NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

DESIGN AND IMPLEMENTATION OF A
"C" COMPILER FOR AN ABSTRACT MACHINE

by

Metin Gursel OZIŞIK

June 1986

Thesis Advisor: Daniel L. Davis

Approved for public release; distribution is unlimited.

ECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
PERFORMING ORGANIZATION REPORT NUMBER(S)	S. MONITORING ORGANIZATION REPORT NUMBER(S)
a. NAME OF PERFORMING ORGANIZATION aval Postgraduate School	6b OFFICE SYMBOL (If applicable) 52
7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School	
c. ADDRESS (City, State, and ZIP Code) onterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000
a. NAME OF FUNDING / SPONSORING ORGANIZATION	8b. OFFICE SYMBOL (If applicable)
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
c. ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO	TASK NO	WORK UNIT ACCESSION NO

1 TITLE (Include Security Classification) UNCLASSIFIED
esign and Implementation of a C Compiler for an Abstract Machine

2 PERSONAL AUTHOR(S)

3a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Masters Thesis	FROM _____ TO _____	1986 June 20	109

6 SUPPLEMENTARY NOTATION

7 COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	"C" Compiler

9 ABSTRACT (Continue on reverse if necessary and identify by block number)

The technique of formal abstraction provides an appropriate tool for specifying an interface between layers of computer hardware and software. An abstract machine called AM has been built to address the problem of portability and reusability of software. This thesis is the design and implementation of a "C" Compiler for this abstract machine.

0 DISTRIBUTION / AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	UNCLASSIFIED	
2a NAME OF RESPONSIBLE INDIVIDUAL Prof. Daniel L. Davis	22b TELEPHONE (Include Area Code) 408 646-3091	22c OFFICE SYMBOL 52Dv

D FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

Approved for public release, distribution unlimited

Design and Implementation of a C Compiler for an Abstract Machine

by

Metin Gursel Ozisik
Ustegmen, Turkish Navy
B.S., Turkish Naval Academy, 1980

Submitted in partial fulfillment of the
requirement for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL PASTGRADUATE SCHOOL
June 1986

ABSTRACT

The technique of formal abstraction provides an appropriate tool for specifying an interface between layers of computer hardware and software. An abstract machine called AM has been built to address the problem of portability and reusability of software. This thesis is the design and implementation of a "C" compiler for this abstract machine.

TABLE OF CONTENTS

I. INTRODUCTION

In today's computer world, portability is a well-known problem which arises in a variety of situations. Since computer software evolves in connection with a particular hardware environment, and often assumes features closely related to characteristics of its own hardware, this problem has been unavoidable.

Formalizing the relationship between hardware and software resources was treated is a previous NPS thesis by Yurchak [Ref. 1], whose efforts resulted in the specification and implementation of an abstract machine, called AM.

The abstraction of a bit mapped display resource was added to AM in another NPS thesis by Hunter. [Ref. 2]

Finally, an abstraction of a formally specified reusable database was added to the same machine by Zang. [Ref. 3]

This presentation is a further extension of the work started by Yurchak and Hunter: An abstract computer and its programming environment. Its major objective is a compiler for a subset of the C language for AM.

A. THE PORTABILITY PROBLEM

It is well-known that moving large programs from one machine to another is frustrating work. And it is also known that once the software has been moved to the new machine, it is not predictable whether or not it will work as before.

Even if it seems to work, it may consume more resources than expected.

For a couple of reasons, the portability problem is getting worse, not better:

- Computer architectures have been changed to make them look like what the programmer wants
- The number of the devices included in modern architectures has been maximized
- Both languages and machines are related to the data they manipulate in an implementation dependent way

These and other factors make the portability problem a difficult task, and in addition, they affect some other difficult issues like language design and software engineering.

B. CURRENT IMPLEMENTATIONS TO SOLVE PORTABILITY PROBLEM

The usage of high level languages provides a degree of high level abstraction, and provides some measure of software standardization and portability. But the portability of high level languages is limited, since all the layers of software below this high level have to be moved, in order to port such a system.

There are other abstraction levels between the computer hardware and the application environments. Especially operating systems represent a software abstraction of physical resources, and support the layers of software built over this level. Starting with CP/M and UNIX, we have seen some good implementations that provide such an abstract

level to some degree. The main idea of the AM machine is to abstract and formally define other physical resources found in typical computing systems.

II. ABSTRACT MACHINE, AM

The Abstract Machine (AM) is a result of Yurchak [Ref. 1] and Hunter's [Ref. 2] efforts to solve the problem of formalizing the relationship between hardware and software resources. It is implemented as a finite state machine interpreter, with an assembler. Details of the newest version of the AM assembler can be found in Zang's [Ref. 3] thesis.

"Abstraction" describes the separation of the defining properties of an object from other, unnecessary details about it. A programmer is primarily concerned with solving a problem. Appropriately, the tools at his disposal, such as programming languages, development aids, and the programming environment, form a problem solving abstraction. The hardware (and some of the software) on which this problem solving abstraction is implemented, however, is an abstraction of a different sort.

The fuzzy area between software and physical resource abstractions, sometimes simplistically perceived as the boundary between hardware and software, exposes a number of shortcomings in language design and computer architecture collectively termed the "semantic gap".

Narrowing the semantic gap requires significant changes in the fundamentals of computer architecture and language design. Three major factors which significantly contribute to this problem are:

- Informally described semantics;
- Representation dependent data types;
- Arbitrarily designed instruction set architectures.

The AM was designed to fill this semantic gap by addressing the above problems. [Ref. 1].

In the AM implementation, a text file representing an assembly language program is translated by the assembler into a relocatable object module. A loader, part of the AM interpreter, loads this object module into the appropriate cells, and AM executes it.

The following presentation is an implementation of a subset of the high level language "C", for that abstract machine. It is a compiler which compiles C source code and generates assembler source code for the AM.

III. DISCUSSION OF "C" SUBSET

Since commercially good compilers are very large programs and it takes on the average six man-years to write one of them, this research work had to be a small subset of the C language.

The goal was to write a small portion of C in the C language itself, and then by feeding the output of this work into itself, to create a native code C compiler.

Since this work was going to be a race against time, the subset had to be as small as possible, but on the other side, had to be large enough to be able to compile its own source code.

The sub-goal was to use a strictly limited number of features to write the compiler, because any new feature used in the code would require implementation of the same feature in the compiler.

The outcome of this work was not sophisticated enough to compile itself. It evolved as a small subset of the C programming language, so called "Tiny-C". And since it was not sufficient to compile its own code, it is used as a cross-compiler from host MS-DOS computers to the target machine AM.

A. TINY-C SUBSET

Tiny-C is a small subset of C, and a thesis project more than a language. There are many features which a real programming language has to have, but Tiny-C does not.

The Tiny-C compiler was written in five months and is considered to have the fundamental structure of a real compiler. Hopefully it will be modified and improved in the future, and may be usable for real applications.

Appendix A is a listing of the Tiny-C language grammar. But this grammar is obviously not the complete "C" language. At least:

- Structure and union specifiers are not included.
- Functions are not allowed to return addresses.
- Assignments inside the expressions are not allowed, because they were considered as making programs "unreadable". For instance:

  "if ((joe= jimmy+5)) 5)" is not allowed in Tiny-C.

- Multiple assignments are not implemented. For instance:

  "joe= jimmy = 15 * mary;" is an invalid statement in Tiny-C.

B. THE TINY-C COMPILER

Even though the Tiny-C language subset was planned within the limits in this thesis, the Tiny-C compiler can only compile and generate code for an even smaller subset of the above grammar.

The Tiny-C compiler implementation can parse the whole Tiny-C subset and give proper error messages if necassary. But,

- due to the time constraints, and
- due to the restricted capabilities of the target AM machine

the Tiny-C compiler cannot generate code for the whole Tiny-C language.

In the Tiny-C compiler;

- Floating point arithmetic is not implemented. Because it is not supported by AM.
- Bitwise and shift expressions are not implemented, since they are not supported by AM.
- Since AM has strictly defined data types and does not allow type conversions, address, pointer and array types are not implemented.
- Since AM is designed as an operating system independent software machine, the "#include" preprecessor is not implemented.
- Since AM does not have a linker yet, external declarations are not implemented.
- Auto, static, register, boolean types are not implemented.

IV. THE DESIGN

This chapter describes the Tiny-C compiler step by step. But obviously the purpose of this presentation is not to teach the "compiler writing art", or to explain the target Abstract Machine's assembler. Complete documentation for the AM Assembler can be found in Yurchak's thesis [Ref. 1]. For a better understanding of the following structures, Ullmann's "Compilers, Techniques and Tools" [Ref. 4] is recommended as a background reference for compiler writing.

The Tiny-C Compiler is written in nine steps. These are:

- Scanner or Lexical Analyzer
- Grammar
- Recursive Descent Parser with Backtracking
- Data Structures for the Parser
- Error Checking and Error Messages
- Emission of Intermediate Code
- Intermediate Code Optimization
- Data Structures for the Code Generator
- Target Code Generation

We will first go through these steps briefly in order to get acquainted with the architecture of the Tiny-C compiler.

A. SCANNER AND LEXICAL ANALYZER

In general, scanners and lexical analyzers are language independent structures. The same scanner may be used for a couple of different compilers. For this reason we will

introduce this structure even before discussing the Tiny-C grammar.

Contrary to the header of this section, Tiny-C does not have a scanner or lexical analyzer in the classical sense.

Even though the most common way of writing compilers is analyzing the input data stream lexically, and after tokenizing, passing tokens to the parser as they are needed, this was not the way scanning was implemented in this compiler. The Tiny-C scanner is made up of a couple of routines used by a recursive descent parser with a backtracking tool. There is no tokenized data stream.

The idea is to read the input stream into a scanner buffer, (which is implemented as a ring buffer) and parse it there. This technique gives an ability to backtrack and makes it possible to write a very simple recursive descent top-down parser. With such a backtracking tool, the grammar does not need to be massaged to a fully LL(1) grammar, that is even if it is ambiguous in the LL(1) sense. In any ambiguous case, the parser can try all possible options by bactracking.

Let's start by introducing our scanner buffer and its initialization.

```
init_buf()      /*  initialize scanner buffer        */
{
    Reads input source file into scanner buffer. Sets the
pointers for the current place (for initializing procedure,
it is simply the begining of the scanner buffer) and for the
very last character in the scanner buffer.
}
```

The scanner may or may not read the whole input stream at once, because its ring buffer has a limited size. Now the next question is how to get a character from this buffer (since tokens are not used, we have to deal with characters), and if it is the end of the characters, how to read some more input into this ring buffer.

```
char    getchr()    /* get character routine    */
{
    Gets the next character from scanner buffer, and loads
it into global "nextch". If it reaches the current end of
the scanner buffer, it reads some more text from the source
into scanner buffer. If it meets the end of file character,
it sets the "file_end" flag TRUE.
}
```

We even can put a character back into the buffer, if needed.

```
ungetchr()      /* un-get character */
{
    Puts a given character back into scanner buffer.
}
```

After initializing the scanner buffer, we can get as many characters from there as we want to. But parsers are higher level concepts, and they shouldn't deal with the low level structures of scanning like getting three more characters or putting back one. Parsers mostly work on tokens. If we had a pure tokenized implementation, we could simply pop a token number from the scanner buffer. But here we need something to give tokens to the parser. Also white characters and comments should be ignored.

String tokens are given to the parser by the following routine.

```
matchtoken(str,whtchk)     /* match to a given string token */
char    str[],             /* string token                  */
        whtchk;   /* boolean variable for white chr. check */
{
    This routine attemps to read the string token from
scanner buffer. A following white character or delimiter is
optional, and this decision is made by the caller, namely
parser. It returns TRUE, if the token matches (and a white
character, optionally), else returns FALSE. In case of
FALSE, it backtracks in the scanner buffer to its previous
place.
}
```

The following routine attempts to match a single character in the scanner buffer and returns a boolean result.

```
match(chr)       /* match to a single character  */
char    chr;     /* character to match           */
{
        delwht();

/* if character matches, return TRUE    */
    if (nextch==chr)
    {
        nextch=getchr();
        return(TRUE);
    }
    return(FALSE);
}
```

Both of these routines delete white characters first. And in case of FALSE, they do not backtrack to their previous places exactly, otherwise the following routines have to skip white characters one more time. So, in the case of the FALSE or "unmatched" case, they backtrack to the very first character which comes just after the white ones.

```
delwht()     /* delete white characters      */
{
```
    Used by both match-character and match-token routines and skips all the following white characters (blank, tab, carriage return and line feed characters) and the comments in the scanner buffer.
```
}
```

B. GRAMMAR

Since there is not a standard C language grammar, we had to first write a grammar to parse. The Tiny-C subset was discussed in the previous chapter, and its complete grammar is presented in Appendix A.

In this grammar (Appendix A), any terminal or non-terminal followed by a '*' character means "none or more," followed by a "+" character means "one or more," and followed by a "?" character means "optional" or "none or one." Under these definitions for example:

program:
⟨pre-precessor⟩* ⟨data-definition⟩* ⟨function-definition⟩+

The non-terminal ⟨program⟩ goes to any number of ⟨pre-precessor⟩, followed by any number of ⟨data-definition⟩ and followed by one or more ⟨function-definition⟩.

