AN APPROACH TO LANGUAGE IMPLEMENTATION
AND CODE GENERATION
FOR MICROCOMPUTERS

By

ANNE MARIE BUTLER

Bachelor of Arts

Harvard University

Cambridge, Massachusetts

1978

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 1987

Thesis
1987
B985a
Cop. 2

AN APPROACH TO LANGUAGE IMPLEMENTATION

AND CODE GENERATION

FOR MICROCOMPUTERS

Thesis Approved:

R. E. Hedrick
Thesis Adviser

Norman N. Durham
Dean of the Graduate College

1275592

PREFACE

A cross-compiler for the Pascal language was developed. The compiler development facilities on the Concurrent XF-610 Unix-based system. The output of the compiler is assembly language code for the Commodore 64 microcomputer. The code was then down-loaded to the Commodore, and run to verify the proper functioning of the compiler.

I have depended on a great number of people for support during the duration of this project. I especially want to thank my major adviser, Dr. G. E. Hedrickfor his support, advice and input to the project. I also appreciate the kindness of my two other committee members, Dr. D. W. Grace, and Dr. K. M. George, for filling in for the two committee members who were unable to participate in my thesis defense. I also would like to thank my friends, Terry Johnson and Judy Edgmand, for listening to me, and the moral and financial support of the computer science department at Oklahoma State University.

Special thanks are due to my family, especially my parents, Dr. and Mrs. W. G. McKechnie and my parents-in-law, Mr. and Mrs. L. W. Butler, Jr. for their invaluable help in taking care of my two sons, Thomas and James, and giving me the freedom to pursue this project. I also thank

my small sons for their patience when I had to work. And, finally, my deepest appreciation to my husband, Lindsay W. Butler III, for putting up with the separation and difficulties that living apart has caused.

TABLE OF CONTENTS

Chapter | Page

I. INTRODUCTION.. 1

Definition of the Problem.................... 1
The Language Subset.......................... 4
The Compiler................................. 4
6510 Organization and Machine Language...... 8
Summary of the Project...................... 10

II. LITERATURE SEARCH.............................. 11

III. SYMBOL TABLE ORGANIZATION....................... 16

IV. RUN-TIME STORAGE ADMINISTRATION................. 23

V. SEMANTIC ROUTINES.............................. 26

VI. CODE GENERATION................................ 29

VII. SUMMARY AND CONCLUSIONS........................ 35

Conclusions from the Project............ 35
Future Work............................. 35

BIBLIOGRAPHY.. 38

APPENDICES.. 40

APPENDIX A.................................. 40
APPENDIX B.................................. 41

LIST OF FIGURES

Figure Page

1. LALR(1) Grammar for a Subset of Pascal............. 2

2. Symbol Table Organization......................... 22

3. Memory Map for the Commodore 64................... 30

CHAPTER I

INTRODUCTION

Definition of the Problem

The purpose of this project is to explore the design and implementation of languages for a 6510 microprocessor on the Commodore-64 home computer. At present time there is a paucity of good programming language implementations for this particular machine; in most cases the programmer must choose between Basic and assembly language. Since the former tends to be under-powered and inefficient, and the latter makes coding exceptionally tedious, it is desirable to have a language between these two extremes.

There are a variety of high-level languages in use today including Ada[R], ([R] Ada is a registered trademark of the U.S. DOD,Ada Joint Projects Office (AJPO)) Pascal, PL/I, Cobol, and Algol68. Any of these would extend the usability of this particular machine. Unfortunately, implementation of any of these languages is not a trivial programming task. To simplify this first effort to address this problem, a predefined grammar, outlined by Aho and Ullman[1], will be used. What is contained in this grammar is a useful subset of the Pascal language. The LALR(1) grammar for this subset is shown in Figure 1.

```
 program:
    program IDENTIFIER (id-list);
     declarations
     subprogram-declarations
     compound statement
id-list:
     IDENTIFIER
     | id-list, IDENTIFIER
declarations:
     var declaration-list
     |
declarations-list:
     id-list:type
     | declaration-list id-list:type
type:
     standard type
     | array type
standard type:
    integer
     | real
array-type:
    array [CONSTANT..CONSTANT] OF standard-type
subprogram-declarations:
     subprogram-declarations
     subprogram-declaration
     |
subprogram-declaration:
     subprogram-head
     declarations
     compound-statement
subprogram-head:
    function IDENTIFIER arguments:
    result standard type;
    | procedure IDENTIFIER arguments
arguments:
     (parameter-list)
     |
parameter-list:
     id-list : type
     | parameter-list; id-list:type
compound-statement:
    begin
     statement-list
    end
statement-list:
     statement
     | statement-list ; statement
```

Figure 1. LALR(1) Grammar for a Subset of Pascal

```
statement:
     elementary-statement
     | if expression then restricted-statement
    else statement
     | if expression  then statement
     | while expression do
restricted-statement:
     elementary-statement
     | if expression then restricted-statement
     else restricted-statement
     | while expression do  restricted-statement
elementary-statement:
     variable ASSIGNOP expression
     | procedure-statement
     | compound-statement
variable:
     IDENTIFIER
     | IDENTIFIER  expression
procedure-statement:
     IDENTIFIER
     | IDENTIFIER ( expression-list )
expression-list:
     expression
     | expression-list , expression
expression:
     simple-expression
     | simple-expression RELOP simple-expression
simple-expression:
     term
     | sign term
     | simple-expression ADDOP term
term:
     factor
     | term MULOP factor
factor
     variable:
     | CONSTANT
     | ( expression )
     | function-reference
     | not factor
function-reference:
     IDENTIFIER
     | IDENTIFIER ( expression-list )
sign
     +|-
```

Figure 1. (Continued)

The Language Subset

Although the subset violates none of the conventions for full Pascal, there are several omissions which simplify the subset. The only type declarations possible are integer, real and array. Full Pascal has a variety of types, including user definable types[2]. In the subset, records, structures and sets are omitted and arrays are limited to one dimension. Block structures, with their own variable declarations, are also not included. Since many of the operations that would be made easier by record, set or multi-dimensional array can be done by intelligent use of one dimensional arrays, and block code is never a necessity, these omissions are only minor annoyances. They could be included with some restructuring of the compiler mechanisms, at the expense of making the compiler larger. The subset does cover many of the more interesting features of a high-level language. These include functions and procedures, recursion, parameter passing, and conditional loops. The subset is upwardly compatible with full Pascal, with the revisions noted later.
This compiler therefore could be useful for bootstrapping a more complete Pascal for this machine.

