                  L#2
R  BNZ     L#2
          MOV                      A                    C
R  LDA     A
R  STA     C
          LOCDEF                   L#2
D  L#2
          LOCDEF                   L#1
D  L#1
          GLLNO                    5
0005        1      REPEAT SET C TO FROM 1 TO B ;
          LOCDEF                   L#5
R  02300000 R  24000010 R  14000015 D  00000027 D  00000000 D  00000000
D  L#5
          MOV                      1                    C
R  LDA     1
R  STA     C
          LOCDEF                   L#7
D  L#7
          COMPARE_LT               1                    0                    T$2
R  LDA     0
R  SUB     1
          GOIF_TRUE                T$2                  L#8
R  BP      L#8
R  24000011 R  14000015 D  00000000 R  24000005 R  22000011 R  02400000
          COMPARE_GT               C                    B                    T$3
R  LDA     C
R  SUB     B
          GO                       L#9
R  STA     T$3
R  BRU     L#9
          LOCDEF                   L#8
D  L#8
          COMPARE_LT               C                    B                    T$3
R  LDA     8
R  24000015 R  22000012 R  14000004 R  01000000 D  00000036 R  24000012
R  SUB     C
```

APPENDIX 14 (Page 1 of 3)
SOURCE LISTING OF TEST PROGRAM NUMBER 1

```
LINE     LEVEL

0001        1        |-----------------------------------|
0002        1        |   SAMPLE TESTING PROGRAM NO. 1    |
0003        1        |-----------------------------------|
0004        1    SET A TO 1 ;
0005        1    SET B TO 2 ;
0006        1    IF B > A THEN SET F TO 4 ;
0007        2    END_IF ;
0008        1    REPEAT SET C TO  FROM 1 TO 2 ;
0009        2    IF C = 1 THEN SET D TO 6 ; END_IF ;
0010        2    IF C = 2 THEN SET E TO 7 ; END_IF ;
0011        2    END_REPEAT ;
0012        1    STOP ;
0013        1     END;
```

APPENDIX 14 (Page 2 of 3)
RELOCATABLE FAIRCHILD F24 MACHINE CODE
GENERATED FROM TEST PROGRAM NUMBER 1

```
A  00000001 R  01000000 V  00000001 V  00000001 V  00000001 A +00000000
V  00000001 V  00000001 A +00000001 V  00000001 A +00000002 V  00000001
A +00000004 V  00000001 V  00000001 A +00000006 V  00000001 A +00000007
D  00000001 R  24000010 R  14000007 R  24000012 R  14000011 R  24000011
R  22000007 R  02300000 R  24000014 R  14000013 D  00000030 D  00000000
D  00000000 R  24000010 R  14000015 D  00000000 R  24000005 R  22000010
R  02400000 R  24000015 R  22000012 R  14000003 R  01000000 D  00000037
R  24000012 R  22000015 R  14000003 D  00000043 R  24000003 R  02400000
R  01000000 R  24000015 R  20000010 R  14000015 R  01000035 D  00000050
R  01000000 D  00000051 R  24000015 R  22000010 R  02500064 A  24000200
R  01000065 A  07022024 R  02300000 R  24000017 R  14000016 D  00000065
D  00000000 R  24000015 R  22000012 R  02500075 A  24000200 R  01000076
A  07022024 R  02300000 R  24000021 R  14000020 D  00000076 D  00000000
D  00000000 R  01040057 D  00000056 A  00000200 A  00000200 E
```

APPENDIX 14 (Page 3 of 3)
ABSOLUTE FAIRCHILD MACHINE CODE
GENERATED FROM TEST PROGRAM NUMBER 1

```
 F24SIM VERSION OF JANUARY 29, 1974 - - - - - - - - - - -
OCTOBER 14, 1976        17:35:09.300

NO TRACE PARAMETERS READ.  DEFAULT VALUES USED.