The '|' character means "or". For example:

pre-precessor:
"#define" ⟨file-definition⟩ |
"#include" ⟨file-definition⟩

Thus, ⟨pre-precessor⟩ goes to "#define" followed by ⟨file-definition⟩, or, "#include" followed by ⟨file-definition⟩.

The '!' character means "allowed at most once." For example:

```
switch-statement:
    "switch"  "("  (arithmetic-expression)  ")"  "{"
    (case-stmt)+  "}"

case-stmt:
    "case" | "default"!  (constant-expression)  ":"
    (statement)*
```

Thus, (switch-statement) can go to "default" at most once.

C. PARSER

A very simple form of a working parser is presented in Appendix B. It is a recursive descent parser but with a backtracking feature. There is a one-to-one correspondence between non-terminal names in the grammar and function names in the parser. The reader is encouraged to read the parser with an eye on the grammar. With the grammar's help, it is not difficult to understand the structure of the parser.

In this first version of the Tiny-C parser, all functions backtrack if they fail. In the real Tiny-C environment this is extremely unnecessary, because in the Tiny-C grammar, ambiguity exists in a few places only. The reason this first version is presented in Appendix B is its clarity and simplicity. In the following versions, unnecessary backtracks have been taken out.

In all the routines in the parser, there are two backtracking tools. First, the "oldp" old pointer points to the parser's previous place in the scanner buffer, and second, the "line_no" line number keeps track of the current

line number for error checking purposes. If a function fails, these routines backtrack to their previous states and try to find another legal path to parse.

Appendix C has the routines for the basic nonterminals and terminals of the Tiny-C parser. So, it presents a working version of that parser with Appendix B.

D. DATA STRUCTURES FOR THE PARSER

Now is the time to introduce some data structures to improve the Tiny-C parser. The first one is going to be a name string structure since all the following tables need this structure.

1. Name String Implementation

A name string is basically a big character array (or a string) which holds all the names used in the source file. Tiny-C has two routines to implement this structure:

The first one is used to add a new name into the name string, and the second one is used to look for a given name.

```
add_name()      /* add a name into the name string  */
{
    Adds a new name into the name string from the
"id_name" global variable. The "id_name" variable holds the
current identifier name all the time. The function
"identifier" in the parser sets this variable whenever it
parses an identifier.
}

find_name()          /* find a name in the name string      */
{
    Looks for "id_name" in the name string. If found, it
loads the identifier's address into a pointer and returns
TRUE, else returns FALSE.
}
```

In the current version of Tiny-C, the name string was implemented completely sequentially. Instead, there could have been a hashing mechanism, which would be much more efficient. When testing the whole compiler, it was observed that a large number of predefined constant names and variables was making execution slow.

2. Constant Table

Constants are implicitly declared elements. In the Tiny-C compiler, a constant table is implemented to take care of them. Since every occurence of a constant denotes the same declaration, we do not need to check if a constant occurs more than once. We simply add each constant into the constant table as it occurs.

```
add_num()    /* add an integer number into constant table */
{
    Adds an integer numeric value into the constant table
if it is not in there. And returns its address in a
pointer.
}
```

In the current version of AM, integers are the only numeric type. So it is the only numeric type implemented in Tiny-C, and is the only constant denotation required.

Since input data is an integer for the above routine, and since source file is read as character stream from the scanner buffer, we need a string-to-numeric conversion routine, to convert text input into numeric values.

```
str_num()       /* string to numeric      */
{
    Takes  a  string  "num_name"  (numeric name)  which  is
set  by  the "constant()" routine  in the parser,  calculates
its   numeric   value,   and   returns it  in   the   "num_cnst"
(numeric constant) global as an integer.
}
```

3. Definition Table

In Tiny-C, the preprecessor command "#define" lets us define constant identifiers. So, a definition table is implemented for these identifiers.

In case of a "#define" declaration, we need to add a new constant identifier into the definition table.

```
add_cnid()                  /* add constant identifier      */
{
    First,  checks  if  the given id-name is already  in  the
definition table.  If so it gives an error,  since definition
of the same constant-id more than once is nonsense. Otherwise
it  nadds that given constant identifier into the  definition
table.
}
```

The next problem in implementing constant identifiers is finding the corresponding values for these constant id-names, if they are met when parsing a program.

```
find_cnid()                 /* find constant identifier */
{
    Takes    a  constant   identifier  name and  looks  for it
in  the definition table.  If found,   it sets a  pointer  to
its  place in the definition table and returns TRUE,  else it
returns FALSE.
}
```

4. Scoping Rule

In classical compilers, symbol tables are primarily responsible for establishing the scoping rules. The Tiny-C

compiler solves the scoping problem in a different way.

Our Tiny-C compiler has a variable string which holds all valid variable names in the current scope. When the parser starts parsing a new function or a new compound statement, (namely a new "block" in block structured language literature), the parser puts a mark into the variable string to define the beginning of the new block, and adds the following variable declarations into the same string. Whenever the parser goes out of a block, it deletes the very last block's variables from this string. (Since the Tiny-C compiler is a one-pass compiler, the deletion of the variables for the last block is acceptable in this case). So, any time a variable is used, the compiler looks for this variable in the variable string, starting from the end to the beginning. If found, it finds a pointer to the symbol table for this variable, if not, it gives an error message since that particular variable is unknown (or out of scope).

```
find_var()      /* find a variable in variable string */
{
    Takes an id-name and looks for it in the variable
string.  If found,  it sets a pointer to the symbol table
pointing to its place in there and returns TRUE, otherwise
it returns FALSE.
}
```

We introduced searching for variable names in the variable string before discussing inserting them. The reason is, whenever the parser meets a new variable declaration, it is supposed to add that new variable into both the symbol

table and the variable string. In the Tiny-C compiler, one single routine does both these duties. Since the symbol table is not introduced yet, we didn't meet this routine either.

Here, the theory to satisfy scoping rule is: mark the beginning of a block in the variable string when starting to parse a new block, and delete the most recent block's variables when exiting from it. So any variable which is not in the variable string is automatically out of scope.

5. Symbol Table

In the Tiny-C implementation, the symbol table is responsible for variables, function names, label names, and function arguments.

Let's first start with how to add a new variable into the symbol table when a variable declaration occurs.

```
add_var()       /* add variable     */
{
   Gets a new variable's id_name and gets its type, then adds
it into symbol table and variable string.
}
```

Similarly, label declarations require label names to be added into the symbol table, too. But we shouldn't add labels into the variable string, since in 'C' they do not satisfy the same scoping rules as variables.

```
add_label()     /* add a label into symbol table    */
{
    Gets a label, and adds it to the end of the symbol table.
}
```

Whenever the parser meets a new label declaration, it adds this label into the symbol table by the above routine. But it must be smart enough not to accept duplicate label declarations.

One pointer is assigned to point to the beginning of the very last function in the symbol table. So, when the parser meets a new label declaration, it first starts from the beginning of the last function in the symbol table, and goes all the way down to the end of it, to look for a same label name. If it finds one, it gives a duplicated label declaration error, since the same label is not allowed to be declared twice in the same routine in this language. The following routine does this job in the Tiny-C compiler.

```
dup_lbl()   /* is duplicate label?     */
{
    Checks if the same label name has been declared before.
}
```

6. Label Table

In the C language, any label referenced by a goto statement has to be declared somewhere in the same function. Classically, compilers read the source file twice. But the number of input/output operations is very important for total execution speed. Since the Tiny-C compiler is designed as a "one-pass-compiler", we immediately have this problem: detection of undeclared labels.

Classical two pass compilers read all label declarations in the first pass. So, in the second pass they can check if

"goto label" statements are valid. When our one-pass Tiny-C compiler meets a goto statement, and if the referenced label name has not been declared yet, it is unpredictable if this label is going to be declared in the following statements. To solve this problem, Tiny-C implements a label table, and at the end of every function, it checks if a referenced but undeclared label exists.

Whenever a label is referenced by a goto statement, the compiler saves it in the label table by the following routine.

```
save_lbl()              /* save label into the label table */
{
    Inserts a label which is referenced by a "goto"
statement into the label table for future checking.
}
```

And at the end of every function, the compiler checks if the labels referenced by goto's were ever declared in the function.

```
check_labels()                  /* check labels     */
{
     Called by the parser at the end of every function body.
Checks if labels in the label table are declared in the
symbol table.
}
```

7. Function Calls

Tiny-C keeps function names and their argument counts in the symbol table. In case of a function call, it checks if this function has been called before, and if it has not, enters its name and argument count into the symbol table.

If it has been entered before, it checks if the argument count in the new function call is the same as the one in the symbol table. If the argument counts are not the same, it gives an "inconsistent argument count" error.

```
add_fun(fun_no)      /* add function into symbol table   */
char    *fun_no;
{
```
Adds a function name and its argument count into the symbol table in case of a function call, and if it is the first call of the function. If it is not the first call, the function is already in the symbol table, so, it checks if argument counts match. In both cases, it returns the function's function number (basically symbol table entry number) to the parser, to emit intermediate code.
```
}
```

8. Function Declarations

In the C language, parameter declarations follow a function declaration. Parameter names have to be given inside parentheses immediately following a function name, and then they have to be declared one more time with their types.

The following parameter declarations have to match the ones given with function name. Tiny-C has two routines to get this mechanism to work properly.

```
chk_prmt()      /* check parameter  */
{
```
At the end of a parameter declaration, this routine checks if that parameter was given as one of the function's arguments, or if it is declared more than once! If everything is proper, it enters the parameters' type into the symbol table, since the parameter name was already entered before (when parsing the parameter list following the function name).
```
}
```

And, at the end of all parameter declarations, compiler has to make sure that all the arguments given with function name were declared as parameters.

```
chk_parms()      /* check all the parameters */
{
    When parameter declarations are done, checks if there is
any parameter name in the symbol table, without its type.
Since parameter names are entered into the symbol table when
parsing the parameter list, and types are entered in there
when parsing the following parameter declarations, if there
is any parameter with its type missing, that means it is not
declared.
}
```

These are all the data structures, used by the parser to manage variables, constants, labels, function names and arguments, and all remaining structures in the Tiny-C parser. The following section improves the parser one more step, and handles the error checking mechanism.

E. ERROR CHECKING

A list of error and warning messages used in the Tiny-C compiler is given in Appendix D.

Error and warning messages are given by the following routines:

```
err_msg(msg_no)      /* error messages    */
char    msg_no;
{
/* increment error counter          */
    ++err_cnt;

/* give line number of the error    */
    printf("%d error! ",line_no);
```

```
/* and give the error message        */

  switch(msg_no)
    {
    case list for all error messages described in Appendix D.
    }
}


warning(msg_no)      /* warning messages */
char    msg_no;
{
/* give line number of the warning     */
    printf("%d warning! ",line_no);

/* give the message                    */
  switch(msg_no)
  {
  case list for all warning messages described in Appendix D.
  }
}
```

F. INTERMEDIATE CODE GENERATION

In order to generate code for the target machine, first the compiler has to build a parse tree. Appendix E is a list of nodes that form Tiny-C parse trees.

Now, the same old heavy-duty parser can shoulder one more job: emissions of intermediate code.

The following routine does the intermediate code emissions, when called by the parser. It takes two arguments; the node itself, and the number of the children of this node. If there is not any error up to that time, the parser emits the code into an emission table, (which is in fact a flattened parse tree) and increments the emit-counter.

```
emit(node,child)     /* emit intermediate code                */
char    node,        /* node kind to emit                     */
        child;  /* # of the children belonging to this node*/
```

```
{
/* if there is not any error, give emissions     */

        if (!err_cnt )
        {
                emitstr[emit_cnt] = node;
                emitchl[emit_cnt] = child;
                ++emit_cnt;
        }
}
```

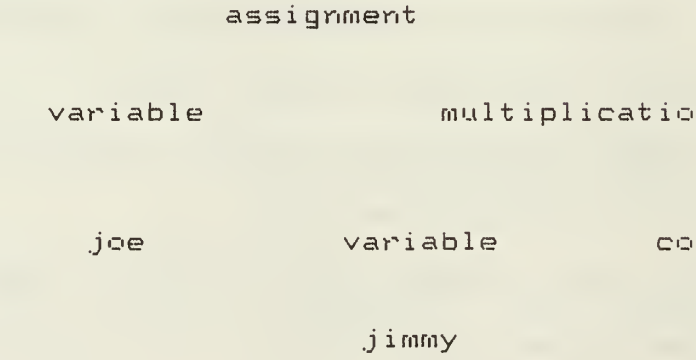
G. POSTPONED EMISSIONS

There are times when we do not want to emit code in the same order as we parse. An assignment statement is a good example for this situation.