The Compiler

The compiler consists of three major parts: first is the lexical analyzer, to recognize the tokens of this language. For a language of any size, the lexical analyzer

usually is a simulation of a finite state automaton. It may either be written by the programmer or be engineered with the use of a lexical analyzer constructor such as LEX, available on UNIX based systems[3]. Although Kernighan and Pike[4], and Waite and Goos[5] indicate that the programmer can usually write shorter and more efficient analyzers than those that are mechanically produced, it is far simpler to let LEX do most of the work, so this compiler uses a LEX engineered lexical analyzer. The second part of the compiler is the syntactic analyzer. This is where the code is analyzed for conformity to the rules of the grammar defined for that language. This is another sort of recognizer, a pushdown automaton. The grammar of the language provides rules for reducing subtrees of related nodes. If the program conforms to the grammar, the final reduction is to the root of the tree, and a correct program is recognized by the automaton. The automaton contains a stack for storing incompletely recognized parts of rules. Once a reduction is made, part of the stack is replaced by the left hand side of the grammar rule, until all that is left on the stack is the root, or start symbol of the grammar. For a language of any size, this automaton is a sizable piece of code. The programmer may choose to write this by hand, but using an automatic constructor such as YACC, also available on UNIX systems[6], is an alternative. Even if the lexical analyzer is handwritten to increase efficiency, the the amount of programming time consumed by coding this phase manually make it much more attractive to automate.

It is during the syntactic analysis phase that the incoming code is transformed into another representation. This transformation is acomplished by semantic actions. In YACC, semantic actions may be appended to the grammar rules. These actions are executed just prior to reduction and produce the transformation of the input program some form more applicable to the target. The simplest transformation is to have the output of the syntax analyzer be the output of the program being translated. In this case, no further transformation is necessary as the output of the program is immediately available. Kernighan and Pike[4], demonstrating the function of YACC on the UNIX system create a series of demonstration compilers, and the first of these, hoc1, does this. Unfortunately, this simplistic approach is only workable when the value of the output can be determined from the input directly. It is not useful for languages with control-flow constructions such as IF-THEN statements. The difficulty with these constructions is that the position of the next statement to be executed is not known at the time that the IF-THEN statement is being analyzed.One method of coping with this is to have the syntax analyzer generate incomplete code, which can then be "fixed up" with the address of the next statement after that statement is discovered.

The code that the syntax analyzer generates may be either executable code for the target machine, or some variety of intermediate code that can be interpreted to produce executable code. There are several other choices,

including another high level language, and tables for a linking loader. The latter, according to Barrett and Couch[7] is the form used by many commercial compilers because it allows segments of a large program to be compiled separately, which cuts costs for debugging. This technique assumes the existance and availability of a linking loader, however, and such is not readily available for the 6510.

Compiling to another high level language has the advantage of simplicity, but it masks the function of the actual machine. In addition, program performance would be dependent on the compiler for the target language. Since some constructions are difficult to translate from one high level language to another the resulting program might function inefficiently.

Machine code has the advantage of eliminating one step in the compilation process; however, in doing so, the possibility of rewriting the output code to improve efficiency is sacrificed. It does allow the program to be executed immediately, but makes it more difficult to discover and to correct any errors in the code generation process without disassembling the instructions to determine where and why the error happened.

Generating intermediate code that may be manipulated in the interpretation step is a more generalized approach. Not only may the efficiency of the final code be improved, but it also may be easily changed to run on another machine by altering the interpreter[7].

All varieties of intermediate code require further

interpretation to produce executable code. There are a number of different types of intermediate code ranging from fairly elementary to quite detailed. One simple approach is to have the syntax analyzer produce modules which consist of an operator, up to two operands, and a location for the result of the operation. These are called three address modules, or quadruples, referring to the four fields in the module. They have the advantage of being easily generalizable, but need a fair amount of reinterpretation to produce object code. Other choices include P-code, which is quite similiar to Pascal itself[8], the table building language, TBL, described by Anklam et.al.,[9], and some variety of assembly language or machine code for the target machine. Since the main aim of this project is to produce code to be used to bootstrap a more nearly complete compiler, the intermediate language is quadruples for generality. The target language is 6510 assembly code because it gives most of the speed and power of machine code with a degree of readability that makes verification of the compiler functions easier.

6510 Organization and Machine Language

As discussed above, the third stage of this compiler is translation of the generated quadruples into code for the target machine. The interpreter is responsible for choosing a starting address for the code. Since the 6510, as many microprocessors,must support many functions such as screen

display, buffers for printing and data transfer in a relatively small memory, the program must be placed so that it does not interfere with any function that the programmer needs at the time that program is to be executed. In addition, the interpreter must take care of such issues as register management, and code optimization.

Because the target machine is a microprocessor, it is a fairly simple machine. The 6510 has 64K of available RAM, some of which is unusable because it is the fixed location for the screen, disk and printer utilities. It has an accumulator A and two registers: X and Y. All are 8 bits. The two registers differ slightly in their use for indirect addressing, but otherwise are the same. Having so few registers to use limits their usefulness for register optimization. The small size of the accumulator forces real number calculations to use memory locations as accumulators, consequently real number arithmetic is relatively slow.