TRACE-LEVEL =                    2
CURRENT EXECUTION TIME =         0
EXECUTION TIME LIMIT =        3000

***  ENTERING LOADER.
***200#
***00000001 01000222 00000000 00000000 00000000 00000000 00000000 00000000
***00000001 00000000 00000002 00000000 00000004 00000000 00000000 00000006
***00000000 00000007 24000210 14000207 24000212 14000211 24000211 22000207
***02300233 24000214 14000213 24000210 14000215 24000205 22000210 02400244
***24000215 22000212 14000203 01000247 24000212 22000215 14000203 24000203
***02400256 01000257 24000215 20000210 14000215 01000235 01000302 24000215
***22000210 02500264 24000200 01000265 07022024 02300270 24000217 14000216
***24000215 22000212 02500275 24000200 01000276 07022024 02300301 24000221
***14000220 01040257 00000200 00000200 00000000 00000000 00000000 00000000
***00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
***00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
***00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
***00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
***00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
***00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
***00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
***00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
***00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
***00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
***00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
***00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
***00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
***00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
***00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
***201##

***  LEAVING LOADER WITH PC=0000001000001        ERROR_LEVEL=    0

* ERROR LEVEL IS     0

*  DUMP_C
*  DUMP_COUNT =     1
*  INITIAL SAVE OF MEMORY.

*  END OF DUMP_C
```

APPENDIX 15 (Page 1 of 3)
SOURCE LISTING OF TEST PROGRAM NUMBER 2

```
LINE      LEVEL

                  |-------------------------------------------------|
0001        1     | SAMPLE TESTING PROGRAM PROGRAM NO. 2            |
0002        1     |-------------------------------------------------|
0003        1     SET B TO 0 ;
0004        1     REPEAT SET A TO -1,0,1 ;
0005        1     IF A < B THEN SET XX TO  1 ; END_IF ;
0006        2     IF A <=B THEN SET XX TO  2 ; END_IF ;
0007        2     IF A = B THEN SET XX TO  3 ; END_IF ;
0008        2     IF A ¬=B THEN SET XX TO  4 ; END_IF ;
0009        2     IF A >=B THEN SET XX TO  5 ; END_IF ;
0010        2     IF A > B THEN SET XX TO  6 ; END_IF ;
0011        2     END_REPEAT ;
0012        2  END;
0013        1
```

APPENDIX 15 (Page 2 of 3)
RELOCATABLE FAIRCHILD F24 MACHINE CODE
GENERATED FROM TEST PROGRAM NUMBER 2

```
A  00000001 R  01000000 V  00000001 V  00000001 V  00000001 V  00000001
A +00000000 V  00000001 A +00000001 V  00000001 A +00000002 A +00000003
A +00000004 A +00000005 A +00000006 D  00000001 R  24000006 R  14000005
D  00000000 R  24000010 R  22000010 R  22000010 R  14000003 R  14000007
R  01000000 D  00000000 R  24000006 R  14000007 R  01000026 D  00000000
R  24000010 R  14000007 R  01000031 D  00000000 R  01000000 D  00000034
R  24000005 R  22000007 R  02300000 R  24000010 R  14000011 D  00000040
D  00000000 R  24000005 R  22000007 R  02400050 R  02100050 A  24000200
R  02300000 R  24000012 R  14000011 D  00000050 D  00000000 R  24000007
R  22000005 R  02500060 A  24000200 R  01000061 A  07022024 R  02300000
R  24000013 R  14000011 D  00000061 D  00000000 R  24000007 R  22000005
R  02200070 A  24000200 R  02300000 R  24000014 R  14000011 D  00000070
D  00000000 R  24000007 R  22000005 R  02400100 R  02100100 A  24000200
R  02300000 R  24000015 R  14000011 D  00000100 D  00000000 R  24000007
R  22000005 R  02300000 R  24000016 R  14000011 D  00000105 D  00000000
D  00000000 R  01040036 D  00000035 A  00000200 E
```

APPENDIX 15 (Page 3 of 3)
ABSOLUTE FAIRCHILD F24 MACHINE CODE
GENERATED FROM TEST PROGRAM NUMBER 2

```
 F24SIM VERSION OF JANUARY 29, 1974 - - - - - - - - - - -
OCTOBER 14, 1976        17:33:46.970

NO TRACE PARAMETERS READ.  DEFAULT VALUES USED.