Suppose we have the assignment:

joe = jimmy * 5;

The parse tree for this statement is:



Since our parse tree is in flattened form, the order of the intermediate code emissions for the above tree, should be:

jimmy, variable, 5, constant, multiplication, joe, variable, assignment

But this is not the same order we parse! There may be some quick solutions for this particular problem. But the

case might be worse than the above one. Consider the following statement:

```
joe[ (jimmy*15) % mary++ ] = joe[5];
```

Here, the left value is not a simple variable. It is an array element with a complex index expression.

Summarizing, there are cases, when we simply do not want to give emissions immediately. We want to save them, and then at the end of some certain expressions we want to emit them. This type of emission is called "postponed emission."

Up to now, our recursive descent parser has been suffering the same problem. But for the sake of simplicity, we ignored it. Now is the time to build some mechanisms to make the parser be able to postpone emissions.

First of all, we have to make our emission tool more flexible. The following is revised version of our "emit-code" function.

```
emit(node,child)
int     node,
        child;
{
/* if there are not any errors, give emissions    */

      if (!err_cnt )
      {
              * (emitptr[0] + (*( emitptr[2] ))) = node;
              * (emitptr[1] + (*( emitptr[2] ))) = child;
              ++(* (emitptr[2]));
      }
}
```

As can be seen, this revised version is not restricted to emit code into emission table all the time. It can emit code

into any table which is addressed by "emitptr" pointers. That is, by setting these pointers somewhere else, we can "redirect" the emissions.

The following routine directs emissions into a given pointer set. This given pointer set is supposed to be pointing to a table, of the same type as the emission table.

```
drct_emit(emit_ptr, ptr1, ptr2, ptr3) /* direct emits */
int    *emit_ptr[],     /* pointer set to emissions     */
       ptr1[]      ,
       ptr2[]      ,
       *ptr3       ;    /* pointers to new direction    */

{
       emit_ptr[0]=ptr1;
       emit_ptr[1]=ptr2;
       emit_ptr[2]=ptr3;
}
```

As we have seen before, our emit-code routine emits into a table, pointed to by the "emitptr" global emission pointers. But, if we redirect these pointers into somewhere else, don't we lose the address of the previous table? So, we have to be able to save our previous emission addresses somewhere. The following routine saves these pointer addresses in given ones.

```
rplc_emits(ptr_a,ptr_b)      /* saving emit pointers    */
int    *ptr_a[],
       *ptr_b[];        /* pointer sets to both emit-tables */
{
       ptr_a[0]=ptr_b[0];
       ptr_a[1]=ptr_b[1];
       ptr_a[2]=ptr_b[2];
}
```

And the very last problem: We are able to redirect our "emitptr" emission pointers into some tables (then obviously

successive emissions are then entered into these tables). We are able to save the previous value of these pointers. But what about the "postponed emissions". Namely the ones we saved somewhere else other than our emission table. The following routine transfers previously saved emissions from one table into another.

```
trns_emits(emit_a, emit_b)      /* transfer emits          */
int     *emit_a[],              /* destination table pointers   */
        *emit_b[];              /* source table pointers        */
{
        char    i;

        for (i=0; i( ( *(emit_b[2])); ++i)
        {
            *( emit_a[0] + ( *(emit_a[2]))) = *(emit_b[0]+i);
            *( emit_a[1] + ( *(emit_a[2]))) = *(emit_b[1]+i);
            ++( *(emit_a[2]));
        }

}
```

H. CODE OPTIMIZATION

Under normal conditions, code optimization can be done on both intermediate code and target code. When generating target code, compilers attempt to find the best code generation sequence, eliminate common sub-expressions, minimize the number of temporary variables. And after code generation is done, they pass through it again one or two times, for peep-hole optimization, jump optimization, etc.

Our Tiny-C intermediate code has a flattened tree structure; it is possible to traverse it as a tree. In order to do this, we will need some interface routines between

this flattened form and a real tree structure. Then we can logically look at it as a tree and travel from root to leaves or vice-versa.

In this thesis work, it was decided to generate code as quickly and simply as possible. So the Tiny-C compiler uses sequential code generation, even though it is not the best way to do it.

Since our code is going to be source code for the AM assembler, it is not going to be easy to work on a "text" file, to optimize it. At this point, we can work on our intermediate code to make it more effective. So, contrary to the classical compilers, our code optimization is going to be only on intermediate code, instead of both intermediate and target codes.

There are several things we do in the code optimization phase:

- Removing dead code
- Label/jump optimization
- Emitting imbedded assignments

The last one cannot be classified as part of code optimization phase, although we deliberately left it to this point. We will see why pretty soon.

1. Dead Code Elimination

In some cases, the Tiny-C compiler generates dead-code. For instance:

In the intermediate code list, there is a node, called "DUMMY". Sometimes our parser may emit some code, but then it may realize that this code is not necessary. In that case emitting a "DUMMY" node makes this previous code "out of concern" or a "dummy statement".

In fact, such a tool is not truly necessary, but was used in early versions of the compiler. In the following phases this "DUMMY" node was used only in the "case" statement. Due to constraints on time, it has not been removed.

As we discussed before, this thesis is a presentation of the first version of the Tiny-C compiler, and hopefully a reference for its future authors, rather than a discussion about compiler writing techniques.

Nevertheless, to simplify the tree we can remove this "DUMMY" node and its children.

In addition, there may be dead-code that is generated by the compiler. An example:

```
        joe = 5;
        goto there;
        joe = jimmy*5;
        ++jimmy;
there:
```

Here two statements, in the third and fourth lines are dead code. They will never be used. So, we can remove this dead code from the parse tree.

2. Dead Label Elimination

In general, any label declaration is automatically the beginning of a new basic block. However if there is no

"goto" for this label, then such a label is part of a larger basic block.

Having basic blocks as large as possible removes the amount of data transfer between registers and memory. In other words it reduces the number of "register cleaning" operations.

So, if we detect labels, which are declared but never used, removing them is going to be an improvement.

3. Temporary Variables in the Front End

There is one more thing that has to be done when passing over the intermediate code for optimizing purposes.

In the parser, arithmetic expressions following a "switch" reserved word are assigned to some temporary variables. These temporary variables are represented by "TVAR" nodes, with a temporary variable number. Since the result of those arithmetic expressions are assigned to "TVAR" nodes, and these variables are compared with "case" labels, we have to allocate memory for these nodes just as we are going to do for normal variables. The values of "TVAR" nodes may or may not reside in their allocated memory locations, they may be kept in registers, too. The register manager in the following section will treat them just like variable nodes.

In fact, all variables are referred to by their symbol numbers, or their symbol table entry numbers. And at this point, we know our symbol table length. So we can assign

some new symbol numbers to these "TVAR" nodes, and change their names to "VARB" variable nodes. Then the register manager can take care of the rest.

4. Code Optimization, Phase 1.

The following routine is the first part of the intermediate code optimization, and is called just after the parser.

```
frstopt()   /* first pass of optimization   */
{
    - Detects  dead-code  and replaces it  with  "NOOP"  no
operation  nodes.

    - Detects  unused  labels and replaces them  with  "NOOP"
nodes.

    - Replaces  "TVAR" nodes with "VARB"  nodes  and  assigns
them  new symbol numbers starting from the last symbol number
in symbol table.
}
```

5. Separation of Front End and Code Generator

Up to now, our intermediate code has been in memory, in its allocated location (emission table). The emission table has to be large enough to be able to keep the largest size program in it, because of its fixed size. If the input source file is too big to fit into our emission table, Tiny-C responds with an error message. (This is one of the reasons it is called Tiny-C).

It is possible to pass this emission table to the second, target machine dependent part of compiler, but it would not be efficient.

There is a logical separation between parser/ intermediate code generator and target code generator. The first part is totally language dependent and machine independent, and the second part is machine dependent but language independent. So, putting a physical separation between these logically independent units is always a good idea, and has been implemented in most compilers.

For this reason we should end the first part of this compiler here. But before doing this, we have to pass the outcome of this part to the second part of compiler (basically, the code generator of Tiny-C).

The code generator is going to need intermediate code, a symbol table, a constant table, and the number of temporary variables used by the parser. All this information has to be written in some place for later access by the code generator.

But we have a last minute problem here, which we deliberately ignored up to now. This is "imbedded assignments."

6. Imbedded Assignments

In the C language the statement;

```
joe = jimmy++ * 5;
```

is in fact two different statements:

```
joe = jimmy * 5; and a following:
++jimmy;
```

The second statement here is an "imbedded assignment." We didn't emit code for imbedded assignments up to now, and in fact we have ignored this problem on purpose. Because right now, when writing intermediate code into a quad file, we can simply emit these codes without any effort.

7. Code Optimization, Phase 2. The Quad File Filter

The following routine is the second part of the intermediate code optimizer. It is called just after the first-pass optimizer.

```
scndopt()       /* second-pass optimization   */
    Creates a quad file named "TC.QQQ" and:

  - Writes intermediate code in this file, without
    "NOOP" codes and with additional imbedded assignments.

  - Marks end of intermediate code

  - Writes symbol table

  - Writes number of the temporary variables (TVARs)

  - Writes constant table

  - Writes name string

  - And closes that quad file.
}
```

I. DATA STRUCTURES FOR CODE GENERATION

The final step is code generation for the Abstract Machine.

As discussed before, the output of this compiler is not going to be binary code which is ready to be linked and run. It is going to be a source file for the AM assembler, so it will be readable.

Since this is the second part of the compiler, it receives the work done in the first part. The following routine reads a Tiny-C quad file from the disk.

```
read_quad()     /* read quad file       */
{
    Reads quad file from disk in a sequence of intermediate
code, symbol table, constant table and name string.
}
```

Now the compiler has all the information it needs to go ahead and generate code. But right now it does not have any tools to do this. We build some tools first, to help the code generation phase.

The target machine AM theoretically has an unlimited number of registers. This is not realistic. So, the Tiny-C compiler considers that AM has a reasonable number of registers, and tries to manage them properly.

Keeping all the variables and all the intermediate results in registers would be awfully nice. But since this is impossible and we are going to run out of registers after generating a piece of code, we will need a "register manager" to handle the limited number of registers properly. Tiny-C compiler does not have a single "register manager" routine. Instead, we will introduce a couple of routines, which manage AM registers properly.

1. Address Descriptors

As it is known, a compiler cannot keep all the variables in registers all the time. So, it is obvious

that a variable may be in a register, or in its allocated memory location, or both, at a particular time. A compiler needs a mechanism to keep track of the current addresses of all variables. The following routine sets symbol addresses by given parameters.

```
addr_dscr(sym_no,status,reg_no) /* symbol addr. descriptor*/
char    sym_no,                 /* symbol number          */
        status,                 /* address status         */
        reg_no;                 /* register number        */
{
    Sets current addresses of variables.  All variables
have an  8-bit  value  address descriptor.  Status may  be
"in-register",  "in-memory"  or "in-both".  If  7th  bit  of
this  descriptor  is  1,  that means  variable  is  in  its
allocated memory location.  If the value stored in bits 0 to
6  is zero,  means variable is not in any register.  If it it
different from zero,  that value minus one gives the register
number which symbol is stored in.
}
```

Exactly the same problem exists for constants. Even though constant values are fixed and they reside in a constant table all the time, the compiler should not transfer a constant value into a register if it is already in one. The following routine sets a constant address descriptor.

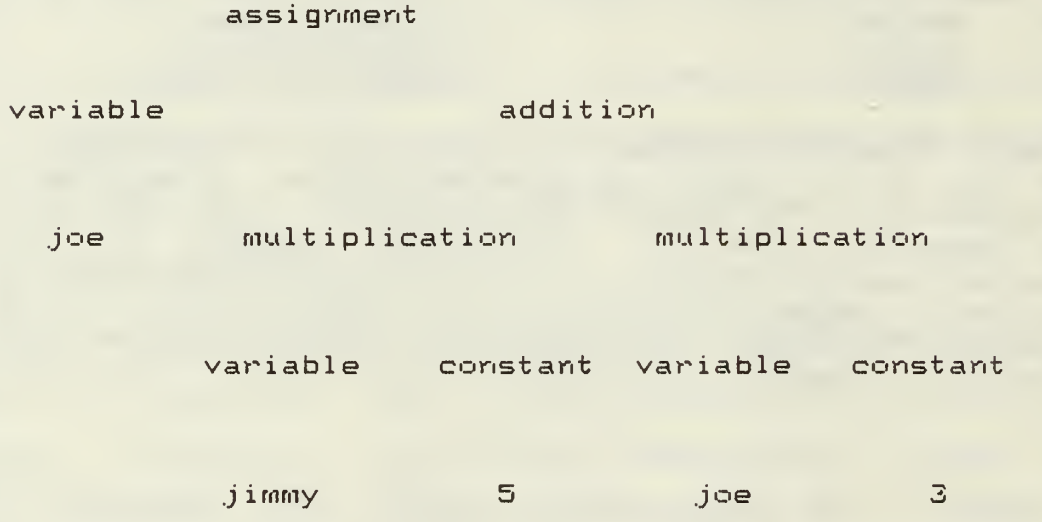
```
cnst_adr_dscr(cnst_no,status,reg_no)
int     cnst_no;              /* constant number        */
char    status,               /* status                 */
        reg_no;               /* register number        */
{
    Sets current addresses of constants.  All constants have
an  8-bit value address descriptor.  If 7th bit of this value
is 1,  and all others are zero,  that means the constant  is
not in any register.  Otherwise the value  of this descriptor
gives the register number which the constant  resides in.
}
```

2. Temporary Management

There is one more address problem. When calculating an arithmetic expression, we may have a couple of temporary results. For instance:

"joe=jimmy * 5 + joe * 3;"

The statement has the following parse tree:



Here, the compiler calculates "jimmy * 5" and "joe * 3" first. Since it has to keep these results somewhere temporarily, we have to manage these temporaries and keep track of their addresses.