The instruction set for this machine is also somewhat limited. The only arithmetic functions are addition and subtraction of the contents of the accumulator with another number. Division, multiplication, mod and all other arithmetic functions must be supplied by the code generator. One rather interesting feature of this machine, however, is the availability of routines in the kernal. The kernal is a set of machine language subroutines, primarily I/O utilities, that are grouped together. The term kernal refers to those routines that may be located in different locations in the different varieties of 6502 microprocessor (including

the 6510 discussed here), but whose address is guaranteed to be in a specified location. These routines provide various utilities for Basic when the computer is running under that language, and many of these routines may easily be linked into Assembly language programs. Some of the kernal routines assist in data entry from the keyboard or an external storage device, in this case, a floppy disk drive. Other routines are available in the Basic ROM, and these routines do data conversions, It is not the intent of this compiler to rewrite Basic, but the availibility of these subroutines in ROM certainly simplify much of the translation, and would be of great help in adding more features to the Pascal subset.

Summary of the Project

In summary, the project is to design and to execute a cross-compiler for the subset of Pascal described by Aho and Ullman[1]. The compiler will use the versions of LEX and YACC available on the Concurrent XF-610 research computer to construct a parser and to generate quadruples, which then will be translated, using a code generator written in C, also on the Concurrent XF-610. The output of the interpreter will be 6510 assembly code, which will then be transferred to a Commodore 64 home computer for verification.

CHAPTER II

LITERATURE SEARCH

As indicated in chapter 1, there are quite a few choices the designer must make. The form of grammar to be used, how many passes through the code the compiler should make, the type of intermediate language, if any, to be employed, and the exact formulation of the symbol table are just a few of the numerous decisions involved. It seems logical, therefore, that before deciding anything, an examination of what other compiler writers have used, and their justification for using it should be undertaken.

A description of the construction of a lexical analyzer can be found in Aho and Ullman(1), Kernighan and Pike(4), and Barrett and Couch(10), but the construction of the lexical analyzer is so much simpler than the syntax analyzer that there is not much to report. A discussion of the pros and cons of automating this rather than hand-coding is found in Kernighan and Pike(4) and the Lex manual(3). Probably the best support for automation is in Johnson(6).

The rest of compiler design is not so straightforward. Limiting the scope of examination only to Pascal compilers still results in a wealth of material written. Waite and Goos(5) describe the first Pascal compilers, Pascal-P and Pascal-6000, which were completed in 1973-4. Pascal-P

produces code for a generalized stack machine, and Pascal-6000 for the CDC 6000 series computer. These were single pass, recursive descent parsers, written in Pascal. There was no explicit symbol table; symbols were stored as packed character arrays. This symbol table organization slows access time, and may require more space for searching than a symbol table, thus it does not represent the best choice for identifier storage. A discussion of the Pascal-6000 and its relation to the standard is found in Jensen and Wirth(10).

Another Pascal compiler was the IBM 360/370 bootstrap described by Russell and Sue(11). They took the original Pascal compiler on a CDC 6000, and rewrote the code generator to produce object code for the target IBM 360. Then they rewrote it in PL.I, a language that already existed on the IBM machine. This produced a compiler for Pascal that was inefficient, but could be used to translate the Pascal code for the original CDC compiler. Once translated, the resulting Pascal compiler produced translations of an acceptable caliber. This is a good example of cross-compilation, which is the basis of the project discussed in this paper.

Further examples of bootstrapping compilers can be found in Anklam, et. al.(9), and and Grosse-Lindemann and Nagel(12). The latter contains an accounting of just how much work a sizable compiler requires; their bootstrap of the Pascal-P compiler to a DECsystem-10, and subsequent additions to make it a more attractive language for general purpose usage took about 2 1/2 man years of effort. The

smaller task of writing a compiler for the Aho and Ullman(1) subset seems a more reasonable project for a single person.

The idea of using a relatively small language is justified further in the software principles outlined by Richard and Ledgard(13). They argue that a language should be simple, rather than complicated with extra structures, and limited in size.

Although good descriptions of the general principle of syntax analysis and the related topic of generating intermediate code exist in many texts on compiler writing, Aho and Ullman(1) take a generalized theoretical approach to the topic that is especially useful in bootstrapping to a new machine. Several other papers, including Beatty(14), and DeRemer and Pennello(15) explore the design, construction, and verification of the properties of grammars.

The design of Pascal compiler symbol tables has undergone quite a transition since their inception. The Pascal-P compiler had no symbol table as such. Knuth's analysis(16) of performance indicates that a hash-table based system would produce the best lookup-performance, although would require more space than other schemes. This choice is defended by Barrett and Couch(7), who further propose a stack access type storage for ease in exiting the different levels. Reiss(17) outlines a method for the automatic construction of an appropriate mechanism, based on the language specifications.

For the generation of semantic actions, the theoretical approach in Aho and Ullman(1) clearly outlines appropriate

actions for most major features of any programming language. Other discussions may be found in Waite and Goos(5) and Barrett and Couch(7). The former contains a detailed account of semantic analysis, or the examination of the input program for conformity to the general semantics of the language.Their account includes such topics as type and level checking, which cannot be included in the grammar itself easily.

Optimization of the code to improve performance can improve run time significantly. When Russell and Sue(11) ran their Pascal front end through the PLIX optimizing compiler, rather than the PL/I(F) compiler, its run time was three times faster. Optimization is certainly something desirable to include in a compiler, even though extra passes are required to achieve it. A general description of optimization techniques can be found in Aho and Ullmann(1). Davidson and Fraser(18) designed a peephole optimizer that examines two and three instruction sequences to see if they can be replaced by more efficient code. Tannenbaum, et. al. (19) demonstrated that this sort of optimization could be even more effective when applied to the intermediate code, rather than the object code.

Finally, a description of the architecture of the Commodore 64 may be found in the Commodore Reference Guide(20). The assembler to be used is described in Bush and Holmes.(21) This includes the description of the ROM routines that may be accessed by kernal jumps.

The literature on cross compilers outlines many

techniques to use in constructing such a compiler. There is also documentation on writing machine language for home computers, but very little on the construction of compilers for small, single-user machines. Indeed, there are few sources that fully document the types of decisions involved in making a compiler except the texts on compiler writing. One of the aims of this project is to address more concretely, the design decisions made, and the reasoning behind them.