TRACE-LEVEL =                   2
CURRENT EXECUTION TIME =        0
EXECUTION TIME LIMIT =       3000

***  ENTERING LOADER.
***200#
***00000001 01000217 00000000 00000000 00000000 00000000 00000000 00000000
***00000001 00000000 00000002 00000003 00000004 00000005 00000006 24000206
***14000205 24000210 22000210 22000210 14000203 14000207 01000236 24000206
***14000207 01000236 24000210 14000207 01000236 01000311 24000205 22000207
***02300243 24000210 14000211 24000205 22000207 02400250 02100250 24000200
***02300253 24000212 14000211 24000207 22000205 02500260 24000200 01000261
***07022024 02300264 24000213 14000211 24000207 22000205 02200270 24000200
***02300273 24000214 14000211 24000207 22000205 02400300 02100300 24000200
***02300303 24000215 14000211 24000207 22000205 02300310 24000216 14000211
***01040236 00000200 00000000 00000000 00000000 00000000 00000000 00000000
***00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
***00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
***00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
***00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
***00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
***00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
***00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
***00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
***00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
***00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
***00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
***00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
***00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
***00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
***201##

***  LEAVING LOADER WITH PC=00000010000001        ERROR_LEVEL=    0

* ERROR LEVEL IS    0

*  DUMP_C
*  DUMP_COUNT =     1
*  INITIAL SAVE OF MEMORY.

*  END OF DUMP_C
```

BIBLIOGRAPHY

BOOKS

1. Donovan, John J., System Programming, New York, Mcgraw-Hill, 1972

2. Gries, Davis, Compiler Construction for Digital Computers, New York, John-Wiley, 1971

3. Hopgood, F. R. A., Compiling Techniques, London, Macdonald, 1971

4. International Business Machines, PL/1 Language Specifications, IBM Form C28-6571, 1974

5. International Business Machines, FORTRAN Programming Guide, IBM Form C28-6835,1973

6. Mckeeman, W. M., A Compiler Generator, New Jersey, Pretice-Hall, 1970

7. Waite, W. M., Compiler Construction, New York, Springer-Verlag, 1974

ARTICLES

1. Aho, A. V., "A Formal Approach to Code Optimization," SIGPLAN Notices, July 1974, pp. 86-100

2. Beatty, J. C., "Register Assignment Algorithm For Generation of Highly Optimized Object Code," IBM J. Res. Develop., 1974, pp. 20-39

3. Cocke, J., "Global Common Sub-Expression Elimination," SIGPLAN Notice, July 1970, pp. 20-24

4. Dijkstra, E. W., "ALGOL60 Translation," ALGOL Bulletin 10, 1960

5. Early, J., "An Efficient Context-Free Parsing Algorithm," Comm. ACM 13, 1970, pp.94-102

6. NEWLY, M., "Abstract Machine Modeling to Produce Portable Software," Software, Practice And Experience 2, 1972, pp. 107-136

7. Yourdon, Edward, "A Brief Look at Structure Programming and Top-Down Programming Design," Modern Data, 1974, pp. 30-34

UNPUBLISHED MATERIALS

1. Isaacson, Portia, A Compiler for This Programming Language, Department of Computer science, North Texas State University, Denton, Texas 1972

2. Isaacson, Portia, User's Manual for This Programming Language, Department of Computer Science, North Texas State University, Denton, Texas 1972

3. Leinius, R. P., Error Detection and Recovery for Syntax Directed Compiler System, PH.D Dissertation, University of Wisconsin, 1970