Tiny-C compiler manages temporaries' addresses exactly in the same way as it does for variables. In addition, it may dispose a temporary, so we can use the same temporary number somewhere else later.

```
dispose_temp(temp_no)    /* dispose temporary    */
char    temp_no;
{
    Disposes the given temporary variable.
}
```

When the code generator finishes a statement completely, there is no need for any temporary, in Tiny-C's sequential code generation order. So at the end of every statement, the compiler disposes of temporary variables.

```
clean_temps()          /* clean all temporaries */
{
    Disposes all the temporaries.
}
```

Compiler needs a new temporary every time it calculates a temporary result. So, the following routines provide new temporaries to the code generator.

```
get_a_temp(temp_no)        /* get a temporary variable  */
char    *temp_no;
{
    Finds an unused temporary, returns its number to the code
generator, and marks it "used."
}
```

3. Finding Current Addresses

The compiler should be able to figure out any given token's address at any time. Tiny-C uses the following routines for this purpose.

```
is_inreg(token_no,kind)     /* is token in a register?     */
int     token_no;           /* token number               */
char    kind;               /* token kind                 */
{
    Takes token kind (variable, constant or a temporary
variable) and its token number, returns TRUE if it is stored
in a register, else returns FALSE.
}
```

If a particular token is in a register, it can be figured out which register this one is.

```
get_reg_num(token_no,reg,kind)  /* get register number     */
int     token_no;                 /* token number           */
char    *reg,                     /* register number        */
        kind;                     /* token kind             */
{
    Takes a token number and its kind, and returns its
register number in "reg" pointer.
}
```

After some operations, variable values may be only in registers, and may not be in their allocated memory locations. The compiler should figure out if a given varible is in memory, to avoid transferring it into its memory location unnecessarily.

```
isinmem(sym_no)                 /* is symbol in memory?     */
char    sym_no;                 /* variable's symbol number */
{
    Checks variable's address descriptor, returns TRUE if it
is in memory, else returns FALSE.
}
```

4. Register Management

A register can hold just one single value. But in the Tiny-C compiler, this value can belong to more than one token at the same time; for instance the same register can keep two variables, one constant and two temporary variables in it if they all have the same value at that particular time.

We will define the structure of the register manager like this:

```
#define  MXREG  16      /* # of target machine's registers */
#define  MXVAR  5       /* maximum # of variables that
                           one single register can hold    */
int     *regtr[MXREG],  /* pointers to register variables  */
reg_arr [MXREG*MXVAR];  /* register variable array         */
```

So, every register has an amount MXVAR of register array (reg_arr) locations. These are token descriptors and shows which tokens (variables, constants and temporaries) that particular register has at any time. The size of the register array is MXREG times MXVAR.

The register array keeps the names of the tokens which are loaded in some registers.

Since particular parts of the register array belong to particular registers, we can easily figure out which tokens are in which registers, or which register has which tokens.

In order to calculate a new result, the compiler has to find an unused register to load the value. The following routine provides free registers to the code generator.

```
get_a_reg(reg)              /* get a register              */
char    *reg;               /* register number            */
{
    Checks every registers register array locations. If finds
a blank one, returns this register to code generator. If they
are all occupied, evacuates one of them randomly, and returns
it.
}
```

The compiler should be able to load a token from its memory location into one of the registers. The following routine is used for this purpose.

```
load_in_reg(token_no,reg,kind) /* load into a register   */
int     token_no;              /* token number           */
char    *reg,                  /* register number        */
        kind;                  /* token kind             */
{
    Takes a register, a token number and its kind, and
generates code to load it into that given register.
}
```

After loading this token into a register, its address descriptor has to be set as "in both register and memory", and the register manager should set the members of this particular register.

Suppose we load an integer value "3" into a register. The register manager should know that the register is keeping a constant value "3", or which constant number from our constant table is in that register.

Then, suppose we assign this constant to a variable, like in the statement: "joe=3."

The register manager should mark that this particular register has a constant and a variable in it.

The following routine helps the register manager to state that a register is now holding a given token.

```
occupy_reg(token_no,reg,kind)   /* occupy register          */
int     token_no;               /* token number            */
char    *reg,                   /* register number         */
        kind;                   /* token kind              */
{
    Enters given token into given register's register array
location, to mark that this register is holding that given
token in it.
}
```

There may be times when the compiler assigns a new value to a variable but that particular variable may have been stored in a different register before. Since we want to bind a new register to the old variable, we want to release its old register.

```
rel_sym_reg(sym_no)    /* release symbols's register */
int     sym_no;
{
    Takes  a variable,  finds its register,  and deletes  its
membership to this register.
}
```

Sometimes the compiler has to store a token from its register into its memory location. The following two routines do this chore.

```
eva_symbol(sym_no,reg_no)/* evacuate register from symbol */
char    sym_no, reg_no;
{
    Generates  code  to  transfer symbol from  register  into
memory.  Then  sets  the symbol's address descriptor  as  "in
memory" only.
}
```

```
eva_temp(temp_no,reg_no) /* take temporary out of register*/
char    temp_no, reg_no;
{
    Generates  code to transfer temporary from register  into
memory. Then sets its address descriptor as "in memory" only.
}
```

And there are some cases when compiler wants to empty a register completely. For instance, we may do this to release a register.

```
eva_reg(reg_no)          /* evacuate register    */
char    reg_no;
{
    Takes  a  register number,  finds all its members in  the
register array,  and generates code to transfer those members
to  their  memory locations if they are  not  already  there.
(Uses above two routines, actually).
}
```

Before getting out of a basic block, the compiler should empty all registers. The following routine does this task.

```
clean_regs()                /* clean registers      */
{
    Calls "evacuate register" routine for all registers.
}
```

5. Operands for the Operators

In the actual code generation phase, the compiler looks for an operator, and according to operator's type, requests the registers for operands. The following two routines return integer operands in registers.

```
load_two_oprnd(j,r1,r2,step)  /* load two integer operand  */
int     j;                    /* pointer to int. code      */
char    *r1, *r2,             /* registers                 */
        *step;                /* # of the total steps taken*/
{
    Gets two operands from intermediate code, loads them
into two available registers, and returns these register
numbers to the code generator. Since our parse tree is in
a flattened form, the code generator needs to know where it
came in that array-tree, after loading these operands. So,
the "step" is a variable that tells how many steps have
been consumed in the intermediate code.
}
```

```
load_one_oprnd(i,reg,step)    /* load one integer operand  */
int     i;                    /* pointer to int. code      */
char    *reg,                 /* register number           */
        *step;                /* # of the total steps taken*/
{
    Loads the next operand in the parse tree into a
register, and returns the register number with the number of
steps walked in the parse tree.
}
```

The Abstract Machine AM, has some boolean operators that accept only boolean operands. But everything in Tiny-C has integer type. So the code generator should have some tools to convert integer values into booleans. The

following two routines provide boolean operands for boolean operators, whenever they are needed.

```
two_bool(j,r1,r2,step)  /* load two boolean operand  */
int     j;              /* pointer to int. code      */
char    *r1, *r2,       /* registers                 */
        *step;          /* # of the total steps taken*/
{
    Loads two operands.  If they have integer values it loads
the  corresponding boolean values into registers and  returns
them to the code generator.
}


one_bool(i,reg_no,step) /* load one boolean oprnd    */
int     i;              /* pointer to int. code      */
char    *reg_no,        /* register number           */
        *step;          /* # of the total steps taken*/
{
    Returns one boolean operand into a register.
}
```

J. CODE GENERATION

In the Tiny-C compiler, the main routine in the code generator is a large switch statement as is used in most compilers. The compiler generates code for the data segment first, which is just a memory allocation routine for the symbols. Then the code segment comes as the actual code generation phase. The following routine is a subset of the code generation routine for the code segment. Each case element dispatches to the code emitter for that case.

```
code_seg()              /* give code segment                 */
{
    int     i;          /* index variable                    */
    char    r1,r2,      /* register numbers                  */

/* walk emit array from beginning to emit-end     */
for (i=0; i(emitend; ++i)
```

```
/* if node has children, (if it is not a leaf) */
  if (emitchl[i]!=0 )

     switch ( emitstr[i] )
    {
          .
          .
          .
        case  IADD  :  /* integer addition */
                       code_iadd(i,&step);
                       break;
          .
          .
          .
        case  MEND  :  /* end of main function  */
                       fprintf(fl,"    stop\n");
                       break;
  }
}
```

The following routine is used by the above "code_seg()" routine and emits code for integer additions.

```
code_iadd(i,step)     /* integer addition                    */
int      i;
char     *step;       /* # of the steps taken on int. code */

{
    char     r1,r2,    /* register numbers                    */
             temp_no; /* temporary variable number           */

/* load two operands      */
    load_two_oprnd(i-1,&r1,&r2,step);

/* they both might be in the same register,
                    if so, allocate one more register */
    if (r1==r2)
    {
      get_a_reg(&r1);
      fprintf(fl,"    mov     r(0:%d),r(0:%d)\n",r2,r1);
    }

/* since addition will be loaded in r1, evacuate it first */
    eva_reg(r1);

/* code for integer addition      */
   fprintf(fl,"       add     r(0:%d),r(0:%d)\n",r2,r1);
```

```
/* give a number to this temporary result     */
    get_a_temp(&temp_no);
    occupy_reg(temp_no,r1,TEMP);

/* set temporary's address descriptor    */
   temp_var[temp_no]=r1+1;

/* validate emission array for sequential code generation */
   emitstr[i]=TEMP;
   emitchl[i]=*step+1;
   emitstr[i-1]=temp_no;
}
```

Some sample C programs and the code generated for them by the Tiny-C compiler can be found in Appendix F.

V. CONCLUSION

Precise, understandable and enforcable interface standards can provide a way to improve efforts toward portable software. In the Tiny-C implementation we showed a way to improve the programming capabilities of AM, and encouraged programmers to use such a portable and standardizable machine in high level languages.

Unfortunately, this implementation is not completely satisfactory. Because of restricted capabilities in the target AM machine, the Tiny-C compiler does not fully support application programming. Some of these restrictions are:

- Based on the principle of resource abstraction, AM has strictly defined data types. Since it presently does not support conversion between two types, it is a higher level concept than the "C" language. So, contrary to usual implementations, this thesis had an opposite direction: production of a lower level tool in a higher level environment.

- The AM abstract machine does not yet have a complete linker. So, the user is forced to keep the whole program and input/output library in one single module, which is extremely inconvenient in application environments.

- The current version of AM is an emulator, rather than hardware. Even though this is convenient for a development phase, it is not going to be an easy-to-use product for users.

So, further development that could be done for an improved AM environment might include:

- A linker for AM

- Type conversion between AM data types

- An input/output library for the Tiny-C compiler

- A Tiny-C code generator for AM machine code (instead of a source generator for the AM Asembler)

- Given improvements in AM, an extended version of the Tiny-C compiler to cover the whole Tiny-C language grammar

- A compiler version of AM.