CHAPTER III

SYMBOL TABLE ORGANIZATION

The organization of the symbol table depends on a number of factors, including the amount of space available, the specific requirements of the target language and conveniences provided by the language and operating system of the front-end environment. One of the pecularities of a cross-compiler is that the contents of the symbol table are not available automatically to the target machine; any information necessary for the target code to execute properly must be downloaded with the code. This pecularity must be considered in choosing the organization and devices that create and manipulate the symbol table.

In the organization of the symbol table for this Pascal compiler, there is sufficient memory available to the front end of the compiler that it is possible to create a static symbol table entry, keeping all of the information about the symbols readily available. This approach has the virtue of simplicity of code, for it eliminates the necessity for several symbol table manipulation routines, consequently it is the approach used here. Because the name will not be discarded, it is stored as a field in the structure that defines the symbol, rather than in a separate array; although this choice was made on the basis of simplicity of

code rather than efficient use of storage resources.

Another issue to be addressed in the symbol table organization is the possible duplication of names in different procedures. Several options exist to ensure that the proper entry for a given symbol is accessed. Aho and Ullman's description of run-time storage allocation for ALGOL suggests that the entries for each procedure level be allocated space on a stack, and the stack be searched in reverse order of entry until the symbol is located. Since later entries will be closer to the top of the stack than previous declarations, the correct access is assured. One drawback to this approach, however, is that the entries for procedures at the same level should occupy essentially the same space on the stack. Since it is necessary to keep the entries for each procedure available for subsequent passes, this requires that all the entries be moved to some accessible location; Aho and Ullman suggest that the bottom of the stack array be allocated as an inactive area to hold these entries.

This type of approach has several disadvantages. First, because the entries are on a stack, a good deal of linear searching would be necessary to locate a particular entry. Second, the entries must be located in a statically defined stack in order to implement the removal of entries when they no longer need to be available for reference. Since compilers should be able to handle efficiently very small as well as very large programs, it is difficult to choose an array size that both is sufficient for a large program and

not wastful for the requirements of a small program. Also, the necessity of moving entries from one area to another seems unattractive, for it requires several routines to maintain this storage stack. The requirements of this organization seemed unappealing in an examination of the resources of this compilers environment, so a different approach was taken to address these concerns.

Intuitively, the primary disadvantage of Aho and Ullman's scheme is the ineffiency of looking up a variable by searching the stack from the top down. Other data structures have much better overall access statistics than a stack. Of the choices, a hash table is best. It does require a fixed size array be set up, but if the entries in the hash table are simply pointers to locations where entries actually are found, then the space needed for this array is actually fairly small. Since the UNIX system has excellent facilities for the dynamic allocation of storage, only the amount of storage required for a given program needs be allocated. To resolve collisions, the entries are chained at each hash location, locating the most recent entry for each name as the closest to the table to make searching as efficient as possible. Although this chain must be searched linearly by following the links from one entry to the next, this is superior to searching through all the entries on a stack before accessing the next level, and decreases search time by keeping the number of entries accessed before the correct entry is found to a minimum.

Addressing the problem of the need to keep track of all

of the declarations for a given procedure added two more details to the symbol table. The first was an array to point to the entries for each procedure. As a procedure is entered, it is assigned a unique procedure number, and once the first symbol for that procedure is encountered, the array element corresponding to that level number is set to point to the address of that particular entry. An extra field is in each entry to allow each subsequent entry to be chained to the last entry made. This chain can be followed easily when the procedure is ended to remove these inactive entries from the chains for the hash table while still leaving them accessible from the chains in the procedure table. This accessibility keeps the symbols available for use during the second pass when they may be used to calculate storage locations on the target machine.

In addition to the address of the symbols for the procedure, the procedure array entries have a field to determine whether that procedure is active at a given time, a field to keep track of the total storage requirements for that procedure, and an array of procedures that were active at the time the procedure was entered. The latter is commonly referred to as a display. These were added because the language being compiled requires dynamic runtime storage allocation to support recursion, and these fields will supply information to the code generator to enable this dynamic allocation to be done.

The symbol table mechanism defined above functioned well except in one area. In Pascal, all variables must be

declared, followed by all procedure declarations, followed by code. Since the code may be separated from the declarations by other declarations, the procedure number is not accessible when temporary variables and constants from the code must be installed during compilation of the code. The remedy for this difficulty is interposing a level array between the procedure number array and the symbol table. The entries in the level table are the procedure numbers. As a procedure definintion is encountered, it is assigned a unique procedure number which is installed in the level table at the current level. The procedure table pointers are accessed through this level reference. Having this level table is advantagous when creating the display at run-time, as it is possible to determine and record the surrounding procedures for an array during the initial analysis. The level can be decremented when the end of the procedure definintion is encountered.

All the reserved words for the language are installed in the symbol table initially, to eliminate the possibility of redeclaration of these key words. Other words are installed in the symbol table as encountered.

Having defined the mechanisms, the entry itself must be considered. The basic elements of the entry are the name, the type and the value. All are installed statically. The procedure number is also necessary to ensure correct access when searching for a reference to a variable or procedure. Because of the mechanism, two pointers were also included, one for the chain from the hash table, and the other from

the procedure.

The symbol table, therefore, is as shown in Figure 2. The structure of the table addresses the concerns of efficiency of symbol look-up, and retention of the appropriate information for code generation. It basically consists of a hash table whose entries are pointers to nodes holding the information about the symbols. These nodes are also threaded to a procedure counter to allow them to be marked as inactive when a procedure is exited. Storage requirements for each procedure are kept in a separate table.