APPENDIX A

GRAMMAR FOR TINY-C LANGUAGE

PROGRAM :

```
program:
        (pre-precessor)*   (data-definition)*
        (function-definition)+
```

PRE-PRECESSOR :

```
pre-precessor:
        "#define"    (file-definition)   |
        "#include"   (file-definition)

file-definition:
        '"'   (filename)   '"'        |
        '('   (filename)   ')'

filename:
        (identifier)   (filetype)

filetype:
        '.'   (identifier)
```

DATA DEFINITIONS :

```
data-definition:
        (sc-specifier)?   (declaration)

sc-specifier:
        "auto"       |
        "static"     |
        "extern"     |
        "register"

declaration:
        (type-specifier)   (variable-declaration-list)   ";"

type-specifier:
        "char"       |
        "short"      |
        "int"        |
        "long"       |
        "unsigned"   |
        "float"      |
        "double"
```

```
variable-declaration-list:
        ⟨variable-declaration⟩   ⟨more-variable-declarations⟩*

more-variable-declarations:
        ','   ⟨variable-declaration⟩
```

DECLARATIONS :

```
variable-declaration:
        "*"?   ⟨identifier⟩   ⟨index-declaration⟩?   ⟨initializer⟩?

index-declaration:
        "["   ⟨constant-expression⟩(1)   "]"

initializer:
        "="   ⟨primary⟩

primary:
        ⟨identifier⟩                |
        ⟨constant⟩                  |
        ⟨char-definition⟩           |
        ⟨string⟩

char-definition:
        "'"   ⟨character⟩   "'"

string:
        """   ⟨character⟩*   """
```

FUNCTION DEFINITION :

```
function-definition:
        ⟨type-specifier⟩?   ⟨function-declaration⟩   ⟨function-body⟩

function-declaration:
        ⟨identifier⟩   "("   ⟨identifier-list⟩?   ")"

identifier-list:
        ⟨identifier⟩   ⟨more-identifiers⟩*

more-identifiers:
        ','   ⟨identifier⟩

function-body:
        ⟨type-decl-list⟩   ⟨compound-statement⟩
```

PARAMETER DECLARATIONS :

```
type-declaration-list:
        <parameter-declaration>+

parameter-declaration:
        <type-specifier>   <parameter-declaration-list>   ';'

parameter-declaration-list:
        <parameter>    <more-parameters>

more-parameters:
        ','   <parameter>

parameter:
        '*'?   <identifier>   <index-declaration>?
```

STATEMENTS :

```
statement:
        <compound-statement>          |
        <function-call>   ";"         |
        <assignment-statement>   ";"  |
        <if-statement>                |
        <while-statement>             |
        <do-statement>                |
        <for-statement>               |
        <switch-statement>            |
        <break-statement>             |
        "continue"   ";"              |
        <return-statement>            |
        <goto-statement>              |
        <label>                       |
        ";"

compound-statement:
        "{"   <declaration>*   <statement>+   "}"

function-call:
        <identifier>   '('   <expression-list>   ')'

expression-list:
        <expression>   <more-expressions>*

more-expressions:
        ','   <expression>

assignment-statement:
        <assignment>                  |
        <incremental-expression>
```

```
assignment:
        <lvalue>   "="   <logic-expression>                                  |
        <lvalue>   <shift-assignment-op>   <shift_expression>                |
        <lvalue>   <bitwise-assignment-op>   <bitwise-expression>


shift-assignment-op:
        "+=" | "-=" | "*=" | "/=" | "%=" | ">>=" | "<<="

bitwise-assignment-op:
        "&=" | "^=" | "|="

incremental-expression:
        "++"   <lvalue>              |
        "--"   <lvalue>              |
        <lvalue>   "++"              |
        <lvalue>   "--"

if-statement:
        "if"   "("   <logic-expression>   ")"   <statement>
        <else-statement>?

else-statement:
        "else"   <statement>

while-statement:
        "while"   "("   <logic-expression>   ")"   <statement>

do-statement:
        "do"   <statement>   "while"   '('   <logic-expression>   ')'
        ";"

for-statement:
        "for"   "("   <assignment-list>?   ";"   <logic-expression>
        ";"   <assignment-list>?   ")"   <statement>

assignment-list:
        <assignment-statement>   <more-assignments>*

more-assignments:
        ','   <assignment-statement>

switch-statement:
        "switch"   "("   <arithmetic-expression>   ")"   "{"
        <case-stmt>+   "}"

case-stmt:
        "case" | "default"!   <constant-expression>   ":"
        <statement>*

break-statement:
        "break"   ';'
```

```
return-statement:
        "return"   <expression>   ';'

goto-statement:
        "goto"   <identifier>   ";"

label:
        <identifier>    ":"
```

EXPRESSIONS :

```
expression:
        <string>                                |
        <pointer-expression>                    |
        <address-expression>                    |
        <logic-expression>                      |
        <incremental-expression>

pointer-expression:
        "*"   <array-element>                   |
        "*"   <identifier>                      |
        "*"   "("   <arith-expr>   ")"

address-expression:
        "&"   <array-element>                   |
        "&"   <identifier>

logic-expression:
        <logic-term>   <more-logic-terms>*

more-logic-terms:
        "||"   <logic_term>

logic-term:
        <logic-factor>   <more-logic-factors>

more-logic-factors:
       "&&"   <logic-factor>

logic-factor:
        '!'?   <bitwise-expression>                     |
        '!'?   "("   <logic-expression>   ")"

bitwise_expression:
        "~"?   <bitwise-term>   <more-bitwise-terms>*

more-bitwise-terms:
        "|"   <bitwise-term>

bitwise-term:
        <bitwise-factor>   <more-bitwise-factors>*
```

```
more-bitwise-factors:
        "^"   <bitwise-factor>

bitwise-factor:
        <bitwise-element>   <more-bitwise-elements>*

more-bitwise-elements:
        '&'   <bitwise-element>


bitwise-element:
        <compare-exrp>                          |
        "("   <bitwise-expression>   ")"

compare-expression:
        <compare-term>   <more-compare-terms>*

more-compare-terms:
         <equality-op>   <compare-term>

equality-op:
        "=="   |   "!="

compare-term:
        <compare-factor>   more-compare-factors>*

more-compare-factors:
       <relation-op>   <compare-factor>

relation-op:
        "<" | ">" | "<=" | ">="

compare-factor:
        <shift-expression>                              |
        "("   <compare-expression>   ")"

shift-expression:
        <lvalue>   <shift-op>   <arith-expression>  |
        <arith-expression>

shift-op:
        ">>"   |   "<<"

arith-expression:
        '-'?   <term>   <more-terms>*

more-terms:
        <add-op>   <term>

add-op:
        "-"   |   "+"
```

```
term:
        <factor>   <more-factors>*

more-factors:
        <mult-op>   <factor>

mult-op:
        "*"   |   "/"   |   "%"

factor:
        "("   <arith-expr>   ")"    |
        <constant-expression>       |
        <character-definition>      |
        <function-call>             |
        <incremental-expression>    |
        <lvalue>

constant-expression:
        <constant>                  |
        <constant-identifier>

lvalue:
        <array-element>             |
        <identifier>                |
        <pointer-expression>

array-element:
        <identifier>   <index>

index:
        "["   <arith-expression>   "]"
```

SEMANTIC CONSTRAINTS :

(i) Prohibited for extern and parameter declarations. Mandatory for others.

APPENDIX B

TINY-C PARSER VERSION 1

```
extern          char    buf[], nextch, func_end;
extern          int     bufp, glbptr, line_no;


program()               /* Tiny-C Program        */
{

    while (preprcs())
    ;
    while (data_def())
    ;
    if (!func_def())            goto quit;

    while (!match(EOF))
    {
      func_end=FALSE;

      if (!func_def())
                        goto quit;
    }

        return(TRUE);
quit: return(FALSE);

}


preprcs()               /* pre-precessor         */
{
    int  oldp=bufp,  linep=line_no;

    glbptr=bufp;

    if (matchtoken("#define "))
    {
      if (!cnst_id())           goto quit
      ;
      if (!constant())          goto quit
      ;
    }
    else if (matchtoken("#include "))
    {
      if (!file_def())          goto quit;
    }
    else                        goto quit;
```

```
        return(TRUE);
quit:  bufp=oldp;       line_no=linep;  nextch=buf[bufp];
        return(FALSE);
}


file_def()              /* file definition      */
{
    int  oldp=bufp,  linep=line_no;
    char limiter;

    if (match('"'))             limiter='"'
    ;
    else if (match('('))        limiter='('
    ;
    else                        goto quit
    ;

    if (!filename())  goto quit;


    if (limiter=='"')
    {
       if (!match('"'))                 goto quit;
    }
    else if (!match(')'))       goto quit
    ;

        return(TRUE);
quit:  bufp=oldp;       line_no=linep;  nextch=buf[bufp];
        return(FALSE);
}


filename()              /* file name            */
{

    if (!id())                  return(FALSE);

    if (filetype())
    ;

    return(TRUE);
}


filetype()              /* file type            */
{
    int  oldp=bufp,  linep=line_no;


    if (!match('.'))  goto quit;
```

```
    if (!id())                  goto quit;

        return(TRUE);
quit: bufp=oldp;        line_no=linep;  nextch=buf[bufp];
        return(FALSE);
}


data_def()              /* data definition      */
{
    int oldp=bufp, linep=line_no;


    glbptr=bufp;

    if (sc_spcfr())
    ;
    if (dclrtion())
          return(TRUE);


quit: bufp=oldp;        line_no=linep;  nextch=buf[bufp];
        return(FALSE);
}


sc_spcfr()              /* sc specifier         */
{

    if (matchtoken("auto "))
    ;
    else if (matchtoken("static "))
    ;
    else if (matchtoken("extern "))
    ;
    else if (matchtoken("register "))
    ;
    else      return(FALSE)
    ;

    return(TRUE);
}


dclrtion()              /* declaration          */
{
    int oldp=bufp, linep=line_no;


    if (!typ_spf())           goto quit;

    if (!var_dec_list())      goto quit;
```

```
    if (!match(';'))            goto quit;

        return(TRUE);
quit:   bufp=oldp;  line_no=linep;    nextch=buf[bufp];
        return(FALSE);
}


typ_spf()               /* type specifier       */
{

    if (matchtoken("char "))
    ;
    else if (matchtoken("short "))
    ;
    else if (matchtoken("int "))
    ;
    else if (matchtoken("long "))
    ;
    else if (matchtoken("unsigned "))
    ;
    else if (matchtoken("float "))
    ;
    else if (matchtoken("double "))
    ;
    else      return(FALSE)
    ;

    return(TRUE);
}


var_dec_list()      /* variable declaration list */
{

    if (!vardclr())             return(FALSE);

    while (morevardcls())
    ;

    return(TRUE);
}


morevardcls()           /* more variable declarations   */
{
    int oldp=bufp, linep=line_no;


    if (!match(','))            goto quit;

    if (!vardclr())             goto quit;
```

```
        return(TRUE);
quit:   bufp=oldp;      line_no=linep;  nextch=buf[bufp];
        return(FALSE);
}


vardclr()               /* variable declaration          */
{
    int oldp=bufp, linep=line_no;

    if (match('*'))
    ;
    if (!id())          goto quit;

    if(indxdclr())
    ;
    if (initializer())
    ;

        return(TRUE);
quit:   bufp=oldp;      line_no=linep;  nextch=buf[bufp];
        return(FALSE);
}


indxdclr()              /* index declaration             */
{
    int oldp=bufp,      linep=line_no;


    if (!match('['))            goto quit;

    if (cnst_expr())
    ;
    if (!match(']'))            goto quit;


        return(TRUE);
quit:   bufp=oldp;      nextch=buf[bufp];       line_no=linep;
        return(FALSE);
}


initializer()           /* initializer  */
{
    int oldp=bufp, linep=line_no;


    if (!match('='))            goto quit;

    if (match('{'))
```

```
        {
          if (!expression())        goto quit;

          while(!moreexpr())
          ;
          if (!match('}'))          goto quit;
        }
        else if (!expression())     goto quit;


             return(TRUE);
    quit:    bufp=oldp;     line_no=linep;   nextch=buf[bufp];
             return(FALSE);
    }


    func_def()                          /* function definition  */
    {
        int oldp=bufp, linep=line_no;


        glbptr=bufp;

        if (typ_spf())
        ;
        if (!func_dclr())           goto quit;

        glbptr=bufp;

        if (!func_body())           goto quit;

        func_end=TRUE;


             return(TRUE);
quit:        bufp=oldp;     nextch=buf[bufp];        line_no=linep;
             return(FALSE);

    }


    func_dclr()                         /* function declaration */
    {
        int oldp=bufp, linep=line_no;


        if (!id())                  goto quit;

        if (!match('('))   goto quit;

        if (idnfrs())
        ;
        if (!match(')'))   goto quit;
```

```
        return(TRUE);
quit:   bufp=oldp;    nextch=buf[bufp];       line_no=linep;
        return(FALSE);

}


idnfrs()                        /* identifiers  */
{

    if (!id())                  return(FALSE)
    ;
    while (more_id())
    ;

    return(TRUE);

}


more_id()                       /* more identifiers     */
{
    int oldp=bufp, linep=line_no;


    if (!match(','))  goto quit;

    if (!id())                  goto quit;

        return(TRUE);
quit:   bufp=oldp;    nextch=buf[bufp];       line_no=linep;
        return(FALSE);

}


func_body()                     /* function body        */
{
    int oldp=bufp, linep=line_no;


    if (!type_dec_lst())        goto quit;

    glbptr=bufp;

    if (!cmpn_stmt())           goto quit;

        return(TRUE);
quit:   bufp=oldp;    nextch=buf[bufp];       line_no=linep;
        return(FALSE);

}
```

```
type_dec_lst()          /* type declaration list */
{
     int oldp=bufp, linep=line_no;


     if (par_dclrtion())
     ;
     while (par_dclrtion())
     ;

          return(TRUE);
quit:     bufp=oldp;  line_no=linep;     nextch=buf[bufp];
          return(FALSE);
}


par_dclrtion()    /* parameter declarations */
{
     int oldp=bufp, linep=line_no;


     if (!typ_spf())              goto quit;

     if (!par_dec_list())         goto quit;

     if (!match(';'))             goto quit;

        return(TRUE);
quit:   bufp=oldp;     line_no=linep;    nextch=buf[bufp];
        return(FALSE);
}


par_dec_list()          /* parameter declaration list */
{

     if (!parameter()) return(FALSE)
     ;
     while (morepardcls())
     ;

     return(TRUE);
}


morepardcls()           /* more parameter declarations  */
{
     int oldp=bufp, linep=line_no;


     if (!match(','))  goto quit;

     if (!parameter()) goto quit;
```

```
            return(TRUE);
    quit:   bufp=oldp;     line_no=linep; nextch=buf[bufp];
            return(FALSE);
    }


    parameter()             /* parameter     */
    {
        int oldp=bufp, linep=line_no;

        if (match('*'))
        ;
        if (!id())          goto quit;

        if (indxdclr())
        ;

            return(TRUE);
quit:       bufp=oldp;     line_no=linep;  nextch=buf[bufp];
            return(FALSE);
    }


    stmt()                                  /* statement     */
    {
        if (cmpn_stmt())
        ;
        else if (if_stmt())
        ;
        else if (while_stmt())
        ;
        else if (do_stmt())
        ;
        else if (for_stmt())
        ;
        else if (swtc_stmt())
        ;
        else if (break_stmt())
        ;
        else if (matchtoken("continue ",1))
        { if (!match(';'))          goto quit;
          }
        else if (rtrn_stmt())
        ;
        else if (goto_stmt())
        ;
        else if (func_call())
        { if (!match(';'))          goto quit;
          }
        else if (asnmt())
        { if (!match(';'))          goto quit;
          }
        else if (label())
```

```
        ;
        else if (match(';'))
        ;
        else                            goto quit
        ;
            return(TRUE);
    quit:   return(FALSE);
    }


    cmpn_stmt()                         /* compound statement    */
    {
        int oldp=bufp, linep=line_no;

        if (!match('{'))                goto quit;

        while (dclrtion())
        ;
        if (!stmt())                    goto quit;

        while (stmt())
        ;
        if (!match('}'))                goto quit;

            return(TRUE);
quit:       bufp=oldp;    nextch=buf[bufp];       line_no=linep;
            return(FALSE);
    }


    func_call()             /* function call         */
    {
        int oldp=bufp, linep=line_no;


        if (!id())                      goto quit;

        if (!match('('))                goto quit;

        if (expr_lst())
        ;
        if (!match(')'))                goto quit;


            return(TRUE);
    quit:   bufp=oldp;    nextch=buf[bufp];       line_no=linep;
            return(FALSE);
    }
```

```
expr_lst()              /* expression list      */
{

    if (!expression())          return(FALSE)
    ;
    while (moreexpr())
    ;

    return(TRUE);
}


moreexpr()              /* more expressions     */
{
    int oldp=bufp, linep=line_no;


    if (!match(','))            goto quit;

    if (!expression())          goto quit;


        return(TRUE);
quit:   bufp=oldp;    nextch=buf[bufp];       line_no=linep;
        return(FALSE);
}



asnmt()                         /* assignment statement */
{
    if (assign())
    ;
    else if (incr_stmt())
    ;
    else      return(FALSE)
    ;

    return(TRUE);
}



assign()                /* simple assignment    */
{
    int oldp=bufp, linep=line_no;


    if (!lvalue())                      goto quit;

    if (match('='))
    {  if (!lgc_expr())                 goto quit;
    }
```

```
    else if (shf_asm_op())
    {   if (!shf_expr())                goto quit;
    }
    else if (btw_asm_op())
    {   if (!btw_expr())                goto quit;
    }
    else                                goto quit
    ;

        return(TRUE);
quit:   bufp=oldp;    nextch=buf[bufp];       line_no=linep;
        return(FALSE);
}


shf_asm_op()            /* shift assignment operator     */
{

    if (matchtoken("+= ",0))
    ;
    else if (matchtoken("-= ",0))
    ;
    else if (matchtoken("*= ",0))
    ;
    else if (matchtoken("/= ",0))
    ;
    else if (matchtoken("%= ",0))
    ;
    else if (matchtoken(")>= ",0))
    ;
    else if (matchtoken("(<= ",0))
    ;
    else      return(FALSE)
    ;

    return(TRUE);
}


btw_asm_op()            /* bitwise assignment operator   */
{

    if (matchtoken("&= ",0))
    ;
    else if (matchtoken("^= ",0))
    ;
    else if (matchtoken("|= ",0))
    ;
    else                return(FALSE)
    ;

    return(TRUE);
}
```

```
incr_stmt()                    /* incremental statement*/
{

    int     oldp=bufp, linep=line_no;
    char    pre_op=TRUE;                /* pre-operator */


    if (matchtoken("++ ",0))
    ;
    else if (matchtoken("-- ",0))
    ;
    else                                pre_op=FALSE
    ;

    if (!lvalue())                      goto quit;


    if (!pre_op)
    {
       if (matchtoken("++ ",0))
       ;
       else if (matchtoken("-- ",0))
       ;
       else                             goto quit;
    }


        return(TRUE);
quit:   bufp=oldp;    nextch=buf[bufp];       line_no=linep;
        return(FALSE);
}


if_stmt()               /* if statement          */
{

    if (!matchtoken("if ",1)) goto quit;

    if (!match('('))          goto quit;

    if (!lgc_expr())          goto quit;

    if (!match(')'))          goto quit;

    if (!stmt())              goto quit;

    if (else_stmt())
    ;

        return(TRUE);
quit:   return(FALSE);
}
```

```
else_stmt()             /* else statement       */
{

    if (!matchtoken("else ",1))         goto quit;

    if (!stmt())                        goto quit;


        return(TRUE);
quit:   return(FALSE);
}


while_stmt()            /* while statement      */
{

    if (!matchtoken("while ",1))        goto quit;

    if (!match('('))                    goto quit;

    if (!lgc_expr())                    goto quit;

    if (!match(')'))                    goto quit;

    if (!stmt())                        goto quit;

        return(TRUE);
quit:   return(FALSE);
}


do_stmt()               /* do statement         */
{

    if (!matchtoken("do ",1))           goto quit;

    if (!stmt())                        goto quit;

    if (!matchtoken("while ",1))        goto quit;

    if (!match('('))                    goto quit;

    if (!lgc_expr())                    goto quit;

    if (!match(')'))                    goto quit;

    if (!match(';'))                    goto quit;


        return(TRUE);
quit:   return(FALSE);
}
```

```
for_stmt()              /* for statement          */
{

    if (!matchtoken("for ",1))          goto quit;

    if (!match('('))                    goto quit;

    if (asn_lst())
    ;
    if (!match(';'))                    goto quit;

    if (!lgc_expr())                    goto quit;

    if (!match(';'))                    goto quit;

    if (asn_lst())
    ;
    if (!match(')'))                    goto quit;

    if (!stmt())                        goto quit;


        return(TRUE);
quit:   return(FALSE);
}


asn_lst()               /* assignment list        */
{

    if (!asnmt())               return(FALSE)
    ;
    while (more_asnmt())
    ;

    return(TRUE);
}


more_asnmt()            /* more assignments       */
{
    int oldp=bufp, linep=line_no;


    if (!match(','))            goto quit;

    if (!asnmt())               goto quit;


        return(TRUE);
quit:   bufp=oldp;    nextch=buf[bufp];       line_no=linep;
        return(FALSE);
}
```

```
swtc_stmt()             /* switch statement      */
{

    if (!matchtoken("switch ",1))      goto quit;

    if (!match('('))                   goto quit;

    if (!art_expr())                   goto quit;

    if (!match(')'))                   goto quit;

    if (!match('{'))                   goto quit;

    if (!case_stmt())                  goto quit;

    while (case_stmt())
    ;
    if (!match('}'))                   goto quit;


        return(TRUE);
quit:   return(FALSE);
}


case_stmt()             /* case statement        */
{

    if (matchtoken("case ",1))
    { if (!cnst_expr())                goto quit;
    }
    else if (matchtoken("default ",0))
    ;
    else                               goto quit
    ;

    if (!match(':'))                   goto quit;

    while (stmt())
    ;

        return(TRUE);
quit:   return(FALSE);
}


break_stmt()            /* break statement       */
{

    if (!matchtoken("break ",1))       goto quit;

    if (!match(';'))                   goto quit;
```

```
        return(TRUE);
quit:   return(FALSE);
}


rtrn_stmt()             /* return statement      */
{
    if (!matchtoken("return ",1))      goto quit;

    if (expression())
    ;
    if (!match(';'))                   goto quit;


        return(TRUE);
quit:   return(FALSE);
}


goto_stmt()             /* goto statement        */
{
    if (!matchtoken("goto ",1))        goto quit;

    if (!id())                         goto quit;

    if (!match(';'))                   goto quit;


        return(TRUE);
quit:   return(FALSE);
}


label()                         /* label         */
{
    int oldp=bufp, linep=line_no;

    if (!id())                      goto quit;

    if (!match(':'))                goto quit;


        return(TRUE);
quit:   bufp=oldp;    nextch=buf[bufp];       line_no=linep;
        return(FALSE);
}
```

```
expression()            /* expression    */
{

    if (string())
    ;
    else if (pntr_expr())
    ;
    else if (addr_expr())
    ;
    else if (lgc_expr())
    ;
    else if (incr_stmt())
    ;
    else      return(FALSE)
    ;

    return(TRUE);
}



pntr_expr()             /* pointer expression    */
{
    int oldp=bufp, linep=line_no;


    if (!match('*'))            goto quit;

    if (array_elm())
    ;
    else if (id())
    ;
    else if (match('('))
    {
      if (!art_expr())          goto quit;

      if (!match(')'))          goto quit;
    }
    else                        goto quit
    ;

        return(TRUE);
quit:   bufp=oldp;   nextch=buf[bufp];        line_no=linep;
        return(FALSE);
}



addr_expr()             /* address expression    */
{
    int oldp=bufp, linep=line_no;
```

```
    if (!match('&'))            goto quit;

    if (array_elm())
    ;
    else if (id())
    ;
    else                        goto quit
    ;

        return(TRUE);
quit:   bufp=oldp;    nextch=buf[bufp];       line_no=linep;
        return(FALSE);
}


lgc_expr()              /* logic expression     */
{

    if (!lgc_trm())             return(FALSE)
    ;
    while (lg_trms())
    ;

    return(TRUE);
}


lg_trms()               /* logic terms          */
{
    int oldp=bufp, linep=line_no;


    if (!matchtoken("|| ",0)) goto quit;

    if (!lgc_trm())             goto quit;


        return(TRUE);
quit:   bufp=oldp;    nextch=buf[bufp];       line_no=linep;
        return(FALSE);
}


lgc_trm()               /* logic term           */
{

    if (!lgc_fct())             return(FALSE)
    ;
    while (lg_fcts())
    ;

    return(TRUE);
}
```

```
lg_fcts()               /* logic factors        */
{
    int oldp=bufp, linep=line_no;


    if (!matchtoken("&& ",0)) goto quit;

    if (!lgc_fct())           goto quit;

        return(TRUE);
quit:   bufp=oldp;    nextch=buf[bufp];       line_no=linep;
        return(FALSE);
}


lgc_fct()               /* logic factor         */
{
    int oldp=bufp, linep=line_no;

    if ( match('!'))          /* unary operator       */
    ;
    if (btw_expr())
    ;
    else if (match('('))
    {
       if (!lgc_expr())               goto quit;

       if (!match(')'))               goto quit;
    }
    else                              goto quit
    ;

        return(TRUE);
quit:   bufp=oldp;    nextch=buf[bufp];       line_no=linep;
        return(FALSE);
}


btw_expr()              /* bitwise expression   */
{
    int oldp=bufp, linep=line_no;

    if (match('~'))
    ;
    if (!btw_trm())           goto quit;

    while (bt_trms())
    ;

        return(TRUE);
quit:   bufp=oldp;    nextch=buf[bufp];       line_no=linep;
        return(FALSE);
}
```

```
bt_trms()               /* bitwise terms        */
{
    int oldp=bufp, linep=line_no;


    if (!match('|'))            goto quit;

    if (!btw_trm())             goto quit;


        return(TRUE);
quit:   bufp=oldp;    nextch=buf[bufp];       line_no=linep;
        return(FALSE);
}


btw_trm()               /* bitwise term         */
{

    if (!btw_fct())             return(FALSE)
    ;
    while (bt_fcts())
    ;

    return(TRUE);
}


bt_fcts()               /* bitwise factors      */
{
    int oldp=bufp, linep=line_no;


    if (!match('^'))                    goto quit;

    if (!btw_fct())                     goto quit;

        return(TRUE);
quit:   bufp=oldp;    nextch=buf[bufp];       line_no=linep;
        return(FALSE);
}


btw_fct()               /* bitwise factor       */
{

    if (!btw_elm())             return(FALSE)
    ;
    while (bt_elms())
    ;

    return(TRUE);
}
```

```
bt_elms()               /* bitwise elements      */
{
    int oldp=bufp, linep=line_no;


    if (!match('&'))            goto quit;

    if (!btw_elm())             goto quit;


        return(TRUE);
quit:   bufp=oldp;    nextch=buf[bufp];       line_no=linep;
        return(FALSE);
}


btw_elm()               /* bitwise element       */
{
    int oldp=bufp, linep=line_no;

    if (cmp_expr())
    ;
    else if (match('('))
    {
       if (!btw_expr())                 goto quit;

       if (!match(')'))                 goto quit;
    }
    else                                goto quit
    ;

        return(TRUE);
quit:   bufp=oldp;    nextch=buf[bufp];       line_no=linep;
        return(FALSE);
}


cmp_expr()              /* compound expression   */
{
    int oldp=bufp, linep=line_no;


    if (!cmp_trm())             return(FALSE)
    ;
    while (cp_trms())
    ;

    return(TRUE);
}
```

```
cp_trms()               /* compound terms        */
{
    int oldp=bufp, linep=line_no;

    if (!equ_op())              goto quit;

    if (!cmp_trm())             goto quit;


        return(TRUE);
quit: bufp=oldp;        nextch=buf[bufp];       line_no=linep;
        return(FALSE);
}


equ_op()                /* equality operators    */
{

    if (matchtoken("== ",0))
    ;
    else if (matchtoken("!= ",0))
    ;
    else      return(FALSE)
    ;

    return(TRUE);
}


cmp_trm()      /* compound term         */
{

    if (!cmp_fct())             return(FALSE)
    ;
    while (cp_fcts())
    ;

    return(TRUE);
}


cp_fcts()               /* compound factors      */
{

    if (!rel_op())                      goto quit;

    if (!cmp_fct())                     goto quit;


        return(TRUE);
quit:   return(FALSE);
}
```

```
rel_op()                /* relational operator  */
{

    if (match('<'))
    { if (match('='));
    }
    else if (match('>'))
    { if (match('='));
    }
    else      return(FALSE)
    ;

    return(TRUE);
}


cmp_fct()               /* compound factor      */
{
    int oldp=bufp, linep=line_no;


    if (shf_expr())
    ;
    else if (match('('))
    {
      if (!cmp_expr())                goto quit;

      if (!match(')'))                goto quit;
    }
    else                              goto quit
    ;

        return(TRUE);
quit:   bufp=oldp;    nextch=buf[bufp];       line_no=linep;
        return(FALSE);
}


shf_expr()              /* shift expression     */
{
    int oldp=bufp, linep=line_no;

    if (shf_init())
    ;
    if (!art_expr())        goto quit
    ;

         return(TRUE);
quit:   bufp=oldp;    nextch=buf[bufp];       line_no=linep;
        return(FALSE);
}
```

```
shf_init()              /* shift expression-initial     */
{
    int oldp=bufp, linep=line_no;


    if (!lvalue())              goto quit;

    if (!shf_op())              goto quit;


        return(TRUE);
quit:   bufp=oldp;    nextch=buf[bufp];       line_no=linep;
        return(FALSE);
}


shf_op()                /* shift operator       */
{

    if (matchtoken(")) ",0))
    ;
    else if (matchtoken("(( ",0))
    ;
    else      return(FALSE)
    ;

    return(TRUE);
}


art_expr()              /* arithmetic expression */
{
    int oldp=bufp, linep=line_no;

    if (match('-'))             /* unary operator       */
    ;
    if (!term())                goto quit;

    while (more_term())
    ;

        return(TRUE);
quit:   bufp=oldp;    nextch=buf[bufp];       line_no=linep;
        return(FALSE);
}


more_term()             /* more terms           */
{

    if (!add_op())              goto quit;

    if (!term())                goto quit;
```

```
        return(TRUE);
quit:   return(FALSE);
}


add_op()                /* additional operator  */
{

    if (match('+'))
    ;
    else if (match('-'))
    ;
    else       return(FALSE)
    ;

    return(TRUE);
}


term()                          /* term         */
{

    if (!factor())              return(FALSE)
    ;
    while (more_fcts())
    ;

    return(TRUE);
}


more_fcts()             /* more factors */
{
    int oldp=bufp, linep=line_no;


    if (!mul_op())              goto quit;

    if (!factor())              goto quit;


        return(TRUE);
quit:   bufp=oldp;      nextch=buf[bufp];       line_no=linep;
        return(FALSE);
}


mul_op()                /* multiplicational operator    */
{
    if (match('*'))
    ;
    else if (match('/'))
    ;
```

```
    else if (match('%'))
    ;
    else        return(FALSE)
    ;

    return(TRUE);
}


factor()                /* factor       */
{
    int oldp=bufp, linep=line_no;


    if (match('('))
    {
      if (!art_expr())          goto quit;

      if (!match(')'))          goto quit;
    }
    else if (func_call())
    ;
    else if (cnst_expr())
    ;
    else if (char_def())
    ;
    else if (incr_stmt())
    ;
    else if (lvalue())
    ;
    else                        goto quit
    ;

        return(TRUE);
quit:   bufp=oldp;    nextch=buf[bufp];       line_no=linep;
        return(FALSE);
}


cnst_expr()             /* constant expression  */
{

    if ( constant())
    ;
    else if (!cnst_id())
                return(FALSE);

    return(TRUE);
}
```

```
lvalue()                    /* left value              */
{
    int     oldp=bufp, linep=line_no;
    char    prnthsis=FALSE;


    if (match('('))             prnthsis=TRUE;

    if (array_elm())
    ;
    else if (id())
    ;
    else if (pntr_expr())
    ;
    else                        goto quit
    ;

    if (prnthsis)
      if (!match(')'))          goto quit;

        return(TRUE);
quit:   bufp=oldp;    nextch=buf[bufp];       line_no=linep;
        return(FALSE);
}


array_elm()             /* array element           */
{
    int oldp=bufp, linep=line_no;


    if (!id())                  goto quit;

    if (!index())               goto quit;


        return(TRUE);
quit:   bufp=oldp;    nextch=buf[bufp];       line_no=linep;
        return(FALSE);
}


index()                 /* index expression for arrays  */
{
    int oldp=bufp,    linep=line_no;


    if (!match('['))            goto quit;

    if (art_expr())
    ;
    if (!match(']'))            goto quit;
```

```
        return(TRUE);
quit:   bufp=oldp;      nextch=buf[bufp];       line_no=linep;
        return(FALSE);
}


primary()               /* primary expression   */
{

    if (cnst_expr())
    ;
    else if (array_elm())
    ;
    else if (id())
    ;
    else if (char_def())
    ;
    else if (string())
    ;
    else      return(FALSE)
    ;

    return(TRUE);
}
```