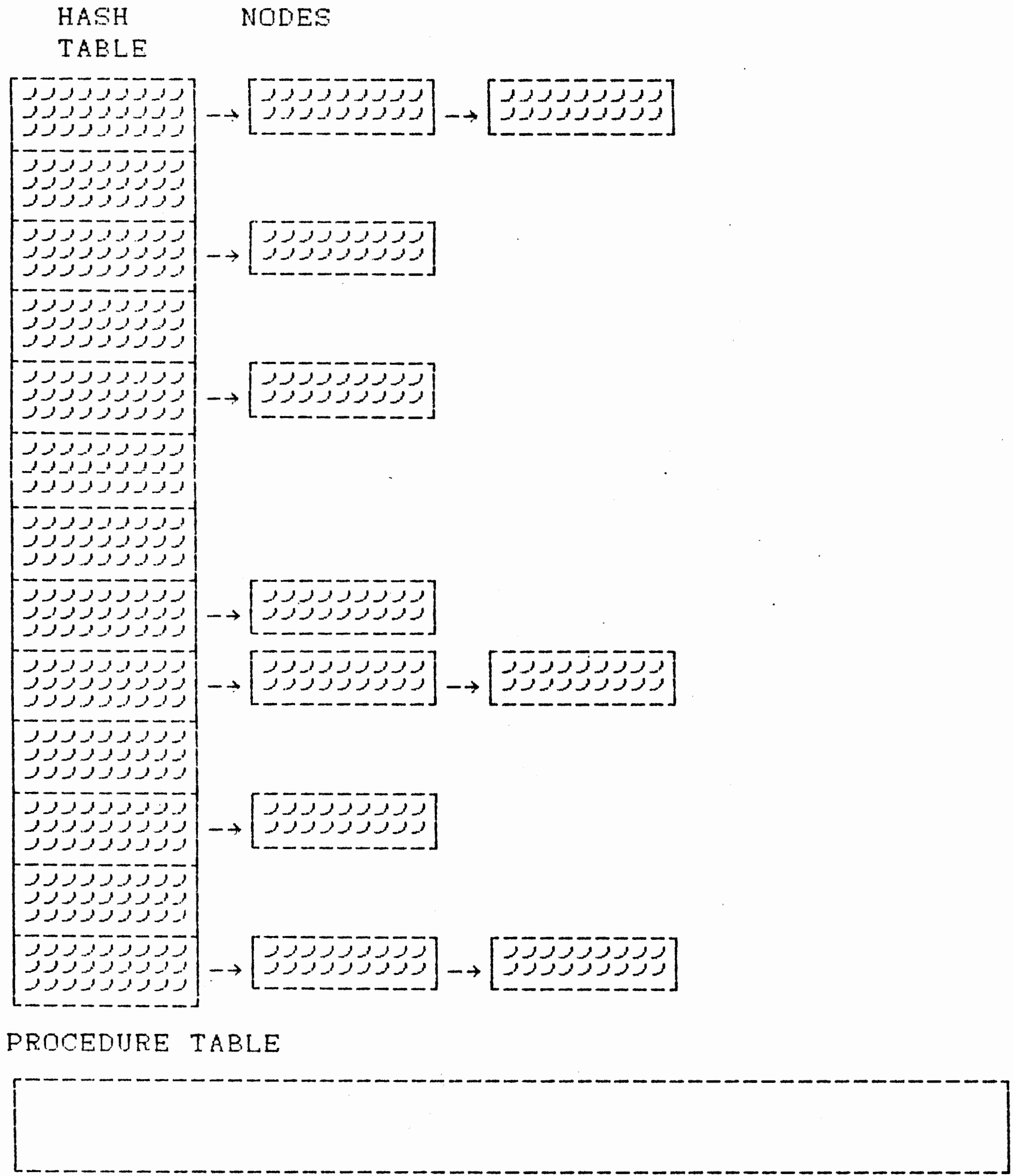

Figure 2. Symbol Table Organization

CHAPTER IV

RUN-TIME STORAGE ADMINISTRATION

One of the difficulties with a cross-compiler is that the symbol table and all other descriptors generated during the lexical and syntactic analysis are not readily available to the run-time environment. Since the subset of Pascal used by this compiler allows recursion and requires a dynamic storage allocation capability, some of the information stored by the front end of the compiler must be passed to the target machine. Specifically, the table of storage requirements for each procedure, along with its display vector, and the location of any static variables referenced by the procedure should be available.

The generated code for the microprocessor must contain both the information required for this storage allocation, and the routines to manipulate it. There are several options for providing this. The routines could be stored in a separate file accessible to the microprocessor, but then the user would be compelled to load routines prior to run time, in a linking phase. The compiler front end would still have to provide the templates for procedure storage requirements and addresses for static storage locations. Another option,

at the expense of slightly more time for translation, is to have the compiler code generator regenerate these routines for each compilation, and include them with the code generated for the program being compiled. This option is less demanding of user interaction, and thus simpler. The storage requirements for each procedure can be computed during the first pass, and be allocated via code included in the main body of the program. Since the routines discussed are small, the second approach is the one used for this compiler.

Another complexity in storage allocation arises in the area of addressing. All formal parameters for this compiler are accessed by the call by reference technique. During dynamic storage allocation, a quantity in a given storage location can either be a temporary parameter or the address of some other location in the environment of the procedure. The normal method for dealing with accessing variables in other storage locations is to use some sort of indirect addressing, by putting the base address of the routine containing the parameter in question into a register. Unfortunately, the instruction set of the 6510 has very limited indirect addressing capabilities. First, the registers on the 6510 are only 8 bits, and addresses are 16 bits, so cannot be put in a register. The only option for doing an indirect address is to place the address to be accessed in some location in the zero page of the machine memory; load the offset of the variable being

accessed in the Y register, then use one of the few indirect indexed instructions. Zero page is approximately 99% full, though. Most of the locations are occupied by parameters required by the operating system and basic routines. There are a few locations available; specifically, locations 251-255 are reserved for the user. By using the bottom two locations for the current stack pointer, and the second location for the address of another block being accessed, indirect addressing is possible, though not neat. This is recognized as an area where performance could be improved by some compiler optimization.

Storage manipulation is accomplished by the simple expedient of storing the current stack pointer in an easily accessible location in page zero. When a procedure is called, this value is updated via code generated from the procedure storage requirements calculated in the first pass. The level of the procedure is also available from the first pass, and is passed in the code to determine which values from the display of the calling procedure need to be copied onto the new storage block.

CHAPTER V

SEMANTIC ROUTINES

The first pass of the compiler is done via the YACC-generated parser. One of the advantages of YACC, in addition to simplifying construction of a parser from the grammar, is that it provides a built-in stack to hold values that can be returned from the lexical analyzer. The stack can be redefined, but if it is not, it is of type integer. The value stack runs concurrently with the token stack and is a very convenient place to keep the locations of the symbols that hold various parts of the rule during translation. The difficulty is that since the symbols were allocated dynamically, they are located via a pointer which cannot be stored directly on a stack of type integer. One remedy for this would be to put the hash location, which is an integer, on the stack and at the time that the symbol must be accessed to find it from its hash address. This would require a bit of searching which could be eliminated if the address of the symbol node could be directly stored on the stack. C provides for this sort of coercion, using a cast to specify the type of a value, when it is assigned to a variable of a different type. By using casts the pointer value is stored on an integer stack, and can then be referenced when needed, and coerced back into an address, if

necessary.

The grammar itself provides other instances where the stack is useful. Relational operators can be treated in the same manner as far as semantic routines, but must be distinguishable to generate the appropriate quadruple. This is accomplished by returning the same value to the token stack for all relational operators and letting the value on the value stack hold the particular symbol. This is also true of the classes of additive operators and multiplicative operators.

The original grammar required some alteration to conform to standard Pascal. The grammar, as written, allows algebraic combination of arithmetic and Boolean expressions, which is not allowable in Pascal. To eliminate the possibility of this sort of construction, a flag could be set in the semantic routines, but this is messy. At the expense of slightly more complexity in the pushdown automaton the two classes can be differentiated in the grammar rules, and thus eliminate the recognition of unallowable constructions. This caused another revision to separate the logical operators, and the resulting Boolean expressions by their application.

One other revision was to eliminate the unary plus. Since this does not have much use in any language, it was removed. The unary minus was included in the arithmetic expressions.

In other cases distinction via grammatical revision seemed inappropriate. Declarations of arguments to

procedures and functions are the same as the declarations in the program, but they must generate a different set of quadruples. To do this, a flag is set in the grammar when a subroutine header is set. The value of the flag determines whether a declaration generates a quadruple or a symbol table entry.

In summary, the semantic routines are based on compiler writing concepts outlined in several texts in the field. The grammar itself was slightly revised to accomodate the conventions of standard Pascal. Distinctions not possible from the grammar as written are resolved by setting flags. Output of this section is a set of quadruples that represent the semantic content of the translated code.

CHAPTER VI

CODE GENERATION

Writing the code generation phase of a compiler is a nebulous task. There is not much written about generalized techniques for this task, since each machine has its own unique instruction set. It is almost entirely left to the designer to use the assets and liabilities of the particular machine that is the compiler's target.

Home microcomputers, in particular, are a rather interesting environment for a compiler. A home computer is designed by the manufacturer to stand alone, and to contain all that is necessary to operate the machine for the average home user to do wo unassisted. It contains a number of resident routines in its ROM to accomplish such tasks as input, arithmetic, and output and a basic language interpreter, in addition to the operating system. A table of the memory map is shown in figure 3.

There are a number of ways of approaching dealing with these utilities. All but the necessary operating system routines may be overwritten by resetting the pointers that determine what space is available to be used for machine language routines. In particular, the basic interpreter may be dispensed with fairly easily. In eliminating this, the code generated by the compiler must supply all the

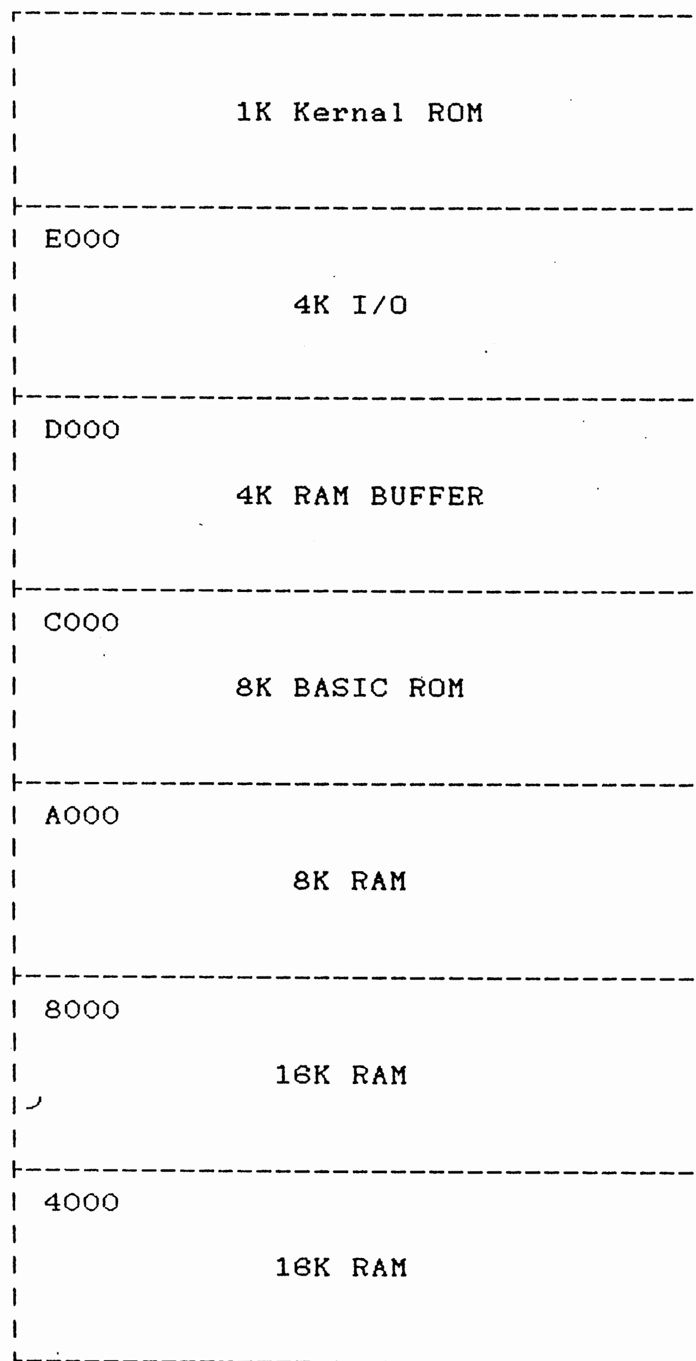

Figure 3. Memory Organization of the C-64

routines that are necessary for language functions.