APPENDIX C

TERMINALS AND BASIC NONTERMINALS

```
isaltr(c)          /* is a letter?              */
int      c;        /* character to test         */
{
    return (( c>='A' && c<='Z' ) || ( c>='a' && c<='z' ));
}


iscapch(c)         /* is a capital letter?      */
int      c;        /* character to test         */
{
    return ( c>='A' && c<='Z');
}


isadgt(c)          /* is a digit?               */
int      c;        /* character to test         */
{
    return ( c>='0' && c<='9' );
}


isidch(c)          /* is identifier character? */
int     c;
{
    return( isaltr(c) || isadgt(c) || c=='_' );
}


delimiter()        /* is next-character a delimiter?   */
{
    return( nextch== '=' ||  nextch== '(' ||  nextch== ')' ||
            nextch== '+' ||  nextch== '-' ||  nextch== '|' ||
            nextch== '*' ||  nextch== '/' ||  nextch== '&' ||
            nextch== ';' ||  nextch== ',' ||  nextch== ':' ||
            nextch== ')' ||  nextch== '^' ||  nextch== '!' ||
            nextch== '%' ||  nextch== '(' ||  nextch== ')' ||
            nextch== '[' ||  nextch== ']' ||  nextch=='.'  ||
            whtchr(nextch)  );
}


whtchr(c)          /* is a white-character?     */
int      c;        /* character to test         */
{
    return ( c==' ' || c==TAB || c==CR || c==LF );
}
```

```
char_def()      /* is character definition? */
{

    char    blank=FALSE;        /* boolean var. for blanks */

    delwht();                   /* skip white characters   */

/* character definition should start with "'" character    */
    if (!match('\''))       return(FALSE)
    ;

/* consume the following white characters, if there is any  */

    while ( whtchr(nextch) )
    {
        nextch = getchr();
        blank = TRUE;
    }


    if ( nextch=='\'')
    {
/* check if character body is empty     */

        if ( !blank )
/* illegal character definition */  err_msg(ICDF)
        ;

/* else it is a blank character */
        else
        {
            nextch=getchr();
            return(TRUE);
        }
    }

/* if met '\' character, parse one more */
    if ( nextch=='\\' )     nextch=getchr()
    ;

/* parse the original character */
    nextch=getchr();


/* should finish with "'" character!    */

   if (!match('\''))
/* illegal character definition */  err_msg(ICDF)
   ;

    return(TRUE);
}
```

```
string()         /* is string? */
{

    char    i;          /* index variable        */

    delwht();           /* skip white characters     */


/* string must start with '"' character */
    if (!match('"'))        return(FALSE)
    ;


/* since strings not implemented in Tiny-C, just consume it */

    for (i=0; ( nextch!='"') && ( i<=MXSTR ); ++i )
        nextch=getchr();


/* check if it is too long  */

    if ( i > MXSTR )
/* string length too long  */   err_msg(SLTL)
    ;

/* should finish with '"' character */
    match('"');

    return(TRUE);
}


constant()  /* integer constant */
{
    char    i=0;            /* index variable        */

    delwht();           /* skip white characters     */

/* it should start with a digit     */
    if (!isadgt(nextch))    return(FALSE)
    ;

    while (isadgt(nextch))      /* parse the number */
    {

/* check if number length is too long   */

        if (i>=MXNML)
/* number length too long   */  err_msg(NLTL)
        ;
        else                    num_name[i++]=nextch
        ;
```

```
        nextch=getchr();
    }

    if (nextch!=' ' && !delimiter() )
/* delimiter was expected   */   err_msg(DWEX)
    ;

    num_name[i]=' ';

/* convert string "num_name" into numeric value */
    str_num();

/* add number into constant table   */
    add_num();

    return(TRUE);

}


cnst_id()   /* constant identifier  */
{
    int oldp,   linep;
    char    i=0;            /* index variable       */

    delwht();           /* skip white characters    */

    oldp=bufp;  linep=line_no;
    nextch=buf[bufp];


/* first character should be a capital letter   */
    if (!iscapch(nextch))       return(FALSE)
    ;

    while (iscapch(nextch))
    {

/* check if identifier length is too long   */

        if (i)=MXIDL)
/* identifier length too long   */  warning(ILTL)
        ;
        else                        id_name[i++]=nextch
        ;

        nextch=getchr();
    }

/* if following character is still a letter, it can be a lower
   letter only. since Tiny-C assumes constant identifiers are
   all capital letters, this cannot be a constant identifier */
```

```
    if (isaltr(nextch))     goto quit
    ;

    if (nextch!=' ' && !delimiter() )
/* delimiter was expected   */   err_msg(DWEX)
    ;

    id_name[i]=' ';
    return(TRUE);

/* backtrack on the scanner buffer, and return FALSE    */

quit:   bufp=oldp;    line_no=linep;    nextch=buf[bufp];
        return(FALSE);
}


id()        /* is identifier?   */
{

    char    i=0;            /* index variable       */

    delwht();           /* skip white characters    */


 /* should start with a letter  */
    if (!isaltr(nextch))        return(FALSE)
    ;

    while (isidch(nextch))
    {

/* check if identifier length is too long   */

        if (i)=MXIDL)
/* identifier length too long   */  warning(ILTL)
        ;
        else        id_name[i++]=nextch
        ;

        nextch=getchr();
    }

/* following character must be a delimiter! */

    if (nextch!=' ' && !delimiter() )
/* delimiter was expected   */   err_msg(DWEX)
    ;

    id_name[i]=' ';

/* if identifier is a reserved word, give error message */
```

```
    if (!rsvr_test())
/* reserved word not expected    */  err_msg(RVNE)
    ;

    return(TRUE);
}
```