There were enough good routines available in the Basic interpreter to sacrifice the space it requires. Preserving the Basic interpreter has the added advantage that there are many string and function subroutines that could be integrated into the compiler with ease, by just setting up the proper jumps when generating code from the quadruples. These subroutines are not strictly part of standard Pascal, but have been included in many modern versions, such as the Waterloo Pascal compiler, and would enhance the language.

The Kernal ROM routines are a collection of utilities that are primarily I/O. The philosophy of the Kernal is that the location of these essential I/O routines should be left up to the manufacturers preference, so the location of the subroutine itself is not fixed, but the location of the address of the subroutine will always be found in the same spot in a jump table. If a manufacturer decides to double memory, the I/O routines can be located to a convenient place, and not take up space in the middle of a chunk of the work area. When the kernal jump table is used, the code may be altered more easily to run on another machine with the same microprocessor base. This asset was relatively unimportant here, because in deciding to include the contents of the Basic interpreter in the generated code, the code will be specific to the Commodore 64 6510 and not portable.

The remaining space, not required by the operating system or by the kernal and Basic ROMs is available to hold

the contents of the compiled code. There is approximaltly 80% of the 64K left, that is managed by the storage management techniques in the previous chapter. One aspect of storage management that affected the code generation in a specific way was the decision to choose a fixed starting location for all the code in the program. This simplified calculations during code generation, and is certainly easier on a home computer, where there is only one user, so no allowances must be made for anything in memory except the requirements of the program currently running.

The workspace for programs is memory locations 2K to 40K which are used by changing the contents of the zero page memory location that holds the location of the top of memory. Normally all of this space is allocated as work space for Basic programs, but the actual upper and lower limits on the size of this space required by Basic is defined by the contents of two memory locations. If the contents of the top of memory pointer is altered, the space is deallocated from Basic, and available to hold machine language programs, so this is a good location for the compiled code.

Other locations are available. The 4K bytes from address 49152 are unused, so make a good location to hold the storage stack. Choosing the fixed location 49152 as the start of the stack allows the front end of the compiler to generate addresses more easily, even though all procedure calls generate storage space dynamically. A few other routines are stored in the cassette buffer at location 828.

One of the difficulties in code generation for home computers is the lack of tools usually found on larger machines, namely editors and linking loaders. The orientation of stand-alone home computers is that they need to run only programs entered via the resident monitor, which puts bytes of memory, or through the Basic interpreter, which has some editor functions, but is unsuitable for data entry for arbitrary files. This is not usually a problem for applications that execute solely on the 6510, but makes down-loading from another computer difficult. A loader and a data entry program must be provided to facilitate these functions. The kernal routines make it fairly easy to write a program that will input a string of numeric digits from either the keyboard, or a disk file, store these digits, and then use the routine at address 48371 in the Basic ROM to convert the string to a floating point number. This can be converted, using the routine at address 48282 to an integer value, if this is the necessary form.

The choice to use the Basic ROM routines also dictated the size of real numbers and integers. The Basic routines for numerical manipulation use locations in zero page as two floating point accumulators, and the Basic ROM routines make extensive use of these locations. Since the floating point representation in the floating point accumulator (referred to as the FAC) and the alternate floating point accumulator (the AFAC) are 6 bytes, the real variables and constants in the generated code are also 6 bytes. Likewise, the Basic routines assume 2 byte integers, and this convention is used

in the Pascal compiler.

The downloading of the code itself presented another challenge. When the project was first conceived, it seemed that assembly code would be more readable, and generally easier to handle that machine code. After further analysis, it was determined that it is easier to output machine code, which could be downloaded via the terminal program available on the microcomputer. Because it is completely numeric, it requires no translation and can be easily loaded via a short Basic routine. It has the further virtue of being immediately executable.

In summary, then, code generation for this microcomputer is not difficult, but the issues in transferring the code and required data are complex. The home computer is not designed to interface easily with other machines and thus has few tools to make this easy. The programmer interested in cross-compilers must be concerned with loading and running the output of the compiler, and this can be difficult.

CHAPTER VII

SUMMARY AND CONCLUSIONS

Conclusions from the Project

The analysis of the requirements of a cross-compiler from a larger time-sharing computer to a home computer produced different difficulties than originally anticipated. In the course of this project it became apparent that the idea of bootstrapping a Pascal compiler for a home computer was not practical. The larger environment is too different than the 6510 environment, and the amount of effort to cause the environment to emulate the 6510 is substantial. Without this emulation, the output of the front-end will not produce usable code for the microcomputer.

Future Work

The construction of a compiler using the existing ROM on the 6510 is an interesting idea. If it were approached as a project that executes only on the 6510, it would eliminate the difficulties of code transferral encountered in this project. The front end could be manually translated from the C code that is produced by Yacc, and rewritten in 6510 machine or assembly code.

Anouther approach to future work would be to expand the compiler created in this project to include a more

complete subset of Pascal. This might involve the addition of multi-dimensional arrays, user-defined types and record types. The subset used in this project allows only variable parameters to arrays and functions, thus limiting the types of arguments that can be passed to subroutines. Adding this would not be trivial, as the complexity of code generation from parameter statements would be much more difficult than the single type of parameter passing allowed in this subset.

There are features available in the Basic ROM routines in the 6510 that could be incorporated in a compiler of the sort addressed here, though the definition of Pascal does not include them. These come from the orientation of the home computer towards graphics and simplicity, and provide such utilities as graphics design, sound generation, and screen output. Basic also has functions to handle string manipulation easily. One of the shortcomings of standard Pascal is the lack of string-handling functions. A compiler including string-handling routines borrowed from Basic would not be much more difficult to write since the routines are already available, and would extend the usefulness of this compiler.

More compilers for this microcomputer have become available since the inception of this project, so the original reason for approaching this project has become less important. There is always room for improvement in the field of compiler construction, though, and

certainly the issue of compiler construction for this particular microcomputer is not closed.

REFERENCES

1. Aho, Alfred and Ullmann, Jeffrey, (1979) Principles of Compiler Design, Addison-Wesley Publishing Co., Reading, Mass., pp.563-567.

2. Wirth, N. (1971) "The Design of a Pascal Compiler", Software Practice and Experience 1 309-333.

3. Lesk, M.E. and Schmidt, E. (1982), Lex- A Lexical Analyzer Generator, Technical Report, Bell Laboratories, Murray Hill, N.J.

4. Kernighan, Brian, and Pike, Rob, (1978) The UNIX Programming Environment, Prentice-Hall, Inc, Englewood Cliffs, N.J.

5. Waite, W. and Goos, G. (1984), Compiler Construction, Springer-Verlag.

6. Johnson, S.C. (1978), YACC: Yet Another Compiler-Compiler, Technical Report, Bell Laboratories, Murray Hill, N.J.

7. Barrett, W. and Couch, J. (1979), Compiler Construction: Theory and Practice, SRA, Inc., Chicago.

8. Amman, U. "On Code Generation in a Pascal Compiler", Software- Practice and Experience 7, 1977.

9. Anklam, P., Cutler, D. Heinen, R. Jr., MacLaren, M. (1982), Engineering a Compiler, Addison Wesley, Reading, Mass.

10. Jensen, Kathleen and Wirth, Niklaus, Pascal User Manual and Report, Springer-Verlag, New York 1974.

11. Russell, David L. and Sue, Jeffrey Y., "Implementation of Pascal Compiler for the IBM 360", in Software- Practice and Experience 6, 1976.

12. Grosse-Lindemann, C.O., and Nagel, H.H. "Postlude to a Pascal-Compiler Bootstrap on a DECSystem-10", in Software- Practice and Experience 6 1976.

13. Richard, Frederic, and Ledgard, Henry, "A Reminder for Language Designers", in Lecture Notes 54- Design and Implementation of Programming Languages, ed. G. Goos and J. Hartmans, Springer-Verlag, N.Y., 1977.

14. Beatty, John, "On the Relationship Between the LL(1) and the LR(1), in Journal of the ACM, Vol. 29, No. 4, October 1982.

15. DeRemer, Frank, and Pennello, Thomas, "Efficient Computation of LALR(1) Look-Ahead Sets", in ACM Transactions on Programming Languages and Systems, Vol. 4, No. 4, October 1982.

16. Knuth, D. E., The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley, Reading, Mass., 1973.

17. Reiss, Steven, "Generation of Symbol Table Mechanisms from Specifications", in ACM Transactions on Programming Languages, Vol. 5, No.2, April 1983.

18. Davidson, Jack W., and Fraser, Christopher W., "The Design and Application of a Retargetable Peephole Optimizer", in ACM Transactions on Programming Languages, Vol. 2, No. 2, April 1980.

19. Tannenbaum, Andrew S., vanStaveren, Hans, and Stevenson, Johan W., "Using Peephole Optimization on Intermediate Code", in ACM Transactions on Programming Languages and Systems, Vol. 4, No. 1, January, 1982.

20. Commodore Business Machines, Commodore 64 Programmer's Reference Guide, Howard and Sons, Inc., 1983.

21. Bush, Derek and Holmes, Peter, Commodore 64 Assembly Language Programming, Hayden Press, N.J., 1984.

APPENDIX A

USER GUIDE FOR THE COMPILER

The following are directions for using the compiler developed during this project.

1. Enter the desired Pascal file and store as a file on the Concurrent XF-610

2. Run the file Compascal, using the Pascal program file as input, and designating an output file, to store the assembly code output of the compiler.

3. Download the assembly code to a floppy disk drive attached to the Commodore 64 computer.

4. Load the assembly code from the disk file in the memory of the Commodore 64.

5. Load the assembler from the file LADS, and run the code through the assembler to develop an executable program, also stored on disk.

6. Load and run the executable module. Data can either be entered on the keyboard or stored separately in a file. Output is displayed to the video screen attached to the Commodore 64.

APPENDIX B

SAMPLE PROGRAMS USED FOR VERIFICATION

```
program one (input, output);
var
 i: integer
 d:array [1..10] of real;
begin
 i := 1;
 while i<=10 do
   begin
   read(array[i]);
   write(array[i];
   end
end.


program two (output);
var
  a:integer;
procedure reverse (var f:integer);
  begin
     if f = 0 then
        write(f)
     else
        begin
          f := f-1;
          reverse(f)
        end;
end;
begin {main program}
   a := 10;
   reverse (a);
end.
```

```
program three (output);
var
   a,b: integer;
   c,d: real;
begin
   a := 1;
   b := a;
   c := 3.2
   d := a + (b mod a) div 4 + c;
   write (a,b,c,d);
end.
```

```
program four (output)
var
   a,b: integer;
   c,d: real;
begin
   read (a,b,c,d);
   if a < c then
      a := a * c;
   if d < 0.0 then
      while d <= 0.0 do
         d := d + 1.0;
end.
```

VITA

Anne Marie Butler

Candidate for the Degree of

Master of Science

Thesis: AN APPROACH TO LANGUAGE IMPLEMENTATION AND CODE GENERATION FOR MICROCOMPUTERS

Major Field: Computing and Information Sciences

Biographical:

Personal Data: Born in Baltimore, Maryland, December 29, 1956, the daughter of Dr. William and Mary McKechnie. Married to Lindsay W. Butler III August 15, 1981.Sons Thomas, born September 7, 1983 and James, born February 7, 1986.

Education: Graduated from Binghamton Central High School, Binghamton, N.Y. in June 1974; received Bachelor of Arts in Applied Math from Harvard University in June, 1978; attended University of Southern California; completed requirements for the Master of Science degree at Oklahoma State University in December, 1986.

Professional Experience: Lecturer, University of Maryland FED, August 1978, to January 1981; Teaching Assistant, Departments of Mathematics and Computing and Information Sciences, Oklahoma State University, September 1981 to 1986; Lecturer, Phillips University, Enid, Oklahoma, Department of Mathematics and Computer Science, August 1981, to August 1983.