APPENDIX D

TINY-C COMPILER ERROR AND WARNING MESSAGES

Error Messages:

```
#define AVNI    1     /* auto variables not implemented          */
#define SVNI    2     /* static variables not implemented        */
#define EVNI    3     /* external variables not implemented      */
#define RVNI    4     /* register variables not implemented      */
#define SMEX    5     /* semicolon was expected                  */
#define LINI    6     /* long integers not implemented           */
#define UINI    7     /* unsigned integers not implemented       */
#define FPNI    8     /* floating points not implemented         */
#define DPNI    9     /* double precisions not implemented       */
#define IDEX    10    /* identifier was expected                 */
#define IBSB    11    /* index body was supposed to be blank     */
#define RSBE    12    /* right square bracket was expected       */
#define CINI    14    /* compound initializers not implemented*/
#define EXEX    15    /* expression was expected                 */
#define LCBE    16    /* left curly bracket was expected         */
#define LPEX    17    /* left parenthesis was expected           */
#define RPEX    18    /* right parenthesis was expected          */
#define IEAC    19    /* identifier was expected after comma     */
#define EEAC    20    /* expression was expected after comma     */
#define PTNI    21    /* pointers not implemented                */
#define ARNI    22    /* arrays not implemented                  */
#define IPNI    55    /* include preprecsr not implemented       */
#define FTEX    23    /* filetype was expected                   */
#define IVFD    24    /* invalid file definition                 */
#define RCBE    25    /* right curly bracket was expected        */
#define PREX    26    /* parameter was expected                  */
#define PREC    27    /* parameter expected after comma          */
#define AEAC    28    /* an assignment expected after comma      */
#define LPEI    29    /* left parenthesis expected after if      */
#define LPEW    30    /* left prnthsis. expected after while     */
#define LPEF    31    /* left parenthesis expected after for     */
#define LPES    41    /* left prnthesis. expected after switch*/
#define ILEI    32    /* illegal logic expression in if          */
#define ILEW    33    /* illegal logic expression in while       */
#define ILEF    34    /* illegal logic expression in for         */
#define WMFD    35    /* while is missing from do                */
#define SSFI    36    /* a statement should follow after if      */
#define SSFE    37    /* a statement should follow after else */
#define SSFW    38    /* a statement should follow after while*/
#define SSFF    39    /* a statement should follow after for     */
#define SSFD    54    /* a statement should follow after do      */
#define SMIF    40    /* semicolon is missing in for             */
#define IAES    42    /* illegal arith. expression in switch     */
#define CSMS    43    /* case statement is missing               */
#define CLIM    44    /* colon is missing                        */
```

```
#define ICEC    45    /* invalid constant exprs. after case   */
#define AENI    46    /* address expression not implemented   */
#define OCNI    47    /* one's complement not implemented     */
#define BONI    48    /* bitwise operators not implemented    */
#define SENI    49    /* shift expressions not implemented    */
#define INPE    50    /* invalid pointer expression           */
#define INAE    51    /* invalid address expression           */
#define UNVR    52    /* unknown variable                     */
#define IAEI    53    /* invalid arith. expr. in array index  */
#define RVNE    56    /* reserved word not expected           */
#define ILFB    57    /* illegal function body                */
#define CBPR    58    /* input couldn't be parsed             */
#define TBBP    59    /* too big block to parse               */
#define UEOF    60    /* unexpected end of file               */
#define CETL    61    /* comment endless or too long          */
#define SETL    62    /* string is endless or too long        */
#define UMPH    63    /* unmatched parenthesis                */
#define SMUP    64    /* semicolon missing/unmatched prnthesis*/
#define CIEX    65    /* constant identifier expected         */
#define CVEX    66    /* constant value expected              */
#define STIF    67    /* symbol table is full                 */
#define NLTL    68    /* numeric length too long              */
#define TBNV    69    /* too big numeric value                */
#define NSIF    70    /* name string is full                  */
#define DTIF    71    /* definition table is full             */
#define CTIF    72    /* constant table is full               */
#define VSIF    73    /* variable string is full              */
#define LTIF    74    /* label table is full                  */
#define DCID    75    /* duplicated cons. id declaration      */
#define DLDC    76    /* duplicated label declaration         */
#define ICDF    77    /* illegal character definition         */
#define SLTL    78    /* string length too long               */
#define DWEX    79    /* delimiter was expected               */
#define UDLB    80    /* undeclared label                     */
#define DPDC    81    /* duplicated parameter declaration     */
#define DPFA    82    /* declared parameter is not a fun. arg.*/
#define UNPE    83    /* undeclared parameter exists          */
#define DFDC    84    /* duplicated function declaration      */
#define ICAN    86    /* inconsistent argument number         */
#define DDDC    87    /* duplicated default declaration       */
#define IVBR    88    /* invalid break usage                  */
#define TMNL    89    /* too many nested level                */
```

Warning Messages:

```
#define AFRI    1     /* all functions return integer */
#define CSIB    2     /* compound statement is blank  */
#define ILTL    3     /* identifier length too long   */
#define TOMF    4     /* main function is missing     */
```

APPENDIX E

INTERMEDIATE CODE DEFINITIONS FOR TINY-C

```
#define   IADD   1        /* integer addition       */
#define   ISUB   2        /* integer subtraction    */
#define   IMUL   3        /* integer multiply       */
#define   IDIV   4        /* integer division       */
#define   MDLS   5        /* integer modulus        */
#define   IMLB   6        /* label declaration      */
#define   JUMP   7        /* unconditional jump     */
#define   JPTR   8        /* jump if true           */
#define   JPFL   9        /* jump if false          */
#define   LGEX   10       /* logic expression       */
#define   CONS   11       /* constant               */
#define   ARID   12       /* array identifier       */
#define   VARB   13       /* variable               */
#define   UNMS   14       /* unary minus            */
#define   LGNT   15       /* unary logic not        */
#define   EQLT   18       /* equality               */
#define   NTEQ   19       /* not equal              */
#define   LSTN   20       /* less than              */
#define   GRTN   21       /* greater than           */
#define   LTEQ   22       /* less than or equal     */
#define   GTEQ   23       /* greater than or equal  */
#define   LGAN   24       /* logic and              */
#define   LGOR   25       /* logic or               */
#define   ASSN   26       /* assignment             */
#define   ADAS   27       /* addition-assignment    */
#define   SBAS   28       /* subtraction-assign     */
#define   MLAS   29       /* multiply-assignment    */
#define   DVAS   30       /* division-assignment    */
#define   MDAS   31       /* modulus-assignment     */
#define   FNCL   32       /* function call          */
#define   ARGM   33       /* argument               */
#define   EXLB   34       /* explicit label         */
#define   GOTO   35       /* jump to exp. label     */
#define   CASE   36       /* case statement         */
#define   TVAR   37       /* temporary variable #   */
#define   SWTC   38       /* switch statement       */
#define   PNTR   39       /* pointer                */
#define   ADDR   40       /* address                */
#define   RTRN   41       /* return                 */
#define   INDX   42       /* index                  */
#define   FNDC   45       /* function declaration   */
#define   STMT   46       /* statement              */
#define   DUMY   47       /* dummy statement        */
#define   BREK   48       /* break statement        */
#define   DFLT   49       /* default case           */
#define   INCR   50       /* increment              */
#define   DCRT   51       /* decrement              */
#define   INCL   52       /* increment, later       */
```

```
#define  DCRL  53        /* decrement, later      */
#define  NOOP  54        /* no operation          */
#define  TEMP  55        /* temporary variable    */
#define  CNVB  56        /* convert to boolean    */
#define  BTEM  57        /* boolean temporary     */
#define  FEND  58        /* function end          */
#define  MAIN  59        /* main function         */
#define  MEND  60        /* end of main function  */
```

APPENDIX F

TEST PROGRAMS FOR THE TINY-C COMPILER

Program 1.

```
main()
{
        int  joe, jimmy;

        joe=5;
        jimmy=15;

        switch ( joe * 5 )
        {
        case 12 : ++joe;
                  break;
        default : joe=jimmy+27;
                  break;
        case 14 : joe= jimmy--;
                  break;
        }

}
```

The Code for the Program 1.

```
codeseg equ     (0:0)
dataseg equ     (1:0)
        org     dataseg
sym1    ds      1
sym2    ds      1

        org     codeseg
        jmp     main
main:
        move    {int,5},r(0:0)
        move    {int,15},r(0:1)
        mov     r(0:0),r(0:2)
        mul     r(0:0),r(0:2)
        move    r(0:0),sym1
        move    r(0:1),sym2
        move    r(0:2),sym3
        jmp     imlbl0
imlbl2:
        move    sym1,r(0:0)
        add     {int,1},r(0:0),r(0:1)
        move    r(0:1),sym1
        jmp     imlbl1
imlbl3:
        move    {int,27},r(0:0)
        move    sym2,r(0:1)
        add     r(0:1),r(0:0)
        move    r(0:0),sym1
        jmp     imlbl1
imlbl4:
        move    sym2,r(0:0)
        sub     {int,1},r(0:0),r(0:1)
        move    r(0:0),sym1
        move    r(0:1),sym2
        jmp     imlbl1
imlbl0:
        move    sym3,r(0:0)
        move    {int,12},r(0:1)
        if      r(0:0)==r(0:1),implbl2
        move    {int,14},r(0:2)
        if      r(0:0)==r(0:2),implbl4
        jmp     implbl3
imlbl1:
        stop
```

Program 2.

```
main()
{
        int  joe, jimmy;

        joe=3;
        jimmy=joe*37;

        if ( joe ) 37 && jimmy(=joe && jimmy )
        {
                jimmy=18;
        }
        else
                jimmy=27;

        joe=jimmy+25;
}
```

The Code for the Program 2.

```
codeseg equ     (0:0)
dataseg equ     (1:0)
        org     dataseg
sym1    ds      1
sym2    ds      1

        org     codeseg
        jmp     main
main:
        move    {int,3},r(0:0)
        move    {int,37},r(0:1)
        mul     r(0:0),r(0:1)
        move    {int,37},r(0:2)
        if      r(0:0))r(0:2),blbl0
        move    {bool,false},r(0:3)
        jmp     blbl1
blbl0:
        move    {bool,true},r(0:3)
blbl1:
        if      r(0:1)(=r(0:0),blbl2
        move    {bool,false},r(0:4)
        jmp     blbl3
blbl2:
        move    {bool,true},r(0:4)
blbl3:
        and     r(0:3),r(0:4)
        if      r(0:1)=={int,0},blbl4
        move    {bool,true},r(0:5)
        jmp     blbl5
blbl4:
        move    {bool,false},r(0:5)
blbl5:
        and     r(0:4),r(0:5)
        move    r(0:0),sym1
        move    r(0:1),sym2
        if      r(0:5)=={bool,false},imlbl0:
        move    {int,18},r(0:0)
        move    r(0:0),sym2
        jmp     imlbl1
imlbl0:
        move    {int,27},r(0:0)
        move    r(0:0),sym2
imlbl1:
        move    {int,25},r(0:0)
        move    sym2,r(0:1)
        add     r(0:1),r(0:0)
        stop
```

Program 3.

```
function(joe,jimmy)
int joe, jimmy;
{


        do
                joe = jimmy++;

        while ( joe == 3);

        --jimmy;
}


main()
{
        int joe, jimmy;


        jimmy=5;

        do
                joe = jimmy--;
        while ( joe == 3);

        function(jimmy,joe);

        ++jimmy;
}
```

The Code for the Program 3.

```
codeseg equ     (0:0)
dataseg equ     (1:0)
        org     dataseg
sym1    ds      1
sym2    ds      1
sym4    ds      1
sym5    ds      1
:
        org     codeseg
        jmp     main

function:
        pop     s(0),r(0:0)
        pop     s(0),r(0:1)
        move    r(0:0),sym1
        move    r(0:1),sym2
imlbl0:
        move    sym2,r(0:0)
        add     {int,1},r(0:0),r(0:1)
        move    {int,3},r(0:2)
        if      r(0:0)==r(0:2),blbl0
        move    {bool,false},r(0:3)
        jump    blbl1
blbl0:
        move    {bool,true},r(0:3)
blbl1:
        move    r(0:0),sym1
        move    r(0:1),sym2
        if      r(0:3)=={bool,true},imlbl0:
imlbl1:
        move    sym2,r(0:0)
        sub     {int,1},r(0:0),r(0:1)
        push    {int,1},s(0)
        rts     s(1)


main:
        move    {int,5},r(0:0)
        move    r(0:0),sym5
        move    r(0:1),sym2
imlbl2:
        move    sym5,r(0:0)
        sub     {int,1},r(0:0),r(0:1)
        move    {int,3},r(0:2)
        if      r(0:0)==r(0:2),blbl2
        move    {bool,false},r(0:3)
        jump    blbl3
blbl2:
        move    {bool,true},r(0:3)
```

```
blbl3:
        move    r(0:0),sym4
        move    r(0:1),sym5
        if      r(0:3)=={bool,true},imlbl2:
imlbl3:
        move    sym4,r(0:0)
        push    r(0:0),s(0)
        move    sym5,r(0:1)
        push    r(0:1),s(0)
        jsr     funcall,s(1)
        pop     s(0),r(0:2)
        add     {int,1},r(0:1),r(0:3)
        stop
```

LIST OF REFERENCES

1. Yurchak, J., The Formal Specification of an Abstract Machine: Design an Implementation, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1984.

2. Hunter, J. E., The Formal Specification of a Visual Display Device: Design and Implementation, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1985.

3. Zang, K. H., The Formal Specification of an Abstract Database: Design and Implementation, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1985.

4. Ullman J.D., Sethi R., Aho A.V., Compilers Principles, Techniques, and Tools, Addison-Wesley, 1986.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5000	2
3.	Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100	1
4.	Computer Technology Programs, Code 37 Naval Postgraduate School Monterey, California 93943-5100	1
5.	Daniel L. DAVIS, Code 52Vv Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100	5
6.	Bruce J. MacLENNAN, Code Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100	1
7.	Deniz Harp Okulu Komutanligi Egitim Daire Baskanligi Tuzla, ISTANBUL TURKEY	2
8.	Metin Gursel OZISIK Rasim Pasa Mah. Taslibayir Sok. Denizsever Apt. 47-8 Kadikoy, ISTANBUL TURKEY	5

219571

Thesis
09965
c.1

Ozisik
Design and implementation of a C compiler for an abstract machine.